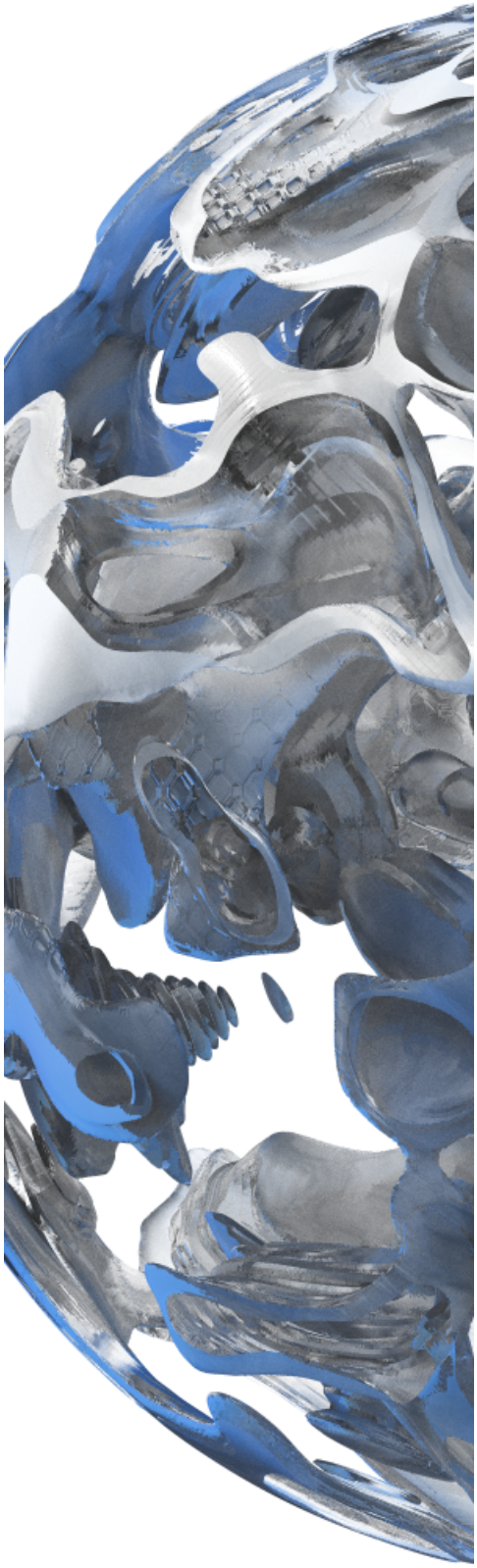


The VTK-m User's Guide

VTK-m version 2.2

Kenneth Moreland

With contributions from:
Vicente Bolea, Hank Childs, Nickolas Davis, Mark Kim, James Kress, Matthew Letter, Li-Ta Lo, Robert Maynard, Sujin Philip, David Pugmire, Nick Thompson, Allison Vacanti, Abhishek Yenpure, and the VTK-m community

August 1, 2024
ORNL/TM-2024/3443

http://m.vtk.org
http://kitware.com





This project has been funded in whole or in part with Federal funds from the Department of Energy, including from Oak Ridge National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories.

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This work was supported in part by the U.S. Department of Energy (DOE) RAPIDS SciDAC project under contract number DE-AC05-00OR22725 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

Moreland, K. (2024). *The VTK-m User's Guide, version 2.2*, Tech report ORNL/TM-2024/3443, Oak Ridge National Laboratory.

CONTRIBUTORS

This book includes contributions from the VTK-m community including the VTK-m development team and the user community. We would like to thank the following people for their significant contributions to this text:

Vicente Bolea, **Nickolas Davis**, **Matthew Letter**, and **Nick Thompson** for their help keeping the user's guide up to date with the VTK-m source code.

Sujin Philip, **Robert Maynard**, **James Kress**, **Abhishek Yenpure**, **Mark Kim**, and **Hank Childs** for their descriptions of numerous filters.

Allison Vacanti for her documentation of several VTK-m features in Sections 26.11 and 26.12 and select filters.

David Pugmire for his documentation of partitioned data sets (Section 7.5) and select filters.

Abhishek Yenpure and **Li-Ta Lo** for their documentation of locator structures (Chapter 30).

Li-Ta Lo for his documentation of random array handles and particle density filters.

James Kress for his documentation on VTK-m's testing classes.

Manish Mathai for his documentation of rendering features (Chapter 10).

Vicente Bolea for his documentation of the VTK-m's usage in distributed systems.

ABOUT THE COVER

The interior cover image represents seismic wave propagation through the Earth. The visualization is provided by Matthew Larsen at Lawrence Livermore National Laboratory.

Join the VTK-m Community at http://m.vtk.org

CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

Part I

Getting Started

CHAPTER

ONE

INTRODUCTION

High-performance computing relies on ever finer threading. Advances in processor technology include ever greater numbers of cores, hyperthreading, accelerators with integrated blocks of cores, and special vectorized instructions, all of which require more software parallelism to achieve peak performance. Traditional visualization solutions cannot support this extreme level of concurrency. Extreme scale systems require a new programming model and a fundamental change in how we design algorithms. To address these issues we created VTK-m: the visualization toolkit for multi-/many-core architectures.

VTK-m supports a number of algorithms and the ability to design further algorithms through a top-down design with an emphasis on extreme parallelism. VTK-m also provides support for finding and building links across topologies, making it possible to perform operations that determine manifold surfaces, interpolate generated values, and find adjacencies. Although VTK-m provides a simplified high-level interface for programming, its template-based code removes the overhead of abstraction.

VTK-m simplifies the development of parallel scientific visualization algorithms by providing a framework of supporting functionality that allows developers to focus on visualization operations. Consider the listings in Figure 1.1 that compares the size of the implementation for the Marching Cubes algorithm in VTK-m with the equivalent reference implementation in the CUDA software development kit. Because VTK-m internally manages the parallel distribution of work and data, the VTK-m implementation is shorter and easier to maintain. Additionally, VTK-m provides data abstractions not provided by other libraries that make code written in VTK-m more versatile.

1.1 How to Use This Guide

This user's guide is organized into 5 parts to help guide novice to advanced users and to provide a convenient reference. Part I, Getting Started, provides a brief overview of using VTK-m. This part provides instructions on building VTK-m and some simple examples of using VTK-m. Users new to VTK-m are well served to read through Part I first to become acquainted with the basic concepts.

The remaining parts, which provide detailed documentation of increasing complexity, have chapters that do not need to be read in detail. Readers will likely find it useful to skip to specific topics of interest.

Part II, Using VTK-m, dives deeper into the VTK-m library. It provides much more detail on the concepts introduced in Part I and introduces new topics helpful to people who use VTK-m's existing algorithms.

Part III, Developing Algorithms, documents how to use VTK-m's framework to develop new or custom visualization algorithms. In this part we dive into the inner workings of filters and introduce the concept of a *worklet*, which is the base unit used to write a device-portable algorithm in VTK-m. Part III also documents many supporting functions that are helpful in implementing visualization algorithms.

Figure 1.1: Comparison of the Marching Cubes algorithm in VTK-m and the reference implementation in the CUDA SDK. Implementations in VTK-m are simpler, shorter, more general, and easier to maintain. (Lines of code (LOC) measurements come from cloc.)

Part IV, Advanced Development, explores in more detail how VTK-m manages memory and devices. This information describes how to adapt VTK-m to custom data structures and new devices.

Part V, Core Development, exposes the inner workings of VTK-m. These concepts allow you to design new algorithmic structures not already available in VTK-m.

1.2 Conventions Used in This Guide

When documenting the VTK-m API, the following conventions are used.

- Filenames are printed in a sans serif font.
- C++ code is printed in a `monospace font`.
- Macros and namespaces from VTK-m are printed in `red`.

- Identifiers from VTK-m are printed in `blue`.
- Signatures, described in Chapter 17, and the tags used in them are printed in `green`.

This guide provides actual code samples throughout its discussions to demonstrate their use. These examples are all valid code that can be compiled and used although it is often the case that code snippets are provided. In such cases, the code must be placed in a larger context.

Did you know?

*In this guide we periodically use these **Did you know?** boxes to provide additional information related to the topic at hand.*

Common Errors

***Common Errors** blocks are used to highlight some of the common problems or complications you might encounter when dealing with the topic of discussion.*

CHAPTER

TWO

BUILD AND INSTALL VTK-M

Before we begin describing how to develop with VTK-m, we have a brief overview of how to build VTK-m, optionally install it on your system, and start your own programs that use VTK-m.

2.1 Getting VTK-m

VTK-m is an open source software product where the code is made freely available. To get the latest released version of VTK-m, go to the VTK-m releases page:

https://gitlab.kitware.com/vtk/vtk-m/-/releases

From there with your favorite browser you may download the source code from any of the recent VTK-m releases in a variety of different archive files such as zip or tar gzip.

For access to the most recent work, the VTK-m development team provides public anonymous read access to their main source code repository. The main VTK-m repository on a GitLab instance hosted at Kitware, Inc. The repository can be browsed from its project web page:

https://gitlab.kitware.com/vtk/vtk-m

We leave access to the git hosted repository as an exercise for the user. Those interested in git access for the purpose of contributing to VTK-m should consult the CONTRIBUTING guidelines documented in the source code.[1]

2.2 Configure VTK-m

VTK-m uses a cross-platform configuration tool named CMake to simplify the configuration and building across many supported platforms. CMake is available from many package distribution systems and can also be downloaded for many platforms from http://cmake.org.

Most distributions of CMake come with a convenient GUI application (cmake-gui) that allows you to browse all of the available configuration variables and run the configuration. Many distributions also come with an alternative terminal-based version (ccmake), which is helpful when accessing remote systems where creating GUI windows is difficult.

[1] https://gitlab.kitware.com/vtk/vtk-m/blob/master/CONTRIBUTING.md

One helpful feature of CMake is that it allows you to establish a build directory separate from the source directory, and the VTK-m project requires that separation. Thus, when you run CMake for the first time, you want to set the build directory to a new empty directory and the source to the downloaded or cloned files. The following example shows the steps for the case where the VTK-m source is cloned from the git repository. (If you extracted files from an archive downloaded from the VTK-m web page, the instructions are the same from the second line down.)

Example 2.1: Running CMake on downloaded VTK-m source (Unix commands).

```
tar xvzf ~/Downloads/vtk-m-v2.2.0.tar.gz
mkdir vtkm-build
cd vtkm-build
cmake-gui ../vtk-m-v2.2.0
```

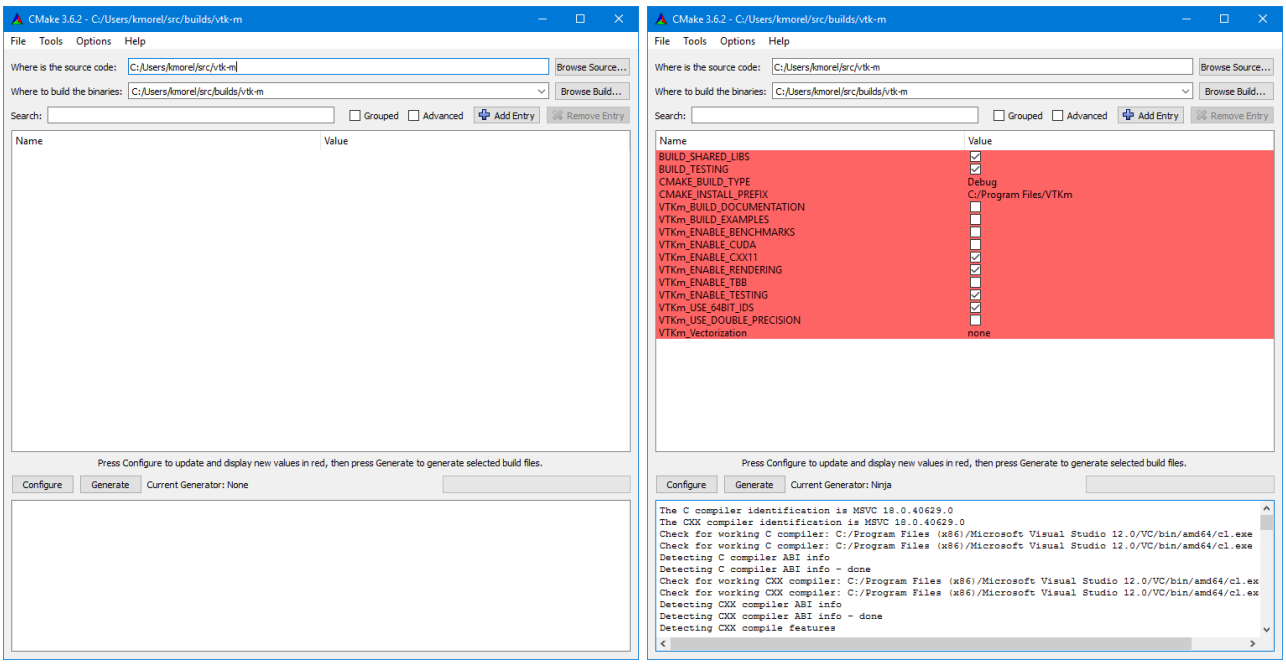


Figure 2.1: The CMake GUI configuring the VTK-m project. At left is the initial blank configuration. At right is the state after a configure pass.

The first time the CMake GUI runs, it initially comes up blank as shown at left in Figure 2.1. Verify that the source and build directories are correct (located at the top of the GUI) and then click the "Configure" button near the bottom. The first time you run configure, CMake brings up a dialog box asking what generator you want for the project. This allows you to select what build system or IDE to use (e.g. make, ninja, Visual Studio). Once you click "Finish," CMake will perform its first configuration. Don't worry if CMake gives an error about an error in this first configuration process.

Common Errors

Most options in CMake can be reconfigured at any time, but not the compiler and build system used. These must be set the first time configure is run and cannot be subsequently changed. If you want to change the compiler or the project file types, you will need to delete everything in the build directory and start over.

After the first configuration, the CMake GUI will provide several configuration options as shown in Figure 2.1 on the right. You now have a chance to modify the configuration of VTK-m, which allows you to modify both the behavior of the compiled VTK-m code as well as find components on your system. Using the CMake GUI is usually an iterative process where you set configuration options and re-run "Configure." Each time you configure, CMake might find new options, which are shown in red in the GUI.

It is often the case during this iterative configuration process that configuration errors occur. This can occur after a new option is enabled but CMake does not automatically find the necessary libraries to make that feature possible. For example, to enable TBB support, you may have to first enable building TBB, configure for TBB support, and then tell CMake where the TBB include directories and libraries are.

Once you have set all desired configuration variables and resolved any CMake errors, click the "Generate" button. This will create the build files (such as makefiles or project files depending on the generator chosen at the beginning). You can then close the CMake GUI.

There are a great number of configuration parameters available when running CMake on VTK-m. The following list contains the most common configuration parameters.

BUILD_SHARED_LIBS Determines whether static or shared libraries are built.

CMAKE_BUILD_TYPE Selects groups of compiler options from categories like Debug and Release. Debug builds are, obviously, easier to debug, but they run *much* slower than Release builds. Use Release builds whenever releasing production software or doing performance tests.

CMAKE_INSTALL_PREFIX The root directory to place files when building the install target.

VTKm_ENABLE_EXAMPLES The VTK-m repository comes with an examples directory. This macro determines whether they are built.

VTKm_ENABLE_BENCHMARKS If on, the VTK-m build includes several benchmark programs. The benchmarks are regression tests for performance.

VTKm_ENABLE_CUDA Determines whether VTK-m is built to run on CUDA GPU devices.

VTKm_CUDA_Architecture Specifies what GPU architecture(s) to build CUDA for. The options include native, fermi, kepler, maxwell, pascal, volta, and turing.

VTKm_ENABLE_KOKKOS Determines whether VTK-m is built using the Kokkos portable library. Kokkos, (https://kokkos.github.io/kokkos-core-wiki/) can be configured to support several backends that VTK-m can leverage.

VTKm_ENABLE_OPENMP Determines whether VTK-m is built to run on multi-core devices using OpenMP pragmas provided by the C++ compiler.

VTKm_ENABLE_RENDERING Determines whether to build the rendering library.

VTKm_ENABLE_TBB Determines whether VTK-m is built to run on multi-core x86 devices using the Intel Threading Building Blocks library.

VTKm_ENABLE_TESTING If on, the VTK-m build includes building many test programs. The VTK-m source includes hundreds of regression tests to ensure quality during development.

VTKm_ENABLE_TUTORIALS If on, several small example programes used for the VTK-m tutorial are built.

VTKm_USE_64BIT_IDS If on, then VTK-m will be compiled to use 64-bit integers to index arrays and other lists. If off, then VTK-m will use 32-bit integers. 32-bit integers take less memory but could cause failures on larger data.

VTKm_USE_DOUBLE_PRECISION If on, then VTK-m will use double precision (64-bit) floating point numbers for calculations where the precision type is not otherwise specified. If off, then single precision (32-bit) floating point numbers are used. Regardless of this setting, VTK-m's templates will accept either type.

2.3 Building VTK-m

Once CMake successfully configures VTK-m and generates the files for the build system, you are ready to build VTK-m. As stated earlier, CMake supports generating configuration files for several different types of build tools. Make and ninja are common build tools, but CMake also supports building project files for several different types of integrated development environments such as Microsoft Visual Studio and Apple XCode.

The VTK-m libraries and test files are compiled when the default build is invoked. For example, if Makefiles were generated, the build is invoked by calling make in the build directory. Expanding on Example 2.1

Example 2.2: Using make to build VTK-m.

```
1 tar xvzf ~/Downloads/vtk-m-v2.2.0.tar.gz
2 mkdir vtkm-build
3 cd vtkm-build
4 cmake-gui ../vtk-m-v2.2.0
5 make -j
6 make install
```

Did you know?

The Makefiles and other project files generated by CMake support parallel builds, which run multiple compile steps simultaneously. On computers that have multiple processing cores (as do almost all modern computers), this can significantly speed up the overall compile. Some build systems require a special flag to engage parallel compiles. For example, make requires the -j flag to start parallel builds as demonstrated in Example 2.2.

Did you know?

Example 2.2 assumes that a make build system was generated, which is the default on most system. However, CMake supports many more build systems, which use different commands to run the build. If you are not sure what the appropriate build command is, you can run cmake --build to allow CMake to start the build using whatever build system is being used.

Common Errors

CMake allows you to switch between several types of builds including default, Debug, and Release. Programs and libraries compiled as release builds can run much *faster than those from other types of builds. Thus, it is important to perform Release builds of all software released for production or where runtime is a concern. Some integrated development environments such as Microsoft Visual Studio allow you to specify the different build types within the build system. But for other build programs, like make, you have to*

specify the build type in the CMAKE_BUILD_TYPE CMake configuration variable, which is described in Section 2.2.

CMake creates several build "targets" that specify the group of things to build. The default target builds all of VTK-m's libraries as well as tests, examples, and benchmarks if enabled. The test target executes each of the VTK-m regression tests and verifies they complete successfully on the system. The install target copies the subset of files required to use VTK-m to a common installation directory. The install target may need to be run as an administrator user if the installation directory is a system directory.

Did you know?

VTK-m contains a significant amount of regression tests. If you are not concerned with testing a build on a given system, you can turn off building the testing, benchmarks, and examples using the CMake configuration variables described in Section 2.2. This can shorten the VTK-m compile time.

2.4 Linking to VTK-m

Ultimately, the value of VTK-m is the ability to link it into external projects that you write. The header files and libraries installed with VTK-m are typical, and thus you can link VTK-m into a software project using any type of build system. However, VTK-m comes with several CMake configuration files that simplify linking VTK-m into another project that is also managed by CMake. Thus, the documentation in this section is specifically for finding and configuring VTK-m for CMake projects.

VTK-m can be configured from an external project using the `find_package` CMake function. The behavior and use of this function is well described in the CMake documentation. The first argument to `find_package` is the name of the package, which in this case is `VTKm`. CMake configures this package by looking for a file named VTKmConfig.cmake, which will be located in the lib/cmake/vtkm-2.2 directory of the install or build of VTK-m. The configurable CMake variable CMAKE_PREFIX_PATH can be set to to the build or install directory, or VTKm_DIR can be set to the directory that contains this file.

Example 2.3: Loading VTK-m configuration from an external CMake project.

```
1 find_package(VTKm REQUIRED)
```

Did you know?

The CMake `find_package` *function also supports several features not discussed here including specifying a minimum or exact version of VTK-m and turning off some of the status messages. See the CMake documentation for more details.*

When you load the VTK-m package in CMake, several libraries are defined. Projects building with VTK-m components should link against one or more of these libraries as appropriate, typically with the target_link_libraries command.

Example 2.4: Linking VTK-m code into an external program.

```
1 find_package(VTKm REQUIRED)
```

```
2
3 add_executable(myprog myprog.cxx)
4 target_link_libraries(myprog vtkm_filter)
```

Several library targets are provided, but most projects will need to link in one or more of the following.

vtkm::cont Contains the base objects used to control VTK-m. This library must always be linked in.

vtkm::filter Contains VTK-m's pre-built filters including but not limited to CellAverage, CleanGrid, Contour, ExternalFaces, and PointAverage. Applications that are looking to use VTK-m filters will need to link to this library. The filters are further broken up into several smaller library packages (such as vtkm::filter_contour , vtkm::filter_flow , vtkm::filter_field_transform , and many more. vtkm::filter is actually a meta library that links all of these filter libraries to a CMake target.

vtkm::io Contains VTK-m's facilities for interacting with files. For example, reading and writing png, NetBPM, and VTK files.

vtkm::rendering Contains VTK-m's rendering components. This library is only available if VTKm_ENABLE_RENDERING is set to true.

vtkm::source Contains VTK-m's pre-built dataset generators including but not limited to Wavelet, Tangle, and Oscillator. Most applications will not need to link to this library.

Did you know?

The "libraries" made available in the VTK-m do more than add a library to the linker line. These libraries are actually defined as external targets that establish several compiler flags, like include file directories. Many CMake packages require you to set up other target options to compile correctly, but for VTK-m it is sufficient to simply link against the library.

Common Errors

Because the VTK-m CMake libraries do more than set the link line, correcting the link libraries can do more than fix link problems. For example, if you are getting compile errors about not finding VTK-m header files, then you probably need to link to one of VTK-m's libraries to fix the problem rather than try to add the include directories yourself.

The following is a list of all the CMake variables defined when the `find_package` function completes.

VTKm_FOUND Set to true if the VTK-m CMake package is successfully loaded. If `find_package` was not called with the `REQUIRED` option, then this variable should be checked before attempting to use VTK-m.

VTKm_VERSION The version number of the loaded VTK-m package. The package also sets VTKm_VERSION_MAJOR, VTKm_VERSION_MINOR, and VTKm_VERSION_PATCH to get the individual components of the version. There is also a VTKm_VERSION_FULL that is augmented with a partial git SHA to identify snapshots in between releases.

VTKm_ENABLE_CUDA Set to true if VTK-m was compiled for CUDA.

VTKm_ENABLE_Kokkos Set to true if VTK-m was compiled with Kokkos.

VTKm_ENABLE_OPENMP Set to true if VTK-m was compiled for OpenMP.

VTKm_ENABLE_TBB Set to true if VTK-m was compiled for TBB.

VTKm_ENABLE_RENDERING Set to true if the VTK-m rendering library was compiled.

VTKm_ENABLE_MPI Set to true if VTK-m was compiled with MPI support.

These package variables can be used to query whether optional components are supported before they are used in your CMake configuration.

Example 2.5: Using an optional component of VTK-m.

```
1 find_package(VTKm REQUIRED)
2
3 if (NOT VTKm_ENABLE_RENDERING)
4   message(SEND_ERROR "VTK-m must be built with rendering on.")
5 endif()
6
7 add_executable(myprog myprog.cxx)
8 target_link_libraries(myprog vtkm_cont vtkm_rendering)
```

CHAPTER

THREE

QUICK START

In this chapter we go through the steps to create a simple program that uses VTK-m. This "hello world" example presents only the bare minimum of features available. The remainder of this book documents dives into much greater detail.

We will call the example program we are building VTKmQuickStart. It will demonstrate reading data from a file, processing the data with a filter, and rendering an image of the data. Readers who are less interested in an explanation and are more interested in browsing some code can skip to Section 3.5 on page 17.

3.1 Initialize

The first step to using VTK-m is to initialize the library. Although initializing VTK-m is *optional*, it is recommend to allow VTK-m to configure devices and logging. Initialization is done by calling the `vtkm::cont::Initialize` function. The `Initialize` function is defined in the vtkm/cont/Initialize.h header file.

`Initialize` takes the `argc` and `argv` arguments that are passed to the `main` function of your program, find any command line arguments relevant to VTK-m, and remove them from the list to make further command line argument processing easier.

Example 3.1: Initializing VTK-m.

```
1 int main(int argc, char* argv[])
2 {
3   vtkm::cont::Initialize(argc, argv);
```

`Initialize` has many options to customize command line argument processing. See Chapter 6 for more details.

Did you know?

Don't have access to `argc` *and* `argv`*? No problem. You can call* `vtkm::cont::Initialize` *with no arguments.*

3.2 Reading a File

VTK-m comes with a simple I/O library that can read and write files in VTK legacy format. These files have a ".vtk" extension.

VTK legacy files can be read using the `vtkm::io::VTKDataSetReader` object, which is declared in the vtkm/io/VTKDataSetReader.h header file. The object is constructed with a string specifying the filename (which for this example we will get from the command line). The data is then read in by calling the `VTKDataSetReader::ReadDataSet` method.

Example 3.2: Reading data from a VTK legacy file.

```
1 vtkm::io::VTKDataSetReader reader(argv[1]);
2 vtkm::cont::DataSet inData = reader.ReadDataSet();
```

The `ReadDataSet` method returns the data in a `vtkm::cont::DataSet` object. The structure and features of a `DataSet` object is described in Chapter 7. For the purposes of this quick start, we will treat `DataSet` as a mostly opaque object that gets passed to and from operations in VTK-m.

More information about VTK-m's file readers and writers can be found in Chapter 8.

3.3 Running a Filter

Algorithms in VTK-m are encapsulated in units called *filters*. A filter takes in a `DataSet`, processes it, and returns a new `DataSet`. The returned `DataSet` often, but not always, contains data inherited from the source data.

VTK-m comes with many filters, which are documented in Chapter 9. For this example, we will demonstrate the use of the `vtkm::filter::MeshQuality` filter, which is defined in the vtkm/filter/MeshQuality.h header file. The `MeshQuality` filter will compute for each cell in the input data will compute a quantity representing some metric of the cell's shape. Several metrics are available, and in this example we will find the area of each cell.

Like all filters, `MeshQuality` contains an `Execute` method that takes an input `DataSet` and produces an output `DataSet`. It also has several methods used to set up the parameters of the execution. Section 9.1.9 provides details on all the options of `MeshQuality`. Suffice it to say that in this example we instruct the filter to find the area of each cell, which it will output to a field named "area."

Example 3.3: Running a filter.

```
1 vtkm::filter::mesh_info::MeshQuality cellArea;
2 cellArea.SetMetric(vtkm::filter::mesh_info::CellMetric::Area);
3 vtkm::cont::DataSet outData = cellArea.Execute(inData);
```

3.4 Rendering an Image

Although it is possible to leverage external rendering systems, VTK-m comes with its own self-contained image rendering algorithms. These rendering classes are completely implemented with the parallel features provided by VTK-m, so using rendering in VTK-m does not require any complex library dependencies.

Even a simple rendering scene requires setting up several parameters to establish what is to be featured in the image including what data should be rendered, how that data should be represented, where objects should be placed in space, and the qualities of the image to generate. Consequently, setting up rendering in VTK-m involves many steps. Chapter 10 goes into much detail on the ways in which a rendering scene is specified. For now, we just briefly present some boilerplate to achieve a simple rendering.

Example 3.4: Rendering data.

```
1 vtkm::rendering::Actor actor(
2   outData.GetCellSet(), outData.GetCoordinateSystem(), outData.GetField("area"));
3
```

```
4   vtkm::rendering::Scene scene;
5   scene.AddActor(actor);
6
7   vtkm::rendering::MapperRayTracer mapper;
8
9   vtkm::rendering::CanvasRayTracer canvas(1280, 1024);
10
11  vtkm::rendering::View3D view(scene, mapper, canvas);
12
13  view.Paint();
14
15  view.SaveAs("image.png");
```

The first step in setting up a render is to create a *scene*. A scene comprises some number of *actors*, which represent some data to be rendered in some location in space. In our case we only have one `DataSet` to render, so we simply create a single actor and add it to a scene as shown in lines 1–5.

The second step in setting up a render is to create a *view*. The view comprises the aforementioned scene, a *mapper*, which describes how the data are to be rendered, and a *canvas*, which holds the image buffer and other rendering context. The view is created in line 11. The image generation is then performed by calling `Paint` on the view object (line 13). However, the rendering done by VTK-m's rendering classes is performed offscreen, which means that the result does not appear on your computer's monitor. The easiest way to see the image is to save it to an image file using the `SaveAs` method (line 15).

3.5 The Full Example

Putting together the examples from Sections 3.1 to 3.4, here is a complete program for reading, processing, and rendering data with VTK-m.

Example 3.5: Simple example of using VTK-m.

```
1  #include <vtkm/cont/Initialize.h>
2
3  #include <vtkm/io/VTKDataSetReader.h>
4
5  #include <vtkm/filter/mesh_info/MeshQuality.h>
6
7  #include <vtkm/rendering/Actor.h>
8  #include <vtkm/rendering/CanvasRayTracer.h>
9  #include <vtkm/rendering/MapperRayTracer.h>
10 #include <vtkm/rendering/Scene.h>
11 #include <vtkm/rendering/View3D.h>
12
13 int main(int argc, char* argv[])
14 {
15   vtkm::cont::Initialize(argc, argv);
16
17   if (argc != 2)
18   {
19     std::cerr << "USAGE: " << argv[0] << " <file.vtk>" << std::endl;
20     return 1;
21   }
22
23   // Read in a file specified in the first command line argument.
24   vtkm::io::VTKDataSetReader reader(argv[1]);
25   vtkm::cont::DataSet inData = reader.ReadDataSet();
26
27   // Run the data through the elevation filter.
28   vtkm::filter::mesh_info::MeshQuality cellArea;
29   cellArea.SetMetric(vtkm::filter::mesh_info::CellMetric::Area);
```

```
30    vtkm::cont::DataSet outData = cellArea.Execute(inData);
31
32    // Render an image and write it out to a file.
33    vtkm::rendering::Actor actor(
34      outData.GetCellSet(), outData.GetCoordinateSystem(), outData.GetField("area"));
35
36    vtkm::rendering::Scene scene;
37    scene.AddActor(actor);
38
39    vtkm::rendering::MapperRayTracer mapper;
40
41    vtkm::rendering::CanvasRayTracer canvas(1280, 1024);
42
43    vtkm::rendering::View3D view(scene, mapper, canvas);
44
45    view.Paint();
46
47    view.SaveAs("image.png");
48
49    return 0;
50  }
```

3.6 Build Configuration

Now that we have the program listed in Example 3.5, we still need to compile it with the appropriate compilers and flags. By far the easiest way to compile VTK-m code is to use CMake. CMake commands that can be used to link code to VTK-m are discussed in Section 2.4. The following example provides a minimal CMakeLists.txt required to build this program.

Example 3.6: CMakeLists.txt to build a program using VTK-m.

```
1  cmake_minimum_required(VERSION 3.13)
2  project(VTKmQuickStart CXX)
3
4  find_package(VTKm REQUIRED)
5
6  add_executable(VTKmQuickStart VTKmQuickStart.cxx)
7  target_link_libraries(VTKmQuickStart vtkm::filter vtkm::rendering)
```

The first two lines contain boilerplate for any CMakeLists.txt file. They all should declare the minimum CMake version required (for backward compatibility) and have a `project` command to declare which languages are used.

The remainder of the commands find the VTK-m library, declare the program begin compiled, and link the program to the VTK-m library. These steps are described in detail in Section 2.4.

Part II

Using VTK-m

CHAPTER

FOUR

BASE TYPES

It is common for a framework to define its own types. Even the C++ standard template library defines its own base types like `std::size_t` and `std::pair`. VTK-m is no exception.

In fact VTK-m provides a great many base types. It is the general coding standard of VTK-m to not directly use the base C types like `int` and `float` and instead to use types declared in VTK-m. The rational is to precisely declare the representation of each variable to prevent future trouble.

Consider that you are programming something and you need to declare an integer variable. You would declare this variable as `int`, right? Well, maybe. In C++, the declaration `int` does not simply mean "an integer." `int` means something much more specific than that. If you were to look up the C++11 standard, you would find that `int` is an integer represented in 32 bits with a two's complement signed representation. In fact, a C++ compiler has no less than 8 standard integer types.[1]

So, `int` is nowhere near as general as the code might make it seem, and treating it as such could lead to trouble. For example, consider the MPI standard, which, back in the 1990's, implicitly selected `int` for its indexing needs. Fast forward to today where there is a need to reference buffers with more than 2 billion elements, but the standard is stuck with a data type that cannot represent sizes that big.[2]

Consequently, we feel that with VTK-m it is best to declare the intention of a variable with its declaration, which should help both prevent errors and future proof code. All the types presented in this chapter are declared in vtkm/Types.h, which is typically included either directly or indirectly by all source using VTK-m.

4.1 Floating Point Types

VTK-m declares 2 types to hold floating point numbers: `vtkm::Float32` and `vtkm::Float64`. These, of course, represent floating point numbers with 32-bits and 64-bits of precision, respectively. These should be used when the precision of a floating point number is predetermined.

When the precision of a floating point number is not predetermined, operations usually have to be overloaded or templated to work with multiple precisions. In cases where a precision must be set, but no particular precision is specified, `vtkm::FloatDefault` should be used. `vtkm::FloatDefault` will be set to either `vtkm::Float32` or `vtkm::Float64` depending on whether the CMake option VTKM_USE_DOUBLE_PRECISION was set when VTK-m was compiled, as discussed in Section 2.2. Using `vtkm::FloatDefault` makes it easier for users to trade off precision and speed.

[1] I intentionally use the phrase "no less than" for our pedantic readers. One could argue that `char` and `bool` are treated distinctly by the compiler even if their representations match either `signed char` or `unsigned char`. Furthermore, many modern C++ compilers have extensions for less universally accepted types like 128-bit integers.

[2] To be fair, it is *possible* to represent buffers this large in MPI, but it is extraordinarily awkward to do so.

4.2 Integer Types

The most common use of an integer in VTK-m is to index arrays. For this purpose, the `vtkm::Id` type should be used. (The width of `vtkm::Id` is determined by the VTKm_USE_64BIT_IDS CMake option.)

VTK-m also has a secondary index type named `vtkm::IdComponent`, which is smaller and typically used for indexing groups of components within a thread. For example, if you had an array of 3D points, you would use `vtkm::Id` to reference each point, and you would use `vtkm::IdComponent` to reference the respective x, y, and z components.

Did you know?

The VTK-m index types, `vtkm::Id` and `vtkm::IdComponent` use signed integers. This breaks with the convention of other common index types like the C++ standard template library `std::size_t`, which use unsigned integers. Unsigned integers make sense for indices as a valid index is always 0 or greater. However, doing things like iterating in a for loop backward, representing relative indices, and representing invalid values is much easier with signed integers. Thus, VTK-m chooses to use a signed integer for indexing.

VTK-m also has types to declare an integer of a specific width and sign. The types `vtkm::Int8`, `vtkm::Int16`, `vtkm::Int32`, and `vtkm::Int64` specify signed integers of 1, 2, 4, and 8 bytes, respectively. Likewise, the types `vtkm::UInt8`, `vtkm::UInt16`, `vtkm::UInt32`, and `vtkm::UInt64` specify unsigned integers of 1, 2, 4, and 8 bytes, respectively.

4.3 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides a collection of base types to represent these short vectors, which are collectively referred to as `Vec` types.

`vtkm::Vec2f`, `vtkm::Vec3f`, and `vtkm::Vec4f` specify floating point `Vec`s of 2, 3, and 4 components, respectively. The precision of the floating point numbers follows that of `vtkm::FloatDefault` (which, from what is said in Section 4.1, is specified by the VTKm_USE_DOUBLE_PRECISION compile option). Components of these and other `Vec` types can be references through the `[ ]` operator, much like a C array. `Vec`s also support basic arithmetic operators so that they can be used much like their scalar-value counterparts.

Example 4.1: Simple use of `Vec` objects.

```
1  vtkm::Vec2f A(1);          // A is (1, 1)
2  A[1] = 3;                  // A is (1, 3) now
3  vtkm::Vec2f B = { 4, 5 }; // B is (4, 5)
4  vtkm::Vec2f C = A + B;     // C is (5, 8)
5  vtkm::FloatDefault manhattanDistance = C[0] + C[1];
```

You can also specify the precision for each of these vector types by appending the bit size of each component. For example, `vtkm::Vec3f_32` and `vtkm::Vec3f_64` represent 3-component floating point vectors with each component being 32 bits and 64 bits respectively. Note that the precision number refers to the precision of each component, not the vector as a whole. So `vtkm::Vec3f_32` contains 3 32-bit (4-byte) floating point components, which means the entire `vtkm::Vec3f_32` requires 96 bits (12 bytes).

To help with indexing 2-, 3-, and 4- dimensional arrays, VTK-m provides the types vtkm::Id2, vtkm::Id3, and vtkm::Id4, which are Vecs of type vtkm::Id. Likewise, VTK-m provides vtkm::IdComponent2, vtkm::IdComponent3, and vtkm::IdComponent4.

VTK-m also provides types for Vecs of integers of all varieties described in Section 4.2. vtkm::Vec2i, vtkm::Vec3i, and vtkm::Vec4i are vectors of signed integers whereas vtkm::Vec2ui, vtkm::Vec3ui, and vtkm::Vec4ui are vectors of unsigned integers. All of these sport components of a width equal to vtkm::Id. The width can be specified by appending the desired number of bits in the same way as the floating point Vecs. For example, vtkm::Vec4ui_8 is a Vec of 4 unsigned bytes.

These types really just scratch the surface of the Vec types available in VTK-m and the things that can be done with them. See Chapter 19 for more information on Vec types and what can be done with them.

CHAPTER

FIVE

VTK-M VERSION

As the VTK-m code evolves, changes to the interface and behavior will inevitably happen. Consequently, code that links into VTK-m might need a specific version of VTK-m or changes its behavior based on what version of VTK-m it is using. To facilitate this, VTK-m software is managed with a versioning system and advertises its version in multiple ways. As with many software products, VTK-m has three version numbers: major, minor, and patch. The major version represents significant changes in the VTK-m implementation and interface. Changes in the major version include backward incompatible changes. The minor version represents added functionality. Generally, changes in the minor version to not introduce changes to the API (although the early 1.X versions of VTK-m violate this). The patch version represents fixes provided after a release occurs. Patch versions represent minimal change and do not add features.

If you are writing a software package that is managed by CMake and load VTK-m with the `find_package` command as described in Section 2.4, then you can query the VTK-m version directly in the CMake configuration. When you load VTK-m with `find_package`, CMake sets the variables VTKm_VERSION_MAJOR, VTKm_VERSION_MINOR, and VTKm_VERSION_PATCH to the major, minor, and patch versions, respectively. Additionally, VTKm_VERSION is set to the "major.minor" version number and VTKm_VERSION_FULL is set to the "major.minor.patch" version number. If the current version of VTK-m is actually a development version that is in between releases of VTK-m, then and abbreviated SHA of the git commit is also included as part of VTKm_VERSION_FULL.

Did you know?

If you have a specific version of VTK-m required for your software, you can also use the version option to the `find_package` *CMake command. The* `find_package` *command takes an optional version argument that causes the command to fail if the wrong version of the package is found.*

It is also possible to query the VTK-m version directly in your code through preprocessor macros. The vtkm/Version.h header file defines the following preprocessor macros to identify the VTK-m version. `VTKM_VERSION_MAJOR`, `VTKM_VERSION_MINOR`, and `VTKM_VERSION_PATCH` are set to integer numbers representing the major, minor, and patch versions, respectively. Additionally, `VTKM_VERSION` is set to the "major.minor" version number as a string and `VTKM_VERSION_FULL` is set to the "major.minor.patch" version number (also as a string). If the current version of VTK-m is actually a development version that is in between releases of VTK-m, then and abbreviated SHA of the git commit is also included as part of `VTKM_VERSION_FULL`.

Common Errors

Note that the CMake variables all begin with `VTKm_` *(lowercase "m") whereas the preprocessor macros begin with* `VTKM_` *(all uppercase). This follows the respective conventions of CMake variables and preprocessor macros.*

Note that vtkm/Version.h does not include any other VTK-m header files. This gives your code a chance to load, query, and react to the VTK-m version before loading any VTK-m code proper.

CHAPTER

SIX

INITIALIZATION

When it comes to running VTK-m code, there are a few ways in which various facilities, such as logging device connections, and device configuration parameters, can be initialized. The preferred method of initializing these features is to run the `vtkm::cont::Initialize` function. Although it is not strictly necessary to call `Initialize`, it is recommended to set up state and check for available devices.

`Initialize` can be called without any arguments, in which case VTK-m will be initialized with defaults. But it can also optionally take the `argc` and `argv` arguments to the `main` function to parse some options that control the state of VTK-m. VTK-m accepts arguments that, for example, configure the compute device to use or establish logging levels. Any arguments that are handled by VTK-m are removed from the `argc`/`argv` list so that your program can then respond to the remaining arguments.

`Initialize` takes an optional third argument that specifies some options on the behavior of the argument parsing. The options are specified as a bit-wise "or" of fields specified in the `vtkm::cont::InitializeOptions` enum. The available initialize options are

`None` Placeholder used when no options are enabled. This is the value used when the third argument to `Initialize` is not provided.

`RequireDevice` Issue an error if the device argument is not specified.

`DefaultAnyDevice` If no device is specified, treat it as if the user gave "--device=Any". This means that `DeviceAdapterTagUndefined` will never be return in the result.

`AddHelp` Add a help option. If "-h" or "--help" is provided, prints a usage statement. Of course, the usage statement will only print out arguments processed by VTK-m, which is why help is not given by default. A string with usage help is returned from `Initialize` so that the calling program can provide VTK-m's help in its own usage statement.

`ErrorOnBadOption` If an unknown option is encountered, the program terminates with an error. If this option is not provided, any unknown options are returned in `argv`. If this option is used, it is a good idea to use `AddHelp` as well.

`ErrorOnBadArgument` If an extra argument is encountered, the program terminates with an error. If this option is not provided, any unknown arguments are returned in `argv`.

`Strict` If supplied, Initialize treats its own arguments as the only ones supported by the application and provides an error if not followed exactly. This is a convenience option that is a combination of `ErrorOnBadOption`, `ErrorOnBadArgument`, and `AddHelp`.

As stated earlier, `vtkm::cont::Initialize` removes parsed options from the `argc`/`argv` passed to it so that the calling program can further respond to command line arguments. Additionally, `Initialize` returns an `vtkm::cont::InitializeResult` object that contains the following information.

`Device` A `vtkm::cont::DeviceAdapterId` that represents the device specified by the command line arguments. (See Chapter 12 for details on how VTK-m represents devices.) If no device is specified in the command line options, `vtkm::cont::DeviceAdapterTagUndefined` is returned (unless the `DefaultAnyDevice` option is given, in which case `vtkm::cont::DeviceAdapterTagAny` is returned).

Example 6.1: Calling `Initialize`.

```
#include <vtkm/cont/Initialize.h>

int main(int argc, char** argv)
{
  vtkm::cont::InitializeOptions options =
    vtkm::cont::InitializeOptions::ErrorOnBadArgument |
    vtkm::cont::InitializeOptions::DefaultAnyDevice;
  vtkm::cont::InitializeResult config = vtkm::cont::Initialize(argc, argv, options);

  if (argc != 2)
  {
    std::cerr << "USAGE: " << argv[0] << " [options] filename" << std::endl;
    std::cerr << "Available options are:" << std::endl;
    std::cerr << config.Usage << std::endl;
    return 1;
  }
  std::string filename = argv[1];

  // Do something cool with VTK-m
  // ...

  return 0;
}
```

CHAPTER

SEVEN

DATA SETS

A *data set*, implemented with the `vtkm::cont::DataSet` class, contains and manages the geometric data structures that VTK-m operates on. A data set comprises the following 3 data structures.

Cell Set A cell set describes topological connections. A cell set defines some number of points in space and how they connect to form cells, filled regions of space. A data set has exactly one cell set.

Field A field describes numerical data associated with the topological elements in a cell set. The field is represented as an array, and each entry in the field array corresponds to a topological element (point, edge, face, or cell). Together the cell set topology and discrete data values in the field provide an interpolated function throughout the volume of space covered by the data set. A cell set can have any number of fields.

Coordinate System A coordinate system is a special field that describes the physical location of the points in a data set. Although it is most common for a data set to contain a single coordinate system, VTK-m supports data sets with no coordinate system such as abstract data structures like graphs that might not have positions in a space. `DataSet` also supports multiple coordinate systems for data that have multiple representations for position. For example, geospatial data could simultaneously have coordinate systems defined by 3D position, latitude-longitude, and any number of 2D projections.

In addition to the base `vtkm::cont::DataSet`, VTK-m provides `vtkm::cont::PartitionedDataSet` to represent data partitioned into multiple domains. A `PartitionedDataSet` is implemented as a collection of `DataSet` objects. Partitioned data sets are described later in Section 7.5.

7.1 Building Data Sets

Before we go into detail on the cell sets, fields, and coordinate systems that make up a data set in VTK-m, let us first discuss how to build a data set. One simple way to build a data set is to load data from a file using the `vtkm::io` module. Reading files is discussed in detail in Chapter 8.

This section describes building data sets of different types using a set of classes named `DataSetBuilder*`, which provide a convenience layer on top of `vtkm::cont::DataSet` to make it easier to create data sets.

Did you know?

To simplify the introduction of `DataSet`s, this section uses the simplest mechanisms. In many cases this involves loading data in a `std::vector` and passing that to VTK-m, which usually causes the data to be copied. This is not the most efficient method to load data into VTK-m. Although it is sufficient for small

data or data that come from a "slow" source, such as a file, it might be a bottleneck for large data generated by another library. It is possible to adapt VTK-m's `DataSet` *to externally defined data. This is done by wrapping existing data into what is called* `ArrayHandle`*, but this is a more advanced topic that will not be addressed in this chapter.* `ArrayHandle`*s are introduced in Chapter 16 and more adaptive techniques are described in later chapters.*

7.1.1 Creating Uniform Grids

Uniform grids are meshes that have a regular array structure with points uniformly spaced parallel to the axes. Uniform grids are also sometimes called regular grids or images.

The `vtkm::cont::DataSetBuilderUniform` class can be used to easily create 2- or 3-dimensional uniform grids. `DataSetBuilderUniform` has several versions of a method named `Create` that takes the number of points in each dimension, the origin, and the spacing. The origin is the location of the first point of the data (in the lower left corner), and the spacing is the distance between points in the x, y, and z directions. The `Create` methods also take an optional name for the coordinate system and an optional name for the cell set.

The following example creates a `vtkm::cont::DataSet` containing a uniform grid of $101 \times 101 \times 26$ points.

Example 7.1: Creating a uniform grid.

```
1  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3  vtkm::cont::DataSet dataSet = dataSetBuilder.Create(vtkm::Id3(101, 101, 26));
```

If not specified, the origin will be at the coordinates $(0,0,0)$ and the spacing will be 1 in each direction. Thus, in the previous example the width, height, and depth of the mesh in physical space will be 100, 100, and 25, respectively, and the mesh will be centered at $(50,50,12.5)$. Let us say we actually want a mesh of the same dimensions, but we want the z direction to be stretched out so that the mesh will be the same size in each direction, and we want the mesh centered at the origin.

Example 7.2: Creating a uniform grid with custom origin and spacing.

```
1  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3  vtkm::cont::DataSet dataSet =
4    dataSetBuilder.Create(vtkm::Id3(101, 101, 26),
5                          vtkm::Vec3f(-50.0, -50.0, -50.0),
6                          vtkm::Vec3f(1.0, 1.0, 4.0));
```

7.1.2 Creating Rectilinear Grids

A rectilinear grid is similar to a uniform grid except that a rectilinear grid can adjust the spacing between adjacent grid points. This allows the rectilinear grid to have tighter sampling in some areas of space, but the points are still constrained to be aligned with the axes and each other. The irregular spacing of a rectilinear grid is specified by providing a separate array each for the x, y, and z coordinates.

The `vtkm::cont::DataSetBuilderRectilinear` class can be used to easily create 2- or 3-dimensional rectilinear grids. `DataSetBuilderRectilinear` has several versions of a method named `Create` that takes these coordinate arrays and builds a `vtkm::cont::DataSet` out of them. The arrays can be supplied as either standard C arrays or as `std::vector` objects, in which case the data in the arrays are copied into the `DataSet`. These arrays can also be passed as `ArrayHandle` objects (introduced later in this book), in which case the data are shallow copied.

The following example creates a vtkm::cont::DataSet containing a rectilinear grid with $201 \times 201 \times 101$ points with different irregular spacing along each axis.

Example 7.3: Creating a rectilinear grid.

```
// Make x coordinates range from -4 to 4 with tighter spacing near 0.
std::vector<vtkm::Float32> xCoordinates;
for (vtkm::Float32 x = -2.0f; x <= 2.0f; x += 0.02f)
{
  xCoordinates.push_back(vtkm::CopySign(x * x, x));
}

// Make y coordinates range from 0 to 2 with tighter spacing near 2.
std::vector<vtkm::Float32> yCoordinates;
for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
{
  yCoordinates.push_back(vtkm::Sqrt(y));
}

// Make z coordinates rangefrom -1 to 1 with even spacing.
std::vector<vtkm::Float32> zCoordinates;
for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
{
  zCoordinates.push_back(z);
}

vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;

vtkm::cont::DataSet dataSet =
  dataSetBuilder.Create(xCoordinates, yCoordinates, zCoordinates);
```

7.1.3 Creating Explicit Meshes

An explicit mesh is an arbitrary collection of cells with arbitrary connections. It can have multiple different types of cells. Explicit meshes are also known as unstructured grids. Explicit meshes can contain cells of different shapes. The shapes that VTK-m currently supports are listed in Figure 7.1.

The cells of an explicit mesh are defined with the following 3 arrays, which are depicted graphically in Figure 7.2.

Shapes An array of ids identifying the shape of the cell. Each value is a vtkm::UInt8 and should be set to one of the vtkm ::CELL_SHAPE_* constants. The shapes and their identifiers are shown in Figure 7.1. The size of this array is equal to the number of cells in the set.

Connectivity An array that lists all the points that comprise each cell. Each entry in the array is a vtkm::Id giving the point id associated with a vertex of a cell. The points for each cell are given in a prescribed order for each shape, which is also shown in Figure 7.1. The point indices are stored consecutively from the first cell to the last.

Offsets An array of vtkm::Id s pointing to the index in the connectivity array where the points for a particular cell starts. The size of this array is equal to the number of cells in the set plus 1. The first entry is expected to be 0 (since the connectivity of the first cell is at the start of the connectivity array). The last entry, which does not correspond to any cell, should be the size of the connectivity array.

One important item that is missing from this list of arrays is a count of the number of indices associated with each cell. This is not explicitly represented in VTK-m's mesh structure because it can be implicitly derived from the offsets array by subtracting consecutive entries. However, it is usually the case when building an explicit

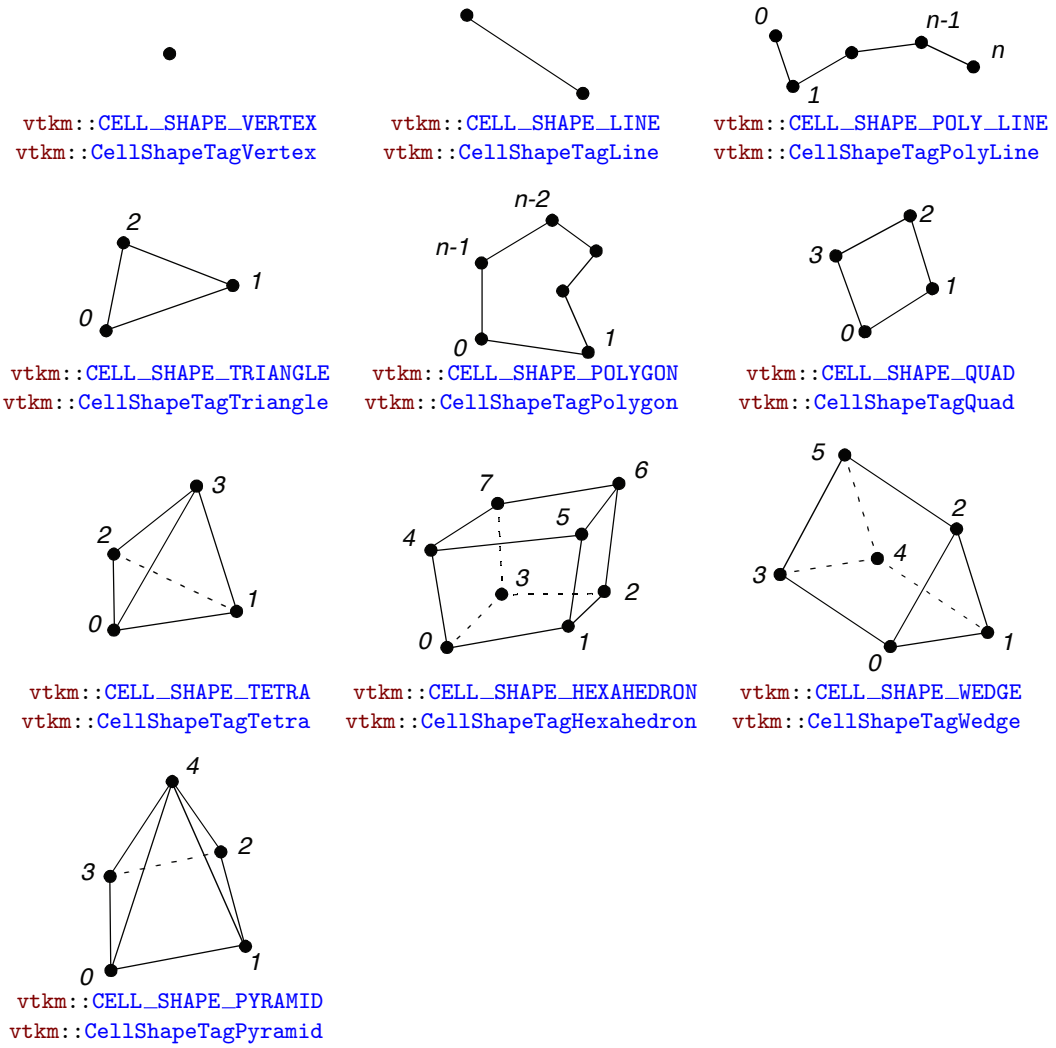


Figure 7.1: Basic Cell Shapes

mesh that you will have an array of these counts rather than the offsets. It is for this reason that VTK-m contains mechanisms to build an explicit data set with a "num indices" arrays rather than an offsets array.

The `vtkm::cont::DataSetBuilderExplicit` class can be used to create data sets with explicit meshes. `DataSetBuilderExplicit` has several versions of a method named `Create`. Generally, these methods take the shapes, number of indices, and connectivity arrays as well as an array of point coordinates. These arrays can be given in `std::vector` objects, and the data are copied into the `DataSet` created.

The following example creates a mesh like the one shown in Figure 7.2.

Example 7.4: Creating an explicit mesh with `DataSetBuilderExplicit`.

```
1 // Array of point coordinates.
2 std::vector<vtkm::Vec3f_32> pointCoordinates;
3 pointCoordinates.push_back(vtkm::Vec3f_32(1.1f, 0.0f, 0.0f));
4 pointCoordinates.push_back(vtkm::Vec3f_32(0.2f, 0.4f, 0.0f));
5 pointCoordinates.push_back(vtkm::Vec3f_32(0.9f, 0.6f, 0.0f));
6 pointCoordinates.push_back(vtkm::Vec3f_32(1.4f, 0.5f, 0.0f));
```

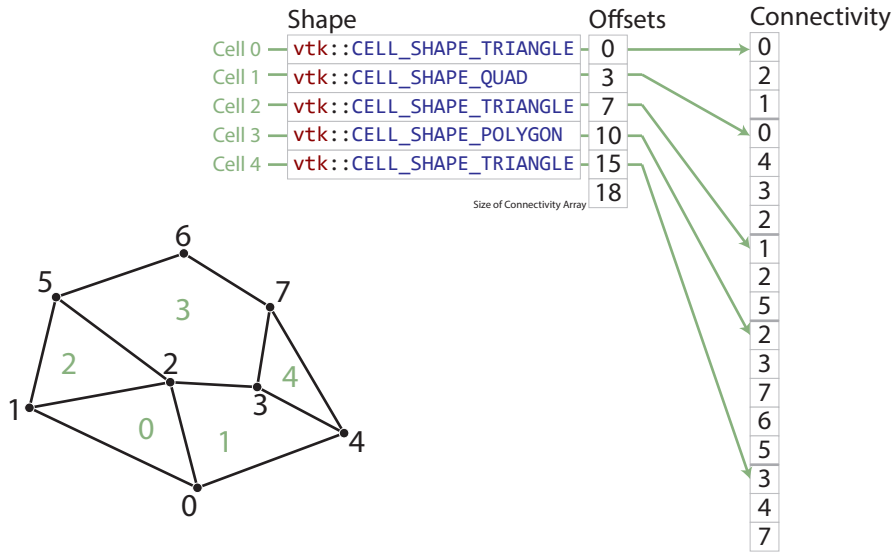


Figure 7.2: An example explicit mesh.

```
7   pointCoordinates.push_back(vtkm::Vec3f_32(1.8f, 0.3f, 0.0f));
8   pointCoordinates.push_back(vtkm::Vec3f_32(0.4f, 1.0f, 0.0f));
9   pointCoordinates.push_back(vtkm::Vec3f_32(1.0f, 1.2f, 0.0f));
10  pointCoordinates.push_back(vtkm::Vec3f_32(1.5f, 0.9f, 0.0f));
11
12  // Array of shapes.
13  std::vector<vtkm::UInt8> shapes;
14  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
15  shapes.push_back(vtkm::CELL_SHAPE_QUAD);
16  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
17  shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
18  shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
19
20  // Array of number of indices per cell.
21  std::vector<vtkm::IdComponent> numIndices;
22  numIndices.push_back(3);
23  numIndices.push_back(4);
24  numIndices.push_back(3);
25  numIndices.push_back(5);
26  numIndices.push_back(3);
27
28  // Connectivity array.
29  std::vector<vtkm::Id> connectivity;
30  connectivity.push_back(0); // Cell 0
31  connectivity.push_back(2);
32  connectivity.push_back(1);
33  connectivity.push_back(0); // Cell 1
34  connectivity.push_back(4);
35  connectivity.push_back(3);
36  connectivity.push_back(2);
37  connectivity.push_back(1); // Cell 2
38  connectivity.push_back(2);
39  connectivity.push_back(5);
40  connectivity.push_back(2); // Cell 3
41  connectivity.push_back(3);
42  connectivity.push_back(7);
43  connectivity.push_back(6);
```

```
44  connectivity.push_back(5);
45  connectivity.push_back(3); // Cell 4
46  connectivity.push_back(4);
47  connectivity.push_back(7);
48
49  // Copy these arrays into a DataSet.
50  vtkm::cont::DataSetBuilderExplicit dataSetBuilder;
51
52  vtkm::cont::DataSet dataSet =
53    dataSetBuilder.Create(pointCoordinates, shapes, numIndices, connectivity);
```

Often it is awkward to build your own arrays and then pass them to `DataSetBuilderExplicit`. There also exists an alternate builder class named `vtkm::cont::DataSetBuilderExplicitIterative` that allows you to specify each cell and point one at a time rather than all at once. This is done by calling one of the versions of `AddPoint` and one of the versions of `AddCell` for each point and cell, respectively. The next example also builds the mesh shown in Figure 7.2 except this time using `DataSetBuilderExplicitIterative`.

Example 7.5: Creating an explicit mesh with `DataSetBuilderExplicitIterative`.

```
1   vtkm::cont::DataSetBuilderExplicitIterative dataSetBuilder;
2
3   dataSetBuilder.AddPoint(1.1, 0.0, 0.0);
4   dataSetBuilder.AddPoint(0.2, 0.4, 0.0);
5   dataSetBuilder.AddPoint(0.9, 0.6, 0.0);
6   dataSetBuilder.AddPoint(1.4, 0.5, 0.0);
7   dataSetBuilder.AddPoint(1.8, 0.3, 0.0);
8   dataSetBuilder.AddPoint(0.4, 1.0, 0.0);
9   dataSetBuilder.AddPoint(1.0, 1.2, 0.0);
10  dataSetBuilder.AddPoint(1.5, 0.9, 0.0);
11
12  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
13  dataSetBuilder.AddCellPoint(0);
14  dataSetBuilder.AddCellPoint(2);
15  dataSetBuilder.AddCellPoint(1);
16
17  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_QUAD);
18  dataSetBuilder.AddCellPoint(0);
19  dataSetBuilder.AddCellPoint(4);
20  dataSetBuilder.AddCellPoint(3);
21  dataSetBuilder.AddCellPoint(2);
22
23  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
24  dataSetBuilder.AddCellPoint(1);
25  dataSetBuilder.AddCellPoint(2);
26  dataSetBuilder.AddCellPoint(5);
27
28  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_POLYGON);
29  dataSetBuilder.AddCellPoint(2);
30  dataSetBuilder.AddCellPoint(3);
31  dataSetBuilder.AddCellPoint(7);
32  dataSetBuilder.AddCellPoint(6);
33  dataSetBuilder.AddCellPoint(5);
34
35  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
36  dataSetBuilder.AddCellPoint(3);
37  dataSetBuilder.AddCellPoint(4);
38  dataSetBuilder.AddCellPoint(7);
39
40  vtkm::cont::DataSet dataSet = dataSetBuilder.Create();
```

7.1.4 Add Fields

In addition to creating the geometric structure of a data set, it is usually important to add fields to the data. Fields describe numerical data associated with the topological elements in a cell. They often represent a physical quantity (such as temperature, mass, or volume fraction) but can also represent other information (such as indices or classifications).

The easiest way to define fields in a data set is to use the `DataSet::AddPointField` and `DataSet::AddCellField` methods. Each of these methods take a requisite field name and the array with with field data.

Both `AddPointField` and `AddCellField` are overloaded to accept arrays of data in different structures. Field arrays can be passed as standard C arrays or as `std::vectors`, in which case the data are copied. Field arrays can also be passed in a `ArrayHandle` (introduced later in this book), in which case the data are not copied.

The following (somewhat contrived) example defines fields for a uniform grid that identify which points and cells are on the boundary of the mesh.

Example 7.6: Adding fields to a `DataSet`.

```
// Make a simple structured data set.
const vtkm::Id3 pointDimensions(20, 20, 10);
const vtkm::Id3 cellDimensions = pointDimensions - vtkm::Id3(1, 1, 1);
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointDimensions);

// Create a field that identifies points on the boundary.
std::vector<vtkm::UInt8> boundaryPoints;
for (vtkm::Id zIndex = 0; zIndex < pointDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < pointDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < pointDimensions[0]; xIndex++)
    {
      if ((xIndex == 0) || (xIndex == pointDimensions[0] - 1) || (yIndex == 0) ||
          (yIndex == pointDimensions[1] - 1) || (zIndex == 0) ||
          (zIndex == pointDimensions[2] - 1))
      {
        boundaryPoints.push_back(1);
      }
      else
      {
        boundaryPoints.push_back(0);
      }
    }
  }
}

dataSet.AddPointField("boundary_points", boundaryPoints);

// Create a field that identifies cells on the boundary.
std::vector<vtkm::UInt8> boundaryCells;
for (vtkm::Id zIndex = 0; zIndex < cellDimensions[2]; zIndex++)
{
  for (vtkm::Id yIndex = 0; yIndex < cellDimensions[1]; yIndex++)
  {
    for (vtkm::Id xIndex = 0; xIndex < cellDimensions[0]; xIndex++)
    {
      if ((xIndex == 0) || (xIndex == cellDimensions[0] - 1) || (yIndex == 0) ||
          (yIndex == cellDimensions[1] - 1) || (zIndex == 0) ||
          (zIndex == cellDimensions[2] - 1))
      {
        boundaryCells.push_back(1);
      }
```

```
45          else
46          {
47            boundaryCells.push_back(0);
48          }
49        }
50      }
51    }
52
53    dataSet.AddCellField("boundary_cells", boundaryCells);
```

7.2 Cell Sets

A cell set determines the topological structure of the data in a data set. Fundamentally, any cell set is a collection of cells, which typically (but not always) represent some region in space. 3D cells are made up of points, edges, and faces. (2D cells have only points and edges, and 1D cells have only points.) Figure 7.3 shows the relationship between a cell's shape and these topological elements. The arrangement of these points, edges, and faces is defined by the *shape* of the cell, which prescribes a specific ordering of each. The basic cell shapes provided by VTK-m are discussed in detail in Section 25.1 starting on page 209.

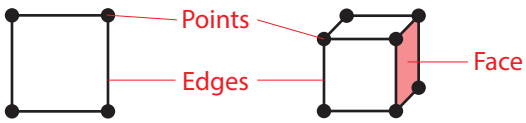


Figure 7.3: The relationship between a cell shape and its topological elements (points, edges, and faces).

There are multiple ways to express the connections of a cell set, each with different benefits and restrictions. These different cell set types are managed by different cell set classes in VTK-m. All VTK-m cell set classes inherit from `vtkm::cont::CellSet`. The two basic types of cell sets are structured and explicit, and there are several variations of these types.

7.2.1 Structured Cell Sets

A `vtkm::cont::CellSetStructured` defines a 1-, 2-, or 3-dimensional grid of points with lines, quadrilaterals, or hexahedra, respectively, connecting them. The topology of a `CellSetStructured` is specified by simply providing the dimensions, which is the number of points in the i, j, and k directions of the grid of points. The number of points is implicitly $i \times j \times k$ and the number of cells is implicitly $(i-1) \times (j-1) \times (k-1)$ (for 3D grids). Figure 7.4 demonstrates this arrangement.

The big advantage of using `vtkm::cont::CellSetStructured` to define a cell set is that it is very space efficient because the entire topology can be defined by the three integers specifying the dimensions. Also algorithms can be optimized for `CellSetStructured`'s regular nature. However, `CellSetStructured`'s strictly regular grid structure also limits its applicability. A structured cell set can only be a dense grid of lines, quadrilaterals, or hexahedra. It cannot represent irregular data well.

Many data models in other software packages, such as the one for VTK, make a distinction between uniform, rectilinear, and curvilinear grids. VTK-m's cell sets do not. All three of these grid types are represented by `CellSetStructured`. This is because in a VTK-m data set the cell set and the coordinate system are defined independently and used interchangeably. A structured cell set with uniform point coordinates makes a uniform grid. A structured cell set with point coordinates defined irregularly along coordinate axes makes a rectilinear grid. And a structured cell set with arbitrary point coordinates makes a curvilinear grid. The point coordinates are defined by the data set's coordinate system, which is discussed in Section 7.4 starting on page 41.

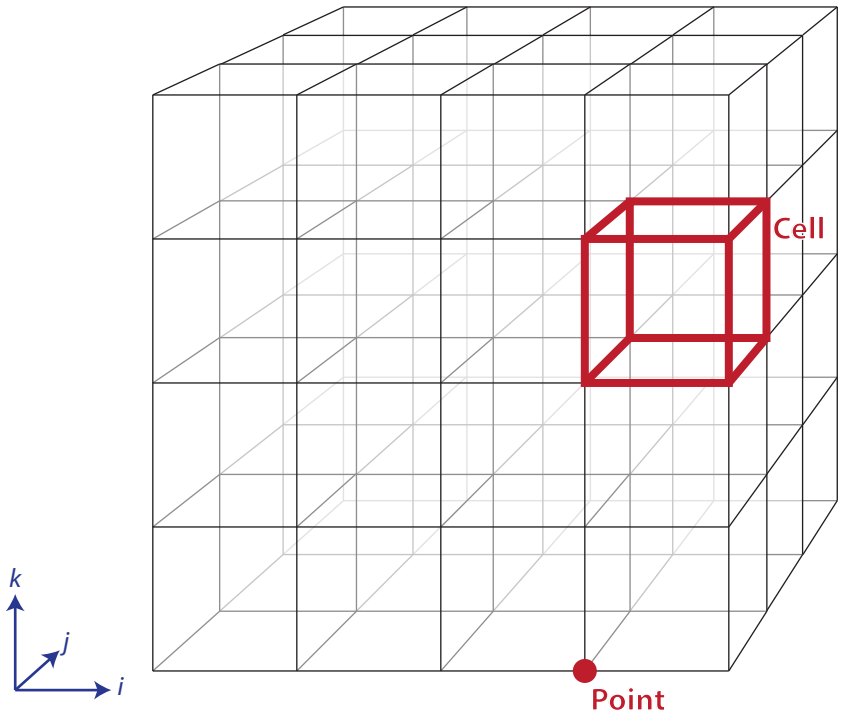


Figure 7.4: The arrangement of points and cells in a 3D structured grid.

7.2.2 Explicit Cell Sets

A `vtkm::cont::CellSetExplicit` defines an irregular collection of cells. The cells can be of different types and connected in arbitrary ways. The types of cell sets are listed in Figure 7.5. This is done by explicitly providing for each cell a sequence of points that defines the cell.

An explicit cell set is defined with a minimum of three arrays. The first array identifies the shape of each cell. (Identifiers for cell shapes are shown in Figure 7.5.) The second array has a sequence of point indices that make up each cell. The third array identifies an offset into the second array where the point indices for each cell is found plus an extra entry at the end set to the size of the second array. Figure 7.6 shows a simple example of an explicit cell set.

An explicit cell set can also identify the number of indices defined for each cell by subtracting consecutive entries in the offsets array. It is often the case when creating a `CellSetExplicit` that you have an array containing the number of indices rather than the offsets. Such an array can be converted to an offsets array that can be used with `CellSetExplicit` by using the `vtkm::cont::ConvertNumComponentsToOffsets` convenience function.

`vtkm::cont::CellSetExplicit` is a powerful representation for a cell set because it can represent an arbitrary collection of cells. However, because all connections must be explicitly defined, `CellSetExplicit` requires a significant amount of memory to represent the topology.

An important specialization of an explicit cell set is `vtkm::cont::CellSetSingleType`. `CellSetSingleType` is an explicit cell set constrained to contain cells that all have the same shape and all have the same number of points. So for example if you are creating a surface that you know will contain only triangles, `CellSetSingleType` is a good representation for these data.

Using `CellSetSingleType` saves memory because the array of cell shapes and the array of point counts no longer

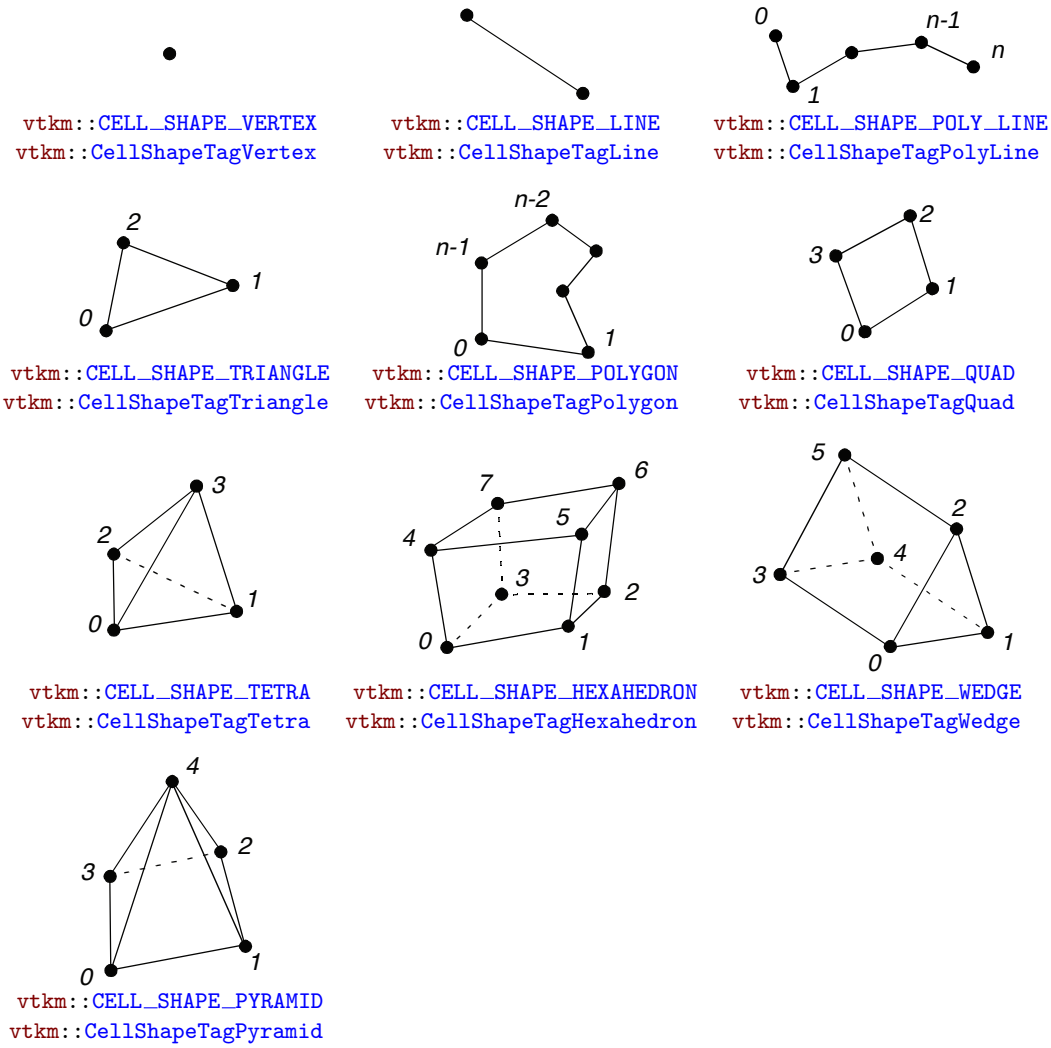


Figure 7.5: Basic Cell Shapes in a `CellSetExplicit`.

need to be stored. `CellSetSingleType` also allows VTK-m to skip some processing and other storage required for general explicit cell sets.

7.2.3 Cell Set Permutations

A `vtkm::cont::CellSetPermutation` rearranges the cells of one cell set to create another cell set. This restructuring of cells is not done by copying data to a new structure. Rather, `CellSetPermutation` establishes a look-up from one cell structure to another. Cells are permuted on the fly while algorithms are run.

A `CellSetPermutation` is established by providing a mapping array that for every cell index provides the equivalent cell index in the cell set being permuted. `CellSetPermutation` is most often used to mask out cells in a data set so that algorithms will skip over those cells when running.

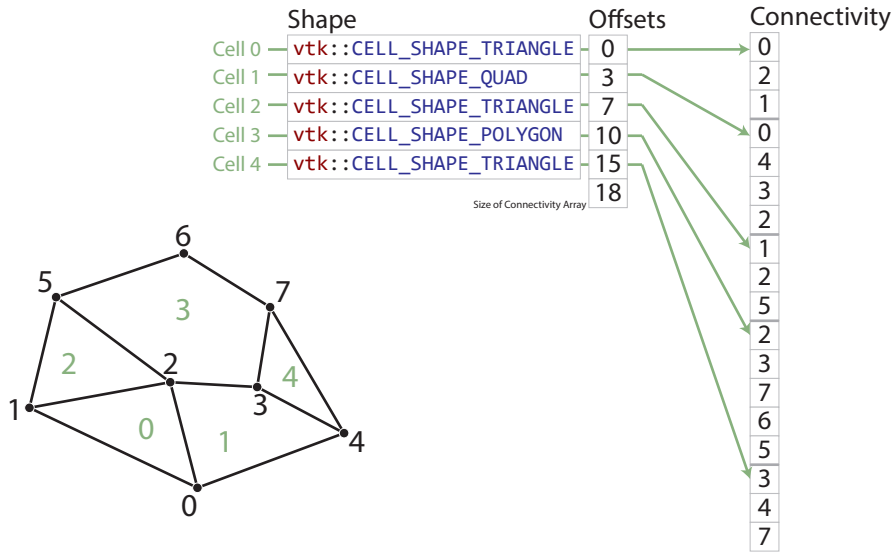


Figure 7.6: Example of cells in a `CellSetExplicit` and the arrays that define them.

Did you know?

Although `CellSetPermutation` *can mask cells, it cannot mask points. All points from the original cell set are available in the permuted cell set regardless of whether they are used.*

The following example uses `vtkm::cont::CellSetPermutation` with a counting array to expose every tenth cell. This provides a simple way to subsample a data set.

Example 7.7: Subsampling a data set with `CellSetPermutation`.

```
// Create a simple data set.
vtkm::cont::DataSetBuilderUniform dataSetBuilder;
vtkm::cont::DataSet originalDataSet = dataSetBuilder.Create(vtkm::Id3(33, 33, 26));
vtkm::cont::CellSetStructured<3> originalCellSet;
originalDataSet.GetCellSet().AsCellSet(originalCellSet);

// Create a permutation array for the cells. Each value in the array refers
// to a cell in the original cell set. This particular array selects every
// 10th cell.
vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);

// Create a permutation of that cell set containing only every 10th cell.
vtkm::cont::CellSetPermutation<vtkm::cont::CellSetStructured<3>,
                               vtkm::cont::ArrayHandleCounting<vtkm::Id>>
  permutedCellSet(permutationArray, originalCellSet);
```

7.2.4 Cell Set Extrude

A `vtkm::cont::CellSetExtrude` defines a 3-dimensional extruded mesh representation from 2-dimensional coordinates in the XZ-plane. This is done by providing 2-dimensional coordinates, the number of planes to extrude along the Y-axis, and whether the resulting wedge cellset representation should be a torus or a cylinder.

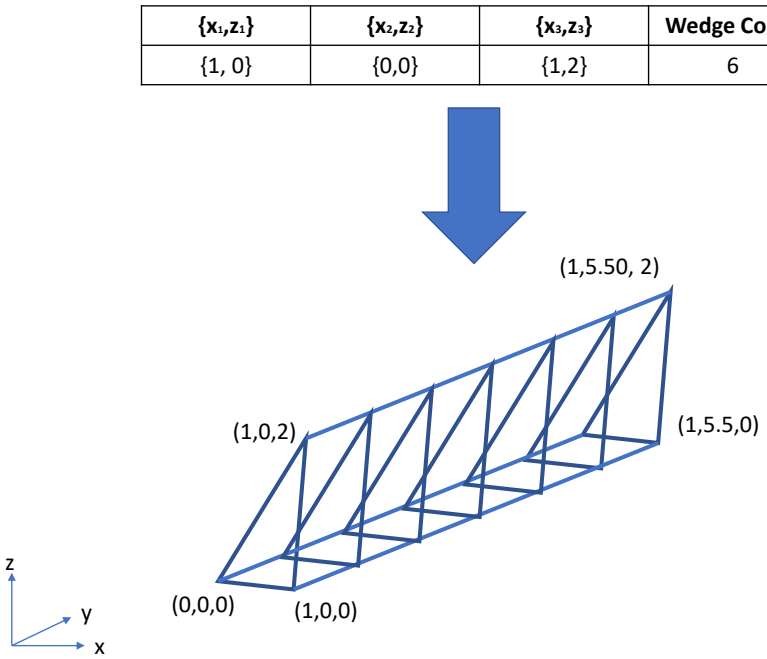


Figure 7.7: An example of an extruded wedge from XZ-plane coordinates. Six wedges are extracted from three XZ-plane points.

The extruded mesh is advantageous because it is represented on-the-fly as required, so no additional memory is required. In contrast other forms of cell sets, such as `vtkm::cont::CellSetExplicit`, need to be explicitly constructed by replicating the vertices and cells. Figure 7.7 shows an example of six wedges extruded from three 2-dimensional coordinates.

7.2.5 Unknown Cell Sets

Each of the aforementioned cell set types are represented by a different class. A `vtkm::cont::DataSet` object must hold one of these cell set objects that represent the cell structure. The actual object used is not determined until run time.

The `DataSet` object manages the cell set object with `vtkm::cont::UnknownCellSet`. When you call `DataSet::GetCellSet`, it returns a `UnknownCellSet`.

The `UnknownCellSet` object provides mechanisms to query the cell set, identify its type, and cast it to one of the concrete `CellSet` types. See Chapter 34 for details on working with `UnknownCellSet`.

7.3 Fields

A field on a data set provides a value on every point in space on the mesh. Fields are often used to describe physical properties such as pressure, temperature, mass, velocity, and much more. Fields are represented in a VTK-m data set as an array where each value is associated with a particular element type of a mesh (such as

points or cells). This association of field values to mesh elements and the structure of the cell set determines how the field is interpolated throughout the space of the mesh.

Fields are manged by the vtkm::cont::Field class. The Field object internally holds a reference to an array in a type-agnostic way. Filters and other VTK-m units will determine the type of the array and pull it out of the Field.

Field has a convenience method named GetRange that finds the range of values stored in the field array. The returned value of GetRange is an ArrayHandle containing vtkm::Range values. The ArrayHandle will have as many values as components in the field. So, for example, calling GetRange on a scalar field will return an ArrayHandle with exactly 1 entry in it. Calling GetRange on a field of 3D vectors will return an ArrayHandle with exactly 3 entries corresponding to each of the components in the range. Details on how to get data from an ArrayHandle them is given in Chapter 27.

7.4 Coordinate Systems

A coordinate system determines the location of a mesh's elements in space. The spatial location is described by providing a 3D vector at each point that gives the coordinates there. The point coordinates can then be interpolated throughout the mesh.

Coordinate systems are managed by the vtkm::cont::CoordinateSystem class. In actuality, a coordinate system is just a field with a special meaning, and so the CoordinateSystem class inherits from the Field class. CoordinateSystem constrains the field to be associated with points and typically has 3D floating point vectors for values.

In addition to all the methods provided by the Field superclass, the CoordinateSystem also provides a GetBounds convenience method that returns a vtkm::Bounds object giving the spatial bounds of the coordinate system.

It is typical for a DataSet to have one coordinate system defined, but it is possible to define multiple coordinate systems. This is helpful when there are multiple ways to express coordinates. For example, positions in geographic may be expressed as Cartesian coordinates or as latitude-longitude coordinates. Both are valid and useful in different ways.

It is also valid to have a DataSet with no coordinate system. This is useful when the structure is not rooted in physical space. For example, if the cell set is representing a graph structure, there might not be any physical space that has meaning for the graph.

7.5 Partitioned Data Sets

A partitioned data set, implemented with vtkm::cont::PartitionedDataSet, comprises a set of vtkm::cont::DataSet objects. The PartitionedDataSet interface allows for adding, replacing, and querying DataSets in its list with the following methods.

GetNumberOfPartitions Returns the number of partitions stored in the PartitionedDataSet.

GetPartition Returns the DataSet at a given index.

GetPartitions Returns all of the DataSets stored in the PartitionedDataSet in a std::vector.

AppendPartition Adds a given DataSet to the end of the list of partitions.

`AppendPartitions` Given a list of `DataSet` objects, appends this list to the end of the list of partitions. This list can be given as a `std::vector` or it can be an initializer list (declared in { } curly braces).

`InsertPartition` Given an index and a `DataSet`, places the `DataSet` at the given index and pushes the remaining partitions after it.

`ReplacePartition` Given an index and a `DataSet`, replaces the partition at that index with the new `DataSet`.

`GetField` Retrieves a `vtkm::cont::Field` object from the `DataSet` at a given index.

The following example creates a `vtkm::cont::PartitionedDataSet` containing two uniform grid data sets.

Example 7.8: Creating a `PartitionedDataSet`.

```
1  // Create two uniform data sets
2  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
3
4  vtkm::cont::DataSet dataSet1 = dataSetBuilder.Create(vtkm::Id3(10, 10, 10));
5  vtkm::cont::DataSet dataSet2 = dataSetBuilder.Create(vtkm::Id3(30, 30, 30));
6
7  // Add the datasets to a multi block
8  vtkm::cont::PartitionedDataSet partitionedData;
9  partitionedData.AppendPartitions({ dataSet1, dataSet2 });
```

It is always possible to retrieve the independent blocks in a `PartitionedDataSet`, from which you can iterate and get information about the data. However, VTK-m provides several helper functions to collect metadata information about the collection as a whole.

`vtkm::cont::BoundsCompute` Queries the bounds of all the `DataSet`s contained in the given `PartitionedDataSet` and returns a `vtkm::Bounds` object encompassing the conglomerate data.

`vtkm::cont::BoundsGlobalCompute` An MPI version of `BoundsCompute` that also finds the bounds around the conglomerate data across all processes. All MPI processes must call this method.

`vtkm::cont::FieldRangeCompute` Given a `PartitionedDataSet`, the name of a field, and (optionally) an association of the field, returns the minimum and maximum value of that field over all the contained blocks. The result is returned in a `ArrayHandle` of `vtkm::Range` objects in the same manner as the `vtkm::cont::Field::GetRange` method (see Section 7.3).

`vtkm::cont::FieldRangeGlobalCompute` An MPI version of `FieldRangeCompute` that also finds the field ranges over all blocks on all processes. All MPI processes must call this method.

The following example illustrates a spatial bounds query and a field range query on a `vtkm::cont::PartitionedDataSet`.

Example 7.9: Queries on a `PartitionedDataSet`.

```
1  // Get the bounds of a multi-block data set
2  vtkm::Bounds bounds = vtkm::cont::BoundsCompute(partitionedData);
3
4  // Get the overall min/max of a field named "cellvar"
5  vtkm::cont::ArrayHandle<vtkm::Range> cellvarRanges =
6    vtkm::cont::FieldRangeCompute(partitionedData, "cellvar");
7
8  // Assuming the "cellvar" field has scalar values, then cellvarRanges has one entry
9  vtkm::Range cellvarRange = cellvarRanges.ReadPortal().Get(0);
```

Did you know?

The aforementioned functions for querying a `PartitionedDataSet` *object also work on* `DataSet` *objects. This is particularly useful with the* `BoundsGlobalCompute` *and* `FieldRangeGlobalCompute` *to manage distributed parallel objects.*

Filters can be executed on `PartitionedDataSet` objects in a similar way they are executed on `DataSet` objects. In both cases, the `Execute` method is called on the filter giving data object as an argument.

Example 7.10: Applying a filter to multi block data.

```
1  vtkm::filter::field_conversion::CellAverage cellAverage;
2  cellAverage.SetActiveField("pointvar", vtkm::cont::Field::Association::Points);
3
4  vtkm::cont::PartitionedDataSet results = cellAverage.Execute(partitionedData);
```

CHAPTER

EIGHT

FILE I/O

Before VTK-m can be used to process data, data need to be loaded into the system. VTK-m comes with a basic file I/O package to get started developing very quickly. All the file I/O classes are declared under the `vtkm::io` namespace.

Did you know?

Files are just one of many ways to get data in and out of VTK-m. In later chapters we explore ways to define VTK-m data structures of increasing power and complexity. In particular, Section 7.1 describes how to build VTK-m data set objects and Section 36.5 documents how to adapt data structures defined in other libraries to be used directly in VTK-m.

8.1 Readers

All reader classes provided by VTK-m are located in the `vtkm::io` namespace. The general interface for each reader class is to accept a filename in the constructor and to provide a `ReadDataSet` method to load the data from disk.

The data in the file are returned in a `vtkm::cont::DataSet` object as described in Chapter 7, but it is sufficient to known that a `DataSet` can be passed around readers, writers, filters, and rendering units.

8.1.1 Legacy VTK File Reader

Legacy VTK files are a simple open format for storing visualization data. These files typically have a `.vtk` extension. Legacy VTK files are popular because they are simple to create and read and are consequently supported by a large number of tools. The format of legacy VTK files is well documented in *The VTK User's Guide*.[1] Legacy VTK files can also be read and written with tools like ParaView and VisIt.

Legacy VTK files can be read using the `vtkm::io::VTKDataSetReader` class. The constructor for this class takes a string containing the filename. The `ReadDataSet` method reads the data from the previously indicated file and returns a `vtkm::cont::DataSet` object, which can be used with filters and rendering.

Example 8.1: Reading a legacy VTK file.

[1] A free excerpt describing the file format is available at http://www.vtk.org/Wiki/File:VTK-File-Formats.pdf.

```
#include <vtkm/io/VTKDataSetReader.h>

vtkm::cont::DataSet OpenDataFromVTKFile()
{
  vtkm::io::VTKDataSetReader reader("data.vtk");

  return reader.ReadDataSet();
}
```

8.1.2 Image Readers

VTK-m provides classes to read images from some standard image formats. These readers will store the data in a `vtkm::cont::DataSet` object with the colors stored as a named point field. The colors are read as 4-component RGBA vectors for each pixel. Each component in the pixel color is stored as a 32-bit float between 0 and 1.

Portable Network Graphics (PNG) files can be read using the `vtkm::io::ImageReaderPNG` class. A `ImageReaderPNG` object is constructed with the name of the file to load. The data from the file are loaded using the `ReadDataSet` method, which returns a `DataSet` object. By default, the colors are stored in a field named "colors", but the name of the field can optionally be changed using the `SetPointFieldName` method.

Example 8.2: Reading an image from a PNG file.

```
#include <vtkm/io/ImageReaderPNG.h>

vtkm::cont::DataSet OpenDataFromPNG()
{
  vtkm::io::ImageReaderPNG imageReader("data.png");
  imageReader.SetPointFieldName("pixel_colors");
  return imageReader.ReadDataSet();
}
```

Portable anymap (PNM) files can be read using the `vtkm::io::ImageReaderPNM` class. Currently, the PNM file reader only supports files using the portable pixmap (PPM) format (with magic number "P6"). These files are most commonly stored with a .ppm extension although the .pnm extension is also valid. Like for PNG files, a `ImageReaderPNM` is constructed with the name of the file to read from. `ImageReaderPNM` also uses the same `ReadDataSet` and optional `SetPointFieldName` methods.

Example 8.3: Reading an image from a PNM file.

```
#include <vtkm/io/ImageReaderPNM.h>

vtkm::cont::DataSet OpenDataFromPNM()
{
  vtkm::io::ImageReaderPNM imageReader("data.ppm");
  imageReader.SetPointFieldName("pixels");
  return imageReader.ReadDataSet();
}
```

8.2 Writers

All writer classes provided by VTK-m are located in the `vtkm::io` namespace. The general interface for each writer class is to accept a filename in the constructor and to provide a `WriteDataSet` method to save data to the disk. The `WriteDataSet` method takes a `vtkm::cont::DataSet` object as an argument, which contains the data to write to the file.

8.2.1 Legacy VTK File Writer

Legacy VTK files can be written using the `vtkm::io::VTKDataSetWriter` class. The constructor for this class takes a string containing the filename. The `WriteDataSet` method takes a `vtkm::cont::DataSet` object and writes its data to the previously indicated file.

Example 8.4: Writing a legacy VTK file.

```
#include <vtkm/io/VTKDataSetWriter.h>

void SaveDataAsVTKFile(vtkm::cont::DataSet data)
{
  vtkm::io::VTKDataSetWriter writer("data.vtk");

  writer.WriteDataSet(data);
}
```

8.2.2 Image Writers

VTK-m provides classes to some standard image formats. These writers store data in a `vtkm::cont::DataSet`. The data must be a 2D structure with the colors stored in a point field. (See Chapter 7 for details on `DataSet` objects.)

Portable Network Graphics (PNG) files can be written using the `vtkm::io::ImageWriterPNG` class. A `ImageWriterPNG` object is constructed with the name of the file to save. The data are written to the file using the `WriteDataSet` method. This method takes a `DataSet` object as an argument. An optional second argument can be given to name the field containing color data to write. If the field is not given, the first point field of the appropriate type is used.

By default, PNG files are written as RGBA colors using 8-bits for each component. You can change the format written using the `SetPixelDepth` method. This takes an item in the `ImageWriterPNG::PixelDepth` enumeration. Valid values are

`PixelDepth::PIXEL_8` 8-bit RGBA values (default).

`PixelDepth::PIXEL_16` 16-bit RGBA values.

Example 8.5: Writing an image to a PNG file.

```
#include <vtkm/io/ImageWriterPNG.h>

void WriteToPNG(const vtkm::cont::DataSet& dataSet)
{
  vtkm::io::ImageWriterPNG imageWriter("data.png");
  imageWriter.SetPixelDepth(vtkm::io::ImageWriterPNG::PixelDepth::PIXEL_16);
  imageWriter.WriteDataSet(dataSet);
}
```

Portable anymap (PNM) files can be written using the `vtkm::io::ImageWriterPNM` class. Currently, the PNM file writer only supports files using the portable pixmap (PPM) format (with magic number "P6"). These files are most commonly stored with a .ppm extensions although the .pnm extension is also valid.

Like for PNG files, a `ImageReaderPNM` is constructed with the name of the file to write. `ImageReaderPNM` also uses the same `WriteDataSet` and `SetPixelDepth`. It also sports the same `ImageReaderPNM::PixelDepth` enumeration.

Example 8.6: Writing an image to a PNM file.

```
#include <vtkm/io/ImageWriterPNM.h>

void WriteToPNM(const vtkm::cont::DataSet& dataSet)
{
  vtkm::io::ImageWriterPNM imageWriter("data.ppm");
  imageWriter.SetPixelDepth(vtkm::io::ImageWriterPNM::PixelDepth::PIXEL_16);
  imageWriter.WriteDataSet(dataSet);
}
```

CHAPTER

NINE

RUNNING FILTERS

Filters are functional units that take data as input and write new data as output. Filters operate on `vtkm::cont::DataSet` objects, which are described in Chapter7.

Did you know?

The structure of filters in VTK-m is significantly simpler than their counterparts in VTK. VTK filters are arranged in a dataflow network (a.k.a. a visualization pipeline) and execution management is handled automatically. In contrast, VTK-m filters are simple imperative units, which are simply called with input data and return output data.

VTK-m comes with several filters ready for use, and in this chapter we will give a brief overview of these filters. All VTK-m filters are currently defined in the `vtkm::filter` namespace. We group filters based on the type of operation that they do and the shared interfaces that they have. Later Part III describes the necessary steps in creating new filters in VTK-m.

Different filters will be used in different ways, but the basic operation of all filters is to instantiate the filter class, set the state parameters on the filter object, and then call the filter's `Execute` method. The `Execute` method takes a `vtkm::cont::DataSet` and returns a new `DataSet`, which contains the modified data. The `Execute` method can alternately take a `vtkm::cont::PartitionedDataSet` object, which is a composite of `DataSet` objects. In this case `Execute` will return another `PartitionedDataSet` object.

The following example provides a simple demonstration of using a filter. It specifically uses the point elevation filter to estimate the air pressure at each point based on its elevation.

Example 9.1: Using `PointElevation`, which is a field filter.

```
1  VTKM_CONT
2  vtkm::cont::DataSet ComputeAirPressure(vtkm::cont::DataSet dataSet)
3  {
4    vtkm::filter::field_transform::PointElevation elevationFilter;
5
6    // Use the elevation filter to estimate atmospheric pressure based on the
7    // height of the point coordinates. Atmospheric pressure is 101325 Pa at
8    // sea level and drops about 12 Pa per meter.
9    elevationFilter.SetLowPoint(0.0, 0.0, 0.0);
10   elevationFilter.SetHighPoint(0.0, 0.0, 2000.0);
11   elevationFilter.SetRange(101325.0, 77325.0);
12
13   elevationFilter.SetUseCoordinateSystemAsField(true);
14
15   elevationFilter.SetOutputFieldName("pressure");
```

```
16
17   vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
18
19   return result;
20 }
```

We see that this example follows the previously described procedure of constructing the filter (line 4), setting the state parameters (lines 9–15), and finally executing the filter on a `DataSet` (line 17).

Every `vtkm::cont::DataSet` object contains a list of *fields*, which describe some numerical value associated with different parts of the data set in space. Fields often represent physical properties such as temperature, pressure, or velocity. Fields are identified with string names. There are also special fields called coordinate systems that describe the location of points in space. Field are mentioned here because they are often used as input data to the filter's operation and filters often generate new fields in the output. This is the case in Example 9.1. In line 13 the coordinate system is set as the input field and in line 15 the name to use for the generated output field is selected.

9.1 Provided Filters

VTK-m comes with the implementation of many filters. Filters in VTK-m are divided into a collection of modules, each with its own namespace and library. This section is organized by each filter module, each of which contains one or more filters that are related to each other.

9.1.1 Cleaning Grids

The `vtkm::filter::clean_grid` module contains filters that resolve issues with mesh structure. This could include finding and merging coincident points, removing degenerate cells, or converting the grid to a known type.

Clean Grid

`vtkm::filter::clean_grid::CleanGrid` is a filter that converts a cell set to an explicit representation and potentially removes redundant or unused data. It does this by iterating over all cells in the data set, and for each one creating the explicit cell representation that is stored in the output. (Explicit cell sets are described in Section 7.2.2.) One benefit of using `CleanGrid` is that it can optionally remove unused points and combine coincident points. Another benefit is that the resulting cell set will be of a known specific type.

Common Errors

The result of `vtkm::filter::clean_grid::CleanGrid` is not necessarily smaller, memory-wise, than its input. For example, "cleaning" a data set with a structured topology will actually result in a data set that requires much more memory to store an explicit topology.

`CleanGrid` provides the following methods.

`SetCompactPointFields/GetCompactPointFields` Sets a Boolean flag that determines whether unused points are removed from the output. If true (the default), then the output data set will have a new coordinate

system containing only those points being used by the cell set, and the indices of the cells will be adjusted to the new ordering of points.

`SetMergePoints/GetMergePoints` Sets a Boolean flag that determines whether points coincident in space are merged into a single point. If true (the default), then the output data set will have a new coordinate system containing containing only points that are unique in space, and the indices of the cells will be adjusted to the new set of points. The tolerance parameters control the proximity used for points to be considered coincident.

`SetTolerance/GetTolerance` Defines the tolerance used when determining whether two points are considered coincident. Because floating point parameters have limited precision, point coordinates that are essentially the same might not be bit-wise exactly the same. Thus, the `CleanGrid` filter has the ability to find and merge points that are close but perhaps not exact. The default tolerance is 10^{-6}.

`SetToleranceIsAbsolute/GetToleranceIsAbsolute` Sets a Boolean flag that determines whether the tolerance parameter should be considered relative to the size of the data set. If false (the default), then the tolerance is multiplied by the length of the diagonal of the bounds of the data being processed. If true, then the tolerance value is used as is.

`SetRemoveDegenerateCellsGetRemoveDegenerateCells` Sets a Boolean flag that determines whether degenerate cells should be removed. If true (the default), then the `CleanGrid` filter will look for repeated points in cells and, if the repeated points cause the cell to drop dimensionality, the cell is removed. This is particularly useful when point merging is on as this operation can create degenerate cells.

`SetFastMerge/GetFastMerge` Sets a Boolean flag that determines whether to use a faster but less accurate method for finding coincident points. If true (the default), some corners are cut when computing coincident points. This will make the point merge step go faster but the tolerance will not be strictly followed. If false, then extra steps will be taken to ensure that all points within tolerance are merged and that only points within tolerance are merged. This flag has no effect if point merging is off.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use when finding coincident points. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.2 Connected Components

Connected components in a mesh are groups of mesh elements that are connected together in some way. For example, if two cells are neighbors, then they are in the same component. Likewise, a cell is also in the same component as its neighbor's neighbors as well as their neighbors and so on. Connected components help identify when features in a simulation fragment or meld.

The `vtkm::filter::connected_components` module contains filters that find groups of cells that are connected. There are different ways to define what it means to be connected. One way is to use the topological connections of the cells. That is, two cells that share a point, edge, or face are connected. Another way is to use a field that classifies each cell, and cells are only connected if they have the same classification.

Cell Connectivity

The vtkm::filter::connected_components::CellSetConnectivity filter finds groups of cells that are connected together through their topology. Two cells are considered connected if they share an edge. CellSetConnectivity identifies some number of components and assigns each component a unique integer.

The result of the filter is a cell field of type vtkm::Id. Each entry in the cell field will be a number that identifies to which component the cell belongs. By default, this output cell field is named "component".

CellSetConnectivity provides the following methods.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Classification Field on Image Data

The vtkm::filter::connected_components::ImageConnectivity filter finds groups of points that have the same field value and are connected together through their topology. Any point is considered to be connected to its Moore neighborhood: 8 neighboring points for 2D and 26 neighboring points for 3D. As the name implies, ImageConnectivity only works on data with a structured cell set. You will get an error if you use any other type of cell set.

The active field passed to the filter must be associated with the points.

The result of the filter is a point field of type vtkm::Id. Each entry in the point field will be a number that identifies to which component the cell belongs. By default, this output point field is named "component".

ImageConnectivity provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.3 Contouring

The vtkm::filter::contour module contains filters that extract regions that match some field or spatial criteria. Unlike entity extraction filters (Section 9.1.5), the geometry will be clipped or sliced to extract the exact matching region. (In contrast, entity extraction filters will pull unmodified points, edges, faces, or cells from the input.)

Contour

Contouring is one of the most fundamental filters in scientific visualization. A contour is the locus where a field is equal to a particular value. A topographic map showing curves of various elevations often used when hiking in hilly regions is an example of contours of an elevation field in 2 dimensions. Extended to 3 dimensions, a contour gives a surface. Thus, a contour is often called an *isosurface*. The contouring/isosurface algorithm is implemented by `vtkm::filter::contour::Contour`.

`Contour` provides the following methods.

`SetIsoValue` Specifies the value on which to extract a contour. Multiple iso values can be specified at once. The first parameter to `SetIsoValue` is the index of the iso value (starting at index 0) and the second is the value to set. If only one iso value is needed, the index does not need to be specified.

`GetIsoValue` Returns the iso value for a given index.

`SetNumberOfIsoValues` Sets the number of iso values to create contours from. If the number of iso values is not specified, the number is automatically set to accommodate the largest index given to `SetIsoValue`.

`GetNumberOfIsoValues` Returns the number of iso values (and consequently the number of contours to be created).

`SetMergeDuplicatePoints/GetMergeDuplicatePoints` Specifies whether coincident points in the data set should be merged. Because the contour filter (like all filters in VTK-m) runs in parallel, parallel threads can (and often do) create duplicate versions of points. When this flag is set to true, a secondary operation will find all duplicated points and combine them together.

`SetGenerateNormals/GetGenerateNormals` Specifies whether to generate normal vectors for the surface. Normals are used in shading calculations during rendering and can make the surface appear more smooth. By default, the generated normals are based on the gradient of the field being contoured and can be quite expensive to compute. A faster method is available that computes the normals based on the faces of the isosurface mesh, but the normals do not look as good as the gradient based normals. Fast normals can be enabled using the flags described bellow.

`SetComputeFastNormalsForStructured/GetComputeFastNormalsForStructured` Specifies whether to use the fast method of normals computation for Structured data sets. This is only valid if the generate normals flag is set.

`SetComputeFastNormalsForUnstructured/GetComputeFastNormalsForUnstructured` Specifies whether to use the fast method of normals computation for unstructured data sets. This is only valid if the generate normals flag is set.

`SetNormalArrayName/GetNormalArrayName` Specifies the name used for the normals field if it is being created.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.2: Using `Contour`, which is a data set with field filter.

```
1  vtkm::filter::contour::Contour contour;
2
3  contour.SetActiveField("pointvar");
4  contour.SetIsoValue(10.0);
5
6  vtkm::cont::DataSet isosurface = contour.Execute(inData);
```

Slice

A slice operation intersects a mesh with a surface. The `vtkm::filter::contour::Slice` filter uses a `vtkm::ImplicitFunctionGeneral` to specify an implicit surface to slice on. A plane is a common thing to slice on, but other surfaces are available. See Chapter 14 for information on implicit functions.

`Slice` provides the following methods.

`SetImplicitFunction/GetImplicitFunction` Specifies the surface to intersect the input data as an implicit function. See Chapter 14 for documentation on implicit functions provided by VTK-m.

`SetMergeDuplicatePoints/GetMergeDuplicatePoints` Specifies whether coincident points in the data set should be merged. Because the contour filter (like all filters in VTK-m) runs in parallel, parallel threads can (and often do) create duplicate versions of points. When this flag is set to true, a secondary operation will find all duplicated points and combine them together.

`SetGenerateNormals/GetGenerateNormals` Specifies whether to generate normal vectors for the surface. Normals are used in shading calculations during rendering and can make the surface appear more smooth. By default, the generated normals are based on the gradient of the field being contoured and can be quite expensive to compute. A faster method is available that computes the normals based on the faces of the isosurface mesh, but the normals do not look as good as the gradient based normals. Fast normals can be enabled using the flags described bellow.

`SetComputeFastNormalsForStructured/GetComputeFastNormalsForStructured` Specifies whether to use the fast method of normals computation for Structured data sets. This is only valid if the generate normals flag is set.

`SetComputeFastNormalsForUnstructured/GetComputeFastNormalsForUnstructured` Specifies whether to use the fast method of normals computation for unstructured data sets. This is only valid if the generate normals flag is set.

`SetNormalArrayName/GetNormalArrayName` Specifies the name used for the normals field if it is being created.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use when computing normals and other geometric features. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Clip with Field

Clipping is an operation that removes regions from the data set based on a user-provided value or function. The `vtkm::filter::contour::ClipWithField` filter takes a clip value as an argument and removes regions where a

named scalar field is below (or above) that value. (A companion filter that discards a region of the data based on an implicit function is described later.)

The result of `ClipWithField` is a volume. If a cell has field values at its vertices that are all below the specified value, then it will be discarded entirely. Likewise, if a cell has field values at its vertices that are all above the specified value, then it will be retained in its entirety. If a cell has some vertices with field values below the specified value and some above, then the cell will be split into the portions above the value (which will be retained) and the portions below the value (which will be discarded).

This operation is sometimes called an *isovolume* because it extracts the volume of a mesh that is inside the iso-region of a scalar. This is in contrast to an *isosurface*, which extracts only the surface of that iso-value. `ClipWithField` is also similar to a threshold operation, which extracts cells based on the value of field. The difference is that threshold will either keep or remove entire cells based on the field values whereas clip with carve cells that straddle the valid regions. (See section 9.1.5 for threshold extraction.)

`ClipWithField` provides the following methods.

`SetClipValue/GetClipValue` Specifies the field value for the clip operation. Regions where the active field is less than this value are clipped away from each input cell.

`SetInvertClip` Specifies if the result for the clip filter should be inverted. If set to false (the default), regions where the active field is less than the specified clip value are removed. If set to true, regions where the active field is more than the specified clip value are removed.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.3: Using `ClipWithField`.

```
1  // Create an instance of a clip filter that discards all regions with scalar
2  // value less than 25.
3  vtkm::filter::contour::ClipWithField clip;
4  clip.SetClipValue(25.0);
5  clip.SetActiveField("pointvar");
6
7  // Execute the clip filter
8  vtkm::cont::DataSet outData = clip.Execute(inData);
```

Clip with Implicit Function

The `vtkm::filter::contour::ClipWithImplicitFunction` function takes an implicit function and removes all parts of the data that are inside (or outside) that function. See Chapter 14 for more detail on how implicit functions are represented in VTK-m. (A companion filter that discards a region of the data based on the value of a scalar field is described previously.)

The result of `ClipWithImplicitFunction` is a volume. If a cell has its vertices positioned all outside the implicit function, then it will be discarded entirely. Likewise, if a cell its vertices all inside the implicit function, then it will be retained in its entirety. If a cell has some vertices inside the implicit function and some outside, then the cell will be split into the portions inside (which will be retained) and the portions outside (which will be discarded).

`ClipWithImplicitFunction` provides the following methods.

`SetImplicitFunction/GetImplicitFunction` Specifies the implicit function to be used to perform the clip operation. See Chapter 14 for documentation on implicit functions provided by VTK-m.

`SetInvertClip` Specifies whether the result of the clip filter should be inverted. If set to false (the default), all regions where the implicit function is negative will be removed. If set to true, all regions where the implicit function is positive will be removed.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use when comparing spatial locations for the implicit function. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

In the example provided below the `vtkm::Sphere` implicit function is used. This function evaluates to a negative value if points from the original dataset occur within the sphere, evaluates to 0 if the points occur on the surface of the sphere, and evaluates to a positive value if the points occur outside the sphere.

Example 9.4: Using `ClipWithImplicitFunction`.

```
1  // Parameters needed for implicit function
2  vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);
3
4  // Create an instance of a clip filter with this implicit function.
5  vtkm::filter::contour::ClipWithImplicitFunction clip;
6  clip.SetImplicitFunction(implicitFunction);
7
8  // By default, ClipWithImplicitFunction will remove everything inside the sphere.
9  // Set the invert clip flag to keep the inside of the sphere and remove everything
10 // else.
11 clip.SetInvertClip(true);
12
13 // Execute the clip filter
14 vtkm::cont::DataSet outData = clip.Execute(inData);
```

9.1.4 Density Estimation

Density estimation takes a collection of samples and estimates the density of the samples in each part of the domain (or estimate the probabilty that a sample would be at a location in the domain). The domain of samples could be a physical space, such as with particle density, or in an abstract place, such as with a histogram. The `vtkm::filter::density_estimate` module contains filters that estimate density in a variety of ways.

Histogram

The `vtkm::filter::density_estimate::Histogram` filter computes a histogram of a given scalar field. The histogram divides the range of the field into an even number of bins and counts the number of instances occur in each bin.

The default name for the output fields is "histogram". The name can be overridden as always using the `SetOutputFieldName` method.

`Histogram` provides the following methods.

`SetRange/GetRange` Specifies an explicit range to use to generate the histogram. If no range is set then the global range of the field is computed during filter execution and that range is used.

`SetNumberOfBins/GetNumberOfBins` Specifies the number of bins to divide the range. The range of data will be split evenly into this number of bins, and the number of items landing in each bin will be counted. The default number of bins is 10.

`GetBinDelta` Get the size of each bin from the last computed field. This value is only valid after a call to `Execute`.

`GetComputedRange` Get the range used during the last call to `Execute`. If `SetRange` was called, then this range will be returned. Otherwise, the computed range is returned. This value is only valid after a call to `Execute`.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Nearest Grid Point

The `vtkm::filter::density_estimate::ParticleDensityNearestGridPoint` filter defines a 3D grid of bins. It then takes from the input a collection of particles, identifies which bin each particle lies in, and sums some attribute from a field of the input (or the particles can simply be counted). Once the sum of the attribute is computed for each grid cell, it is optionally divided by the volume of the cell. Thus, the density will be computed as the units of the scalar field per the cubic units of the coordinate system.

This operation is helpful in the analysis of particle-based simulation where the data often requires *conversion* or *deposition* of particles' attributes, such as mass, to an overlaying mesh. This allows further identification of regions of interest based on the spatial distribution of particles attributes, for example, high density regions could be considered as *clusters* or *halos* while low density regions could be considered as *bubbles* or *cavities* in the particle data.

Since there is no specific `CellSet` for particles in VTK-m, `ParticleDensityNearestGridPoint` treats the `CoordinateSystem` of the input `DataSet` as the positions of the particles while ignoring the details of the `CellSet`. Particles are infinitesimal in size with finite mass (or other scalar attributes such as charge). The filter estimates density by imposing a regular grid of bins.

`ParticleDensityNearestGridPoint` provides the following methods.

`SetActiveField/GetActiveFieldName` Specify the particle attribute to be deposited.

`SetDivideByVolume/GetDivideByVolume` If you just want a sum of the attribute in each cell, turn off the DivideByVolume feature of this filter with `SetDivideByVolume(false)`.

`SetComputeNumberDensity/GetComputeNumberDensity` Simply count the number of particles in each cell instead of summing a field variable by calling `SetComputeNumberDensity(true)`.

`SetDimension/GetDimension` Specify the number of bins to estimate particles in with a `vtkm::Id3`. The numbers specify the number of *bins* (i.e. cells in the output mesh) in each dimensions, not the number of points in the output mesh.

`SetOrigin/GetOrigin` Specify the minimum (lower-left) corner of the domain of density estimation.

`SetSpacing/GetSpacing` Specify the spacing of the grid points used to form the grid for density estimation. The size of each bin is equivalent to the spacing.

`SetBounds/GetBounds` Specify the domain of space to estimate point density as a `vtkm::Bounds` object. `SetDimension` must be called before the bounds are set. Calling `SetDimension` will change the ranges of the bounds.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

Cloud in Cell

The `vtkm::filter::density_estimate::ParticleDensityCloudInCell` filter defines a 3D grid of bins. It then takes from the input a collection of particles, identifies which bin each particle lies in, and then redistributes each particle's attribute to the 8 vertices of the containing bin. The filter then sums up all the contributions of particles for each vertex in the grid. This contribution is either an attribute value from an input field or a simple count. This creates a point centered density field. Once the sum of the attribute is computed for each grid points, it is optionally divided by the volume of each cell. Thus, the density will be computed as the units of the scalar field per the cubic units of the coordinate system.

This operation is helpful in the analysis of particle-based simulation where the data often requires *conversion* or *deposition* of particles' attributes, such as mass, to an overlaying mesh. This allows further identification of regions of interest based on the spatial distribution of particles attributes, for example, high density regions could be considered as *clusters* or *halos* while low density regions could be considered as *bubbles* or *cavities* in the particle data.

Since there is no specific `CellSet` for particles in VTK-m, `ParticleDensityCloudInCell` treats the `CoordinateSystem` of the input `DataSet` as the positions of the particles while ignoring the details of the `CellSet`. Particles are infinitesimal in size with finite mass (or other scalar attributes such as charge). The filter estimates density by imposing a regular grid of bins.

`ParticleDensityCloudInCell` provides the following methods.

`SetActiveField/GetActiveFieldName` Specify the particle attribute to be deposited.

`SetDivideByVolume/GetDivideByVolume` If you just want a sum of the attribute in each cell, turn off the DivideByVolume feature of this filter with `SetDivideByVolume(false)`.

`SetComputeNumberDensity/GetComputeNumberDensity` Simply count the number of particles in each cell instead of summing a field variable by calling `SetComputeNumberDensity(true)`.

`SetDimension/GetDimension` Specify the number of bins to estimate particles in with a `vtkm::Id3`. The numbers specify the number of *bins* (i.e. cells in the output mesh) in each dimensions, not the number of points in the output mesh.

`SetOrigin/GetOrigin` Specify the minimum (lower-left) corner of the domain of density estimation.

`SetSpacing/GetSpacing` Specify the spacing of the grid points used to form the grid for density estimation. The size of each bin is equivalent to the spacing.

`SetBounds/GetBounds` Specify the domain of space to estimate point density as a `vtkm::Bounds` object. `SetDimension` must be called before the bounds are set. Calling `SetDimension` will change the ranges of the bounds.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

9.1.5 Entity Extraction

VTK-m contains a collection of filters that extract a portion of one `DataSet` and construct a new `DataSet` based on that portion of the geometry. These filters are collected in the `vtkm::filter::entity_extraction` module.

External Faces

`vtkm::filter::entity_extraction::ExternalFaces` is a filter that extracts all the external faces from a polyhedral data set. An external face is any face that is on the boundary of a mesh. Thus, if there is a hole in a volume, the boundary of that hole will be considered external. More formally, an external face is one that belongs to only one cell in a mesh.

`ExternalFaces` provides the following methods.

`SetCompactPoints/GetCompactPoints` Specifies whether point fields should be compacted. If on, the filter will remove from the output all points that are not used in the resulting surface. If off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

`SetPassPolyData/GetPassPolyData` Specifies how polygonal data (polygons, lines, and vertices) will be handled. If on (the default), these cells will be passed to the output. If off, these cells will be removed from the output. (Because they have less than 3 topological dimensions, they are not considered to have any "faces.")

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

External Geometry

The vtkm::filter::entity_extraction::ExternalGeometry filter extracts all of the cells in a DataSet that is inside or outside of an implicit function. Implicit functions are described in Chapter 14. They define a function in 3D space that follow a geometric shape. The inside of the implicit function is the region of negative values.

ExternalGeometry differs from ClipWithImplicitFunction in that the clip filter will subdivide boundary cells into new cells whereas this filter will not. Thus, the output of ExternalGeometry will produce a more "crinkly" output.

ExternalGeometry provides the following methods.

`SetImplicitFunction/GetImplicitFunction` Specifies the implicit function that specifies the region of space to extract geometry from.

`SetExtractInside/GetExtractInside` Specify whether to extract the geometry that is on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

`ExtractInsideOnExtractInsideOff` Set the extract inside flag on (true) or off (false).

`SetExtractBoundaryCells/GetExtractBoundaryCells` The implicit function used to extract geometry is likely to intersect some of the cells of the input. If this flag is true, then any cells intersected by the implicit function are extracted and included in the output. This flag is false by default.

`ExtractBoundaryCellsOn/ExtractBoundaryCellsOff` Set the Extract boundary cells flag to on (true) or off (false).

`SetExtractOnlyBoundaryCells/GetExtractOnlyBoundaryCells` Specify whether to extract only the cells intersected by the implicit function (where the function equals 0). By default this flag is off.

`ExtractOnlyBoundaryCellsOn/ExtractOnlyBoundaryCellsOff` Set the extract only boundary cells flag to on (true) or off (false).

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use for the positions of the geometry. The coordinate system should match that of the implicit function.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Extract Structured

vtkm::filter::entity_extraction::ExtractStructured is a filter that extracts a volume of interest (VOI) from a structured data set. In addition the filter is able to subsample the VOI while doing the extraction.

The output of this filter is a structured dataset. The filter treats input data of any topological dimension (i.e., point, line, plane, or volume) and can generate output data of any topological dimension.

Typical applications of this filter are to extract a slice from a volume for image processing, subsampling large volumes to reduce data size, or extracting regions of a volume with interesting data.

ExtractStructured provides the following methods.

`SetVOI/GetVOI` Specifies what volume of interest (VOI) should be extracted by the filter. By default the VOI is the entire input.

`SetSampleRate/GetSampleRate` Specifies the sample rate of the VOI. Supports sub-sampling on a per dimension basis.

`SetIncludeBoundary/GetIncludeBoundary` Specifies if the VOI is inclusive or exclusive on the boundary of the VOI.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Ghost Cell Removal

`vtkm::filter::entity_extraction::GhostCellRemove` is a filter that is used to remove cells from a data set according to a cell centered field that is provided to the filter. The default field used for removal is "vtkmGhostCells". The field is of type `vtkm::UInt8`, and represents a bit-field to classify each cell. By default, if the input is a structured data set the filter will attempt to output a structured data set. If this is not possible, an explicit data set is produced. The field specified for cell removal is not passed to the output.

`GhostCellRemove` provides the following methods.

`RemoveAllGhost` Remove all cells where the value is a ghost cell (i.e. `vtkm::CellClassification::GHOST`).

`RemoveByType` Remove cells specified by the `vtkm::UInt8` using a bitwise "and" operation with the type field. The values in `vtkm::CellClassification` can be combined with a logical "or" operation to specify the type. Current values of `vtkm::CellClassification` include: `NORMAL`, `GHOST`, and `INVALID`.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as the ghost cell levels.

Threshold

A threshold operation removes topology elements from a data set that do not meet a specified criterion. The `vtkm::filter::entity_extraction::Threshold` filter removes all cells where the a field is between a range of values.

Note that `Threshold` either passes an entire cell or discards an entire cell. This can consequently lead to jagged surfaces at the interface of the threshold caused by the shape of cells that jut inside or outside the removed region. See Section 9.1.3 for a clipping filter that will clip off a smooth region of the mesh.

`Threshold` provides the following methods.

`SetLowerThreshold/GetLowerThreshold` Specifies the lower scalar value. Any cells where the scalar field is less than this value are removed.

`SetUpperThreshold/GetUpperThreshold` Specifies the upper scalar value. Any cells where the scalar field is more than this value are removed.

`SetThresholdBelow` Sets the threshold criterion to pass any value less than or equal to the provided value.

`SetThresholdAbove` Sets the threshold criterion to pass any value greater than or equal to the provided value.

`SetThresholdBetween` Sets the threshold criterion to pass any value between the given values (inclusive). This method is equivalent to calling `SetLowerThreshold` with the first argument and `SetUpperThreshold` with the second argument.

`SetAllInRange/GetAllInRange` When thresholding on a point field, each cell must consider the multiple values associated with all incident points. When this flag is false (the default), the cell is passed if *any* of the incident points matches the threshold criterion. When this flag is true, the cell is passed only if *all* the incident points match the threshold criterion.

`SetComponentToTest` Specifies which vector component to apply the threshold criterion to (default 0). When thresholding on a vector field (which has more than one component per entry), the `Threshold` filter will by default compare the threshold criterion to the first component of the vector (component index 0). Use `SetComponentToTest` to change the component to test against.

`SetComponentToTestToAny` Specifies that the threshold criterion should be applied to all the components of the input vector field and a cell will pass if *any* the components match.

`SetComponentToTestToAll` Specifies that the threshold criterion should be applied to all the components of the input vector field and a cell will pass only if *all* the components match.

`SetInvert/GetInvert` When set to true, the threshold result is inverted. That is, cells that would have been in the output with this option set to false (the default) are excluded while cells that would have been excluded from the output are included.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.6 Field Conversion

Field conversion modifies a field of a `DataSet` to have roughly equivalent values but with a different structure. These filters allow the field to be used in places where they otherwise would not be applicable.

Cell Average

vtkm::filter::field_conversion::CellAverage is the cell average filter. It will take a data set with a collection of cells and a field defined on the points of the data set and create a new field defined on the cells. The values of this new derived field are computed by averaging the values of the input field at all the incident points. This is a simple way to convert a point field to a cell field.

The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName` method.

CellAverage provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Point Average

vtkm::filter::field_conversion::PointAverage is the point average filter. It will take a data set with a collection of cells and a field defined on the cells of the data set and create a new field defined on the points. The values of this new derived field are computed by averaging the values of the input field at all the incident cells. This is a simple way to convert a cell field to a point field.

The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName` method.

PointAverage provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.7 Field Transform

VTK-m provides multiple filters to convert fields through some mathematical relationship.

Cylindrical Coordinate System Transform

vtkm::filter::field_transform::CylindricalCoordinateTransform is a coordinate system transformation filter. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a Cartesian coordinate system to a cylindrical coordinate system. The order for cylindrical coordinates is (R, θ, Z). The output coordinate system will be set to the new computed coordinates.

CylindricalCoordinateTransform provides the following methods.

`SetCartesianToCylindrical` This method specifies a transformation from cartesian to cylindrical coordinates.

`SetCylindricalToCartesian` This method specifies a transformation from cylindrical to cartesian coordinates.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Field to Colors

vtkm::filter::FieldToColors takes a field in a data set, looks up each value in a color table, and writes the resulting colors to a new field. The color to be used for each field value is specified using a vtkm::cont::ColorTable object. ColorTable objects are also used with VTK-m's rendering module and are described in Section 10.8.

FieldToColors has three modes it can use to select how it should treat the input field. These input modes are contained in FieldToColors::InputMode.

FieldToColors::InputMode::Scalar Treat the field as a scalar field. It is an error to provide a field of any type that cannot be directly converted to a basic floating point number (such as a vector).

FieldToColors::InputMode::Magnitude Given a vector field, take the magnitude of each field value before looking it up in the color table.

FieldToColors::InputMode::Component Select a particular component of the vectors in a field to map to colors.

Additionally, FieldToColors has different modes in which it can represent colors in its output. These output modes are contained in FieldToColors::OutputMode.

FieldToColors::OutputMode::RGB Output colors are represented as RGB values with each component represented by an unsigned byte. Specifically, these are vtkm::Vec3ui_8 values.

`FieldToColors::OutputMode::RGBA` Output colors are represented as RGBA values with each component represented by an unsigned byte. Specifically, these are `vtkm::Vec4ui_8` values.

`FieldToColors` provides the following methods.

`SetColorTable/GetColorTable` Specifies the `vtkm::cont::ColorTable` object to use to map field values to colors.

`SetMappingMode/GetMappingMode` Specifies the input mapping mode. The value is one of the `FieldToColors::InputMode::Scalar`, `FieldToColors::InputMode::Magnitude`, or `FieldToColors::InputMode::Component` selectors described previously.

`SetMappingToScalar` Sets the input mapping mode to scalar. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::InputMode::Scalar )`.

`SetMappingToMagnitude` Sets the input mapping mode to vector. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::InputMode::Magnitude )`.

`SetMappingToComponent` Sets the input mapping mode to component. Shortcut for `SetMappingMode(vtkm::filter::FieldToColors::InputMode::Component )`.

`IsMappingScalar` Returns true if the input mapping mode is scalar (`FieldToColors::InputMode::Scalar`).

`IsMappingMagnitude` Returns true if the input mapping mode is magnitude (`FieldToColors::InputMode::Magnitude`).

`IsMappingComponent` Returns true if the input mapping mode is component (`FieldToColors::InputMode::Component`).

`SetMappingComponent/GetMappingComponent` Specifies the component of the vector to use in the mapping. This only has an effect if the input mapping mode is set to `FieldToColors::InputMode::Component`.

`SetOutputMode/GetOutputMode` Specifies the output representation of colors. The value is one of the `FieldToColors::OutputMode::RGB` or `FieldToColors::OutputMode::RGBA` selectors described previously.

`SetOutputToRGB` Sets the output representation to 8-bit RGB. Shortcut for `SetOutputMode(vtkm::filter::FieldToColors::OutputMode::RGB )`.

`SetOutputToRGBA` Sets the output representation to 8-bit RGBA. Shortcut for `SetOutputMode(vtkm::filter::FieldToColors::OutputMode::RGBA )`.

`IsOutputRGB` Returns true if the output representation is 8-bit RGB (`FieldToColors::OutputMode::RGB`).

`IsOutputRGBA` Returns true if the output representation is 8-bit RGBA (`FieldToColors::OutputMode::RGBA`).

`SetNumberOfSamplingPoints/GetNumberOfSamplingPoints` Specifies how many samples to use when looking up color values. The implementation of `FieldToColors` first builds an array of color samples to quickly look up colors for particular values. The size of this lookup array can be adjusted with this parameter. By default, an array of 256 colors is used.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Generate Ids

`vtkm::filter::field_transform::GenerateIds` is a filter that creates point and/or id fields that mimic the identifier for the respective element.

`GenerateIds` provides the following methods.

`SetPointFieldName/GetPointFieldName` Specify the name of the point id field generated. By default, this is set to "pointids".

`SetCellFieldName/GetCellFieldName` Specify the name of the cell id field generated. By default, this is set to "cellids".

`SetGeneratePointIds/GetGeneratePointIds` Specify whether to generate the the point id field. By default, this is true, meaning the field will be generated.

`SetGenerateCellIds/GetGenerateCellIds` Specify whether to generate the the cell id field. By default, this is true, meaning the field will be generated.

`SetUseFloat/GetUseFloat` Specify whether to generate fields of integers or floats. If true, the geneated fields will be of type `vtkm::FloatDefault`. If false, the default, the generated fields will be of type `vtkm::Id`.

Point Elevation

`vtkm::filter::field_transform::PointElevation` computes the "elevation" of a field of point coordinates in space. The filter will take a data set and a field of 3 dimensional vectors and compute the distance along a line defined by a low point and a high point. Any point in the plane touching the low point and perpendicular to the line is set to the minimum range value in the elevation whereas any point in the plane touching the high point and perpendicular to the line is set to the maximum range value. All other values are interpolated linearly between these two planes. This filter is commonly used to compute the elevation of points in some direction, but can be repurposed for a variety of measures. Example 9.1 gives a demonstration of the elevation filter.

The default name for the output field is "elevation", but that can be overridden as always using the `SetOutputFieldName` method.

`PointElevation` provides the following methods.

`SetLowPoint/SetHighPoint` This pair of methods is used to set the low and high points, respectively, of the elevation. Each method takes three floating point numbers specifying the x, y, and z components of the low or high point.

`SetRange` Sets the range of values to use for the output field. This method takes two floating point numbers specifying the low and high values, respectively.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Point Transform

`vtkm::filter::field_transform::PointTransform` is the point transform filter. The filter will take a data set and a field of 3 dimensional vectors and perform the specified point transform operation. Multiple point transformations can be accomplished by subsequent calls to the filter and specifying the result of the previous transform as the input field.

The default name for the output field is "transform", but that can be overridden as always using the `SetOutputFieldName` method. By default, produced field replaces the coordinate system.

`PointTransform` provides the following methods.

`SetTranslation` This method translates, or moves, each point in the input field by a given direction. This method takes either a three component vector of floats, or the x, y, z translation values separately.

`SetRotation` This method is used to rotate the input field about a given axis. This method takes a single floating point number to specify the degrees of rotation and either a vector representing the rotation axis, or the x, y, z axis components separately.

`SetRotationX` This method is used to rotate the input field about the x axis. This method takes a single floating point number to specify the degrees of rotation.

`SetRotationY` This method is used to rotate the input field about the y axis. This method takes a single floating point number to specify the degrees of rotation.

`SetRotationZ` This method is used to rotate the input field about the $z0$ axis. This method takes a single floating point number to specify the degrees of rotation.

`SetScale` This method is used to scale the input field. This method takes either a single float to scale each vector component of the field equally, or the x, y, z scaling values as separate floats, or a three component vector.

`SetTransform` This is a generic transform method. This method takes a $4x4$ matrix and applies this to the input field.

`SetChangeCoordinateSystem/GetChangeCoordinateSystem` When this flag is on, the default, the coordinate system in the output `DataSet` is replaced with the transformed point coordinates .

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Spherical Coordinate System Transform

`vtkm::filter::field_transform::SphericalCoordinateTransform` is a coordinate system transformation filter. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a cartesian coordinate system to a spherical coordinate system. The order for spherical coordinates is (R, θ, ϕ). The output coordinate system will be set to the new computed coordinates.

`CylindricalCoordinateTransform` provides the following methods.

`SetCartesianToSpherical` This method specifies a transformation from cartesian to spherical coordinates.

`SetSphericalToCartesian` This method specifies a transformation from spherical to cartesian coordinates.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Warp Scalar

`vtkm::filter::field_transform::WarpScalar` is a specialized point transformation filter. The filter transforms points by moving them based on a scalar field and a constant scale factor. This filter is useful for creating carpet plots.

The `WarpScalar` filter will take a data set, a normal field, a scalar field, and a constant scale factor. The coordinates will be scaled based on the scalar field and the scale factor. If no explicit normal field is provided the filter will search for a field named “normal”. If no explicit scalar field is provided the filter will search for a field named “scalarfactor”.

The default name for the output field is "warpscalar", but that can be overridden as always using the `SetOutputFieldName` method.

In addition to the standard `SetOutputFieldName` and `Execute` methods, `WarpScalar` provides the following methods.

`SetNormalField` This method allows the user to select the name of the normal field. The normal field is the B field in the warp equation of $A + B \times scaleAmount \times scalarFactor$ (where A is the original position of the point).

`SetScalarFactorField` This method allows the user to select the name of the scale factor field. The scale factor field is the $scalarFactor$ field in the warp equation of $A + B \times scaleAmount \times scalarFactor$ (where A is the original position of the point).

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Warp Vector

`vtkm::filter::field_transform::WarpVector` is a specialized point transformation filter. The filter transforms points by moving them based on a vector field and a constant scale factor. This filter can be used to highlight interesting features such as flow or deformations.

The `WarpScalar` filter will take a data set, a vector field, and a constant scale factor. The coordinates will be scaled based on the vector field and the scale factor. If no explicit vector field is provided the filter will search for a field named "normal".

The default name for the output field is "warpvector", but that can be overridden as always using the `SetOutputFieldName` method.

In addition the standard `SetOutputFieldName` and `Execute` methods, `WarpVector` provides the following methods.

`SetVectorField` This method allows the user to select the name of the vector field. The vector field is the B field in the warp equation of $A + B$ (where A is the original position of the point).

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.8 Geometry Refinement

Geometry refinement modifies the geometry of a `DataSet`. It might add, change, or remove components of the structure, but the general representation will be the same.

Split Sharp Edges

`vtkm::filter::geometry_refinement::SplitSharpEdges` splits sharp manifold edges where the feature angle between the adjacent surfaces are larger than a threshold value. When an edge is split, it adds a new point to the coordinates and updates the connectivity of an adjacent surface.

For example, consider two adjacent triangles (0,1,2) and (2,1,3). Edge (1,2) needs to be split. Two new points 4 (duplication of point 1) and 5 (duplication of point 2) would be added and the later triangle's connectivity would be changed to (5,4,3). By default, all old point's fields would be copied to the new point.

Note that "split" edges do not have space added between them. They are still adjacent visually, but the topology becomes disconnectered there. Splitting sharp edges is most useful to duplicate normal shading vectors to make a sharp shading effect.

`SplitSharpEdges` contains the following methods.

`SetFeatureAngle/GetFeatureAngle` Specifies the angle, in degrees, to split edges. Any time the normal vectors to 2 adjacent polygons are greater than this angle, the points of this edge are duplicated. The default angle is 30 degrees.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Tetrahedralize

`vtkm::filter::geometry_refinement::Tetrahedralize` converts all the polyhedra in a `DataSet` into tetrahedra.

`Tetrahedralize` has the following methods.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Triangulate

`vtkm::filter::geometry_refinement::Triangulate` converts all the polyhedra in a `DataSet` into tetrahedra.

`Triangulate` has the following methods.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Tube

`vtkm::filter::geometry_refinement::Tube` generates a tube around each line and polyline in the input data set. The radius, number of sides, and end capping can be specified for each tube. The orientation of the geometry of the tube are computed automatically using a heuristic to minimize the twisting along the input data set.

`Tube` provides the following methods.

`SetRadius` Specifies the radius of the tube.

`SetNumberOfSides` Specifies the number of sides for the tube geometry.

`SetCapping` Specifies if the ends of the tube should be capped.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.5: Using `Tube`, which is a data set with field filter.

```
1  vtkm::filter::geometry_refinement::Tube tubeFilter;
2
3  tubeFilter.SetRadius(0.5f);
4  tubeFilter.SetNumberOfSides(7);
5  tubeFilter.SetCapping(true);
6
7  vtkm::cont::DataSet output = tubeFilter.Execute(inData);
```

Vertex Clustering

vtkm::filter::vertex_clustering::VertexClustering is a filter that simplifies a polygonal mesh. It does so by dividing space into a uniform grid of bin and then merges together all points located in the same bin. The smaller the dimensions of this binning grid, the fewer polygons will be in the output cells and the coarser the representation. This surface simplification is an important operation to support level of detail (LOD) rendering in visualization applications.

VertexClustering provides the following methods.

`SetNumberOfDivisions/GetNumberOfDimensions` Specifies the dimensions of the uniform grid that establishes the bins used for clustering. Setting smaller numbers of dimensions produces a smaller output, but with a coarser representation of the surface. The dimensions are provided as a vtkm::Id3.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.6: Using VertexClustering.

```
1  vtkm::filter::geometry_refinement::VertexClustering vertexClustering;
2
3  vertexClustering.SetNumberOfDivisions(vtkm::Id3(128, 128, 128));
4
5  vtkm::cont::DataSet simplifiedSurface = vertexClustering.Execute(originalSurface);
```

9.1.9 Mesh Information

VTK-m provides several filters that derive information about the structure of the geometry. This can be information about the shape of cells or their connections.

Ghost Cell Classification

vtkm::filter::mesh_info::GhostCellClassify adds a cell centered field to the input data set that marks each cell as either vtkm::CellClassification::Normal or vtkm::CellClassification::Ghost. The outer layer of cells are marked as `Ghost`, and the remainder are marked as *CellClassificationNormal. This filter only supports uniform and rectilinear data sets. The default field is "vtkmGhostCells".

GhostCellClassify provides the following methods.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Mesh Quality Metrics

vtkm::filter::MeshQuality is a filter that can calculate various metrics for evaluating mesh quality. It generates a new field as output, based on the cells of an input data set. The metrics for this filter come from the Verdict library, and full mathematical descriptions for each metric can be found in the Verdict documentation.[1]

[1] https://www.csimsoft.com/download?file=Documents/sand20071751.pdf

The following table lists the supported mesh quality metrics. The constructor for `vtkm::filter::MeshQuality` takes an argument from the enum `vtkm::filter::CellMetric`, which specifies the desired metric to calculate. The possible values for the `CellMetric` enum are listed in the following table. Also listed in the table is the default name for the field being created, which can be overridden using the `SetOutputFieldName` method. Also, because most metrics only work on select types of cells, the table specifies on which cell types each metric works using the abbreviations "Tri" for triangles, "Quad" for quadrilaterals, "Tet" for tetrahedrons, "Pyr" for pyramids, "Wed" for wedges, and "Hex" for hexahedrons.

Constructor Argument	Default Output Name	Supported Cell Types	Brief Description
`CellMetric::Area`	area	Tri, Quad	Area of a 2D cell
`CellMetric::AspectGamma`	aspectGamma	Tet	Compare root-mean-square edge length to volume
`CellMetric::AspectRatio`	aspectRatio	Tri, Quad, Tet, Hex	Ratio involving longest edge and circumradius
`CellMetric::Condition`	condition	Tri, Quad, Tet, Hex	Condition number of weighted Jacobian matrix
`CellMetric::DiagonalRatio`	diagonalRatio	Quad, Hex	Ratio of minimum and maximum diagonals
`CellMetric::Dimension`	dimension	Hex	Designed specifically for Sandia's Pronto code
`CellMetric::Jacobian`	jacobian	Quad, Tet, Hex	Minimum determinant of Jacobian matrix, over corners and cell center
`CellMetric::MaxAngle`	maxAngle	Tri, Quad	Maximum angle within cell, in degrees
`CellMetric::MaxDiagonal`	maxDiagonal	Hex	Length of maximum diagonal in cell
`CellMetric::MinAngle`	minAngle	Tri, Quad	Minimum angle within cell, in degrees
`CellMetric::MinDiagonal`	minDiagonal	Hex	Length of minimum diagonal in cell
`CellMetric::Oddy`	oddy	Quad, Hex	Maximum deviation of a metric tensor from an identity matrix, over all corners and cell center

Constructor Argument	Default Output Name	Supported Cell Types	Brief Description
CellMetric::-RelativeSizeSquared	relativeSizeSquared	Tri, Quad, Tet, Hex	The ratio of the area/volume of this cell compared to mesh average
CellMetric::ScaledJacobian	scaledJacobian	Tri, Quad, Tet, Hex	Derived from the Jacobian metric, with normalization involving edge length
CellMetric::Shape	shape	Tri, Quad, Tet, Hex	Varies by shape type, including incorporating Jacobian or Condition Consult Verdict manual
CellMetric::ShapeAndSize	shapeAndSize	Tri, Quad, Tet, Hex	Shape metric multiplied by relative size squared metric
CellMetric::Shear	shear	Quad, Hex	Min. value of Jacobian at each corner, divided by length of adjacent edges
CellMetric::Skew	skew	Quad, Hex	Maximum angle between principle axes
CellMetric::Stretch	stretch	Quad, Hex	Ratio of minimum edges and maximum diagonal, normalized for unit cube
CellMetric::Taper	taper	Quad	Maximum ratio of cross-derivative with associated principal axis
CellMetric::Volume	volume	Tet, Pyr, Wed, Hex	Volume of a 3D cell
CellMetric::Warpage	warpage	Quad	Captures angles between diagonals

Finally, MeshQuality provides the following methods.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.10 Multi-Block

Data with multiple blocks are stored in vtkm::cont::PartitionedDataSet objects. Most VTK-m filters operate correctly on PartitionedDataSet just like they do with DataSet. However, there are some filters that are designed with operations specific to multi-block datasets.

AMR Arrays

An AMR mesh is a `vtkm::cont::PartitionedDataSet` with a special structure in the partitions. Each partition has a `vtkm::cont::CellSetStructured`. The partitions form a hierarchy of grids where each level of the hierarchy refines the one above.

`PartitionedDataSet` does not explicitly store the structure of an AMR grid. The `vtkm::filter::AmrArrays` filter determines the hierarchical structure of the AMR partitions and stores information about them in cell field arrays on each partition. Cell fields with the following names are created for each partition.

vtkAmrLevel The AMR level at which the partition resides (with 0 being the most coarse level). All the values for a particular partition are set to the same value.

vtkAmrIndex A unique identifier for each partition of a particular level. Each partition of the same level will have a unique index, but the indices will repeat across levels. All the values for a particular partition are set to the same value.

vtkCompositeIndex A unique identifier for each partition. This index is the same as the index used for the partition in the containing `PartitionedDataSet`. All the values for a particular partition are set to the same value.

vtkGhostType It is common for refinement levels in an AMR structure to overlap more coarse grids. In this case, the overlapped coarse cells have invalid data. The vtkGhostType field will track which cells are overlapped and should be ignored. This array will have a 0 value for all valid cells and a non-zero value for all invalid cells. (Specifically, if the bit specified by `vtkm::CellClassification::BLANKED` is set, then the cell is overlapped with a cell in a finer level.)

Did you know?

The names of these arrays (e.g. vtkAmrLevel) are chosen to be compatible with the equivalent arrays in VTK. This is why they use the prefix of "vtk" instead of "vtkm". Likewise, the flags used for vtkGhostType are compatible with VTK.

`AmrArrays` provides the following methods.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.11 Vector Analysis

VTK-m's vector analysis filters compute operations on fields related to vectors (usually in 3-space).

Cross Product

`vtkm::filter::cross_product::CrossProduct` computes the cross product of two vector fields for every element in the input data set. The cross product filter computes (PrimaryField × SecondaryField), where both

the primary and secondary field are specified using methods on the CrossProduct class. The cross product computation works for both point and cell centered vector fields.

CrossProduct provides the following methods.

SetPrimaryField/GetPrimaryFieldName Specifies the name of the field to use as input for the primary (first) value of the cross product.

SetUseCoordinateSystemAsPrimaryField/GetUseCoordinateSystemAsPrimaryField Specifies a Boolean flag that determines whether to use point coordinates as the primary input field. Set to false by default. When true, the name for the primary field is ignored.

SetPrimaryCoordinateSystem/GetPrimaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the primary input field. The default index is 0, which is the first coordinate system.

SetSecondaryField/GetSecondaryFieldName Specifies the name of the field to use as input for the secondary (second) value of the cross product.

SetUseCoordinateSystemAsSecondaryField/GetUseCoordinateSystemAsSecondaryField Specifies a Boolean flag that determines whether to use point coordinates as the secondary input field. Set to false by default. When true, the name for the secondary field is ignored.

SetSecondaryCoordinateSystem/GetSecondaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the secondary input field. The default index is 0, which is the first coordinate system.

SetOutputFieldName/GetOutputFieldName Specifies the name of the output field generated.

Execute Takes a data set, executes the filter on a device, and returns a data set that contains the result.

SetFieldsToPass/GetFieldsToPass Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Dot Product

vtkm::filter::vector_analysis::DotProduct computes the dot product of two vector fields for every element in the input data set. The dot product filter computes (PrimaryField · SecondaryField), where both the primary and secondary field are specified using methods on the DotProduct class. The dot product computation works for both point and cell centered vector fields.

DotProduct provides the following methods.

SetPrimaryField/GetPrimaryFieldName Specifies the name of the field to use as input for the primary (first) value of the dot product.

SetUseCoordinateSystemAsPrimaryField/GetUseCoordinateSystemAsPrimaryField Specifies a Boolean flag that determines whether to use point coordinates as the primary input field. Set to false by default. When true, the name for the primary field is ignored.

SetPrimaryCoordinateSystem/GetPrimaryCoordinateSystemIndex Specifies the index of which coordinate system to use as the primary input field. The default index is 0, which is the first coordinate system.

SetSecondaryField/GetSecondaryFieldName Specifies the name of the field to use as input for the secondary (second) value of the dot product.

`SetUseCoordinateSystemAsSecondaryField/GetUseCoordinateSystemAsSecondaryField` Specifies a Boolean flag that determines whether to use point coordinates as the secondary input field. Set to false by default. When true, the name for the secondary field is ignored.

`SetSecondaryCoordinateSystem/GetSecondaryCoordinateSystemIndex` Specifies the index of which coordinate system to use as the secondary input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Gradients

`vtkm::filter::vector_analysis::Gradient` computes the gradient of a point based input field for every element in the input data set. The gradient computation can either generate cell center based gradients, which are fast but less accurate, or more accurate but slower point based gradients. The default for the filter is output as cell centered gradients, but can be changed by using the `SetComputePointGradient` method. The default name for the output fields is "Gradients", but that can be overridden as always using the `SetOutputFieldName` method.

`Gradient` provides the following methods.

`SetComputePointGradient/GetComputePointGradient` Specifies whether we are computing point or cell based gradients. The output field(s) of this filter will be point based if this is enabled.

`SetComputeDivergence/GetComputeDivergence` Specifies whether the divergence field will be generated. By default the name of the array will be "Divergence" but can be changed by using `SetDivergenceName`. The field will be a cell field unless `ComputePointGradient` is enabled. The input array must have 3 components in order to compute this. The default is off.

`SetComputeVorticity/GetComputeVorticity` Specifies whether the vorticity field will be generated. By default the name of the array will be "Vorticity" but can be changed by using `SetVorticityName`. The field will be a cell field unless `ComputePointGradient` is enabled. The input array must have 3 components in order to compute this. The default is off.

`SetComputeQCriterion/GetComputeQCriterion` Specifies whether the Q-Criterion field will be generated. By default the name of the array will be "QCriterion" but can be changed by using `SetQCriterionName`. The field will be a cell field unless `ComputePointGradient` is enabled. The input array must have 3 components in order to compute this. The default is off.

`SetComputeGradient/GetComputeGradient` Specifies whether the actual gradient field is written to the output. When processing fields that have 3 components it is desirable to compute information such as Divergence, Vorticity, or Q-Criterion without incurring the cost of also having to write out the 3x3 gradient result. The default is on.

`SetColumnMajorOrdering/SetRowMajorOrdering` When processing input fields that have 3 components, the output will be a a 3x3 gradient. By default VTK-m outputs all matrix like arrays in Row Major ordering (C-Ordering). The ordering can be changed when integrating with libraries like VTK or with FORTRAN codes that use Column Major ordering. The default is Row Major. This setting is only relevant for 3 component input fields when `SetComputeGradient` is enabled.

`SetDivergenceName/GetDivergenceName` Specifies the output cell normals field name. The default is "Divergence".

`SetVorticityName/GetVorticityName` Specifies the output Vorticity field name. The default is "Vorticity"

`SetQCriterionName/GetQCriterionName` Specifies the output Q-Criterion field name. The default is "QCriterion".

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Surface Normals

`vtkm::filter::vector_analysis::SurfaceNormals` computes the surface normals of a polygonal data set at its points and/or cells. The filter takes a data set as input and by default, uses the active coordinate system to compute the normals. Optionally, a coordinate system or a point field of 3d vectors can be explicitly provided to the `Execute` method. The point and cell normals may be oriented to a point outside of the manifold surface by setting the auto orient normals option (`SetAutoOrientNormals`), or they may point inward by also setting flip normals (`SetFlipNormals`) to true. Triangle vertices will be wound counter-clockwise around the cell normals when the consistency option (`SetConsistency`) is enabled. For non-polygonal cells, a zeroed vector is assigned. The point normals are computed by averaging the cell normals of the incident cells of each point.

The default name for the output fields is "Normals", but that can be overridden using the `SetCellNormalsName` and `SetPointNormalsName` methods. The filter will also respect the name in `SetOutputFieldName` if neither of the others are set.

`SurfaceNormals` provides the following methods.

`SetGenerateCellNormals/GetGenerateCellNormals` Specifies whether the cell normals should be generated. This is off by default.

`SetGeneratePointNormals/GetGeneratePointNormals` Specifies whether the point normals should be generated. This is on by default.

`SetNormalizeCellNormals/GetNormalizeCellNormals` Specifies whether cell normals should be normalized (made unit length). This is on by default. The intended use case of this flag is for faster, approximate point normals generation by skipping the normalization of the face normals. Note that when set to false, the result cell normals will not be unit length normals and the point normals will be different.

`SetAutoOrientNormals/GetAutoOrientNormals` If true, any generated point and/or cell normals will be oriented to point outwards from the surface. This requires a closed manifold surface or else the behavior is undefined. This is off by default.

`SetFlipNormals/GetFlipNormals` When AutoOrientNormals is true, this option will reverse the point and cell normals to point inward. This is off by default.

`SetConsistency/GetConsistency` When GenerateCellNormals is true, this option will ensure that the triangle vertices in the output dataset are wound counter-clockwise around the generated cell normal. This only affects triangles. This is on by default.

`SetCellNormalsName/GetCellNormalsName` Specifies the output cell normals field name. If no cell or point normal name is specified, "Normals" is used.

`SetPointNormalsName/GetPointNormalsName` Specifies the output point normals field name. If no cell or point normal name is specified, "Normals" is used.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Vector Magnitude

`vtkm::filter::vector_analysis::VectorMagnitude` takes a field comprising vectors and computes the magnitude for each vector. The vector field is selected as usual with the `SetActiveField` method. The default name for the output field is "magnitude", but that can be overridden as always using the `SetOutputFieldName` method.

`VectorMagnitude` provides the following methods.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.12 ZFP Compression

vtkm::filter::zfp::ZFPCompressor takes a 1D, 2D, or 3D field and compresses the values using the compression algorithm ZFP. The field is selected as usual with the `SetActiveField` method. The rate of compression is set using `SetRate`. The default name for the output field is "compressed"

ZFPCompressor provides the following methods:

`SetRate/GetRate` Specifies the rate of compression.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

vtkm::filter::zfp::ZFPDecompressor takes a field of compressed values and decompresses into scalar values using the compression algorithm ZFP. The field is selected as usual with the `SetActiveField` method. The rate of compression is set using `SetRate`. The default name for the output field is "decompressed"

ZFPDecompressor provides the following methods:

`SetRate` Specifies the rate of compression.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored and the active coordinate system is used instead.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as the input field. The default index is 0, which is the first coordinate system.

`SetOutputFieldName/GetOutputFieldName` Specifies the name of the output field generated.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.13 Lagrangian Coherent Structures

Lagrangian coherent structures (LCS) are distinct structures present in a flow field that have a major influence over nearby trajectories over some interval of time. Some of these structures may be sources, sinks, saddles, or vortices in the flow field. Identifying Lagrangian coherent structures is part of advanced flow analysis and is an important part of studying flow fields. These structures can be studied by calculating the finite time Lyapunov exponent (FTLE) for a flow field at various locations, usually over a regular grid encompassing the entire flow field. If the provided input dataset is structured, then by default the points in this dataset will be used as seeds for advection.

The `vtkm::filter::LagrangianStructures` filter is used to compute the FTLE of a flow field. `LagrangianStructures` has the following methods.

`SetStepSize/GetStepSize` Set or retrieve the step size for a single advection step for the particles used to calculate the FTLE.

`SetNumberOfSteps/GetNumberOfSteps` Set or retrieve the maximum number of steps a particle is allowed to traverse while calculating the FTLE field.

`SetAdvectionTime/GetAdvectionTime` Set or retrieve the time interval of advection. The FTLE field is calculated over some finite time, and the advection time determines that interval of time used.

`SetUseAuxiliaryGrid/GetUseAuxiliaryGrid` Set or retrieve the flag to use auxiliary grids. When this flag is off (the default), then the points of the mesh representing the vector field are advected and used for computing the FTLE. However, if the mesh is too coarse, the FTLE will likely be inaccurate. Or if the mesh is unstructured the FTLE may be less efficient to compute. When this flag is on, an auxiliary grid of uniformly spaced points is used for the FTLE computation.

`SetAuxiliaryGridDimensions/GetAuxiliaryGridDimensions` Set or retrieve the dimensions of the auxiliary grid for FTLE calculation. Seeds for advection will be placed along the points of this auxiliary grid. This option has no effect unless the UseAuxiliaryGrid option is on.

`SetUseFlowMapOutput/GetUseFlowMapOutput` Set or retrieve the flag to use flow maps instead of advection. If the start and end points for FTLE calculation are known already, advection is an unnecessary step. This flag allows users to bypass advection, and instead use a precalculated flow map. By default this option is off.

`SetFlowMapOutput/GetFlowMapOutput` Set or retrieve the array representing the flow map output to be used for FTLE calculation.

`SetOutputFieldName/GetOutputFieldName` Set or retrieve the name of the output field in the dataset returned by the `LagrangianStructures` filter. By default, the field will be names “FTLE”.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

9.1.14 Stream Tracing

Stream tracing is a visualization technique used to characterize the structure of flow. The flow itself is defined by a vector field of velocities. Stream tracing works by following the path taken by the flow. There are multiple ways in which to represent flow in this manner, and consequently VTK-m contains several filters that trace streams in different ways.

Streamlines

Streamlines are a powerful technique for the visualization of flow fields. A streamline is a curve that is parallel to the velocity vector of the flow field. Individual streamlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field.

`vtkm::filter::Streamline` provides the following methods.

`SetSeeds` Specifies the seed locations for the streamlines. Each seed is advected in the vector field to generate one streamline for each seed. The seeds are specified in an `ArrayHandle` containing `vtkm::Particle` objects.

`SetStepSize` Specifies the step size used for the numerical integrator (4^{th} order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each streamline.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.7: Using `Streamline`, which is a data set with field filter.

```
1  vtkm::filter::flow::Streamline streamlines;
2
3  // Specify the seeds.
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;
5  seedArray.Allocate(2);
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));
8
9  streamlines.SetActiveField("vectorvar");
10 streamlines.SetStepSize(0.1f);
11 streamlines.SetNumberOfSteps(100);
12 streamlines.SetSeeds(seedArray);
13
14 vtkm::cont::DataSet output = streamlines.Execute(inData);
```

Stream Surface

A *stream surface* is defined as a continuous surface that is everywhere tangent to a specified vector field. `vtkm::filter::StreamSurface` computes a stream surface from a set of input points and the vector field of the input data set. The stream surface is created by creating streamlines from each input point and then connecting adjacent streamlines with a series of triangles.

`vtkm::filter::StreamSurface` provides the following methods.

`SetSeeds` Specifies the seed locations for the edge of the stream surface. The seeds are specified in an `ArrayHandle` containing `vtkm::Particle` objects.

`SetStepSize` Specifies the step size used for the numerical integrator (4^{th} order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each seed.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.8: Using `StreamSurface`, which is a data set with field filter.

```
1  vtkm::filter::flow::StreamSurface streamSurface;
2
3  // Specify the seeds.
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;
5  seedArray.Allocate(2);
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));
8
9  streamSurface.SetActiveField("vectorvar");
10 streamSurface.SetStepSize(0.1f);
11 streamSurface.SetNumberOfSteps(100);
12 streamSurface.SetSeeds(seedArray);
13
14 vtkm::cont::DataSet output = streamSurface.Execute(inData);
```

Pathlines

Pathlines are the analog to Streamlines for time varying vector fields. Individual pathlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field. This filter requires two data sets as input. The data set passed into the filter is termed "Previous" and the "Next" data set is specified to the filter using a method.

`vtkm::filter::Pathline` provides the following methods.

`SetPreviousTime` Specifies time value for the input data set.

`SetNextTime` Specifies time value for the next data set.

`SetNextDataSet` Specifies the data set for the next time step.

`SetSeeds` Specifies the seed locations for the pathlines. Each seed is advected in the vector field to generate one streamline for each seed. The seeds are specified in an `ArrayHandle` containing `vtkm::Particle` objects.

`SetStepSize` Specifies the step size used for the numerical integrator (4^{th} order Runge-Kutta method) to integrate the seed locations through the flow field.

`SetNumberOfSteps` Specifies the number of integration steps to be performed on each pathline.

`SetActiveField/GetActiveFieldName` Specifies the name of the field to use as input.

`SetUseCoordinateSystemAsField/GetUseCoordinateSystemAsField` Specifies a Boolean flag that determines whether to use point coordinates as the input field. Set to false by default. When true, the values for the active field are ignored.

`SetActiveCoordinateSystem/GetActiveCoordinateSystemIndex` Specifies the index of which coordinate system to use as when computing spatial locations in the mesh. The default index is 0, which is the first coordinate system.

`Execute` Takes a data set, executes the filter on a device, and returns a data set that contains the result.

`SetFieldsToPass/GetFieldsToPass` Specifies which fields to pass from input to output. By default all fields are passed. See Section 9.2.2 for more details.

Example 9.9: Using `Pathline`, which is a data set with field filter.

```
1  vtkm::filter::flow::Pathline pathlines;
2
3  // Specify the seeds.
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;
5  seedArray.Allocate(2);
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));
8
9  pathlines.SetActiveField("vectorvar");
10 pathlines.SetStepSize(0.1f);
11 pathlines.SetNumberOfSteps(100);
12 pathlines.SetSeeds(seedArray);
13 pathlines.SetPreviousTime(0.0f);
14 pathlines.SetNextTime(1.0f);
15 pathlines.SetNextDataSet(inData2);
16
17 vtkm::cont::DataSet pathlineCurves = pathlines.Execute(inData1);
```

9.2 Advanced Field Management

Most filters work with fields as inputs and outputs to their algorithms. Although in the previous discussions of the filters we have seen examples of specifying fields, these examples have been kept brief in the interest of clarity. In this section we revisit how filters manage fields and provide more detailed documentation of the controls.

Note that not all of the discussion in this section applies to all the aforementioned filters. For example, not all filters have a specified input field. But where possible, the interface to the filter objects is kept consistent.

9.2.1 Input Fields

Many of VTK-m's filters have a method named `SetActiveField`, which selects a field in the input data to use as the data for the filter's algorithm. We have already seen how `SetActiveField` takes the name of the field as an argument. However, `SetActiveField` also takes an optional second argument that specifies which topological elements the field is associated with (such as points or cells). If specified, this argument is one of the following.

`vtkm::cont::Field::Association::Any` Any field regardless of the association. (This is the default if no association is given.)

`vtkm::cont::Field::Association::Points` A field that applies to points. There is a separate field value attached to each point. Point fields usually represent samples of continuous data that can be reinterpolated through cells. Physical properties such as temperature, pressure, density, velocity, etc. are usually best represented in point fields. Data that deals with the points of the topology, such as displacement vectors, are also appropriate for point data.

`vtkm::cont::Field::Association::Cells` A field that applies to cells. There is a separate field value attached to each cell in a cell set. Cell fields usually represent values from an integration over the finite cells of the mesh. Integrated values like mass or volume are best represented in cell fields. Statistics about each cell like strain or cell quality are also appropriate for cell data.

`vtkm::cont::Field::Association::WholeMesh` A "global" field that applies to the whole mesh. These often contain summary or annotation information. An example of a whole mesh field could be the volume that the mesh covers.

Example 9.10: Setting a field's active filter with an association.

```
1 | filter.SetActiveField("pointvar", vtkm::cont::Field::Association::Points);
```

Common Errors

It is possible to have two fields with the same name that are only differentiable by the association. That is, you could have a point field and a cell field with different data but the same name. Thus, it is best practice to specify the field association when possible. Likewise, it is poor practice to have two fields with the same name, particularly if the data are not equivalent in some way. It is often the case that fields are selected without an association.

It is also possible to set the active scalar field as a coordinate system of the data. A coordinate system essentially provides the spatial location of the points of the data and they have a special place in the `vtkm::cont::DataSet` structure. (See Section 7.4 for details on coordinate systems.) You can use a coordinate system as the active scalars by calling the `SetUseCoordinateSystemAsField` method with a true flag. Since a `DataSet` can have multiple coordinate systems, you can select the desired coordinate system with `SetActiveCoordinateSystem`. (By default, the first coordinate system will be used.)

9.2.2 Passing Fields from Input to Output

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. By default, the filter will automatically pass all fields from input to output (performing whatever

transformations are necessary). You can control which fields are passed (and equivalently which are not) with the `SetFieldsToPass` methods of `vtkm::filter::Filter`.

There are multiple ways to to use `Filter::SetFieldsToPass` to control what fields are passed. If you want to turn off all fields so that none are passed, call `SetFieldsToPass` with `vtkm::filter::FieldSelection::Mode::None`.

Example 9.11: Turning off the passing of all fields when executing a filter.

```
1 |     filter.SetFieldsToPass(vtkm::filter::FieldSelection::Mode::None);
```

If you want to pass one specific field, you can pass that field's name to `SetFieldsToPass`.

Example 9.12: Setting one field to pass by name.

```
1 |     filter.SetFieldsToPass("pointvar");
```

Or you can provide a list of fields to pass by giving `SetFieldsToPass` an initializer list of names.

Example 9.13: Using a list of fields for a filter to pass.

```
1 |     filter.SetFieldsToPass({ "pointvar", "cellvar" });
```

If you want to instead select a list of fields to *not* pass, you can add `vtkm::filter::FieldSelection::Mode::Exclude` as an argument to `SetFieldsToPass`.

Example 9.14: Excluding a list of fields for a filter to pass.

```
1 |     filter.SetFieldsToPass({ "pointvar", "cellvar" },
2 |                            vtkm::filter::FieldSelection::Mode::Exclude);
```

Ultimately, `Filter::SetFieldsToPass` takes a `vtkm::filter::FieldSelection` object. You can create one directly to select (or exclude) specific fields and their associations.

Example 9.15: Using `vtkm::filter::FieldSelection`.

```
1 |     vtkm::filter::FieldSelection fieldSelection;
2 |     fieldSelection.AddField("scalars");
3 |     fieldSelection.AddField("cellvar", vtkm::cont::Field::Association::Cells);
4 |
5 |     filter.SetFieldsToPass(fieldSelection);
```

It is also possible to specify field attributions directly to `Filter::SetFieldsToPass`. If you only have one field, you can just specify both the name and attribution. If you have multiple fields, you can provide an initializer list of `std::pair` or `vtkm::Pair` containing a `std::string` and a `vtkm::cont::Field::AssociationEnum`. In either case, you can add an optional last argument of `vtkm::filter::FieldSelection::Mode::Exclude` to exclude the specified filters instead of selecting them.

Example 9.16: Selecting one field and its association for a filter to pass.

```
1 |     filter.SetFieldsToPass("pointvar", vtkm::cont::Field::Association::Points);
```

Example 9.17: Selecting a list of fields and their associations for a filter to pass.

```
1 |     filter.SetFieldsToPass(
2 |       { vtkm::make_Pair("pointvar", vtkm::cont::Field::Association::Points),
3 |         vtkm::make_Pair("cellvar", vtkm::cont::Field::Association::Cells),
4 |         vtkm::make_Pair("scalars", vtkm::cont::Field::Association::Any) });
```

Note that coordinate systems in a `DataSet` are simply links to point fields, and by default filters will pass coordinate systems regardless of the field selection flags. To prevent a filter from passing a coordinate system if its associated field is not selected, use the `SetPassCoordinateSystems` method. When this flag is set to false, coordinate systems are only passed if their associated field is selected.

Example 9.18: Turning off the automatic selection of fields associated with a DataSet's coordinate system.

```
1 |     filter.SetPassCoordinateSystems(false);
```

CHAPTER

TEN

RENDERING

Rendering, the generation of images from data, is a key component to visualization. To assist with rendering, VTK-m provides a rendering package to produce imagery from data, which is located in the `vtkm::rendering` namespace.

The rendering package in VTK-m is not intended to be a fully featured rendering system or library. Rather, it is a lightweight rendering package with two primary use cases:

1. New users getting started with VTK-m need a "quick and dirty" render method to see their visualization results.
2. In situ visualization that integrates VTK-m with a simulation or other data-generation system might need a lightweight rendering method.

Both of these use cases require just a basic rendering platform. Because VTK-m is designed to be integrated into larger systems, it does not aspire to have a fully featured rendering system.

Did you know?

VTK-m's big sister toolkit VTK is already integrated with VTK-m and has its own fully featured rendering system. If you need more rendering capabilities than what VTK-m provides, you can leverage VTK instead.

10.1 Scenes and Actors

The primary intent of the rendering package in VTK-m is to visually display the data that is loaded and processed. Data are represented in VTK-m by `vtkm::cont::DataSet` objects, which are described in Chapter 7. They are also the unit created from I/O operations (Chapter 8 and filters (Chapter 9).

To render a `DataSet`, the data are wrapped in a `vtkm::rendering::Actor` class. The `Actor` holds the components of the `DataSet` to render (a cell set, a coordinate system, and a field). A color table can also be optionally be specified, but a default color table will be specified otherwise.

`Actor`s are collected together in an object called `vtkm::rendering::Scene`. An `Actor` is added to a `Scene` with the `AddActor` method. The following example demonstrates creating a `Scene` with one `Actor`.

Example 10.1: Creating an `Actor` and adding it to a `Scene`.

```
vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                             surfaceData.GetCoordinateSystem(),
                             surfaceData.GetField("RandomPointScalars"));

vtkm::rendering::Scene scene;
scene.AddActor(actor);
```

10.2 Canvas

A *canvas* is a unit that represents the image space that is the target of the rendering. The canvas' primary function is to manage the buffers that hold the working image data during the rendering. The canvas also manages the context and state of the rendering subsystem.

`vtkm::rendering::Canvas` is the base class of all canvas objects. Each type of rendering system has its own canvas subclass, but currently the only rendering system provided by VTK-m is the internal ray tracer. The canvas for the ray tracer is `vtkm::rendering::CanvasRayTracer`. `CanvasRayTracer` is typically constructed by giving the width and height of the image to render.

Example 10.2: Creating a canvas for rendering.

```
vtkm::rendering::CanvasRayTracer canvas(1920, 1080);
```

10.3 Mappers

A *mapper* is a unit that converts data (managed by an `Actor`) and issues commands to the rendering subsystem to generate images. All mappers in VTK-m are a subclass of `vtkm::rendering::Mapper`. Different rendering systems (as established by the `Canvas`) often require different mappers. Also, different mappers could render different types of data in different ways. For example, one mapper might render polygonal surfaces whereas another might render polyhedra as a translucent volume. Thus, a mapper should be picked to match both the rendering system of the `Canvas` and the data in the `Actor`.

The following mappers are provided by VTK-m.

`vtkm::rendering::MapperRayTracer` Uses VTK-m's built in ray tracing system to render the visible surface of a mesh. `MapperRayTracer` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperCylinder` Uses VTK-m's built in ray tracing system to render cylinders as lines of a mesh. `MapperCylinder` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperGlyphScalar` Uses VTK-m's built in ray tracing system to render glyphs at the visible points/vertices of a mesh. It suports rendering points as spheres, axis-aligned cubes, axes and camera-facing quads, coloring the points based on the values of a scalar field. `MapperGlyphScalar` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperGlyphVector` Uses VTK-m's built in ray tracing system to render arrows at the visible points/vertices of a mesh. It suports rendering points by coloring the points based on the values of a vector field. `MapperGlyphVector` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperQuad` Uses VTK-m's built in ray tracing system to render the visible quadrilaterals of a mesh. `MapperQuad` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperVolume` Uses VTK-m's built in ray tracing system to render polyhedra as a translucent volume. `MapperVolume` only works in conjunction with `CanvasRayTracer`.

`vtkm::rendering::MapperWireframer` Uses VTK-m's built in ray tracing system to render the cell edges (i.e. the "wireframe") of a mesh. `MapperWireframer` only works in conjunction with `CanvasRayTracer`.

10.4 Views

A *view* is a unit that collects all the structures needed to perform rendering. It contains everything needed to take a `Scene` (Section 10.1) and use a `Mapper` (Section 10.3) to render it onto a `Canvas` (Section 10.2). The view also annotates the image with spatial and scalar properties.

The base class for all views is `vtkm::rendering::View`. `View` is an abstract class, and you must choose one of the three provided subclasses, `vtkm::rendering::View3D`, `vtkm::rendering::View2D`, and `vtkm::rendering::View3D`, depending on the type of data being presented. All three view classes take a `Scene`, a `Mapper`, and a `Canvas` as arguments to their constructor.

Example 10.3: Constructing a `View`.

```
vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                             surfaceData.GetCoordinateSystem(),
                             surfaceData.GetField("RandomPointScalars"));

vtkm::rendering::Scene scene;
scene.AddActor(actor);

vtkm::rendering::MapperRayTracer mapper;
vtkm::rendering::CanvasRayTracer canvas(1920, 1080);

vtkm::rendering::View3D view(scene, mapper, canvas);
```

The `View` also maintains a *background color* (the color used in areas where nothing is drawn) and a *foreground color* (the color used for annotation elements). By default, the `View` has a black background and a white foreground. These can be set in the view's constructor, but it is a bit more readable to set them using the `View::SetBackground` and `View::SetForeground` methods. In either case, the colors are specified using the `vtkm::rendering::Color` helper class, which manages the red, green, and blue color channels as well as an optional alpha channel. These channel values are given as floating point values between 0 and 1.

Example 10.4: Changing the background and foreground colors of a `View`.

```
view.SetBackgroundColor(vtkm::rendering::Color(1.0f, 1.0f, 1.0f));
view.SetForegroundColor(vtkm::rendering::Color(0.0f, 0.0f, 0.0f));
```

Common Errors

Although the background and foreground colors are set independently, it will be difficult or impossible to see the annotation if there is not enough contrast between the background and foreground colors. Thus, when changing a `View`'s background color, it is always good practice to also change the foreground color.

Once the `View` is constructed, intialized, and set up, it is ready to render. This is done by calling the `View::Paint` method.

Example 10.5: Using `Canvas::Paint` in a display callback.

```
view.Paint();
```

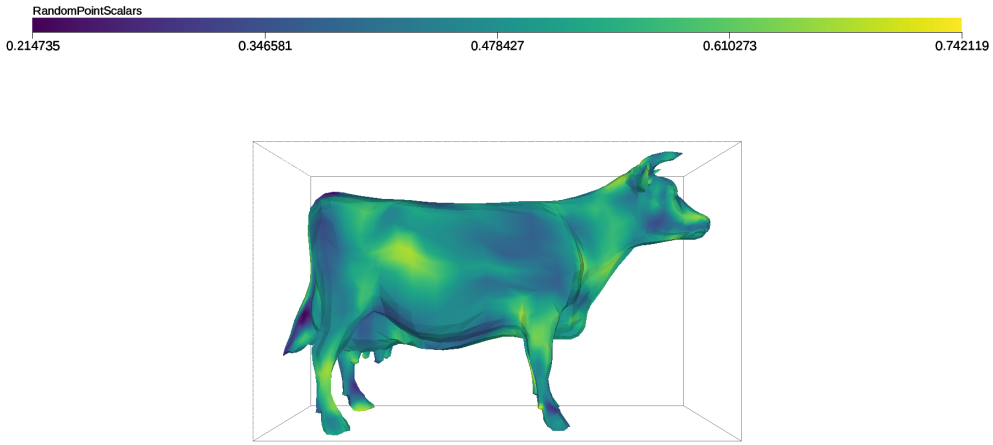


Figure 10.1: Example output of VTK-m's rendering system.

Putting together Examples 10.3, 10.4, and 10.5, the final render of a view looks like that in Figure 10.1.

Of course, the `vtkm::rendering::CanvasRayTracer` created in 10.3 is an offscreen rendering buffer, so you cannot immediately see the image. When doing batch visualization, an easy way to output the image to a file for later viewing is with the `View::SaveAs` method. This method can save the image in either PNG or in the portable pixelmap (PPM) format.

Example 10.6: Saving the result of a render as an image file.

```
1 | view.SaveAs("BasicRendering.png");
```

We visit doing interactive rendering in a GUI later in Section 10.7.

10.5 Changing Rendering Modes

Example 10.3 constructs the default mapper for ray tracing, which renders the data as an opaque solid. However, you can change the rendering mode by using one of the other mappers listed in Section 10.3. For example, say you just wanted to see a wireframe representation of your data. You can achieve this by using `vtkm::rendering::MapperWireframer`.

Example 10.7: Creating a mapper for a wireframe representation.

```
1 | vtkm::rendering::MapperWireframer mapper;
2 | vtkm::rendering::View3D view(scene, mapper, canvas);
```

Alternatively, perhaps you wish to render just the points of mesh. `vtkm::rendering::MapperGlyphScalar` renders the points as glyphs and also optionally can scale the glyphs based on field values.

Example 10.8: Creating a mapper for point representation.

```
1 | vtkm::rendering::MapperGlyphScalar mapper;
2 | mapper.SetGlyphType(vtkm::rendering::GlyphType::Cube);
3 | mapper.SetScaleByValue(true);
```

```
4   mapper.SetScaleDelta(10.0f);
5
6   vtkm::rendering::View3D view(scene, mapper, canvas);
```

These mappers respectively render the images shown in Figure 10.2. Other mappers, such as those that can render translucent volumes, are also available.

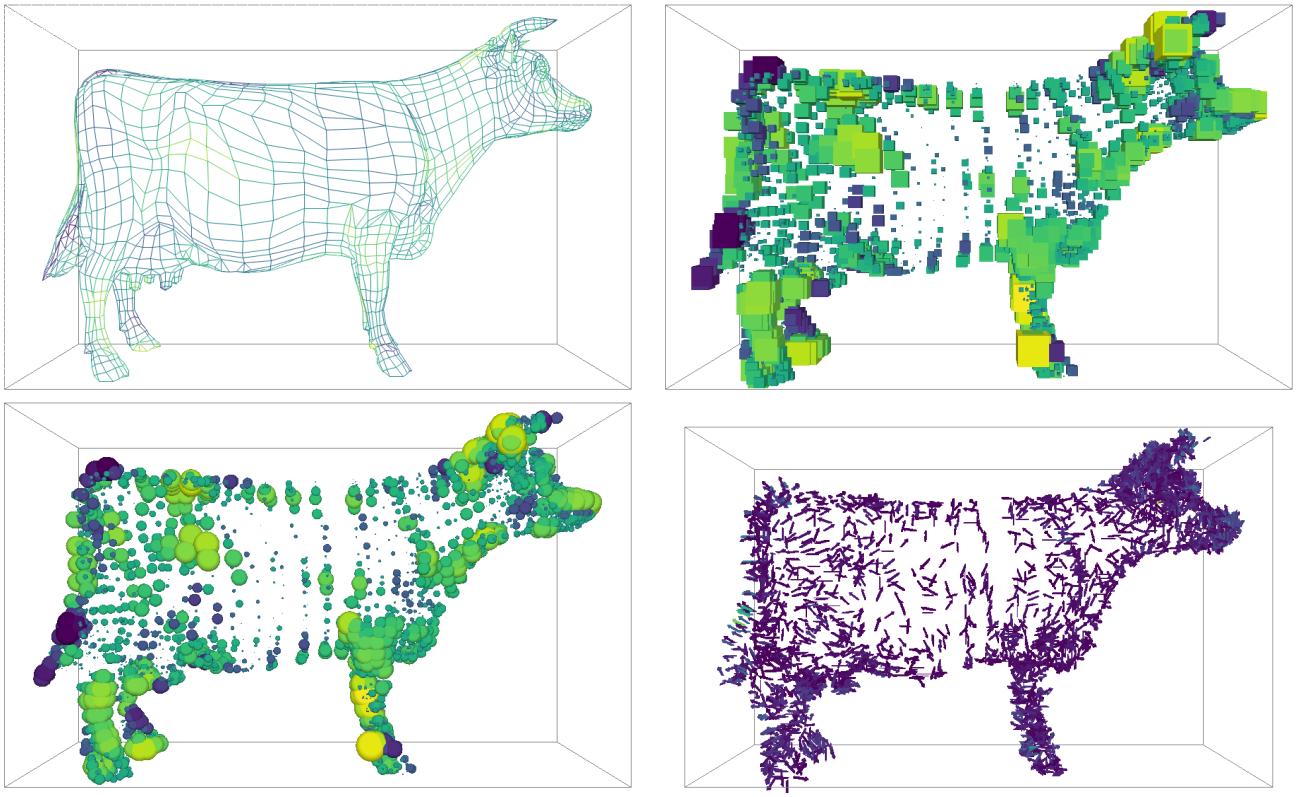


Figure 10.2: Examples of alternate rendering modes using different mappers. The top left image is rendered with `MapperWireframer`. The top right and bottom left images are rendered with `MapperGlyphScalar`. The bottom right image is rendered with `MapperGlyphVector`.

10.6 Manipulating the Camera

The `vtkm::rendering::View` uses an object called `vtkm::rendering::Camera` to describe the vantage point from which to draw the geometry. The camera can be retrieved from the `View::GetCamera` method. That retrieved camera can be directly manipulated or a new camera can be provided by calling `View::SetCamera`. In this section we discuss camera setups typical during view set up. Camera movement during interactive rendering is revisited in Section 10.7.2.

A `Camera` operates in one of two major modes: 2D mode or 3D mode. 2D mode is designed for looking at flat geometry (or close to flat geometry) that is parallel to the x-y plane. 3D mode provides the freedom to place the camera anywhere in 3D space. The different modes can be set with `SetModeTo2D` and `SetModeTo3D`, respectively. The interaction with the camera in these two modes is very different.

10.6.1 2D Camera Mode

The 2D camera is restricted to looking at some region of the x-y plane.

View Range

The vantage point of a 2D camera can be specified by simply giving the region in the x-y plane to look at. This region is specified by calling `Camera::SetViewRange2D`. This method takes the left, right, bottom, and top of the region to view. Typically these are set to the range of the geometry in world space as shown in Figure 10.3.

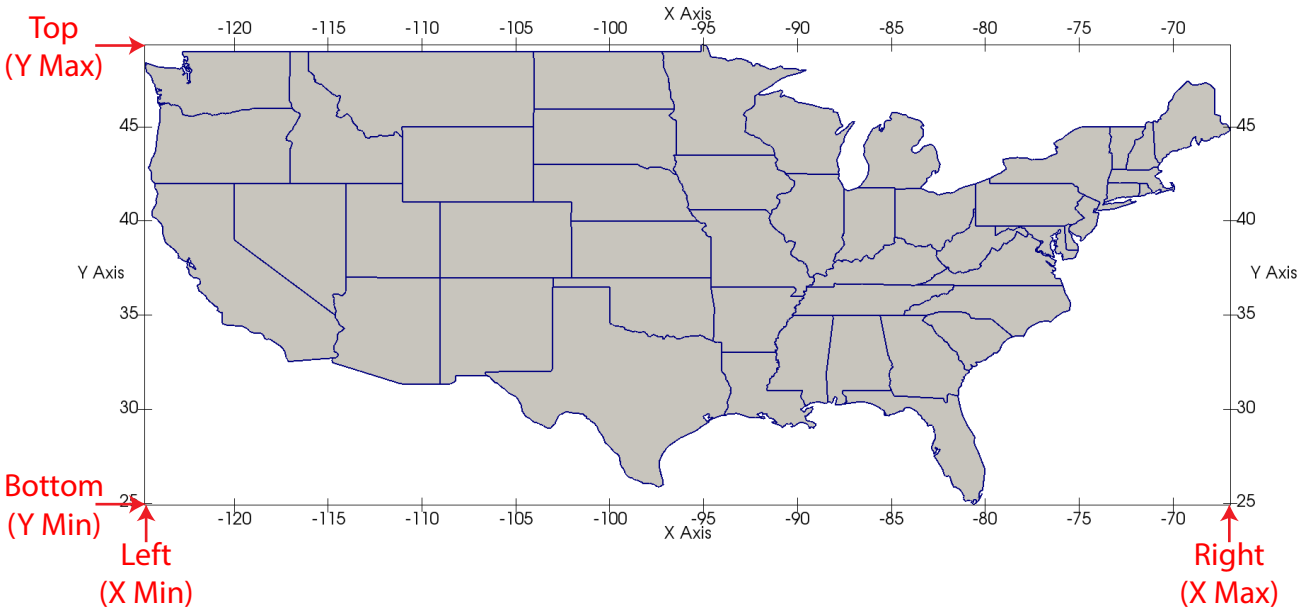


Figure 10.3: The view range bounds to give a `Camera`.

There are 3 overloaded versions of the `SetViewRange2D` method. The first version takes the 4 range values, left, right, bottom, and top, as separate arguments in that order. The second version takes two `vtkm::Range` objects specifying the range in the x and y directions, respectively. The third version trakes a single `vtkm::Bounds` object, which completely specifies the spatial range. (The range in z is ignored.) The `Range` and `Bounds` objects are documented later in Sections 19.3 and 19.4, respectively.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the `Camera::Pan` method. `Pan` takes two arguments: the amount to pan in x and the amount to pan in y.

The pan is given with respect to the projected space. So a pan of 1 in the x direction moves the camera to focus on the right edge of the image whereas a pan of -1 in the x direction moves the camera to focus on the left edge of the image.

Example 10.9: Panning the camera.

```
1 | view.GetCamera().Pan(deltaX, deltaY);
```

Zoom

A camera zoom draws the geometry larger or smaller. A camera zoom is performed by calling the `Camera::Zoom` method. `Zoom` takes a single argument specifying the zoom factor. A positive number draws the geometry larger (zoom in), and larger zoom factor results in larger geometry. Likewise, a negative number draws the geometry smaller (zoom out). A zoom factor of 0 has no effect.

Example 10.10: Zooming the camera.

```
1 | view.GetCamera().Zoom(zoomFactor);
```

10.6.2 3D Camera Mode

The 3D camera is a free-form camera that can be placed anywhere in 3D space and can look in any direction. The projection of the 3D camera is based on the pinhole camera model in which all viewing rays intersect a single point. This single point is the camera's position.

Position and Orientation

The position of the camera, which is the point where the observer is viewing the scene, can be set with the `Camera::SetPosition` method. The direction the camera is facing is specified by giving a position to focus on. This is called either the "look at" point or the focal point and is specified with the `Camera::SetLookAt` method. Figure 10.4 shows the relationship between the position and look at points.

In addition to specifying the direction to point the camera, the camera must also know which direction is considered "up." This is specified with the view up vector using the `Camera::SetViewUp` method. The view up vector points from the camera position (in the center of the image) to the top of the image. The view up vector in relation to the camera position and orientation is shown in Figure 10.4.

Another important parameter for the camera is its field of view. The field of view specifies how wide of a region the camera can see. It is specified by giving the angle in degrees of the cone of visible region emanating from the pinhole of the camera to the `Camera::SetFieldOfView` method. The field of view angle in relation to the camera orientation is shown in Figure 10.4. A field of view angle of 60° usually works well.

Finally, the camera must specify a clipping region that defines the valid range of depths for the object. This is a pair of planes parallel to the image that all visible data must lie in. Each of these planes is defined simply by their distance to the camera position. The near clip plane is closer to the camera and must be in front of all geometry. The far clip plane is further from the camera and must be behind all geometry. The distance to both the near and far planes are specified with the `Camera::SetClippingRange` method. Figure 10.4 shows the clipping planes in relationship to the camera position and orientation.

Example 10.11: Directly setting `vtkm::rendering::Camera` position and orientation.

```
1 | camera.SetPosition(vtkm::make_Vec(10.0, 6.0, 6.0));
2 | camera.SetLookAt(vtkm::make_Vec(0.0, 0.0, 0.0));
3 | camera.SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
4 | camera.SetFieldOfView(60.0);
5 | camera.SetClippingRange(0.1, 100.0);
```

Movement

In addition to specifically setting the position and orientation of the camera, `vtkm::rendering::Camera` contains several convenience methods that move the camera relative to its position and look at point.

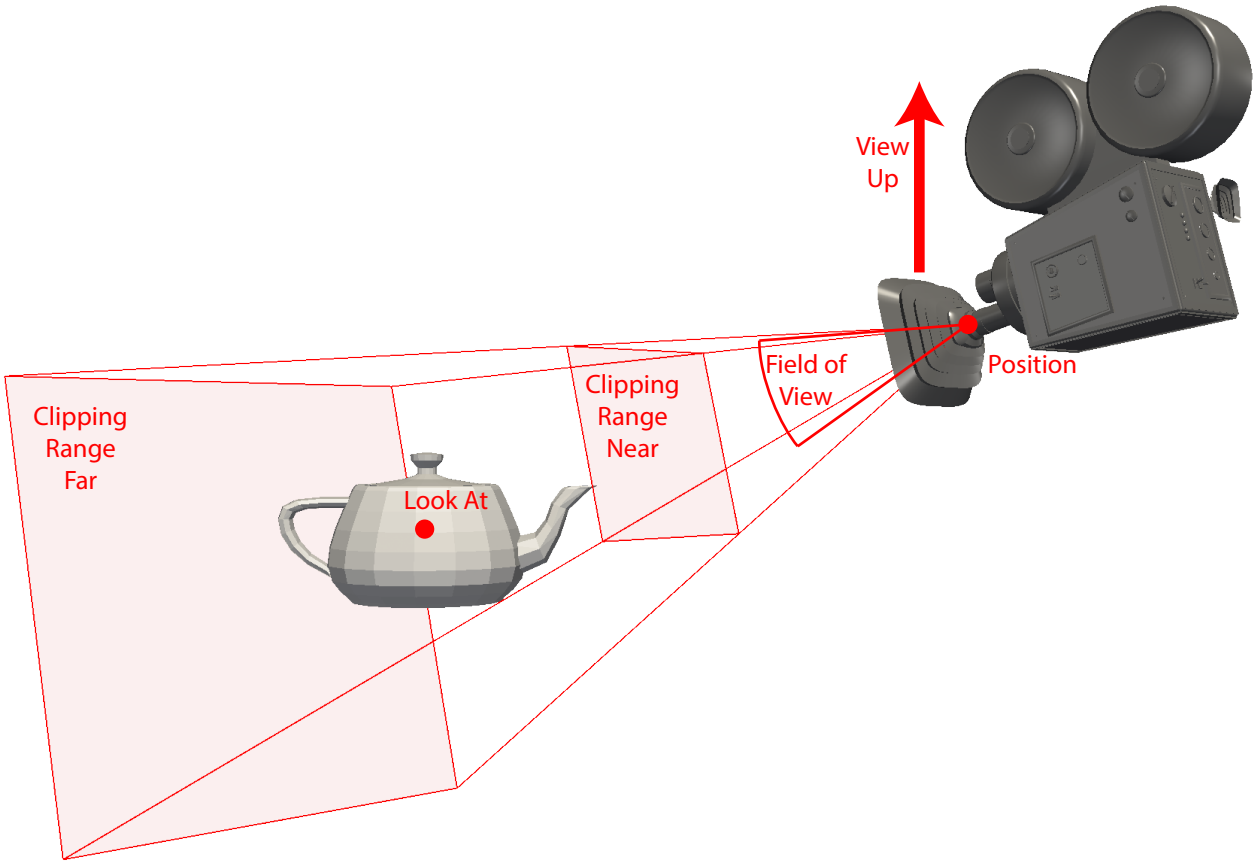


Figure 10.4: The position and orientation parameters for a `Camera`.

Two such methods are elevation and azimuth, which move the camera around the sphere centered at the look at point. `Camera::Elevation` raises or lowers the camera. Positive values raise the camera up (in the direction of the view up vector) whereas negative values lower the camera down. `Camera::Azimuth` moves the camera around the look at point to the left or right. Positive values move the camera to the right whereas negative values move the camera to the left. Both `Elevation` and `Azimuth` specify the amount of rotation in terms of degrees. Figure 10.5 shows the relative movements of `Elevation` and `Azimuth`.

Example 10.12: Moving the camera around the look at point.

```
1  view.GetCamera().Azimuth(45.0);
2  view.GetCamera().Elevation(45.0);
```

Common Errors

The `Camera::Elevation` *and* `Camera::Azimuth` *methods change the position of the camera, but not the view up vector. This can cause some wild camera orientation changes when the direction of the camera view is near parallel to the view up vector, which often happens when the elevation is raised or lowered by about 90 degrees.*

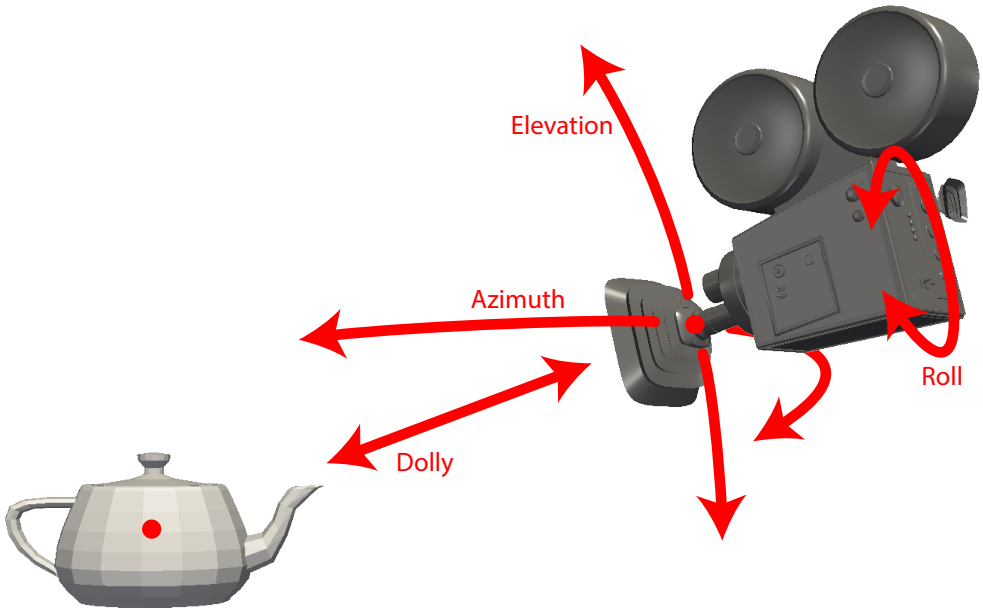


Figure 10.5: `Camera` movement functions relative to position and orientation.

In addition to rotating the camera around the look at point, you can move the camera closer or further from the look at point. This is done with the `Camera::Dolly` method. The `Dolly` method takes a single value that is the factor to scale the distance between camera and look at point. Values greater than one move the camera away, values less than one move the camera closer. The direction of dolly movement is shown in Figure 10.5.

Finally, the `Camera::Roll` method rotates the camera around the viewing direction. It has the effect of rotating the rendered image. The `Roll` method takes a single value that is the angle to rotate in degrees. The direction of roll movement is shown in Figure 10.5.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the `Camera::Pan` method. `Pan` takes two arguments: the amount to pan in x and the amount to pan in y.

The pan is given with respect to the projected space. So a pan of 1 in the x direction moves the camera to focus on the right edge of the image whereas a pan of -1 in the x direction moves the camera to focus on the left edge of the image.

Example 10.13: Panning the camera.

```
1 | view.GetCamera().Pan(deltaX, deltaY);
```

Zoom

A camera zoom draws the geometry larger or smaller. A camera zoom is performed by calling the `Camera::Zoom` method. `Zoom` takes a single argument specifying the zoom factor. A positive number draws the geometry larger (zoom in), and larger zoom factor results in larger geometry. Likewise, a negative number draws the geometry smaller (zoom out). A zoom factor of 0 has no effect.

Example 10.14: Zooming the camera.

```
1 |   view.GetCamera().Zoom(zoomFactor);
```

Reset

Setting a specific camera position and orientation can be frustrating, particularly when the size, shape, and location of the geometry is not known a priori. Typically this involves querying the data and finding a good camera orientation.

To make this process simpler, `vtkm::rendering::Camera` has a convenience method named `Camera::ResetToBounds` that automatically positions the camera based on the spatial bounds of the geometry. The most expedient method to find the spatial bounds of the geometry being rendered is to get the `vtkm::rendering::Scene` object and call `GetSpatialBounds`. The `Scene` object can be retrieved from the `vtkm::rendering::View`, which, as described in Section 10.4, is the central object for managing rendering.

Example 10.15: Resetting a `Camera` to view geometry.

```
1 | void ResetCamera(vtkm::rendering::View& view)
2 | {
3 |   vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
4 |   view.GetCamera().ResetToBounds(bounds);
5 | }
```

The `ResetToBounds` method operates by placing the look at point in the center of the bounds and then placing the position of the camera relative to that look at point. The position is such that the view direction is the same as before the call to `ResetToBounds` and the distance between the camera position and look at point has the bounds roughly fill the rendered image. This behavior is a convenient way to update the camera to make the geometry most visible while still preserving the viewing position. If you want to reset the camera to a new viewing angle, it is best to set the camera to be pointing in the right direction and then calling `ResetToBounds` to adjust the position.

Example 10.16: Resetting a `Camera` to be axis aligned.

```
1 |   view.GetCamera().SetPosition(vtkm::make_Vec(0.0, 0.0, 0.0));
2 |   view.GetCamera().SetLookAt(vtkm::make_Vec(0.0, 0.0, -1.0));
3 |   view.GetCamera().SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
4 |   vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
5 |   view.GetCamera().ResetToBounds(bounds);
```

10.7 Interactive Rendering

So far in our description of VTK-m's rendering capabilities we have talked about doing rendering of fixed scenes. However, an important use case of scientific visualization is to provide an interactive rendering system to explore data. In this case, you want to render into a GUI application that lets the user interact manipulate the view. The full design of a 3D visualization application is well outside the scope of this book, but we discuss in general terms what you need to plug VTK-m's rendering into such a system.

In this section we discuss two important concepts regarding interactive rendering. First, we need to write images into a GUI while they are being rendered. Second, we want to translate user interaction to camera movement.

10.7.1 Rendering Into a GUI

Before being able to show rendering to a user, we need a system rendering context in which to push the images. In this section we demonstrate the display of images using the OpenGL rendering system, which is common for

scientific visualization applications. That said, you could also use other rendering systems like DirectX or even paste images into a blank widget.

Creating an OpenGL context varies depending on the OS platform you are using. If you do not already have an application you want to integrate with VTK-m's rendering, you may wish to start with graphics utility API such as GLUT or GLFW. The process of initializing an OpenGL context is not discussed here.

The process of rendering into an OpenGL context is straightforward. First call `Paint` on the `View` object to do the actual rendering. Second, get the image color data out of the `View`'s `Canvas` object. This is done by calling `Canvas::GetColorBuffer`. This will return a `vtkm::cont::ArrayHandle` object containing the image's pixel color data. (`ArrayHandle`s are discussed in detail in Chapter 16 and subsequent chapters.) A raw pointer can be pulled out of this `ArrayHandle` by casting it to the `vtkm::cont::ArrayHandleBase` subclass and calling the `GetReadPointer` method on that. Third, the pixel color data are pasted into the OpenGL render context. There are multiple ways to do so, but the most straightforward way is to use the `glDrawPixels` function provided by OpenGL. Fourth, swap the OpenGL buffers. The method to swap OpenGL buffers varies by OS platform. The aforementioned graphics libraries GLUT and GLFW each provide a function for doing so.

Example 10.17: Rendering a `View` and pasting the result to an active OpenGL context.

```
1  view.Paint();
2
3  // Get the color buffer containing the rendered image.
4  vtkm::cont::ArrayHandle<vtkm::Vec4f_32> colorBuffer =
5    view.GetCanvas().GetColorBuffer();
6
7  // Pull the C array out of the arrayhandle.
8  const void* colorArray =
9    vtkm::cont::ArrayHandleBasic<vtkm::Vec4f_32>(colorBuffer).GetReadPointer();
10
11 // Write the C array to an OpenGL buffer.
12 glDrawPixels((GLint)view.GetCanvas().GetWidth(),
13              (GLint)view.GetCanvas().GetHeight(),
14              GL_RGBA,
15              GL_FLOAT,
16              colorArray);
17
18 // Swap the OpenGL buffers (system dependent).
```

10.7.2 Camera Movement

When interactively manipulating the camera in a windowing system, the camera is usually moved in response to mouse movements. Typically, mouse movements are detected through callbacks from the windowing system back to your application. Once again, the details on how this works depend on your windowing system. The assumption made in this section is that through the windowing system you will be able to track the x-y pixel location of the mouse cursor at the beginning of the movement and the end of the movement. Using these two pixel coordinates, as well as the current width and height of the render space, we can make several typical camera movements.

Common Errors

Pixel coordinates in VTK-m's rendering system originate in the lower-left corner of the image. However, windowing systems generally report mouse coordinates with the origin in the upper-*left corner. The upshot is that the y coordinates will have to be reversed when translating mouse coordinates to VTK-m image coordinates. This inverting is present in all the following examples.*

Rotate

A common and important mode of interaction with 3D views is to allow the user to rotate the object under inspection by dragging the mouse. To facilitate this type of interactive rotation, `vtkm::rendering::Camera` provides a convenience method named `TrackballRotate`. The `TrackballRotate` method takes a start and end position of the mouse on the image and rotates viewpoint as if the user grabbed a point on a sphere centered in the image at the start position and moved under the end position.

The `TrackballRotate` method is typically called from within a mouse movement callback. The callback must record the pixel position from the last event and the new pixel position of the mouse. Those pixel positions must be normalized to the range -1 to 1 where the position (-1,-1) refers to the lower left of the image and (1,1) refers to the upper right of the image. The following example demonstrates the typical operations used to establish rotations when dragging the mouse.

Example 10.18: Interactive rotations through mouse dragging with `Camera::TrackballRotate`.

```
void DoMouseRotate(vtkm::rendering::View& view,
                   vtkm::Id mouseStartX,
                   vtkm::Id mouseStartY,
                   vtkm::Id mouseEndX,
                   vtkm::Id mouseEndY)
{
  vtkm::Id screenWidth = view.GetCanvas().GetWidth();
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to
  // width/height, to normalized screen coordinates from -1 to 1. Note that y
  // screen coordinates are usually given from the top down whereas our
  // geometry transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
  vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  view.GetCamera().TrackballRotate(startX, startY, endX, endY);
}
```

Pan

Panning can be performed by calling `Camera::Pan` with the translation relative to the width and height of the canvas. For the translation to track the movement of the mouse cursor, simply scale the pixels the mouse has traveled by the width and height of the image.

Example 10.19: Pan the view based on mouse movements.

```
void DoMousePan(vtkm::rendering::View& view,
                vtkm::Id mouseStartX,
                vtkm::Id mouseStartY,
                vtkm::Id mouseEndX,
                vtkm::Id mouseEndY)
{
  vtkm::Id screenWidth = view.GetCanvas().GetWidth();
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to
  // width/height, to normalized screen coordinates from -1 to 1. Note that y
  // screen coordinates are usually given from the top down whereas our
  // geometry transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
```

```
  vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  vtkm::Float32 deltaX = endX - startX;
  vtkm::Float32 deltaY = endY - startY;

  view.GetCamera().Pan(deltaX, deltaY);
}
```

Zoom

Zooming can be performed by calling `Camera::Zoom` with a positive or negative zoom factor. When using `Zoom` to respond to mouse movements, a natural zoom will divide the distance traveled by the mouse pointer by the width or height of the screen as demonstrated in the following example.

Example 10.20: Zoom the view based on mouse movements.

```
void DoMouseZoom(vtkm::rendering::View& view,
                 vtkm::Id mouseStartY,
                 vtkm::Id mouseEndY)
{
  vtkm::Id screenHeight = view.GetCanvas().GetHeight();

  // Convert the mouse position coordinates, given in pixels from 0 to height,
  // to normalized screen coordinates from -1 to 1. Note that y screen
  // coordinates are usually given from the top down whereas our geometry
  // transforms are given from bottom up, so you have to reverse the y
  // coordiantes.
  vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
  vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);

  vtkm::Float32 zoomFactor = endY - startY;

  view.GetCamera().Zoom(zoomFactor);
}
```

10.8 Color Tables

An important feature of VTK-m's rendering units is the ability to pseudocolor objects based on scalar data. This technique maps each scalar to a potentially unique color. This mapping from scalars to colors is defined by a `vtkm::cont::ColorTable` object. A `ColorTable` can be specified as an optional argument when constructing a `vtkm::rendering::Actor`. (Use of `Actor`s is discussed in Section 10.1.)

Example 10.21: Specifying a `ColorTable` for an `Actor`.

```
  vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
                               surfaceData.GetCoordinateSystem(),
                               surfaceData.GetField("RandomPointScalars"),
                               vtkm::cont::ColorTable("inferno"));
```

The easiest way to create a `ColorTable` is to provide the name of one of the many predefined sets of color provided by VTK-m. A list of all available predefined color tables is provided below.

 Viridis	Matplotlib Virdis, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white. This is the default color map.
 Cool to Warm	A color table designed to be perceptually even, to work well on shaded 3D surfaces, and to generally perform well across many uses.
 Cool to Warm Extended	This colormap is an expansion on cool to warm that moves through a wider range of hue and saturation. Useful if you are looking for a greater level of detail, but the darker colors at the end might interfere with 3D surfaces.
 Inferno	Matplotlib Inferno, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.
 Plasma	Matplotlib Plasma, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.
 Black-Body Radiation	The colors are inspired by the wavelengths of light from black body radiation. The actual colors used are designed to be perceptually uniform.
 X Ray	Greyscale colormap useful for making volume renderings similar to what you would expect in an x-ray.
 Green	A sequential color map of green varied by saturation.
 Black - Blue - White	A sequential color map from black to blue to white.
 Blue to Orange	A double-ended (diverging) color table that goes from dark blues to a neutral white and then a dark orange at the other end.
 Gray to Red	A double-ended (diverging) color table with black/gray at the low end and orange/red at the high end.
 Cold and Hot	A double-ended color map with a black middle color and diverging values to either side. Colors go from red to yellow on the positive side and through blue on the negative side.
 Blue - Green - Orange	A three-part color map with blue at the low end, green in the middle, and orange at the high end.
 Yellow - Gray - Blue	A three-part color map with yellow at the low end, gray in the middle, and blue at the high end.
 Rainbow Uniform	A color table that spans the hues of a rainbow. There have been many scientific perceptual studies on the effectiveness of rainbow colors, and they uniformly found them to be ineffective. This color table modifies the hues to make them more perceptually uniform, which should improve the effectiveness of the colors. However, we still recommend the other color tables over this one.
 Jet	A rainbow color table that adds some darkness for greater perceptual resolution. The ends of the jet color table might be too dark for 3D surfaces.
 Rainbow Desaturated	All the badness of the rainbow color table with periodic dark points added, which can help identify rate of change.

CHAPTER

ELEVEN

ERROR HANDLING

VTK-m contains several mechanisms for checking and reporting error conditions.

11.1 Runtime Error Exceptions

VTK-m uses exceptions to report errors. All exceptions thrown by VTK-m will be a subclass of `vtkm::cont::Error`. For simple error reporting, it is possible to simply catch a `vtkm::cont::Error` and report the error message string reported by the `Error::GetMessage` method.

Example 11.1: Simple error reporting.

```
1  int main(int argc, char** argv)
2  {
3    try
4    {
5      // Do something cool with VTK-m
6      // ...
7    }
8    catch (const vtkm::cont::Error& error)
9    {
10     std::cout << error.GetMessage() << std::endl;
11     return 1;
12   }
13   return 0;
14 }
```

There are several subclasses to `vtkm::cont::Error`. The specific subclass gives an indication of the type of error that occurred when the exception was thrown. Catching one of these subclasses may help a program better recover from errors.

`vtkm::cont::ErrorBadAllocation` Thrown when there is a problem accessing or manipulating memory. Often this is thrown when an allocation fails because there is insufficient memory, but other memory access errors can cause this to be thrown as well.

`vtkm::cont::ErrorBadType` Thrown when VTK-m attempts to perform an operation on an object that is of an incompatible type.

`vtkm::cont::ErrorBadValue` Thrown when a VTK-m function or method encounters an invalid value that inhibits progress.

`vtkm::cont::ErrorExecution` Throw when an error is signaled in the execution environment for example when a worklet is being executed.

`vtkm::cont::ErrorInternal` Thrown when VTK-m detects an internal state that should never be reached. This error usually indicates a bug in VTK-m or, at best, VTK-m failed to detect an invalid input it should have.

`vtkm::io::ErrorIO` Thrown by a reader or writer when a file error is encountered.

11.2 Asserting Conditions

In addition to the aforementioned error signaling, the vtkm/Assert.h header file defines a macro named `VTKM_ASSERT`. This macro behaves the same as the POSIX `assert` macro. It takes a single argument that is a condition that is expected to be true. If it is not true, the program is halted and a message is printed. Asserts are useful debugging tools to ensure that software is behaving and being used as expected.

Example 11.2: Using `VTKM_ASSERT`.

```
1 template<typename T>
2 VTKM_CONT T GetArrayValue(vtkm::cont::ArrayHandle<T> arrayHandle, vtkm::Id index)
3 {
4   VTKM_ASSERT(index >= 0);
5   VTKM_ASSERT(index < arrayHandle.GetNumberOfValues());
```

Did you know?

Like the POSIX `assert`*, if the* `NDEBUG` *macro is defined, then* `VTKM_ASSERT` *will become an empty expression. Typically* `NDEBUG` *is defined with a compiler flag (like* `-DNDEBUG`*) for release builds to better optimize the code. CMake will automatically add this flag for release builds.*

Common Errors

A helpful warning provided by many compilers alerts you of unused variables. (This warning is commonly enabled on VTK-m regression test nightly builds.) If a function argument is used only in a `VTKM_ASSERT`*, then it will be required for debug builds and be unused in release builds. To get around this problem, add a statement to the function of the form* `(void)variableName;`*. This statement will have no effect on the code generated but will suppress the warning for release builds.*

11.3 Compile Time Checks

Because VTK-m makes heavy use of C++ templates, it is possible that these templates could be used with inappropriate types in the arguments. Using an unexpected type in a template can lead to very confusing errors, so it is better to catch such problems as early as possible. The `VTKM_STATIC_ASSERT` macro, defined in vtkm/StaticAssert.h makes this possible. This macro takes a constant expression that can be evaluated at compile time and verifies that the result is true.

In the following example, `VTKM_STATIC_ASSERT` and its sister macro `VTKM_STATIC_ASSERT_MSG`, which allows you to give a descriptive message for the failure, are used to implement checks on a templated function that is designed to work on any scalar type that is represented by 32 or more bits.

Example 11.3: Using VTKM_STATIC_ASSERT.

```
template<typename T>
VTKM_EXEC_CONT void MyMathFunction(T& value)
{
  VTKM_STATIC_ASSERT((std::is_same<typename vtkm::TypeTraits<T>::DimensionalityTag,
                                   vtkm::TypeTraitsScalarTag>::value));

  VTKM_STATIC_ASSERT_MSG(sizeof(T) >= 4,
                         "MyMathFunction needs types with at least 32 bits.");
```

Did you know?

In addition to the several trait template classes provided by VTK-m to introspect C++ types, the C++ standard type_traits *header file contains several helpful templates for general queries on types. Example 11.3 demonstrates the use of one such template:* `std::is_same`.

Common Errors

Many templates used to introspect types resolve to the tags `std::true_type` *and* `std::false_type` *rather than the constant values* `true` *and* `false` *that* VTKM_STATIC_ASSERT *expects. The* `std::true_type` *and* `std::false_type` *tags can be converted to the Boolean literal by adding* `::value` *to the end of them. Failing to do so will cause* VTKM_STATIC_ASSERT *to behave incorrectly. Example 11.3 demonstrates getting the Boolean literal from the result of* `std::is_same`.

CHAPTER

TWELVE

MANAGING DEVICES

Multiple vendors vie to provide accelerator-type processors. VTK-m endeavors to support as many such architectures as possible. Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a *device adapter*.

So far in Part II we have been writing code that runs on a local serial CPU. In those examples where we run a filter, VTK-m is launching parallel execution in the execution environment. Internally VTK-m uses a device adapter to manage this execution.

A build of VTK-m generally supports multiple device adapters. In this chapter we describe how to represent and manage devices.

12.1 Device Adapter Tag

A device adapter is identified by a *device adapter tag*. This tag, which is simply an empty struct type, is used as the template parameter for several classes in the VTK-m control environment and causes these classes to direct their work to a particular device. The following device adapter tags are available in VTK-m.

`vtkm::cont::DeviceAdapterTagSerial` Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available. This tag is defined in vtkm/cont/DeviceAdapterSerial.h.

`vtkm::cont::DeviceAdapterTagCuda` Uses a CUDA capable GPU device. For this device to work, VTK-m must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler. This tag is defined in vtkm/cont/cuda/DeviceAdapterCuda.h.

`vtkm::cont::DeviceAdapterTagOpenMP` Uses OpenMP compiler extensions to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. This tag is defined in vtkm/cont/openmp/DeviceAdapterOpenMP.h.

`vtkm::cont::DeviceAdapterTagTBB` Uses the Intel Threading Building Blocks library to run algorithms on multiple threads. For this device to work, VTK-m must be configured to use TBB and the executable must be linked to the TBB library. This tag is defined in vtkm/cont/tbb/DeviceAdapterTBB.h.

`vtkm::cont::DeviceAdapterTagKokkos` Uses the Kokkos library to run algorithms in parallel. For this device to work, VTK-m must be configured to use Kokkos and the executable must be linked to the Kokkos libraries. Vtk-m will use the default execution space of the provided kokkos library build. This tag is defined in vtkm/cont/kokkos/DeviceAdapterKokkos.h.

The following example uses the tag for the Intel Threading Building blocks device adapter to specify a specific device for VTK-m to use. (Details on specifying devices in VTK-m is provided in Section 12.3.)

Example 12.1: Specifying a device using a device adapter tag.

```
1 | vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagTBB{});
```

For classes and methods that have a template argument that is expected to be a device adapter tag, the tag type can be checked with the VTKM_IS_DEVICE_ADAPTER_TAG macro to verify the type is a valid device adapter tag. It is good practice to check unknown types with this macro to prevent further unexpected errors.

12.2 Device Adapter Id

Using a device adapter tag directly means that the type of device needs to be known at compile time. To store a device adapter type at run time, one can instead use vtkm::cont::DeviceAdapterId. DeviceAdapterId is a superclass to all the device adapter tags, and any device adapter tag can be "stored" in a DeviceAdapterId. Thus, it is more common for functions and classes to use DeviceAdapterId then to try to track a specific device with templated code.

In addition to the provided device adapter tags listed previously, a DeviceAdapterId can store some special device adapter tags that do not directly specify a specific device.

vtkm::cont::DeviceAdapterTagAny Used to specify that any device may be used for an operation. In practice this is limited to devices that are currently available.

vtkm::cont::DeviceAdapterTagUndefined Used to avoid specifying a device. Useful as a placeholder when a device can be specified but none is given.

Did you know?

Any device adapter tag can be used where a device adapter id is expected. Thus, you can use a device adapter tag whenever you want to specify a particular device and pass that to any method expecting a device id. Likewise, it is usually more convenient for classes and methods to manage device adapter ids rather than device adapter tag.

DeviceAdapterId contains several helpful methods to get runtime information about a particular device.

`GetName` A `static` method that returns a string description for the device adapter. The string is stored in a type named vtkm::cont::DeviceAdapterNameType, which is currently aliased to `std::string`. The device adapter name is useful for printing information about a device being used.

`GetId` A `static` method taking no arguments that returns a unique integer identifier for the device adapter as a vtkm::Int8.

`IsValueValid` A `static const bool` that is true if the implementation of the integer returned from `GetId` corresponds to a concrete device. So, for example, the `IsValueValid` flag for a DeviceAdapterTagSerial is true whereas the `IsValueValid` flag for a DeviceAdapterTagAny is false.

Did you know?

As a cheat, all device adapter tags actually inherit from the `vtkm::cont::DeviceAdapterId` *class. Thus, all of these methods can be called directly on a device adapter tag.*

Common Errors

Just because the `DeviceAdapterId::IsValueValid` *returns true that does not necessarily mean that this device is available to be run on. It simply means that the device is implemented in VTK-m. However, that device might not be compiled, or that device might not be available on the current running system, or that device might not be enabled. Use the device runtime tracker described in Section 12.3 to determine if a particular device can actually be used.*

12.3 Runtime Device Tracker

It is often the case that you are agnostic about what device VTK-m algorithms run so long as they complete correctly and as fast as possible. Thus, rather than directly specify a device adapter, you would like VTK-m to try using the best available device, and if that does not work try a different device. Because of this, there are many features in VTK-m that behave this way. For example, you may have noticed that running filters, as in the examples of Chapter 9, you do not need to specify a device; they choose a device for you.

However, even though we often would like VTK-m to choose a device for us, we still need a way to manage device preferences. VTK-m also needs a mechanism to record runtime information about what devices are available so that it does not have to continually try (and fail) to use devices that are not available at runtime. These needs are met with the `vtkm::cont::RuntimeDeviceTracker` class. `RuntimeDeviceTracker` maintains information about which devices can and should be run on. VTK-m maintains a `RuntimeDeviceTracker` for each thread your code is operating on. To get the runtime device for the current thread, use the `vtkm::cont::GetRuntimeDeviceTracker` method.

`RuntimeDeviceTracker` has the following methods.

`CanRunOn` Takes a device adapter tag and returns true if VTK-m was configured for the device and it has not yet been marked as disabled.

`DisableDevice` Takes a device adapter tag and marks that device to not be used. Future calls to `CanRunOn` for this device will return false until that device is reset.

`ResetDevice` Takes a `vtkm::cont::DeviceAdapterTag` and resets the state for that device to its default value. Each device defaults to on as long as VTK-m is configured to use that device and a basic runtime check finds a viable device.

`Reset` Resets all devices. This equivocally calls `ResetDevice` for all devices supported by VTK-m.

`ForceDevice` Takes a device adapter tag and enables that device. All other devices are disabled. This method throws a `vtkm::cont::ErrorBadValue` if the requested device cannot be enabled.

`ReportAllocationFailure` A device might have less working memory available than the main CPU. If this is the case, memory allocation errors are more likely to happen. This method is used to report a `vtkm::cont::ErrorBadAllocation` and disables the device for future execution.

`ReportBadDeviceFailure` It is possible that a device may throw a `vtkm::cont::ErrorBadDevice` failure caused by some erroneous device issue. If this occurs, it is possible to catch the `vtkm::cont::ErrorBadDevice` exception and pass it to `ReportBadDeviceFailure` along with the `vtkm::cont::DeviceAdapterId` to forcefully disable a device.

A `RuntimeDeviceTracker` can be used to specify which devices to consider for a particular operation. However, a better way to specify devices is to use the `vtkm::cont::ScopedRuntimeDeviceTracker` class. When a `ScopedRuntimeDeviceTracker` is constructed, it specifies a new set of devices for VTK-m to use. When the `ScopedRuntimeDeviceTracker` is destroyed as it leaves scope, it restores VTK-m's devices to those that existed when it was created.

The following example demonstrates how the `ScopedRuntimeDeviceTracker` is used to force the VTK-m operations that happen within a function to operate exclusively with the TBB device.

Example 12.2: Restricting which devices VTK-m uses per thread.

```
void ChangeDefaultRuntime()
{
  std::cout << "Checking changing default runtime." << std::endl;

  vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagTBB{});

  // VTK-m operations limited to serial devices here...

  // Devices restored as we leave scope.
}
```

In the previous example we forced VTK-m to use the TBB device. This is the default behavior of `ScopedRuntimeDeviceTracker`, but the constructor takes an optional second argument that is a value in the `vtkm::cont::RuntimeDeviceTrackerMode` to specify how modify the current device adapter list.

`RuntimeDeviceTrackerMode::Force` Replaces the current list of devices to try with the device specified to the `ScopedRuntimeDeviceTracker`. This has the effect of forcing VTK-m to use the provided device. This is the default behavior for the `ScopedRuntimeDeviceTracker`.

`RuntimeDeviceTrackerMode::Enable` Adds the provided device adapter to the list of devices to try.

`RuntimeDeviceTrackerMode::Disable` Removes the provided device adapter from the list of devices to try.

As a motivating example, let us say that we want to perform a deep copy of an array (described in Section 16.2). However, we do not want to do the copy on a CUDA device because we happen to know the data is not on that device and we do not want to spend the time to transfer the data to that device. We can use a `vtkm::cont::ScopedRuntimeDeviceTracker` to temporarily disable the CUDA device for this operation.

Example 12.3: Disabling a device with `RuntimeDeviceTracker`.

```
  vtkm::cont::ScopedRuntimeDeviceTracker tracker(
    vtkm::cont::DeviceAdapterTagCuda(),
    vtkm::cont::RuntimeDeviceTrackerMode::Disable);

  vtkm::cont::ArrayCopy(srcArray, destArray);
```

CHAPTER

THIRTEEN

TIMERS

It is often the case that you need to measure the time it takes for an operation to happen. This could be for performing measurements for algorithm study or it could be to dynamically adjust scheduling.

Performing timing in a multi-threaded environment can be tricky because operations happen asynchronously. To ensure that accurate timings can be made, VTK-m provides a `vtkm::cont::Timer` class to provide an accurate measurement of operations that happen on devices that VTK-m can use. By default, `Timer` will time operations on all possible devices.

The timer is started by calling the `Timer::Start` method. The timer can subsequently be stopped by calling `Timer::Stop`. The time elapsed between calls to `Start` and `Stop` (or the current time if `Stop` was not called) can be retrieved with a call to the `Timer::GetElapsedTime` method. Subsequently calling `Start` again will restart the timer.

Example 13.1: Using `vtkm::cont::Timer`.

```
1  vtkm::filter::field_transform::PointElevation elevationFilter;
2  elevationFilter.SetUseCoordinateSystemAsField(true);
3  elevationFilter.SetOutputFieldName("elevation");
4
5  vtkm::cont::Timer timer;
6
7  timer.Start();
8
9  vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
10
11 // This code makes sure data is pulled back to the host in a host/device
12 // architecture.
13 vtkm::cont::ArrayHandle<vtkm::Float64> outArray;
14 result.GetField("elevation").GetData().AsArrayHandle(outArray);
15 outArray.SyncControlArray();
16
17 timer.Stop();
18
19 vtkm::Float64 elapsedTime = timer.GetElapsedTime();
20
21 std::cout << "Time to run: " << elapsedTime << std::endl;
```

Common Errors

Some device require data to be copied between the host CPU and the device. In this case you might want

to measure the time to copy data back to the host. This can be done by "touching" the data on the host by getting a control portal.

The VTK-m `Timer` does its best to capture the time it takes for all parallel operations run between calls to `Start` and `Stop` to complete. It does so by synchronizing to concurrent execution on devices that might be in use.

Common Errors

Because `Timer` synchronizes with devices (essentially waiting for the device to finish executing), that can have an effect on how your program runs. Be aware that using a `Timer` can itself change the performance of your code. In particular, starting and stopping the timer many times to measure the parts of a sequence of operations can potentially make the whole operation run slower.

By default, `Timer` will synchronize with all active devices. However, if you want to measure the time for a specific device, then you can pass the device adapter tag or id to `vtkm::cont::Timer`'s constructor. You can also change the device being used by passing a device adapter tag or id to the `Timer::Reset` method. A device can also be specified through an optional argument to the `Timer::GetElapsedTime` method.

The following methods are provided by `vtkm::cont::Timer`.

`Start` Causes the `Timer` to begin timing. The elapsed time will record an interval beginning when this method is called.

`Started` Returns true if `Start` has been called. It is invalid to try to get the elapsed time if `Started` is not true.

`Stop` Causes the `Timer` to finish timing. The elapsed time will record an interval ending when this method is called. It is invalid to stop the timer if `Started` is not true.

`Stopped` Returns true if `Stop` has been called. If `Stopped` is true, then the elapsed time will no longer increase. If `Stopped` is false and `Started` is true, then the timer is still running.

`Ready` Returns true if the timer has finished the synchronization required to get the timing result from the device.

`GetElapsedTime` Returns the amount of time that has elapsed between calling `Start` and `Stop`. If `Stop` was not called, then the amount of time between calling `Start` and `GetElapsedTime` is returned. `GetElapsedTime` can optionally take a device adapter tag or id to specify for which device to return the elapsed time.

`Reset` Restores the initial state of the `Timer`. All previous recorded time is erased. `Reset` optionally takes a device adapter tag or id that specifies on which device to time and synchronize.

`GetDevice` Returns the id of the device adapter for which this timer is synchronized. If the device adapter has the same id as `vtkm::cont::DeviceAdapterTagAny` (the default), then the timer will synchronize on all devices.

`Synchronize` Synchronizes the device returned by `GetDevice` without starting or stopping the timer. This is useful for ensuring that external events are synchronized to the timer. Note that this method will always block until all pending operations on the device finish even if the `Start` or `Stop` do not actually block.

CHAPTER

FOURTEEN

IMPLICIT FUNCTIONS

VTK-m's implicit functions are objects that are constructed with values representing 3D spatial coordinates that often describe a shape. Each implicit function is typically defined by the surface formed where the value of the function is equal to 0. All `vtkm::ImplicitFunction` s implement `Value` and `Gradient` methods that describe the orientation of a provided point with respect to the `vtkm::ImplicitFunction`'s shape.

`Value` The Value method for a `vtkm::ImplicitFunction` takes a `vtkm::Vec3f` and returns a `vtkm::FloatDefault` representing the orientation of the point with respect to the `vtkm::ImplicitFunction`'s shape. Negative scalar values represent vector points inside of the `vtkm::ImplicitFunction`'s shape. Positive scalar values represent vector points outside the `vtkm::ImplicitFunction`'s shape. Zero values represent vector points that lie on the surface of the `vtkm::ImplicitFunction`

`Gradient` The Gradient method for a `vtkm::ImplicitFunction` takes a `vtkm::Vec3f` and returns a `vtkm::Vec3f` representing the pointing direction from the `vtkm::ImplicitFunction`'s shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

Implicit functions are useful when trying to clip regions from a dataset. For example, it is possible to use `vtkm::filter::ClipWithImplicitFunction` to remove a region in a provided dataset according to the shape of an implicit function. See Section9.1.3 for more information on clipping with implicit functions.

14.1 Provided Implicit Functions

VTK-m has implementations of various implicit functions provided by the following subclasses.

14.1.1 Plane

`vtkm::Plane` defines an infinite plane. The plane is defined by a pair of `vtkm::Vec3f` values that represent the origin, which is any point on the plane, and a normal, which is a vector that is tangent to the plane. These are set with the `SetOrigin` and `SetNormal` methods, respectively. Planes extend infinitely from the origin point in the direction perpendicular form the Normal. An example `vtkm::Plane` is shown in Figure 14.1.

14.1.2 Sphere

`vtkm::Sphere` defines a sphere. The `Sphere` is defined by a center location and a radius, which are set with the `SetCenter` and `SetRadius` methods, respectively. An example `vtkm::Sphere` is shown in Figure 14.2.

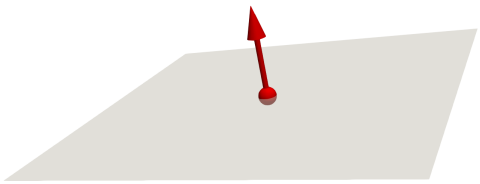


Figure 14.1: Visual Representation of an Implicit Plane. The red dot and arrow represent the origin and normal of the plane, respectively. For demonstrative purposes the plane as shown with limited area, but in actuality the plane extends infinitely.

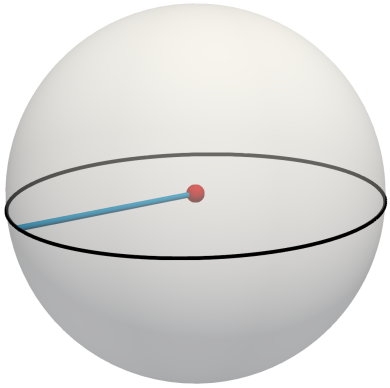


Figure 14.2: Visual Representation of an Implicit Sphere. The red dot represents the center of the sphere. The radius is the length of any line (like the blue one shown here) that extends from the center in any direction to the surface.

14.1.3 Cylinder

`vtkm::Cylinder` defines a cylinder that extends infinitely along its axis. The cylinder is defined with a center point, a direction of the center axis, and a radius, which are set with `SetCenter`, `SetAxis`, and `SetRadius`, respectively. An example `vtkm::Cylinder` is shown in Figure 14.3 with set origin, radius, and axis values.

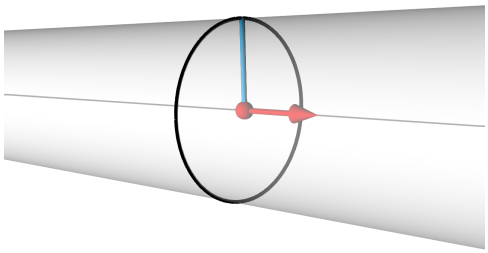


Figure 14.3: Visual Representation of an Implicit Cylinder. The red dot represents the center value, and the red arrow represents the vector that points in the direction of the axis. The radius is the length of any line (like the blue one shown here) that extends perpendicular from the axis to the surface.

14.1.4 Box

vtkm::Box defines an axis-aligned box. The box is defined with a pair of vtkm::Vec3f values that represent the minimum point coordinates and maximum point coordinates, which are set with SetMinPoint and SetMaxPoint, respectively. The Box is the shape enclosed by intersecting axis-parallel lines drawn from each point. Alternately, the Box can be specified with a vtkm::Bounds object using the SetBounds method. An example vtkm::Box is shown in Figure 14.4.

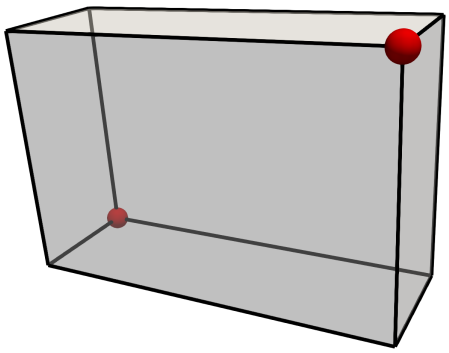


Figure 14.4: Visual Representation of an Implicit Box. The red dots represent the minimum and maximum points.

14.1.5 Frustum

vtkm::Frustum defines a hexahedral region with potentially oblique faces. A Frustum is typically used to define the tapered region of space visible in a perspective camera projection. The frustum is defined by the 6 planes that make up its 6 faces. Each plane is defined by a point and a normal vector, which are set with SetPlane and SetNormal, respectively. Parameters for all 6 planes can be set at once using the SetPlanes and SetNormals methods. Alternately, the Frustum can be defined by the 8 points at the vertices of the enclosing hexahedron using the CreateFromPoints method. The points given to CreateFromPoints must be in hex-cell order where the first four points are assumed to be a plane, and the last four points are assumed to be a plane. An example vtkm::Frustum is shown in Figure 14.5.

14.2 General Implicit Functions

It is often the case when creating code that uses an implicit function that you do not know which implicit function will be desired. For example, the vtkm::filter::ClipWithImplicitFunction filter can be used with any of the implicit functions described here (Plane, Sphere, etc.).

To handle conditions where you want to support multiple implicit functions simultaneously, VTK-m provides vtkm::ImplicitFunctionGeneral. Any of the implicit functions described in this chapter can be copied to a ImplicitFunctionGeneral, which will behave like the specified function. The following example shows shows passing a vtkm::Sphere to ClipWithImplicitFunction, which internally uses ImplicitFunctionGeneral to manage the implicit function types.

Example 14.1: Passing an implicit function to a filter.

```
1 | // Parameters needed for implicit function
```

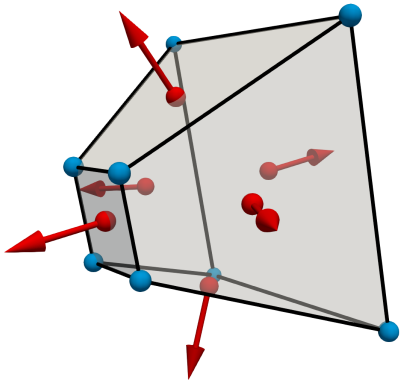


Figure 14.5: Visual Representation of an Implicit Frustum. The red dots and arrows represent the points and normals defining each enclosing plane. The blue dots represent the 8 vertices, which can also be used to define the frustum.

```
2  vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);
3
4  // Create an instance of a clip filter with this implicit function.
5  vtkm::filter::contour::ClipWithImplicitFunction clip;
6  clip.SetImplicitFunction(implicitFunction);
```

Part III

Developing Algorithms

CHAPTER

FIFTEEN

GENERAL APPROACH

VTK-m is designed to provide a *pervasive parallelism* throughout all its visualization algorithms, meaning that the algorithm is designed to operate with independent concurrency at the finest possible level throughout. VTK-m provides this pervasive parallelism by providing a programming construct called a *worklet*, which operates on a very fine granularity of data. The worklets are designed as serial components, and VTK-m handles whatever layers of concurrency are necessary, thereby removing the onus from the visualization algorithm developer. Worklet operation is then wrapped into *filters*, which provide a simplified interface to end users.

A worklet is essentially a functor or kernel designed to operate on a small element of data. (The name "worklet" means a small amount of the work. We mean small in this sense to be the amount of data, not necessarily the amount of instructions performed.) The worklet is constrained to contain a serial and stateless function. These constraints form three critical purposes. First, the constraints on the worklets allow VTK-m to schedule worklet invocations on a great many independent concurrent threads and thereby making the algorithm pervasively parallel. Second, the constraints allow VTK-m to provide thread safety. By controlling the memory access the toolkit can insure that no worklet will have any memory collisions, false sharing, or other parallel programming pitfalls. Third, the constraints encourage good programming practices. The worklet model provides a natural approach to visualization algorithm design that also has good general performance characteristics.

VTK-m allows developers to design algorithms that are run on massive amounts of threads. However, VTK-m also allows developers to interface to applications, define data, and invoke algorithms that they have written or are provided otherwise. These two modes represent significantly different operations on the data. The operating code of an algorithm in a worklet is constrained to access only a small portion of data that is provided by the framework. Conversely, code that is building the data structures needs to manage the data in its entirety, but has little reason to perform computations on any particular element.

Consequently, VTK-m is divided into two *environments* that handle each of these use cases. Each environment has its own API, and direct interaction between the environments is disallowed. The environments are as follows.

Execution Environment This is the environment in which the computational portion of algorithms are executed. The API for this environment provides work for one element with convenient access to information such as connectivity and neighborhood as needed by typical visualization algorithms. Code for the execution environment is designed to always execute on a very large number of threads.

Control Environment This is the environment that is used to interface with applications, interface with I/O devices, and schedule parallel execution of the algorithms. The associated API is designed for users that want to use VTK-m to analyze their data using provided or supplied filters. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

These dual programming environments are partially a convenience to isolate the application from the execution of the worklets and are partially a necessity to support GPU languages with host and device environments. The

control and execution environments are logically equivalent to the host and device environments, respectively, in CUDA and other associated GPU languages.

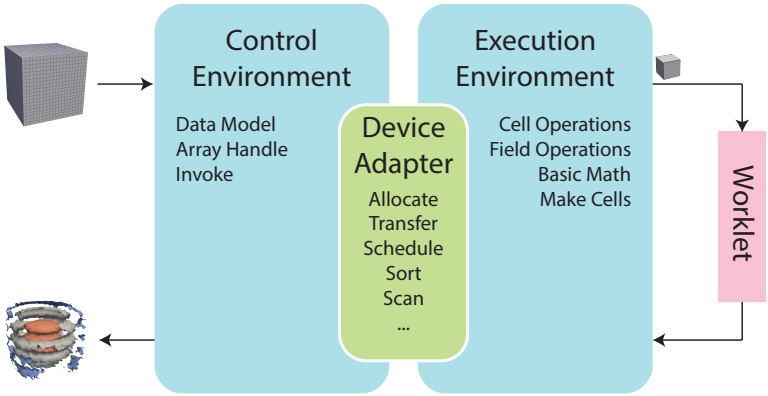


Figure 15.1: Diagram of the VTK-m framework.

Figure 15.1 displays the relationship between the control and execution environment. The typical workflow when using VTK-m is that first the control thread establishes a data set in the control environment and then invokes a parallel operation on the data using a filter. From there the data is logically divided into its constituent elements, which are sent to independent invocations of a worklet. The worklet invocations, being independent, are run on as many concurrent threads as are supported by the device. On completion the results of the worklet invocations are collected to a single data structure and a handle is returned back to the control environment.

Did you know?

Are you only planning to use filters in VTK-m that already exist? If so, then everything you work with will be in the control environment. The execution environment is only used when implementing algorithms for filters.

15.1 Package Structure

VTK-m is organized in a hierarchy of nested packages. VTK-m places definitions in *namespaces* that correspond to the package (with the exception that one package may specialize a template defined in a different namespace).

The base package is named `vtkm` . All classes within VTK-m are placed either directly in the `vtkm` package or in a package beneath it. This helps prevent name collisions between VTK-m and any other library.

As described at the beginning of this chapter, the VTK-m API is divided into two distinct environments: the control environment and the execution environment. The API for these two environments are located in the `vtkm::cont` and `vtkm::exec` packages, respectively. Items located in the base `vtkm` namespace are available in both environments.

Although it is conventional to spell out names in identifiers,[1] there is an exception to abbreviate control and execution to `cont` and `exec`, respectively. This is because it is also part of the coding convention to declare the entire namespace when using an identifier that is part of the corresponding package. The shorter names

[1] VTK-m coding conventions are outlined in the doc/CodingConventions.md file in the VTK-m source code and at https://gitlab.kitware.com/vtk/vtk-m/blob/master/docs/CodingConventions.md

make the identifiers easier to read, faster to type, and more feasible to pack lines in 80 column displays. These abbreviations are also used instead of more common abbreviations (e.g. ctrl for control) because, as part of actual English words, they are easier to type.

Further functionality in VTK-m is built on top of the base `vtkm` , `vtkm::cont` , and `vtkm::exec` packages. Support classes for building worklets, introduced in Chapter 17, are contained in the `vtkm::worklet` package. Other facilities in VTK-m are provided in their own packages such as `vtkm::io` , `vtkm::filter` , and `vtkm::rendering` . These packages are described in Part II.

VTK-m contains code that uses specialized compiler features, such as those with CUDA, or libraries, such as Intel Threading Building Blocks, that will not be available on all machines. Code for these features are encapsulated in their own packages under the `vtkm::cont` namespace: `vtkm::cont::cuda` and `vtkm::cont::tbb` .

VTK-m contains OpenGL interoperability that allows data generated with VTK-m to be efficiently transferred to OpenGL objects. This feature is encapsulated in the `vtkm::opengl` package.

Figure 15.2 provides a diagram of the VTK-m package hierarchy.

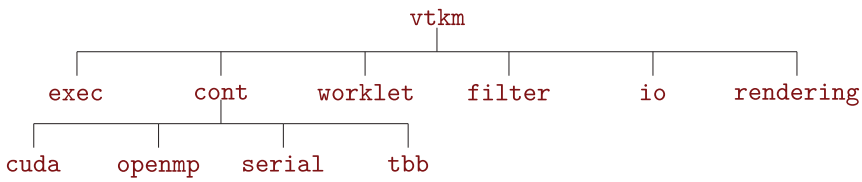


Figure 15.2: VTK-m package hierarchy.

By convention all classes will be defined in a file with the same name as the class name (with a .h extension) located in a directory corresponding to the package name. For example, the `vtkm::cont::DataSet` class is found in the vtkm/cont/DataSet.h header. There are, however, exceptions to this rule. Some smaller classes and types are grouped together for convenience. These exceptions will be noted as necessary.

Within each namespace there may also be `internal` and `detail` sub-namespaces. The `internal` namespaces contain features that are used internally and may change without notice. The `detail` namespaces contain features that are used by a particular class but must be declared outside of that class. Users should generally ignore classes in these namespaces.

15.2 Function and Method Environment Modifiers

Any function or method defined by VTK-m must come with a modifier that determines in which environments the function may be run. These modifiers are C macros that VTK-m uses to instruct the compiler for which architectures to compile each method. Most user code outside of VTK-m need not use these macros with the important exception of any classes passed to VTK-m. This occurs when defining new worklets, array storage, and device adapters.

VTK-m provides three modifier macros, `VTKM_CONT`, `VTKM_EXEC`, and `VTKM_EXEC_CONT`, which are used to declare functions and methods that can run in the control environment, execution environment, and both environments, respectively. These macros get defined by including just about any VTK-m header file, but including vtkm/Types.h will ensure they are defined.

The modifier macro is placed after the template declaration, if there is one, and before the return type for the function. Here is a simple example of a function that will square a value. Since most types you would use this function on have operators in both the control and execution environments, the function is declared for both places.

Example 15.1: Usage of an environment modifier macro on a function.

```
template<typename ValueType>
VTKM_EXEC_CONT ValueType Square(const ValueType& inValue)
{
  return inValue * inValue;
}
```

The primary function of the modifier macros is to inject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control modifiers have `__host__` in them and execution modifiers have `__device__` in them.

It is sometimes the case that a function declared as `VTKM_EXEC_CONT` has to call a method declared as `VTKM_EXEC` or `VTKM_CONT`. Generally functions should not call other functions with incompatible control/execution modifiers, but sometimes a generic `VTKM_EXEC_CONT` function calls another function determined by the template parameters, and the valid environments of this subfunction may be inconsistent. For cases like this, you can use the `VTKM_SUPPRESS_EXEC_WARNINGS` to tell the compiler to ignore the inconsistency when resolving the template. When applied to a templated function or method, `VTKM_SUPPRESS_EXEC_WARNINGS` is placed before the `template` keyword. When applied to a non-templated method in a templated class, `VTKM_SUPPRESS_EXEC_WARNINGS` is placed before the environment modifier macro.

Example 15.2: Suppressing warnings about functions from mixed environments.

```
VTKM_SUPPRESS_EXEC_WARNINGS
template<typename Functor>
VTKM_EXEC_CONT void OverlyComplicatedForLoop(Functor& functor,
                                             vtkm::Id numInterations)
{
  for (vtkm::Id index = 0; index < numInterations; index++)
  {
    functor();
  }
}
```

CHAPTER

SIXTEEN

BASIC ARRAY HANDLES

Chapter 7 describes the basic data sets used by VTK-m. This chapter dives deeper into how VTK-m represents data. Ultimately, data structures like `vtkm::cont::DataSet` can be broken down into arrays of numbers. Arrays in VTK-m are managed by a unit called an *array handle*.

An array handle, which is implemented with the `vtkm::cont::ArrayHandle` class, manages an array of data that can be accessed or manipulated by VTK-m algorithms. It is typical to construct an array handle in the control environment to pass data to an algorithm running in the execution environment. It is also typical for an algorithm running in the execution environment to populate an array handle, which can then be read back in the control environment. It is also possible for an array handle to manage data created by one VTK-m algorithm and passed to another, remaining in the execution environment the whole time and never copied to the control environment.

Did you know?

The array handle may have multiple of the array, one for the control environment and one for each device. However, depending on the device and how the array is being used, the array handle will only have one copy when possible. Copies between the environments are implicit and lazy. They are copied only when an operation needs data in an environment where the data are not.

`vtkm::cont::ArrayHandle` behaves like a shared smart pointer in that when the C++ object is copied, each copy holds a reference to the same array. These copies are reference counted so that when all copies of the `vtkm::cont::ArrayHandle` are destroyed, any allocated memory is released.

An `ArrayHandle` defines the following methods. Note, however, that the brief overview of this chapter will not cover the use of most of these methods. Further descriptions are given in later chapters that explore even further the features of `ArrayHandle`s.

`GetNumberOfValues` Returns the number of entries in the array.

`Allocate` Resizes the array to include the number of entries given. Any previously stored data might be discarded.

`ReleaseResourcesExecution` If the `ArrayHandle` is holding any data on a device (such as a GPU), that memory is released to be used elsewhere. No data is lost from this call. Any data on the released resources is copied to the control environment (the local CPU) before the memory is released.

`ReleaseResources` Releases all memory managed by this `ArrayHandle`. Any data in this memory is lost.

`SyncControlArray` Makes sure any data in the execution environment is also available in the control environment. This method is useful when timing parallel algorithms and you want to include the time to transfer data between parallel devices and their hosts.

`ReadPortal` Returns an array portal that can be used to access the data in the array handle in the control environment. The data in the array portal can only be read. Array portals are described in Section 27.1.

`WritePortal` Returns an array portal that can be used to access the data in the array handle in the control environment. The data in the array portal can be both read and written. Array portals are described in Section 27.1.

`PrepareForInput` Readies the data as input to a parallel algorithm. See Section 27.4 for more details.

`PrepareForOutput` Readies the data as output to a parallel algorithm. See Section 27.4 for more details.

`PrepareForInPlace` Readies the data as input and output to a parallel algorithm. See Section 27.4 for more details.

`DeepCopyFrom` Given an `ArrayHandle` of the same type, deep copies the data from the provided array to this array. The array will be resized as necessary.

`IsOnHost` Returns true if the data are available on the host memory (that is, available in the control environment).

`IsOnDevice` Returns true if the data are available on the specific device.

16.1 Creating Array Handles

`vtkm::cont::ArrayHandle` is templated on the type of values being stored in the array. There are multiple ways to create and populate an array handle. The default `vtkm::cont::ArrayHandle` constructor will create an empty array with nothing allocated in either the control or execution environment. This is convenient for creating arrays used as the output for algorithms.

Example 16.1: Creating an `ArrayHandle` for output data.

```
1 | vtkm::cont::ArrayHandle<vtkm::Float32> outputArray;
```

Chapter 27 describes in detail how to allocate memory and access data in an `ArrayHandle`. However, you can use the `vtkm::cont::make_ArrayHandle` function for a simplified way to create an `ArrayHandle` with data.

`make_ArrayHandle` has many forms. An easy form to use takes an initializer list and creates a basic `ArrayHandle` with it. This allows you to create short `ArrayHandle`s from literals.

Example 16.2: Creating an `ArrayHandle` from initially specified values.

```
1 | auto fibonacciArray = vtkm::cont::make_ArrayHandle({ 0, 1, 1, 2, 3, 5, 8, 13 });
```

One problem with creating an array from an initializer list like this is that it can be tricky to specify the exact value type of the `ArrayHandle`. The value type of the `ArrayHandle` will be the same types as the literals in the initializer list, but that might not match the type you actually need. This is particularly true for types like `vtkm::Id` and `vtkm::FloatDefault`, which can change depending on compile options. To specify the exact value type to use, give that type as a template argument to the `vtkm::cont::make_ArrayHandle` function.

Example 16.3: Creating a typed `ArrayHandle` from initially specified values.

```
1 | vtkm::cont::ArrayHandle<vtkm::FloatDefault> inputArray =
2 |   vtkm::cont::make_ArrayHandle<vtkm::FloatDefault>({ 1.4142f, 2.7183f, 3.1416f });
```

Constructing an ArrayHandle that points to a provided C array is also straightforward. To do this, call make_ArrayHandle with the array pointer, the number of values in the C array, and a vtkm::CopyFlag. This last argument can be either CopyFlag::On to copy the array or CopyFlag::Off to share the provided buffer.

Example 16.4: Creating an ArrayHandle that points to a provided C array.

```
vtkm::Float32 dataBuffer[50];
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
  vtkm::cont::make_ArrayHandle(dataBuffer, 50, vtkm::CopyFlag::On);
```

Likewise, you can use make_ArrayHandle to transfer data from a `std::vector` to an ArrayHandle. This form of make_ArrayHandle takes the `std::vector` as the first argument and a vtkm::CopyFlag as the second argument.

Example 16.5: Creating an ArrayHandle that points to a provided `std::vector`.

```
std::vector<vtkm::Float32> dataBuffer;
// Populate dataBuffer with meaningful data. Perhaps read data from a file.

vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
  vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::On);
```

As hinted at earlier, it is possible to send CopyFlag::Off to make_ArrayHandle to wrap an ArrayHandle around an existing C array or `std::vector`. Doing so allows you to send the data to the ArrayHandle without copying it. It also provides a mechanism for VTK-m to write directly into your array. However, *be aware* that if you change or delete the data provided, the internal state of ArrayHandle becomes invalid and undefined behavior can ensue. A common manifestation of this error happens when a `std::vector` goes out of scope. This subtle interaction will cause the vtkm::cont::ArrayHandle to point to an unallocated portion of the memory heap. The following example provides an erroneous use of ArrayHandle and some ways to fix it.

Example 16.6: Invalidating an ArrayHandle by letting the source `std::vector` leave scope.

```
VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
{
  std::vector<vtkm::Float32> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
    vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::Off);

  return inputArray;
  // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
  // memory. However, inputArray has a pointer to that memory, which becomes an
  // invalid pointer in the returned object. Bad things will happen when the
  // ArrayHandle is used.
}

VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad1()
{
  std::vector<vtkm::Float32> dataBuffer;
  // Populate dataBuffer with meaningful data. Perhaps read data from a file.

  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
    vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::On);

  return inputArray;
  // This is safe.
}

VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad2()
{
  std::vector<vtkm::Float32> dataBuffer;
```

```
31   // Populate dataBuffer with meaningful data. Perhaps read data from a file.
32
33   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
34     vtkm::cont::make_ArrayHandleMove(std::move(dataBuffer));
35
36   return inputArray;
37   // This is safe.
38 }
```

An easy way around the problem of having an `ArrayHandle`'s data going out of scope is to copy the data into the `ArrayHandle`. Simply make the `vtkm::CopyFlag` argument be `On` to copy the data. This solution is shown on line 22 of Example 16.6.

What if you have a `std::vector` that you want to pass to an `ArrayHandle` and then want to only use in the `ArrayHandle`? In this case, it is wasteful to have to copy the data, but you also do not want to be responsible for keeping the `std::vector` in scope. To handle this, there is a special `vtkm::cont::make_ArrayHandleMove` that will move the memory out of the `std::vector` and into the `ArrayHandle`. `make_ArrayHandleMove` takes an "rvalue" version of a `std::vector`. To create an "rvalue", use the `std::move` function provided by C++. Once `make_ArrayHandleMove` is called, the provided `std::vector` becomes invalid and any further access to it is undefined. This solution is shown on line 34 of Example 16.6.

16.2 Deep Array Copies

As stated previously, an `ArrayHandle` object behaves as a smart pointer that copies references to the data without copying the data itself. This is clearly faster and more memory efficient than making copies of the data itself and usually the behavior desired. However, it is sometimes the case that you need to make a separate copy of the data.

The easiest way to copy `ArrayHandle`s is to use the `ArrayHandle::DeepCopyFrom` method.

Example 16.7: Deep copy `ArrayHandle`s of the same type.

```
1   destArray.DeepCopyFrom(srcArray);
```

However, the `DeepCopyFrom` method only works if the two `ArrayHandle`s are the exact same type. To simplify copying the data between `ArrayHandle`s of different types, VTK-m comes with the `vtkm::cont::ArrayCopy` convenience function defined in vtkm/cont/ArrayCopy.h. `ArrayCopy` takes the array to copy from (the source) as its first argument and the array to copy to (the destination) as its second argument. The destination array will be properly reallocated to the correct size.

Example 16.8: Using `ArrayCopy`.

```
1   vtkm::cont::ArrayCopy(srcArray, destArray);
```

16.3 The Hidden Second Template Parameter

We have already seen that `vtkm::cont::ArrayHandle` is a templated class with the template parameter indicating the type of values stored in the array. However, `ArrayHandle` has a second hidden parameter that indicates the *storage* of the array. We have so far been able to ignore this second template parameter because VTK-m will assign a default storage for us that will store the data in a basic array.

Example 16.9: Declaration of the `vtkm::cont::ArrayHandle` templated class.

```
1 template<typename T, typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
2 class ArrayHandle;
```

Changing the storage of an `ArrayHandle` lets us do many weird and wonderful things. We will explore these options in later chapters, but for now we can ignore this second storage template parameter. However, there are a couple of things to note concerning the storage.

First, if the compiler gives an error concerning your use of `ArrayHandle`, the compiler will report the `ArrayHandle` type with not one but two template parameters. A second template parameter of `vtkm::cont::StorageTagBasic` can be ignored.

Second, if you write a function, method, or class that is templated based on an `ArrayHandle` type, it is good practice to accept `ArrayHandle`s with non-default storage types. There are two ways to do this. The first way is to template both the value type and the storage type.

Example 16.10: Templating a function on an `ArrayHandle`'s parameters

```
1 template<typename T, typename Storage>
2 void Foo(const vtkm::cont::ArrayHandle<T, Storage>& array)
3 {
```

The second way is to template the whole array type rather than the sub types. If you create a template where you expect one of the parameters to be an `ArrayHandle`, you should use the `VTKM_IS_ARRAY_HANDLE` macro to verify that the type is indeed an `ArrayHandle`.

Example 16.11: A template parameter that should be an `ArrayHandle`.

```
1 template<typename ArrayType>
2 void Bar(const ArrayType& array)
3 {
4   VTKM_IS_ARRAY_HANDLE(ArrayType);
```

16.4 Mutability

One subtle feature of `ArrayHandle` is that the class is, in principle, a pointer to an array pointer. This means that the data in an `ArrayHandle` is always mutable even if the class is declared `const`. You can change the contents of "constant" arrays via methods like `WritePortal` and `PrepareForOutput`. It is even possible to change the underlying array allocation with methods like `Allocate` and `ReleaseResources`. The upshot is that you can (sometimes) pass output arrays as constant `ArrayHandle` references.

So if a constant `ArrayHandle` can have its contents modified, what is the difference between a constant reference and a non-constant reference? The difference is that the constant reference can change the array's content, but not the array itself. Basically, this means that you cannot perform shallow copies into constant `ArrayHandle`s. This can be a pretty big limitation, and many of VTK-m's internal device algorithms still require non-constant references for outputs.

CHAPTER

SEVENTEEN

SIMPLE WORKLETS

The simplest way to implement an algorithm in VTK-m is to create a *worklet*. A worklet is fundamentally a functor that operates on an element of data. Thus, it is a `class` or `struct` that has an overloaded parenthesis operator (which must be declared `const` for thread safety). However, worklets are also embedded with a significant amount of metadata on how the data should be managed and how the execution should be structured.

Example 17.1: A simple worklet.

```
1  struct PoundsPerSquareInchToNewtonsPerSquareMeterWorklet
2    : vtkm::worklet::WorkletMapField
3  {
4    using ControlSignature = void(FieldIn psi, FieldOut nsm);
5    using ExecutionSignature = void(_1, _2);
6    using InputDomain = _1;
7
8    template<typename T>
9    VTKM_EXEC void operator()(const T& psi, T& nsm) const
10   {
11     // 1 psi = 6894.76 N/m^2
12     nsm = T(6894.76f) * psi;
13   }
14 };
```

As can be seen in Example 17.1, a worklet is created by implementing a `class` or `struct` with the following features.

1. The class must publicly inherit from a base worklet class that specifies the type of operation being performed (line 1).
2. The class must contain a functional type named `ControlSignature` (line 4), which specifies what arguments are expected when invoking the class in the control environment.
3. The class must contain a functional type named `ExecutionSignature` (line 5), which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.
4. The class specifies an `InputDomain` (line 6), which identifies which input parameter defines the input domain of the data.
5. The class must contain an implementation of the parenthesis operator, which is the method that is executed in the execution environment (lines 8–13). The parenthesis operator must be declared `const`.

17.1 Control Signature

The control signature of a worklet is a functional type named `ControlSignature`. The function prototype matches what data are provided when the worklet is invoked (as described in Section 17.5).

Example 17.2: A `ControlSignature`.

```
1 | using ControlSignature = void(FieldIn psi, FieldOut nsm);
```

Did you know?

If the code in Example 17.2 looks strange, you may be unfamiliar with function types. In C++, functions have types just as variables and classes do. A function with a prototype like

```
void functionName(int arg1, float arg2);
```

has the type `void(int,float)`. *VTK-m uses function types like this as a* signature *that defines the structure of a function call.*

The return type of the function prototype is always `void`. The parameters of the function prototype are *tags* that identify the type of data that is expected to be passed to invoke. `ControlSignature` tags are defined by the worklet type and the various tags are documented more fully in Chapter 21. In the case of Example 17.2, the two tags `FieldIn` and `FieldOut` represent input and output data, respectively.

By convention, `ControlSignature` tag names start with the base concept (e.g. `Field` or `Topology`) followed by the domain (e.g. `Point` or `Cell`) followed by `In` or `Out`. For example, `FieldPointIn` would specify values for a field on the points of a mesh that are used as input (read only). Although they should be there in most cases, some tag names might leave out the domain or in/out parts if they are obvious or ambiguous.

17.2 Execution Signature

Like the control signature, the execution signature of a worklet is a functional type named `ExecutionSignature`. The function prototype must match the parenthesis operator (described in Section 17.4) in terms of arity and argument semantics.

Example 17.3: An `ExecutionSignature`.

```
1 | using ExecutionSignature = void(_1, _2);
```

The arguments of the `ExecutionSignature`'s function prototype are tags that define where the data come from. The most common tags are an underscore followed by a number, such as `_1`, `_2`, etc. These numbers refer back to the corresponding argument in the `ControlSignature`. For example, `_1` means data from the first control signature argument, `_2` means data from the second control signature argument, etc.

Unlike the control signature, the execution signature optionally can declare a return type if the parenthesis operator returns a value. If this is the case, the return value should be one of the numeric tags (i.e. `_1`, `_2`, etc.) to refer to one of the data structures of the control signature. If the parenthesis operator does not return a value, then `ExecutionSignature` should declare the return type as `void`.

In addition to the numeric tags, there are other execution signature tags to represent other types of data. For example, the `WorkIndex` tag identifies the instance of the worklet invocation. Each call to the worklet function

will have a unique `WorkIndex`. Other such tags exist and are described in the following section on worklet types where appropriate.

17.3 Input Domain

All worklets represent data parallel operations that are executed over independent elements in some domain. The type of domain is inherent from the worklet type, but the size of the domain is dependent on the data being operated on.

A worklet identifies the argument specifying the domain with a type alias named `InputDomain`. The `InputDomain` must be aliased to one of the execution signature numeric tags (i.e. `_1`, `_2`, etc.). By default, the `InputDomain` points to the first argument, but a worklet can override that to point to any argument.

Example 17.4: An `InputDomain` declaration.

```
1 | using InputDomain = _1;
```

Different types of worklets can have different types of domain. For example a simple field map worklet has a `FieldIn` argument as its input domain, and the size of the input domain is taken from the size of the associated field array. Likewise, a worklet that maps topology has a `CellSetIn` argument as its input domain, and the size of the input domain is taken from the cell set.

Specifying the `InputDomain` is optional. If it is not specified, the first argument is assumed to be the input domain.

17.4 Worklet Operator

A worklet is fundamentally a functor that operates on an element of data. Thus, the algorithm that the worklet represents is contained in or called from the parenthesis operator method.

Example 17.5: An overloaded parenthesis operator of a worklet.

```
1 | template<typename T>
2 | VTKM_EXEC void operator()(const T& psi, T& nsm) const
3 | {
```

There are some constraints on the parenthesis operator. First, it must have the same arity as the `ExecutionSignature`, and the types of the parameters and return must be compatible. Second, because it runs in the execution environment, it must be declared with the `VTKM_EXEC` (or `VTKM_EXEC_CONT`) modifier. Third, the method must be declared `const` to help preserve thread safety.

17.5 Invoking a Worklet

Previously in this chapter we discussed creating a simple worklet. In this section we describe how to run the worklet in parallel.

A worklet is run using the `vtkm::cont::Invoker` class.

Example 17.6: Invoking a worklet.

```
1 | vtkm::cont::ArrayHandle<vtkm::FloatDefault> psiArray;
2 | // Fill psiArray with values...
3 |
```

```
4  vtkm::cont::Invoker invoke;
5
6  vtkm::cont::ArrayHandle<vtkm::FloatDefault> nsmArray;
7  invoke(PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, psiArray, nsmArray);
```

Using an `Invoker` is simple. First, an `Invoker` can be simply constructed with no arguments (line 4). Next, the `Invoker` is called as if it were a function (line 7).

The first argument to the invoke is always an instance of the worklet. The remaining arguments are data that are passed (indirectly) to the worklet. Each of these arguments (after the worklet) match a corresponding argument listed in the `ControlSignature`. So in the invocation on Example 17.6, line 7, the second and third arguments correspond the the two `ControlSignature` arguments given in Example 17.2. `psiArray` corresponds to the `FieldIn` argument and `nmsArray` corresponds to the `FieldOut` argument.

17.6 Preview of More Complex Worklets

This chapter demonstrates the creation of a worklet that performs a very simple math operation in parallel. However, we have just scratched the surface of the kinds of algorithms that can be expressed with VTK-m worklets. There are many more execution patterns and data handling constructs. The following example gives a preview of some of the more advanced features of worklets.

Example 17.7: A more complex worklet.

```
1  struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
4    using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
5    using InputDomain = _1;
6
7    using ScatterType = vtkm::worklet::ScatterCounting;
8
9    template<typename CellShapeTag,
10            typename PointIndexVecType,
11            typename EdgeIndexVecType>
12   VTKM_EXEC void operator()(CellShapeTag cellShape,
13                             const PointIndexVecType& globalPointIndicesForCell,
14                             vtkm::IdComponent edgeIndex,
15                             EdgeIndexVecType& edgeIndices) const
16   {
```

We will discuss the many features available in the worklet framework throughout Part IV.

CHAPTER

EIGHTEEN

BASIC FILTER IMPLEMENTATION

Chapter 17 introduced the concept of a worklet and demonstrated how to create and run one to execute an algorithm on a device. Although worklets provide a powerful mechanism for designing heavily threaded visualization algorithms, invoking them requires quite a bit of knowledge of the workings of VTK-m. Instead, most users execute algorithms in VTK-m using filters. Thus, to expose algorithms implemented with worklets to general users, we need to implement a filter to encapsulate the worklets. In this chapter we will create a filter that encapsulates the worklet algorithm presented in Chapter 17, which converted the units of a pressure field from pounds per square inch (psi) to Newtons per square meter (N/m^2).

Filters in VTK-m are implemented by deriving one of the filter base classes provided in `vtkm::filter` . There are multiple base filter classes that we can choose from. These different classes are documented later in Chapter 22. For this example we will derive the `vtkm::filter::Filter` base class.

The following example shows the declaration of our pressure unit conversion filter. By convention, this declaration would be placed in a header file with a .h extension. VTK-m filters are divided into libraries. In this example, we are assuming this filter is being compiled in a library named vtkm_filter_unit_conversion. By convention, the source files would be placed in a directory named vtkm/filter/unit_conversion.

Example 18.1: Header declaration for a simple filter.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5  namespace unit_conversion
6  {
7
8  class VTKM_FILTER_UNIT_CONVERSION_EXPORT
9    PoundsPerSquareInchToNewtonsPerSquareMeterFilter : public vtkm::filter::Filter
10 {
11 public:
12   VTKM_CONT PoundsPerSquareInchToNewtonsPerSquareMeterFilter();
13
14   VTKM_CONT vtkm::cont::DataSet DoExecute(
15     const vtkm::cont::DataSet& inDataSet) override;
16 };
17
18 }
19 }
20 } // namespace vtkm::filter::unit_conversion
```

It is typical for a filter to have a constructor to set up its initial state. A filter will also override the `DoExecute` method. The `DoExecute` method takes a `vtkm::cont::DataSet` as input and likewise returns a `DataSet` containing the results of the filter operation.

Note that the declaration of the PoundsPerSquareInchToNewtonsPerSquareMeterFilter contains the export macro VTKM_FILTER_UNIT_CONVERSION_EXPORT. This is a macro generated by CMake to handle the appropriate modifies for exporting a class from a library. Remember that this code is to be placed in a library named vtkm_filter_unit_conversion. For this library, CMake creates a header file named vtkm_filter_unit_conversion.h that declares macros like VTKM_FILTER_UNIT_CONVERSION_EXPORT.

Did you know?

A filter can also override the DoExecutePartitions*, which operates on a* vtkm::cont::PartitionedDataSet*. If* DoExecutePartitions *is not overridden, then the filter will call* DoExecute *on each of the partitions and build a new* PartitionedDataSet *with the outputs.*

Once the filter class is declared in the .h file, the implementation filter is by convention given in a separate .cxx file. Given the definition of our filter in Example 18.1, we will need to provide the implementation for the constructor and the DoExecute method. The constructor is quite simple. It initializes the name of the output field name, which is managed by the superclass.

Example 18.2: Constructor for a simple filter.

```
1  VTKM_CONT PoundsPerSquareInchToNewtonsPerSquareMeterFilter::
2    PoundsPerSquareInchToNewtonsPerSquareMeterFilter()
3  {
4    this->SetOutputFieldName("");
5  }
```

In this case, we are setting the output field name to the empty string. This is not to mean that the default name of the output field should be the empty string, which is not a good idea. Rather, as we will see later, we will use the empty string to flag an output name that should be derived from the input name.

The meat of the filter implementation is located in the DoExecute method.

Example 18.3: Implementation of DoExecute for a simple filter.

```
1  VTKM_CONT vtkm::cont::DataSet
2  PoundsPerSquareInchToNewtonsPerSquareMeterFilter::DoExecute(
3    const vtkm::cont::DataSet& inDataSet)
4  {
5    vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
6
7    vtkm::cont::UnknownArrayHandle outArray;
8
9    auto resolveType = [&](const auto& inputArray)
10   {
11     // use std::decay to remove const ref from the decltype of concrete.
12     using T = typename std::decay_t<decltype(inputArray)>::ValueType;
13     vtkm::cont::ArrayHandle<T> result;
14     this->Invoke(
15       PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, inputArray, result);
16     outArray = result;
17   };
18
19   this->CastAndCallScalarField(inField, resolveType);
20
21   std::string outFieldName = this->GetOutputFieldName();
22   if (outFieldName == "")
23   {
24     outFieldName = inField.GetName() + "_N/m^2";
25   }
26
```

```
27   return this->CreateResultField(
28     inDataSet, outFieldName, inField.GetAssociation(), outArray);
29 }
```

The single argument to `DoExecute` is a `vtkm::cont::DataSet` containing the data to operate on, and `DoExecute` returns a derived `DataSet`. The filter must pull the appropriate information out of the input `DataSet` to operate on. This simple algorithm just operates on a single field array of the data. The `Filter` base class provides several methods to allow filter users to select the active field to operate on. The filter implementation can get the appropriate field to operate on using the `GetFieldFromDataSet` method provided by `Filter` as shown in Example 18.3 line 5.

One of the challenges with writing filters is determining the actual types the algorithm is operating on. The `Field` object pulled from the input `DataSet` contains an `ArrayHandle` (see Chapter 16), but you do not know what the template parameters of the `ArrayHandle` are. There are numerous ways to extract an array of an unknown type out of an `ArrayHandle` (many of which will be explored later in Chapter 33), but the `Filter` contains some convenience functions to simplify this.

In particular, this filter operates specifically on scalar fields. For this purpose, `Filter` provides the `CastAndCallScalarField` helper method. The first argument to `CastAndCallScalarField` is the field containing the data to operate on. The second argument is a functor that will operate on the array once it is identified. `CastAndCallScalarField` will pull a `ArrayHandle` out of the field and call the provided functor with that object. `CastAndCallScalarField` is called in Example 18.3 on line 19.

Did you know?

If your filter requires a field containing `vtkm::Vec` *s of a particular size (e.g. 3), you can use the convenience method* `CastAndCallVecField`. `CastAndCallVecField` *works similarly to* `CastAndCallScalarField` *except that it takes a template parameter specifying the size of the* `Vec`. *For example,* `CastAndCallVecField<3>(inField, functor);`.

As previously stated, one of the arguments to `CastAndCallScalarField` is a functor that contains the routine to call with the found `ArrayHandle`. A functor can be created as its own `class` or `struct`, but a more convenient method is to use a C++ lambda. A lambda is an unnamed function defined inline with the code. The lambda in Example 18.3 starts on line 9. Apart from being more convenient than creating a named class, lambda functions offer another important feature. Lambda functions can "capture" variables in the current scope. They can therefore access things like local variables and the `this` reference to the method's class (even accessing private members).

The callback to the lambda function in Example 18.3 first creates an output `ArrayHandle` of a compatible type (line 13), then invokes the worklet that computes the derived field (line 14), and finally captures the resulting array. Note that the `Filter` base class provides an `Invoke` member that can be used to invoke the worklet. (See Section 17.5 for information on invoking a worklet.) Recall that the worklet created in Chapter 17 takes two parameters: an input array and an output array, which are shown in this invocation.

With the output data created, the filter has to build the output structure to return. All implementations of `DoExecute` must return a `vtkm::cont::DataSet`, and for a simple field filter like this we want to return the same `DataSet` as the input with the output field added. The output field needs a name, and we get the appropriate name from the superclass (line 21). However, we would like a special case where if the user does not specify an output field name we construct one based on the input field name. Recall from Example 18.2 that by default we set the output field name to the empty string. Thus, our filter checks for this empty string, and if it is encountered, it builds a field name by appending "_N/M^2" to it.

Finally, our filter constructs the output `DataSet` using one of the `CreateResult` member functions (line 27).

In this particular case, the filter uses `Filter::CreateResultField`, which constructs a `DataSet` with the same structure as the input and adds the computed filter.

Common Errors

The `Filter::CreateResult` *methods do more than just construct a new* `vtkm::cont::DataSet`*. They also set up the structure of the data and pass fields as specified by the state of the filter object. Thus, implementations of* `DoExecute` *should always return* `DataSet`*s that were created with* `Filter::CreateResult` *or a similarly named method in the base filter classes.*

This chapter has just provided a brief introduction to creating filters. There are several more filter superclasses to help express algorithms of different types. After some more worklet concepts to implement more complex algorithms are introduced in Part IV, we will see a more complete documentation of the types of filters in Chapter 22.

Part IV

Advanced Development

CHAPTER

NINETEEN

ADVANCED TYPES

Chapter 4 introduced some of the base data types defined for use in VTK-m. However, for simplicity Chapter 4 just briefly touched the high-level concepts of these types. In this chapter we dive into much greater depth and introduce several more types.

19.1 Single Number Types

As described in Chapter 4, VTK-m provides aliases for all the base C types to ensure the representation matches the variable use. When a specific type width is not required, then the most common types to use are `vtkm::FloatDefault` for floating-point numbers, `vtkm::Id` for array and similar indices, and `vtkm::IdComponent` for shorter-width vector indices.

If a specific type width is desired, then one of the following is used to clearly declare the type and width.

bytes	floating point	signed integer	unsigned integer
1		`vtkm::Int8`	`vtkm::UInt8`
2		`vtkm::Int16`	`vtkm::UInt16`
4	`vtkm::Float32`	`vtkm::Int32`	`vtkm::UInt32`
8	`vtkm::Float64`	`vtkm::Int64`	`vtkm::UInt64`

These VTK-m–defined types should be preferred over basic C types like `int` or `float`.

19.2 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides the `vtkm::Vec <T,Size>` templated type, which is essentially a fixed length array of a given type.

The default constructor of `vtkm::Vec` objects leaves the values uninitialized. All vectors have a constructor with one argument that is used to initialize all components. All `vtkm::Vec` objects also have a constructor that allows you to set the individual components (one per argument). All `vtkm::Vec` objects with a size that is greater than 4 are constructed at run time and support an arbitrary number of initial values. Likewise, there is a `vtkm::make_Vec` convenience function that builds initialized vector types with an arbitrary number of components. Once created, you can use the bracket operator to get and set component values with the same syntax as an array.

Example 19.1: Creating vector types.

```
vtkm::Vec3f_32 A{ 1 };                          // A is (1, 1, 1)
A[1] = 2;                                       // A is now (1, 2, 1)
vtkm::Vec3f_32 B{ 1, 2, 3 };                    // B is (1, 2, 3)
vtkm::Vec3f_32 C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
// Longer Vecs specified with template.
vtkm::Vec<vtkm::Float32, 5> D{ 1 };                  // D is (1, 1, 1, 1, 1)
vtkm::Vec<vtkm::Float32, 5> E{ 1, 2, 3, 4, 5 };      // E is (1, 2, 3, 4, 5)
vtkm::Vec<vtkm::Float32, 5> F = { 6, 7, 8, 9, 10 }; // F is (6, 7, 8, 9, 10)
auto G = vtkm::make_Vec(1, 3, 5, 7, 9);              // G is (1, 3, 5, 7, 9)
```

The types `vtkm::Id2`, `vtkm::Id3`, and `vtkm::Id4` are type aliases of `vtkm::Vec <vtkm::Id,2>`, `vtkm::Vec <vtkm::Id,3>`, and `vtkm::Vec <vtkm::Id,4>`. These are used to index arrays of 2, 3, and 4 dimensions, which is common. Likewise, `vtkm::IdComponent2`, `vtkm::IdComponent4`, and `vtkm::IdComponent4` are type aliases of `vtkm::Vec <vtkm::IdComponent,2>`, `vtkm::Vec <vtkm::IdComponent,3>`, and `vtkm::Vec <vtkm::IdComponent,4>`.

Because declaring `vtkm::Vec <T,Size>` with all of its template parameters can be cumbersome, VTK-m provides easy to use aliases for small vectors of base types. As introduced in Section 4.3, the following type aliases are available.

bytes	size	floating point	signed integer	unsigned integer
default	2	`vtkm::Vec2f`	`vtkm::Vec2i`	`vtkm::Vec2ui`
	3	`vtkm::Vec3f`	`vtkm::Vec3i`	`vtkm::Vec3ui`
	4	`vtkm::Vec4f`	`vtkm::Vec4i`	`vtkm::Vec4ui`
1	2		`vtkm::Vec2i_8`	`vtkm::Vec2ui_8`
	3		`vtkm::Vec3i_8`	`vtkm::Vec3ui_8`
	4		`vtkm::Vec4i_8`	`vtkm::Vec4ui_8`
2	2		`vtkm::Vec2i_16`	`vtkm::Vec2ui_16`
	3		`vtkm::Vec3i_16`	`vtkm::Vec3ui_16`
	4		`vtkm::Vec4i_16`	`vtkm::Vec4ui_16`
4	2	`vtkm::Vec2f_32`	`vtkm::Vec2i_32`	`vtkm::Vec2ui_32`
	3	`vtkm::Vec3f_32`	`vtkm::Vec3i_32`	`vtkm::Vec3ui_32`
	4	`vtkm::Vec4f_32`	`vtkm::Vec4i_32`	`vtkm::Vec4ui_32`
8	2	`vtkm::Vec2f_64`	`vtkm::Vec2i_64`	`vtkm::Vec2ui_64`
	3	`vtkm::Vec3f_64`	`vtkm::Vec3i_64`	`vtkm::Vec3ui_64`
	4	`vtkm::Vec4f_64`	`vtkm::Vec4i_64`	`vtkm::Vec4ui_64`

`vtkm::Vec` supports component-wise arithmetic using the operators for plus (`+`), minus (`-`), multiply (`*`), and divide (`/`). It also supports scalar to vector multiplication with the multiply operator. The comparison operators equal (`==`) is true if every pair of corresponding components are true and not equal (`!=`) is true otherwise. A special `vtkm::Dot` function is overloaded to provide a dot product for every type of vector.

Example 19.2: Vector operations.

```
vtkm::Vec3f_32 A{ 1, 2, 3 };
vtkm::Vec3f_32 B{ 4, 5, 6.5 };
vtkm::Vec3f_32 C = A + B;                 // C is (5, 7, 9.5)
vtkm::Vec3f_32 D = 2.0f * C;              // D is (10, 14, 19)
vtkm::Float32 s = vtkm::Dot(A, B);        // s is 33.5
bool b1 = (A == B);                       // b1 is false
bool b2 = (A == vtkm::make_Vec(1, 2, 3)); // b2 is true

vtkm::Vec<vtkm::Float32, 5> E{ 1, 2.5, 3, 4, 5 };    // E is (1, 2, 3, 4, 5)
vtkm::Vec<vtkm::Float32, 5> F{ 6, 7, 8.5, 9, 10.5 }; // F is (6, 7, 8, 9, 10)
vtkm::Vec<vtkm::Float32, 5> G = E + F; // G is (7, 9.5, 11.5, 13, 15.5)
bool b3 = (E == F);                    // b3 is false
bool b4 = (G == vtkm::make_Vec(7.f, 9.5f, 11.5f, 13.f, 15.5f)); // b4 is true
```

These operators, of course, only work if they are also defined for the component type of the vtkm::Vec. For example, the multiply operator will work fine on objects of type vtkm::Vec <char,3>, but the multiply operator will not work on objects of type vtkm::Vec <std::string,3> because you cannot multiply objects of type std::string.

In addition to generalizing vector operations and making arbitrarily long vectors, vtkm::Vec can be repurposed for creating any sequence of homogeneous objects. Here is a simple example of using vtkm::Vec to hold the state of a polygon.

Example 19.3: Repurposing a vtkm::Vec.

```
1  vtkm::Vec<vtkm::Vec2f_32, 3> equilateralTriangle = { { 0.0f, 0.0f },
2                                                       { 1.0f, 0.0f },
3                                                       { 0.5f, 0.8660254f } };
```

The vtkm::Vec class provides a convenient structure for holding and passing small vectors of data. However, there are times when using Vec is inconvenient or inappropriate. For example, the size of vtkm::Vec must be known at compile time, but there may be need for a vector whose size is unknown until compile time. Also, the data populating a vtkm::Vec might come from a source that makes it inconvenient or less efficient to construct a vtkm::Vec. For this reason, VTK-m also provides several Vec-*like* objects that behave much like vtkm::Vec but are a different class. These Vec-like objects have the same interface as vtkm::Vec except that the NUM_COMPONENTS constant is not available on those that are sized at run time. Vec-like objects also come with a CopyInto method that will take their contents and copy them into a standard Vec class. (The standard Vec class also has a CopyInto method for consistency.)

The first Vec-like object is vtkm::VecC, which exposes a C-type array as a Vec. The constructor for vtkm::VecC takes a C array and a size of that array. There is also a constant version of VecC named vtkm::VecCConst, which takes a constant array and cannot be mutated. The vtkm/Types.h header defines both VecC and VecCConst as well as multiple versions of vtkm::make_VecC to easily convert a C array to either a VecC or VecCConst.

The following example demonstrates converting values from a constant table into a vtkm::VecCConst for further consumption. The table and associated methods define how 8 points come together to form a hexahedron.

Example 19.4: Using vtkm::VecCConst with a constant array.

```
1  VTKM_EXEC
2  vtkm::VecCConst<vtkm::IdComponent> HexagonIndexToIJK(vtkm::IdComponent index)
3  {
4    static const vtkm::IdComponent HexagonIndexToIJKTable[8][3] = {
5      { 0, 0, 0 }, { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 0 },
6      { 0, 0, 1 }, { 1, 0, 1 }, { 1, 1, 1 }, { 0, 1, 1 }
7    };
8
9    return vtkm::make_VecC(HexagonIndexToIJKTable[index], 3);
10 }
11
12 VTKM_EXEC
13 vtkm::IdComponent HexagonIJKToIndex(vtkm::VecCConst<vtkm::IdComponent> ijk)
14 {
15   static const vtkm::IdComponent HexagonIJKToIndexTable[2][2][2] = {
16     {
17       // i=0
18       { 0, 4 }, // j=0
19       { 3, 7 }, // j=1
20     },
21     {
22       // i=1
23       { 1, 5 }, // j=0
24       { 2, 6 }, // j=1
25     }
26   };
```

```
27
28    return HexagonIJKToIndexTable[ijk[0]][ijk[1]][ijk[2]];
29  }
```

Common Errors

The `vtkm::VecC` *and* `vtkm::VecCConst` *classes only hold a pointer to a buffer that contains the data. They do not manage the memory holding the data. Thus, if the pointer given to* `vtkm::VecC` *or* `vtkm::VecCConst` *becomes invalid, then using the object becomes invalid. Make sure that the scope of the* `vtkm::VecC` *or* `vtkm::VecCConst` *does not outlive the scope of the data it points to.*

The next `Vec`-like object is `vtkm::VecVariable`, which provides a `Vec`-like object that can be resized at run time to a maximum value. Unlike `VecC`, `VecVariable` holds its own memory, which makes it a bit safer to use. But also unlike `VecC`, you must define the maximum size of `VecVariable` at compile time. Thus, `VecVariable` is really only appropriate to use when there is a predetermined limit to the vector size that is fairly small.

The following example uses a `vtkm::VecVariable` to store the trace of edges within a hexahedron. This example uses the methods defined in Example 19.4.

Example 19.5: Using `vtkm::VecVariable`.

```
1  vtkm::VecVariable<vtkm::IdComponent, 4> HexagonShortestPath(
2    vtkm::IdComponent startPoint,
3    vtkm::IdComponent endPoint)
4  {
5    vtkm::VecCConst<vtkm::IdComponent> startIJK = HexagonIndexToIJK(startPoint);
6    vtkm::VecCConst<vtkm::IdComponent> endIJK = HexagonIndexToIJK(endPoint);
7
8    vtkm::IdComponent3 currentIJK;
9    startIJK.CopyInto(currentIJK);
10
11   vtkm::VecVariable<vtkm::IdComponent, 4> path;
12   path.Append(startPoint);
13   for (vtkm::IdComponent dimension = 0; dimension < 3; dimension++)
14   {
15     if (currentIJK[dimension] != endIJK[dimension])
16     {
17       currentIJK[dimension] = endIJK[dimension];
18       path.Append(HexagonIJKToIndex(currentIJK));
19     }
20   }
21
22   return path;
23 }
```

VTK-m provides further examples of `Vec`-like objects as well. For example, the `vtkm::VecFromPortal` and `vtkm::VecFromPortalPermute` objects allow you to treat a subsection of an arbitrarily large array as a `Vec`. These objects work by attaching to array portals, which are described in Section 27.1. Another example of a `Vec`-like object is `vtkm::VecRectilinearPointCoordinates`, which efficiently represents the point coordinates in an axis-aligned hexahedron. Such shapes are common in structured grids. These and other data sets are described in Chapter 7.

19.3 Range

VTK-m provides a convenience structure named vtkm::Range to help manage a range of values. The Range struct contains two data members, Min and Max, which represent the ends of the range of numbers. Min and Max are both of type vtkm::Float64. Min and Max can be directly accessed, but Range also comes with the following helper functions to make it easier to build and use ranges. Note that all of these functions treat the minimum and maximum value as inclusive to the range.

IsNonEmpty Returns true if the range covers at least one value.

Contains Takes a single number and returns true if that number is contained within the range.

Length Returns the distance between Min and Max. Empty ranges return a length of 0. Note that if the range is non-empty and the length is 0, then Min and Max must be equal, and the range contains exactly one number.

Center Returns the number equidistant to Min and Max. If the range is empty, NaN is returned.

Include Takes either a single number or another range and modifies this range to include the given number or range. If necessary, the range is grown just enough to encompass the given argument. If the argument is already in the range, nothing changes.

Union A nondestructive version of Include, which builds a new Range that is the union of this range and the argument. The + operator is also overloaded to compute the union.

The following example demonstrates the operation of vtkm::Range.

Example 19.6: Using vtkm::Range.

```
vtkm::Range range;             // default constructor is empty range
bool b1 = range.IsNonEmpty(); // b1 is false

range.Include(0.5);            // range now is [0.5 .. 0.5]
bool b2 = range.IsNonEmpty();  // b2 is true
bool b3 = range.Contains(0.5); // b3 is true
bool b4 = range.Contains(0.6); // b4 is false

range.Include(2.0);            // range is now [0.5 .. 2]
bool b5 = range.Contains(0.5); // b3 is true
bool b6 = range.Contains(0.6); // b4 is true

range.Include(vtkm::Range(-1, 1)); // range is now [-1 .. 2]

range.Include(vtkm::Range(3, 4)); // range is now [-1 .. 4]

vtkm::Float64 lower = range.Min;       // lower is -1
vtkm::Float64 upper = range.Max;       // upper is 4
vtkm::Float64 length = range.Length(); // length is 5
vtkm::Float64 center = range.Center(); // center is 1.5
```

19.4 Bounds

VTK-m provides a convenience structure named vtkm::Bounds to help manage an axis-aligned region in 3D space. Among other things, this structure is often useful for representing a bounding box for geometry. The Bounds struct contains three data members, X, Y, and Z, which represent the range of the bounds along each

respective axis. All three of these members are of type `vtkm::Range`, which is discussed previously in Section 19.3. `X`, `Y`, and `Z` can be directly accessed, but `Bounds` also comes with the following helper functions to make it easier to build and use ranges.

`IsNonEmpty` Returns true if the bounds cover at least one value.

`Contains` Takes a `vtkm::Vec` of size 3 and returns true if those point coordinates are contained within the range.

`Center` Returns the point at the center of the range as a `vtkm::Vec <vtkm::Float64,3>`.

`Include` Takes either a `vtkm::Vec` of size 3 or another bounds and modifies this bounds to include the given point or bounds. If necessary, the bounds are grown just enough to encompass the given argument. If the argument is already in the bounds, nothing changes.

`Union` A nondestructive version of `Include`, which builds a new `Bounds` that is the union of this bounds and the argument. The `+` operator is also overloaded to compute the union.

The following example demonstrates the operation of `vtkm::Bounds`.

Example 19.7: Using `vtkm::Bounds`.

```
1  vtkm::Bounds bounds;           // default constructor makes empty
2  bool b1 = bounds.IsNonEmpty(); // b1 is false
3
4  bounds.Include(vtkm::make_Vec(0.5, 2.0, 0.0));            // bounds contains only
5                                                            // the point [0.5, 2, 0]
6  bool b2 = bounds.IsNonEmpty();                            // b2 is true
7  bool b3 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b3 is true
8  bool b4 = bounds.Contains(vtkm::make_Vec(1, 1, 1));       // b4 is false
9  bool b5 = bounds.Contains(vtkm::make_Vec(0, 0, 0));       // b5 is false
10
11 bounds.Include(vtkm::make_Vec(4, -1, 2)); // bounds is region [0.5 .. 4] in X,
12                                          //                 [-1 .. 2] in Y,
13                                          //              and [0 .. 2] in Z
14 bool b6 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b6 is true
15 bool b7 = bounds.Contains(vtkm::make_Vec(1, 1, 1));       // b7 is true
16 bool b8 = bounds.Contains(vtkm::make_Vec(0, 0, 0));       // b8 is false
17
18 vtkm::Bounds otherBounds(vtkm::make_Vec(0, 0, 0), vtkm::make_Vec(3, 3, 3));
19 // otherBounds is region [0 .. 3] in X, Y, and Z
20 bounds.Include(otherBounds); // bounds is now region [0 .. 4] in X,
21                              //                      [-1 .. 3] in Y,
22                              //                   and [0 .. 3] in Z
23
24 vtkm::Vec3f_64 lower(bounds.X.Min, bounds.Y.Min, bounds.Z.Min);
25 // lower is [0, -1, 0]
26 vtkm::Vec3f_64 upper(bounds.X.Max, bounds.Y.Max, bounds.Z.Max);
27 // upper is [4, 3, 3]
28
29 vtkm::Vec3f_64 center = bounds.Center(); // center is [2, 1, 1.5]
```

19.5 Traits

When using templated types, it is often necessary to get information about the type or specialize code based on general properties of the type. VTK-m uses *traits* classes to publish and retrieve information about types. A traits class is simply a templated structure that provides type aliases for tag structures, empty types used for identification. The traits classes might also contain constant numbers and helpful static functions. See *Effective C++ Third Edition* by Scott Meyers for a description of traits classes and their uses.

19.5.1 Type Traits

The vtkm::TypeTraits <T> templated class provides basic information about a core type. These type traits are available for all the basic C++ types as well as the core VTK-m types described in Chapter 4. vtkm::TypeTraits contains the following elements.

`NumericTag` This type is set to either vtkm::TypeTraitsRealTag or vtkm::TypeTraitsIntegerTag to signal that the type represents either floating point numbers or integers.

`DimensionalityTag` This type is set to either vtkm::TypeTraitsScalarTag or vtkm::TypeTraitsVectorTag to signal that the type represents either a single scalar value or a tuple of values.

`ZeroInitialization` A static member function that takes no arguments and returns 0 (or the closest equivalent to it) cast to the type.

The definition of vtkm::TypeTraits for vtkm::Float32 could like something like this.

Example 19.8: Definition of vtkm::TypeTraits <vtkm::Float32 >.

```
1  namespace vtkm {
2
3  template<>
4  struct TypeTraits<vtkm::Float32>
5  {
6    using NumericTag = vtkm::TypeTraitsRealTag;
7    using DimensionalityTag = vtkm::TypeTraitsScalarTag;
8
9    VTKM_EXEC_CONT
10   static vtkm::Float32 ZeroInitialization() { return vtkm::Float32(0); }
11 };
12
13 }
```

Here is a simple example of using vtkm::TypeTraits to implement a generic function that behaves like the remainder operator (%) for all types including floating points and vectors.

Example 19.9: Using TypeTraits for a generic remainder.

```
1  #include <vtkm/TypeTraits.h>
2
3  #include <vtkm/Math.h>
4
5  template<typename T>
6  T AnyRemainder(const T& numerator, const T& denominator);
7
8  namespace detail
9  {
10
11 template<typename T>
12 T AnyRemainderImpl(const T& numerator,
13                    const T& denominator,
14                    vtkm::TypeTraitsIntegerTag,
15                    vtkm::TypeTraitsScalarTag)
16 {
17   return numerator % denominator;
18 }
19
20 template<typename T>
21 T AnyRemainderImpl(const T& numerator,
22                    const T& denominator,
23                    vtkm::TypeTraitsRealTag,
```

```
24                      vtkm::TypeTraitsScalarTag)
25 {
26   // The VTK-m math library contains a Remainder function that operates on
27   // floating point numbers.
28   return vtkm::Remainder(numerator, denominator);
29 }
30
31 template<typename T, typename NumericTag>
32 T AnyRemainderImpl(const T& numerator,
33                    const T& denominator,
34                    NumericTag,
35                    vtkm::TypeTraitsVectorTag)
36 {
37   T result;
38   for (int componentIndex = 0; componentIndex < T::NUM_COMPONENTS; componentIndex++)
39   {
40     result[componentIndex] =
41       AnyRemainder(numerator[componentIndex], denominator[componentIndex]);
42   }
43   return result;
44 }
45
46 } // namespace detail
47
48 template<typename T>
49 T AnyRemainder(const T& numerator, const T& denominator)
50 {
51   return detail::AnyRemainderImpl(numerator,
52                                   denominator,
53                                   typename vtkm::TypeTraits<T>::NumericTag(),
54                                   typename vtkm::TypeTraits<T>::DimensionalityTag());
55 }
```

19.5.2 Vector Traits

The templated `vtkm::Vec` class contains several items for introspection (such as the component type and its size). However, there are other types that behave similarly to `Vec` objects but have different ways to perform this introspection.

For example, VTK-m contains `Vec`-like objects that essentially behave the same but might have different features. Also, there may be reason to interchangeably use basic scalar values, like an integer or floating point number, with vectors. To provide a consistent interface to access these multiple types that represents vectors, the `vtkm::VecTraits <T>` templated class provides information and accessors to vector types.It contains the following elements.

`ComponentType` This type is set to the type for each component in the vector. For example, a `vtkm::Id3` has `ComponentType` defined as `vtkm::Id`.

`IsSizeStatic` This type is set to either `vtkm::VecTraitsTagSizeStatic` if the vector has a static number of components that can be determined at compile time or set to `vtkm::VecTraitsTagSizeVariable` if the size of the vector is determined at run time. If `IsSizeStatic` is set to `VecTraitsTagSizeVariable`, then `VecTraits` will be missing some information that cannot be determined at compile time.

`HasMultipleComponents` This type is set to either `vtkm::VecTraitsTagSingleComponent` if the vector length is size 1 or `vtkm::VecTraitsTagMultipleComponents` otherwise. This tag can be useful for creating specialized functions when a vector is really just a scalar. If the vector type is of variable size (that is, `IsSizeStatic` is `VecTraitsTagSizeVariable`), then `HasMultipleComponents` might be `VecTraitsTagMultipleComponents` even when at run time there is only one component.

`NUM_COMPONENTS` An integer specifying how many components are contained in the vector. `NUM_COMPONENTS` is not available for vector types of variable size (that is, `IsSizeStatic` is `VecTraitsTagSizeVariable`).

`GetNumberOfComponents` A static method that takes an instance of a vector and returns the number of components the vector contains. The result of `GetNumberOfComponents` is the same value of `NUM_COMPONENTS` for vector types that have a static size (that is, `IsSizeStatic` is `VecTraitsTagSizeStatic`). But unlike `NUM_COMPONENTS`, `GetNumberOfComponents` works for vectors of any type.

`GetComponent` A static method that takes a vector and returns a particular component.

`SetComponent` A static method that takes a vector and sets a particular component to a given value.

`CopyInto` A static method that copies the components of a vector to a `vtkm::Vec`.

The definition of `vtkm::VecTraits` for `vtkm::Id3` could look something like this.

Example 19.10: Definition of `vtkm::VecTraits <vtkm::Id3 >`.

```
namespace vtkm {

template<>
struct VecTraits<vtkm::Id3>
{
  using ComponentType = vtkm::Id;
  static const int NUM_COMPONENTS = 3;
  using IsSizeStatic = vtkm::VecTraitsTagSizeStatic;
  using HasMultipleComponents = vtkm::VecTraitsTagMultipleComponents;

  VTKM_EXEC_CONT
  static vtkm::IdComponent GetNumberOfComponents(const vtkm::Id3&)
  {
    return NUM_COMPONENTS;
  }

  VTKM_EXEC_CONT
  static const vtkm::Id& GetComponent(const vtkm::Id3& vector, int component)
  {
    return vector[component];
  }
  VTKM_EXEC_CONT
  static vtkm::Id& GetComponent(vtkm::Id3& vector, int component)
  {
    return vector[component];
  }

  VTKM_EXEC_CONT
  static void SetComponent(vtkm::Id3& vector, int component, vtkm::Id value)
  {
    vector[component] = value;
  }

  template<vtkm::IdComponent DestSize>
  VTKM_EXEC_CONT static void CopyInto(const vtkm::Id3& src,
                                      vtkm::Vec<vtkm::Id, DestSize>& dest)
  {
    for (vtkm::IdComponent index = 0; (index < NUM_COMPONENTS) && (index < DestSize);
         index++)
    {
      dest[index] = src[index];
    }
  }
};

} // namespace vtkm
```

The real power of vector traits is that they simplify creating generic operations on any type that can look like a vector. This includes operations on scalar values as if they were vectors of size one. The following code uses vector traits to simplify the implementation of less functors that define an ordering that can be used for sorting and other operations.

Example 19.11: Using `VecTraits` for less functors.

```
#include <vtkm/VecTraits.h>

// This functor provides a total ordering of vectors. Every compared vector
// will be either less, greater, or equal (assuming all the vector components
// also have a total ordering).
template<typename T>
struct LessTotalOrder
{
  VTKM_EXEC_CONT
  bool operator()(const T& left, const T& right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
      const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType& rightValue =
        vtkm::VecTraits<T>::GetComponent(right, index);
      if (leftValue < rightValue)
      {
        return true;
      }
      if (rightValue < leftValue)
      {
        return false;
      }
    }
    // If we are here, the vectors are equal (or at least equivalent).
    return false;
  }
};

// This functor provides a partial ordering of vectors. It returns true if and
// only if all components satisfy the less operation. It is possible for
// vectors to be neither less, greater, nor equal, but the transitive closure
// is still valid.
template<typename T>
struct LessPartialOrder
{
  VTKM_EXEC_CONT
  bool operator()(const T& left, const T& right)
  {
    for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
    {
      using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
      const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
      const ComponentType& rightValue =
        vtkm::VecTraits<T>::GetComponent(right, index);
      if (!(leftValue < rightValue))
      {
        return false;
      }
    }
    // If we are here, all components satisfy less than relation.
    return true;
  }
};
```

19.6 List Templates

VTK-m internally uses template metaprogramming, which utilizes C++ templates to run source-generating programs, to customize code to various data and compute platforms. One basic structure often uses with template metaprogramming is a list of class names (also sometimes called a tuple or vector, although both of those names have different meanings in VTK-m).

Many VTK-m users only need predefined lists, such as the type lists specified in Section 19.6.2. Those users can skip most of the details of this section. However, it is sometimes useful to modify lists, create new lists, or operate on lists, and these usages are documented here.

19.6.1 Building Lists

A basic list is defined with the `vtkm::List <T, ...>` template, which is defined in the vtkm/List.h header. It is common (but not necessary) to use the `using` keyword to define an alias for a list with a particular meaning.

Example 19.12: Creating lists of types.

```
#include <vtkm/List.h>

// Placeholder classes representing things that might be in a template
// metaprogram list.
class Foo;
class Bar;
class Baz;
class Qux;
class Xyzzy;

// The names of the following tags are indicative of the lists they contain.

using FooList = vtkm::List<Foo>;

using FooBarList = vtkm::List<Foo, Bar>;

using BazQuxXyzzyList = vtkm::List<Baz, Qux, Xyzzy>;

using QuxBazBarFooList = vtkm::List<Qux, Baz, Bar, Foo>;
```

VTK-m defines the convenience class `vtkm::ListEmpty`, which is simply an empty list (i.e. `vtkm::List <>`).

VTK-m also provides a special identifier named `vtkm::ListUniversal`. `ListUniversal` is a conceptual list containing all possible types. Operations on `ListUniversal` will behave as if it contains all types where possible, but some operations (such as getting the size of the list) are ill-defined and will fail.

19.6.2 Type Lists

One of the major use cases for template metaprogramming lists in VTK-m is to identify a set of potential data types for arrays. The vtkm/TypeList.h header contains predefined lists for known VTK-m types. Although technically all these lists are of C++ types, the types we refer to here are those data types stored in data arrays. The following lists are provided.

`vtkm::TypeListId` Contains the single item `vtkm::Id`.

`vtkm::TypeListId2` Contains the single item `vtkm::Id2`.

`vtkm::TypeListId3` Contains the single item `vtkm::Id3`.

`vtkm::TypeListIdComponent` Contains the single item `vtkm::IdComponent`.

`vtkm::TypeListIndex` A list of all types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`.

`vtkm::TypeListFieldScalar` A list containing types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`).

`vtkm::TypeListFieldVec2` A list containing types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListFieldVec3` A list containing types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListFieldVec4` A list containing types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListField` A list containing all the types generally used for fields. It is the combination of `vtkm::TypeListFieldScalar`, `vtkm::TypeListFieldVec2`, `vtkm::TypeListFieldVec3`, and `vtkm::TypeListFieldVec4`.

`vtkm::TypeListScalarAll` A list of all scalar types. It contains signed and unsigned integers of widths from 8 to 64 bits. It also contains floats of 32 and 64 bit widths.

`vtkm::TypeListVecCommon` A list of the most common vector types. It contains all `vtkm::Vec` class of size 2 through 4 containing components of unsigned bytes, signed 32-bit integers, signed 64-bit integers, 32-bit floats, or 64-bit floats.

`vtkm::TypeListVecAll` A list of all `vtkm::Vec` classes with standard integers or floating points as components and lengths between 2 and 4.

`vtkm::TypeListAll` A list of all types included in vtkm/Types.h with `vtkm::Vec` s with up to 4 components.

`vtkm::TypeListCommon` A list containing only the most used types in visualization. This includes signed integers and floats that are 32 or 64 bit. It also includes 3 dimensional vectors of floats. This is the default list used when resolving the type in arrays of unknown type (described in Chapter 33).

If these lists are not sufficient, it is possible to build new type lists using the existing type lists and the list bases from Section 19.6.1 as demonstrated in the following example.

Example 19.13: Defining new type lists.

```
1  #define VTKM_DEFAULT_TYPE_LIST_TAG MyCommonTypes
2
3  #include <vtkm/List.h>
4  #include <vtkm/TypeList.h>
5
6  // A list of 2D vector types.
7  using Vec2List = vtkm::List<vtkm::Vec2f_32, vtkm::Vec2f_64>;
8
9  // An application that uses 2D geometry might commonly encounter this list of
10 // types.
11 using MyCommonTypes = vtkm::ListAppend<Vec2List, vtkm::TypeListCommon>;
```

The vtkm/TypeList.h header also defines a macro named `VTKM_DEFAULT_TYPE_LIST` that defines a default list of types to use when, for example, determining the type of a field array. This list can be overridden by defining the `VTKM_DEFAULT_TYPE_LIST` macro *before* any VTK-m headers are included. If included after a VTK-m header, the list is not likely to take effect. Do not ignore compiler warnings about the macro being redefined, which you will not get if defined correctly. Example 19.13 also contains an example of overriding the `VTKM_DEFAULT_TYPE_LIST` macro.

19.6.3 Querying Lists

vtkm/List.h contains some templated classes to help get information about a list type. This are particularly useful for lists that are provided as templated parameters for which you do not know the exact type.

The VTKM_IS_LIST does a compile-time check to make sure a particular type is actually a vtkm::List of types. If the compile-time check fails, then a build error will occur. This is a good way to verify that a templated class or method that expects a list actually gets a list.

Example 19.14: Checking that a template parameter is a valid List.

```
template<typename List>
class MyImportantClass
{
  VTKM_IS_LIST(List);
  // Implementation...
};

void DoImportantStuff()
{
  MyImportantClass<vtkm::List<vtkm::Id>> important1; // This compiles fine
  MyImportantClass<vtkm::Id> important2;  // COMPILE ERROR: vtkm::Id is not a list
```

The size of a list can be determined by using the vtkm::ListSize template. The type of the template will resolve to a std::integral_constant<vtkm::IdComponent,N> where N is the number of types in the list. vtkm::ListSize does not work with vtkm::ListUniversal.

Example 19.15: Getting the size of a List.

```
  using MyList = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;

  constexpr vtkm::IdComponent myListSize = vtkm::ListSize<MyList>::value;
  // myListSize is 3
```

The vtkm::ListHas template can be used to determine if a vtkm::List contains a particular type. ListHas takes two template parameters. The first parameter is a form of vtkm::List. The second parameter is any type to check to see if it is in the list. If the type is in the list, then ListHas resolves to std::true_type. Otherwise it resolves to std::false_type. vtkm::ListHas always returns true for vtkm::ListUniversal.

Example 19.16: Determining if a List contains a particular type.

```
  using MyList = vtkm::List<vtkm::Int8, vtkm::Int16, vtkm::Int32, vtkm::Int64>;

  constexpr bool hasInt = vtkm::ListHas<MyList, int>::value;
  // hasInt is true

  constexpr bool hasFloat = vtkm::ListHas<MyList, float>::value;
  // hasFloat is false
```

The vtkm::ListIndexOf template can be used to get the index of a particular type in a vtkm::List. ListIndexOf takes two template parameters. The first parameter is a form of vtkm::List. The second parameter is any type to check to see if it is in the list. The type of the template will resolve to a std::integral_constant<vtkm::IdComponent,N> where N is the index of the type. If the requested type is not in the list, then ListIndexOf becomes std::integral_constant<vtkm::IdComponent,-1>.

Conversely, the vtkm::ListAt template can be used to get the type for a particular index. The two template parameters for ListAt are the List and an index for the list.

Neither vtkm::ListIndexOf nor vtkm::ListAt works with vtkm::ListUniversal.

Example 19.17: Using indices with List.

```
using MyList = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;

constexpr vtkm::IdComponent indexOfInt8 =
  vtkm::ListIndexOf<MyList, vtkm::Int8>::value;
// indexOfInt8 is 0
constexpr vtkm::IdComponent indexOfInt32 =
  vtkm::ListIndexOf<MyList, vtkm::Int32>::value;
// indexOfInt32 is 1
constexpr vtkm::IdComponent indexOfInt64 =
  vtkm::ListIndexOf<MyList, vtkm::Int64>::value;
// indexOfInt64 is 2
constexpr vtkm::IdComponent indexOfFloat32 =
  vtkm::ListIndexOf<MyList, vtkm::Float32>::value;
// indexOfFloat32 is -1 (not in list)

using T0 = vtkm::ListAt<MyList, 0>; // T0 is vtkm::Int8
using T1 = vtkm::ListAt<MyList, 1>; // T1 is vtkm::Int32
using T2 = vtkm::ListAt<MyList, 2>; // T2 is vtkm::Int64
```

19.6.4 Operating on Lists

In addition to providing the base templates for defining and querying lists, vtkm/List.h also contains several features for operating on lists.

The vtkm::ListAppend template joins together 2 or more Lists. The items are concatenated in the order provided to ListAppend. ListAppend does not work with vtkm::ListUniversal.

Example 19.18: Appending Lists.

```
using BigTypes = vtkm::List<vtkm::Int64, vtkm::Float64>;
using MediumTypes = vtkm::List<vtkm::Int32, vtkm::Float32>;
using SmallTypes = vtkm::List<vtkm::Int8>;

using SmallAndBigTypes = vtkm::ListAppend<SmallTypes, BigTypes>;
// SmallAndBigTypes is vtkm::List<vtkm::Int8, vtkm::Int64, vtkm::Float64>

using AllMyTypes = vtkm::ListAppend<BigTypes, MediumTypes, SmallTypes>;
// AllMyTypes is
// vtkm::List<vtkm::Int64, vtkm::Float64, vtkm::Int32, vtkm::Float32, vtkm::Int8>
```

The vtkm::ListIntersect template takes two Lists and becomes a vtkm::List containing all types in both lists. If one of the lists is vtkm::ListUniversal, the contents of the other list used.

Example 19.19: Intersecting Lists.

```
using SignedInts = vtkm::List<vtkm::Int8, vtkm::Int16, vtkm::Int32, vtkm::Int64>;
using WordTypes = vtkm::List<vtkm::Int32, vtkm::UInt32, vtkm::Int64, vtkm::UInt64>;

using SignedWords = vtkm::ListIntersect<SignedInts, WordTypes>;
// SignedWords is vtkm::List<vtkm::Int32, vtkm::Int64>
```

The vtkm::ListApply template transfers all of the types in a vtkm::List to another template. The first template argument of ListApply is the List to apply. The second template argument is another template to apply to. ListApply becomes an instance of the passed template with all the types in the List. ListApply can be used to convert a List to some other template. ListApply cannot be used with vtkm::ListUniversal.

Example 19.20: Applying a List to another template.

```
using MyList = vtkm::List<vtkm::Id, vtkm::Id3, vtkm::Vec3f>;

using MyTuple = vtkm::ListApply<MyList, std::tuple>;
// MyTuple is std::tuple<vtkm::Id, vtkm::Id3, vtkm::Vec3f>
```

The vtkm::ListTransform template applies each item in a vtkm::List to another template and constructs a list from all these applications. The first template argument of ListTransform is the List to apply. The second template argument is another template to apply to. ListTransform becomes an instance of a new vtkm::List containing the passed template each type. ListTransform cannot be used with vtkm::ListUniversal.

Example 19.21: Transforming a List using a custom template.

```
using MyList = vtkm::List<vtkm::Int32, vtkm::Float32>;

template<typename T>
using MakeVec = vtkm::Vec<T, 3>;

using MyVecList = vtkm::ListTransform<MyList, MakeVec>;
// MyVecList is vtkm::List<vtkm::Vec<vtkm::Int32, 3>, vtkm::Vec<vtkm::Float32, 3>>
```

The vtkm::ListRemoveIf template removes items from a vtkm::List given a predicate. The first template argument of ListRemoveIf is the List. The second argument is another template that is used as a predicate to determine if the type should be removed or not. The predicate should become a type with a `value` member that is a static true or false value. Any type in the list that the predicate evaluates to true is removed. ListRemoveIf cannot be used with vtkm::ListUniversal.

Example 19.22: Removing items from a List.

```
using MyList =
  vtkm::List<vtkm::Int64, vtkm::Float64, vtkm::Int32, vtkm::Float32, vtkm::Int8>;

using FilteredList = vtkm::ListRemoveIf<MyList, std::is_integral>;
// FilteredList is vtkm::List<vtkm::Float64, vtkm::Float32>
```

The vtkm::ListCross takes two lists and performs a cross product of them. It does this by creating a new vtkm::List that contains nested Lists, each of length 2 and containing all possible pairs of items in the first list with items in the second list. ListCross is often used in conjunction with another list processing command, such as ListTransform to build templated types of many combinations. ListCross cannot be used with vtkm::ListUniversal.

Example 19.23: Creating the cross product of 2 Lists.

```
using BaseTypes = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;
using BoolCases = vtkm::List<std::false_type, std::true_type>;

using CrossTypes = vtkm::ListCross<BaseTypes, BoolCases>;
// CrossTypes is
//   vtkm::List<vtkm::List<vtkm::Int8, std::false_type>,
//              vtkm::List<vtkm::Int8, std::true_type>,
//              vtkm::List<vtkm::Int32, std::false_type>,
//              vtkm::List<vtkm::Int32, std::true_type>,
//              vtkm::List<vtkm::Int64, std::false_type>,
//              vtkm::List<vtkm::Int64, std::true_type>>

template<typename TypeAndIsVec>
using ListPairToType =
  typename std::conditional<vtkm::ListAt<TypeAndIsVec, 1>::value,
                            vtkm::Vec<vtkm::ListAt<TypeAndIsVec, 0>, 3>,
                            vtkm::ListAt<TypeAndIsVec, 0>>::type;

using AllTypes = vtkm::ListTransform<CrossTypes, ListPairToType>;
// AllTypes is
//   vtkm::List<vtkm::Int8,
//              vtkm::Vec<vtkm::Int8, 3>,
//              vtkm::Int32,
//              vtkm::Vec<vtkm::Int32, 3>,
//              vtkm::Int64,
//              vtkm::Vec<vtkm::Int64, 3>>
```

The vtkm::ListForEach function takes a functor object and a vtkm::List. It then calls the functor object with the default object of each type in the list. This is most typically used with C++ run-time type information to convert a run-time polymorphic object to a statically typed (and possibly inlined) call.

The following example shows a rudimentary version of converting a dynamically-typed array to a statically-typed array similar to what is done in VTK-m classes like vtkm::cont::UnknownArrayHandle (which is documented in Chapter 33).

Example 19.24: Converting dynamic types to static types with ListForEach.

```
struct MyArrayBase
{
  // A virtual destructor makes sure C++ RTTI will be generated. It also helps
  // ensure subclass destructors are called.
  virtual ~MyArrayBase() {}
};

template<typename T>
struct MyArrayImpl : public MyArrayBase
{
  std::vector<T> Array;
};

template<typename T>
void PrefixSum(std::vector<T>& array)
{
  T sum(typename vtkm::VecTraits<T>::ComponentType(0));
  for (typename std::vector<T>::iterator iter = array.begin(); iter != array.end();
       iter++)
  {
    sum = sum + *iter;
    *iter = sum;
  }
}

struct PrefixSumFunctor
{
  MyArrayBase* ArrayPointer;

  PrefixSumFunctor(MyArrayBase* arrayPointer)
    : ArrayPointer(arrayPointer)
  {
  }

  template<typename T>
  void operator()(T)
  {
    using ConcreteArrayType = MyArrayImpl<T>;
    ConcreteArrayType* concreteArray =
      dynamic_cast<ConcreteArrayType*>(this->ArrayPointer);
    if (concreteArray != NULL)
    {
      PrefixSum(concreteArray->Array);
    }
  }
};

void DoPrefixSum(MyArrayBase* array)
{
  PrefixSumFunctor functor = PrefixSumFunctor(array);
  vtkm::ListForEach(functor, vtkm::TypeListCommon());
}
```

19.7 Pair

VTK-m defines a `vtkm::Pair <T1,T2>` templated object that behaves just like `std::pair` from the standard template library. The difference is that `vtkm::Pair` will work in both the execution and control environments, whereas the STL `std::pair` does not always work in the execution environment.

The VTK-m version of `vtkm::Pair` supports the same types, fields, and operations as the STL version. VTK-m also provides a `vtkm::make_Pair` function for convenience.

19.8 Tuple

VTK-m defines a `vtkm::Tuple` templated object that behaves like `std::tuple` from the standard template library. The main difference is that `vtkm::Tuple` will work in both the execution and control environments, whereas the STL `std::tuple` does not always work in the execution environment.

19.8.1 Defining and Constructing

`vtkm::Tuple` takes any number of template parameters that define the objects stored the tuple.

Example 19.25: Defining a `Tuple`.

```
1 vtkm::Tuple<vtkm::Id, vtkm::Vec3f, vtkm::cont::ArrayHandle<vtkm::Int32>> myTuple;
```

You can construct a `vtkm::Tuple` with arguments that will be used to initialize the respective objects. As a convenience, you can use `vtkm::MakeTuple` to construct a `vtkm::Tuple` of types based on the arguments.

Example 19.26: Initializing values in a `Tuple`.

```
1 // Initialize a tuple with 0, [0, 1, 2], and an existing ArrayHandle.
2 vtkm::Tuple<vtkm::Id, vtkm::Vec3f, vtkm::cont::ArrayHandle<vtkm::Float32>>
3   myTuple1(0, vtkm::Vec3f(0, 1, 2), array);
4
5 // Another way to create the same tuple.
6 auto myTuple2 = vtkm::MakeTuple(vtkm::Id(0), vtkm::Vec3f(0, 1, 2), array);
```

19.8.2 Querying

The size of a `vtkm::Tuple` can be determined by using the `vtkm::TupleSize` template, which resolves to an `std::integral_constant`. The types at particular indices can be determined with `vtkm::TupleElement`.

Example 19.27: Querying `Tuple` types.

```
1 using TupleType = vtkm::Tuple<vtkm::Id, vtkm::Float32, vtkm::Float64>;
2
3 // Becomes 3
4 constexpr vtkm::IdComponent size = vtkm::TupleSize<TupleType>::value;
5
6 using FirstType = vtkm::TupleElement<0, TupleType>;  // vtkm::Id
7 using SecondType = vtkm::TupleElement<1, TupleType>; // vtkm::Float32
8 using ThirdType = vtkm::TupleElement<2, TupleType>;  // vtkm::Float64
```

The function `vtkm::Get` can be used to retrieve an element from the `vtkm::Tuple`. `Get` returns a reference to the element, so you can set a `vtkm::Tuple` element by `Get`ing the value.

Example 19.28: Retrieving values from a `Tuple`.

```
  auto myTuple = vtkm::MakeTuple(vtkm::Id3(0, 1, 2), vtkm::Vec3f(3, 4, 5));

  // Gets the value [0, 1, 2]
  vtkm::Id3 x = vtkm::Get<0>(myTuple);

  // Changes the second object in myTuple to [6, 7, 8]
  vtkm::Get<1>(myTuple) = vtkm::Vec3f(6, 7, 8);
```

19.8.3 For Each

`Tuple::ForEach` is a method that takes a function or functor and calls it for each of the items in the tuple. Nothing is returned from ForEach and any return value from the function is ignored. `ForEach` can be used to check the validity of each item.

Example 19.29: Using `Tuple::ForEach` to check the contents.

```
void CheckPositive(vtkm::Float64 x)
{
  if (x < 0)
  {
    throw vtkm::cont::ErrorBadValue("Values need to be positive.");
  }
}

// ...

  vtkm::Tuple<vtkm::Float64, vtkm::Float64, vtkm::Float64> tuple(
    CreateValue(0), CreateValue(1), CreateValue(2));

  // Will throw an error if any of the values are negative.
  tuple.ForEach(CheckPositive);
```

`Tuple::ForEach` can also be used to aggregate values.

Example 19.30: Using `Tuple::ForEach` to aggregate.

```
struct SumFunctor
{
  vtkm::Float64 Sum = 0;

  template<typename T>
  void operator()(const T& x)
  {
    this->Sum = this->Sum + static_cast<vtkm::Float64>(x);
  }
};

// ...

  vtkm::Tuple<vtkm::Float32, vtkm::Float64, vtkm::Id> tuple(
    CreateValue(0), CreateValue(1), CreateValue(2));

  SumFunctor sum;
  tuple.ForEach(sum);
  vtkm::Float64 average = sum.Sum / 3;
```

19.8.4 Transform

`Tuple::Transform` is a method that builds a new Tuple by calling a function or functor on each of the items. The return value is placed in the corresponding part of the resulting Tuple, and the type is automatically created from the return type of the function.

Example 19.31: Transforming a `Tuple`.

```
struct GetReadPortalFunctor
{
  template<typename Array>
  typename Array::ReadPortalType operator()(const Array& array) const
  {
    VTKM_IS_ARRAY_HANDLE(Array);
    return array.ReadPortal();
  }
};

// ...

  auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);

  auto portalTuple = arrayTuple.Transform(GetReadPortalFunctor{});
```

19.8.5 Apply

`Tuple::Apply` is a method that calls a function or functor using the objects in the Tuple as the arguments. If the function returns a value, that value is returned from Apply.

Example 19.32: Applying a `Tuple` as arguments to a function.

```
struct AddArraysFunctor
{
  template<typename Array1, typename Array2, typename Array3>
  vtkm::Id operator()(Array1 inArray1, Array2 inArray2, Array3 outArray) const
  {
    VTKM_IS_ARRAY_HANDLE(Array1);
    VTKM_IS_ARRAY_HANDLE(Array2);
    VTKM_IS_ARRAY_HANDLE(Array3);

    vtkm::Id length = inArray1.GetNumberOfValues();
    VTKM_ASSERT(inArray2.GetNumberOfValues() == length);
    outArray.Allocate(length);

    auto inPortal1 = inArray1.ReadPortal();
    auto inPortal2 = inArray2.ReadPortal();
    auto outPortal = outArray.WritePortal();
    for (vtkm::Id index = 0; index < length; ++index)
    {
      outPortal.Set(index, inPortal1.Get(index) + inPortal2.Get(index));
    }

    return length;
  }
};

// ...

  auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);

  vtkm::Id arrayLength = arrayTuple.Apply(AddArraysFunctor{});
```

If additional arguments are given to `Apply`, they are also passed to the function (before the objects in the `vtkm::Tuple`). This is helpful for passing state to the function.

Example 19.33: Using extra arguments with `Tuple::Apply`.

```
struct ScanArrayLengthFunctor
{
  template<vtkm::IdComponent N, typename Array, typename... Remaining>
  vtkm::Vec<vtkm::Id, N + 1 + vtkm::IdComponent(sizeof...(Remaining))> operator()(
    const vtkm::Vec<vtkm::Id, N>& partialResult,
    const Array& nextArray,
    const Remaining&... remainingArrays) const
  {
    vtkm::Vec<vtkm::Id, N + 1> nextResult;
    std::copy(&partialResult[0], &partialResult[0] + N, &nextResult[0]);
    nextResult[N] = nextResult[N - 1] + nextArray.GetNumberOfValues();
    return (*this)(nextResult, remainingArrays...);
  }

  template<vtkm::IdComponent N>
  vtkm::Vec<vtkm::Id, N> operator()(const vtkm::Vec<vtkm::Id, N>& result) const
  {
    return result;
  }
};

// ...

  auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);

  vtkm::Vec<vtkm::Id, 4> sizeScan =
    arrayTuple.Apply(ScanArrayLengthFunctor{}, vtkm::Vec<vtkm::Id, 1>{ 0 });
```

19.9 Error Codes

For operations that occur in the control environment, VTK-m uses exceptions to report errors as described in Chapter 11. However, when operating in the execution environment, it is not feasible to throw exceptions. Thus, for operations designed for the execution environment, the status of an operation that can fail is returned as an `vtkm::ErrorCode`, which is an `enum`. An `ErrorCode` can be one of the following enumerators.

`ErrorCode::Success` The operation completed successfully.

`ErrorCode::InvalidShapeId` An operation on a cell was given a shape identifier that is not recognized.

`ErrorCode::InvalidNumberOfPoints` The wrong number of points was provided for a given cell type. For example, if a triangle has 4 points associated with it, you are likely to get this error.

`ErrorCode::InvalidPointId` A bad point identifier was detected while operating on a cell.

`ErrorCode::InvalidEdgeId` A bad edge identifier was detected while operating on a cell.

`ErrorCode::InvalidFaceId` A bad face identifier was detected while operating on a cell.

`ErrorCode::SolutionDidNotConverge` An iterative operation did not find an appropriate solution. The result is not likely to be accurate.

`ErrorCode::MatrixFactorizationFailed` A solution was not found for a linear system.

`ErrorCode::DegenerateCellDetected` A cell's parameters have degenerated it to another type. For example, if two vertices of a tetrahedron are the same, it degenerates into a triangle.

`ErrorCode::MalformedCellDetected` The structure of a cell is incorrect. For example, if the vertices of a cell are listed in the wrong order, you might encounter this error.

`ErrorCode::OperationOnEmptyCell` There is an "empty" cell placeholder type to be used when other cell types cannot be applied. Because it is a placeholder, operations on these types of cells are undefined.

`ErrorCode::CellNotFound` A locate operation failed to find a cell given the search criteria.

If a function or method returns an `ErrorCode`, it is a good practice to check to make sure that the returned value is `Success`. If it is not, you can use the `vtkm::ErrorString` function to convert the `ErrorCode` to a descriptive C string. The easiest thing to do from within a worklet is to call the worklet's `RaiseError` method.

Example 19.34: Checking an `ErrorCode` and reporting errors in a worklet.

```
1  vtkm::ErrorCode status = cellLocator.FindCell(point, cellId, parametric);
2  if (status != vtkm::ErrorCode::Success)
3  {
4    this->RaiseError(vtkm::ErrorString(status));
5  }
```

CHAPTER

TWENTY

LOGGING

VTK-m features a logging system that allows status updates and timing. VTK-m uses the loguru project to provide runtime logging facilities.[1] Logging is enabled by setting the CMake variable VTKm_ENABLE_LOGGING. When this flag is enabled, any messages logged to the Info, Warn, Error, and Fatal levels are printed to stderr by default.

20.1 Initializing Logging

Additional logging features are enabled by calling `vtkm::cont::Initialize` as described in Chapter 6. Although calling `Initialize` is not strictly necessary for output messages, initialization adds the following features.

- Set human-readable names for the log levels in the output.
- Allow the stderr logging level to be set at runtime by passing a `-v [level]` argument to the executable (if provided).
- Name the main thread.
- Print a preamble with details of the program's startup (arguments, etc).

Example 20.1 in the following section provides an example of initializing with additional logging setup.

The logging implementation is thread-safe. When working in a multithreaded environment, each thread may be assigned a human-readable name using `vtkm::cont::SetThreadName` (which can later be retrieved with `vtkm::cont::GetThreadName`. This name will appear in the log output so that per-thread messages can be easily tracked.

20.2 Logging Levels

The logging in VTK-m provides several "levels" of logging. Logging levels are ordered by precedence. When selecting which log message to output, a single logging level is provided. Any logging message with that or a higher precedence is output. For example, if warning messages are on, then error messages are also outputted because errors are a higher precedence than warnings. Likewise, if information messages are on, then error and warning messages are also outputted.

[1] A sample of the log output can be found at https://gitlab.kitware.com/snippets/427.

Common Errors

All logging levels are assigned a number, and logging levels with a higher precedence actually have a smaller number.

All logging levels are listed in the `vtkm::cont::LogLevel` enum. The available logging levels, in order of precedence, are as follows.

`LogLevel::Off` A placeholder used to silence all logging.

`LogLevel::Fatal` Fatal errors that should abort execution.

`LogLevel::Error` Important but non-fatal errors, such as device fail-over.

`LogLevel::Warn` Less important user errors, such as out-of-bounds parameters.

`LogLevel::Info` Information messages (detected hardware, etc) and temporary debugging output.

`LogLevel::UserFirst` The first in a range of logging levels reserved for code that uses VTK-m. Internal VTK-m code will not log on these levels but will report these logs.

`LogLevel::UserLast` The last in a range of logging levels reserved for code that uses VTK-m.

`LogLevel::Perf` General timing data and algorithm flow information, such as filter execution, worklet dispatches, and device algorithm calls.

`LogLevel::MemCont` Host-side resource memory allocations and frees such as `ArrayHandle` control buffers.

`LogLevel::MemExec` Device-side resource memory allocations and frees such as `ArrayHandle` device buffers)

`LogLevel::MemTransfer` Transferring of data between a host and device.

`LogLevel::Cast` Report when a dynamic object is (or is not) resolved via a `CastAndCall` or other casting method.

`LogLevel::UserVerboseFirst` The first in a range of logging levels reserved for code that uses VTK-m. Internal VTK-m code will not log on these levels but will report these logs. These are used similarly to those in the `UserFirst` range but are at a lower precedence that also includes more verbose reporting from VTK-m.

`LogLevel::UserVerboseLast` The last in a range of logging levels reserved for code that uses VTK-m.

When VTK-m outputs an entry in its log, it annotates the message with the logging level. VTK-m will automatically provide descriptions for all log levels described in `vtkm::cont::LogLevel`. A custom log level can be described by calling the `vtkm::cont::SetLogLevelName` function. (The log name can likewise be retrieved with `vtkm::cont::GetLogLevelName`.)

Common Errors

The `SetLogLevelName` *function must be called before* `vtkm::cont::Initialize` *to have an effect.*

Common Errors

The descriptions for each log level are only set up if `vtkm::cont::Initialize` *is called. If it is not, then all log levels will be represented with a numerical value.*

If `vtkm::cont::Initialize` is called with `argc`/`argv`, then the user can control the logging level with the "–vtkm-log-level" command line argument. Alternatively, you can control which logging levels are reported with the `vtkm::cont::SetStderrLogLevel`.

Example 20.1: Initializing logging.

```
1  static const vtkm::cont::LogLevel CustomLogLevel = vtkm::cont::LogLevel::UserFirst;
2
3  int main(int argc, char** argv)
4  {
5    vtkm::cont::SetLogLevelName(CustomLogLevel, "custom");
6
7    // For this example we will set the log level manually.
8    // The user can override this with the --vtkm-log-level command line flag.
9    vtkm::cont::SetStderrLogLevel(CustomLogLevel);
10
11   vtkm::cont::Initialize(argc, argv);
12
13   // Do interesting stuff...
```

20.3 Log Entries

Log entries are created with a collection of macros provided in vtkm/cont/Logging.h. In addition to basic log entries, VTK-m logging can also provide conditional logging, scope levels of logs, and generate special logs on crashes.

20.3.1 Basic Log Entries

The main logging entry points are the macros `VTKM_LOG_S` and `VTKM_LOG_F`, which use C++ stream and printf syntax, respectively. Both macros take a logging level as the first argument. The remaining arguments specify the message printed to the log. `VTKM_LOG_S` takes a single argument with a C++ stream expression (so `<<` operators can exist in the expression). `VTKM_LOG_F` takes a C string as its second argument that has printf-style formatting codes. The remaining arguments fulfill those codes.

Example 20.2: Basic logging.

```
1    VTKM_LOG_F(vtkm::cont::LogLevel::Info,
2               "Base VTK-m version: %d.%d",
3               VTKM_VERSION_MAJOR,
4               VTKM_VERSION_MINOR);
5    VTKM_LOG_S(vtkm::cont::LogLevel::Info,
6               "Full VTK-m version: " << VTKM_VERSION_FULL);
```

20.3.2 Conditional Log Entries

The macros `VTKM_LOG_IF_S` `VTKM_LOG_IF_F` behave similarly to `VTKM_LOG_S` and `VTKM_LOG_F`, respectively, except they have an extra argument that contains the condition. If the condition is true, then the log entry is

created. If the condition is false, then the statement is ignored and nothing is recorded in the log.

Example 20.3: Conditional logging.

```
for (size_t i = 0; i < 5; i++)
{
  VTKM_LOG_IF_S(
    vtkm::cont::LogLevel::Info, i % 2 == 0, "Found an even number: " << i);
}
```

20.3.3 Scoped Log Entries

The logging back end supports the concept of scopes. Scopes allow the nesting of log messages, which allows a complex operation to report when it starts, when it ends, and what log messages happen in the middle. Scoped log entries are also timed so you can get an idea of how long operations take. Scoping can happen to arbitrary depths.

Common Errors

Although the timing reported in scoped log entries can give an idea of the time each operation takes, the reported time should not be considered accurate in regards to timing parallel operations. If a parallel algorithm is invoked inside a log scope, the program may return from that scope before the parallel algorithm is complete. See Chapter 13 for information on more accurate timers.

Scoped log entries follow the same scoping of your C++ code. A scoped log can be created with the VTKM_LOG_SCOPE macro. This macro behaves similarly to VTKM_LOG_F except that it creates a scoped log that starts when VTKM_LOG_SCOPE and ends when the program leaves the given scope.

Example 20.4: Scoped logging.

```
for (vtkm::IdComponent trial = 0; trial < numTrials; ++trial)
{
  VTKM_LOG_SCOPE(CustomLogLevel, "Trial %d", trial);

  VTKM_LOG_F(CustomLogLevel, "Do thing 1");

  VTKM_LOG_F(CustomLogLevel, "Do thing 2");

  //...
}
```

It is also common, and typically good code structure, to structure scoped concepts around functions or methods. Thus, VTK-m provides VTKM_LOG_SCOPE_FUNCTION. When placed at the beginning of a function or macro, VTKM_LOG_SCOPE_FUNCTION will automatically create a scoped log around it.

Example 20.5: Scoped logging in a function.

```
void TestFunc()
{
  VTKM_LOG_SCOPE_FUNCTION(vtkm::cont::LogLevel::Info);
  VTKM_LOG_S(vtkm::cont::LogLevel::Info, "Showcasing function logging");
}
```

20.4 Helper Functions

The vtkm/cont/Logging.h header file also contains several helper functions that provide useful functions when reporting information about the system.

Did you know?

Although provided with the logging utilities, these functions can be useful in contexts outside of the logging as well. These functions are available even if VTK-m is compiled with logging off.

The `vtkm::cont::TypeToString` function provides run-time type information (RTTI) based type-name information. `TypeToString` is a templated function for which you have to explicitly declare the type. `TypeToString` returns a `std::string` containing a representation of the type provided. When logging is enabled, `TypeToString` uses the logging back end to demangle symbol names on supported platforms.

The `vtkm::cont::GetHumanReadableSize` takes a size of memory in bytes and returns a human readable string (for example ”64 bytes”, ”1.44 MiB”, ”128 GiB”, etc). `vtkm::cont::GetSizeString` is a similar function that returns the same thing as `GetHumanReadableSize` followed by “(# bytes)” (with # replaced with the number passed to the function). Both `GetHumanReadableSize` and `GetSizeString` take an optional second argument that is the number of digits of precision to display. By default, they display 2 digits of precision.

The `vtkm::cont::GetStackTrace` function returns a string containing a trace of the stack, which can be helpful for debugging. `GetStackTrace` takes an optional argument for the number of stack frames to skip. Reporting the stack trace is not available on all platforms. On platforms that are not supported, a simple string reporting that the stack trace is unavailable is returned.

Example 20.6: Helper log functions.

```
1  template<typename T>
2  void DoSomething(T&& x)
3  {
4    VTKM_LOG_S(CustomLogLevel,
5               "Doing something with type " << vtkm::cont::TypeToString<T>());
6
7    vtkm::Id arraySize = 100000 * sizeof(T);
8    VTKM_LOG_S(CustomLogLevel,
9               "Size of array is " << vtkm::cont::GetHumanReadableSize(arraySize));
10   VTKM_LOG_S(CustomLogLevel,
11              "More precisely it is " << vtkm::cont::GetSizeString(arraySize, 4));
12
13   VTKM_LOG_S(CustomLogLevel, "Stack location: " << vtkm::cont::GetStackTrace());
```

CHAPTER

TWENTYONE

WORKLET TYPE REFERENCE

Chapter 17 introduces worklets and provides a simple example of creating a worklet to run an algorithm on a many core device. Different operations in visualization can have different data access patterns, perform different execution flow, and require different provisions. VTK-m manages these different accesses, execution, and provisions by grouping visualization algorithms into common classes of operation and supporting each class with its own worklet type.

Each worklet type has a generic superclass that worklets of that particular type must inherit. This makes the type of the worklet easy to identify. The following list describes each worklet type provided by VTK-m and the superclass that supports it.

Field Map A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over a mesh, a `WorkletMapField` may actually be applied to any array. Thus, a field map can be used as a basic map operation.

Topology Map A worklet deriving `vtkm::worklet::WorkletMapTopology` or one of its child classes performs a mapping operation that applies a function (the operator in the worklet) on all elements of a particular type (such as points or cells) and creates a new field for those elements. The basic operation is similar to a field map except that in addition to access fields being mapped on, the worklet operation also has access to incident fields.

There are multiple convenience classes available for the most common types of topology mapping. `vtkm::worklet::WorkletVisitCellsWithPoints` calls the worklet operation for each cell and makes every incident point available. This type of map also has access to cell structures and can interpolate point fields. Likewise, `vtkm::worklet::WorkletVisitPointsWithCells` calls the worklet operation for each point and makes every incident cell available.

Point Neighborhood A worklet deriving from `vtkm::worklet::WorkletPointNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all points of a structured mesh. The basic operation is similar to a field map except that in addition to having access to the point being operated on, you can get the field values of nearby points within a neighborhood of a given size. Point neighborhood worklets can only applied to structured cell sets.

Reduce by Key A worklet deriving `vtkm::worklet::WorkletReduceByKey` operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a `Vec`-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

The remainder of this chapter provides details on how to create worklets of each type. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 43.

21.1 Field Map

A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over the mesh, a `WorkletMapField` can actually be applied to any array.

A field map worklet supports the following tags in the parameters of its `ControlSignature`.

`FieldIn` This tag represents an input field. A `FieldIn` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. Each invocation of the worklet gets a single value out of this array.

The worklet's `InputDomain` can be set to a `FieldIn` argument. In this case, the input domain will be the size of the array.

`FieldOut` This tag represents an output field. A `FieldOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldInOut` This tag represents field that is both an input and an output. A `FieldInOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

The worklet's `InputDomain` can be set to a `FieldInOut` argument. In this case, the input domain will be the size of the array.

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`AtomicArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `AtomicArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

`WholeCellSetIn` This tag represents the connectivity of a cell set. A `WholeCellSetIn` argument expects a `vtkm::cont::CellSet` in the associated parameter of the `Invoker`. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A field map worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 21.1: Implementation and use of a field map worklet.

```
1  class ComputeMagnitude : public vtkm::worklet::WorkletMapField
2  {
3  public:
4    using ControlSignature = void(FieldIn inputVectors, FieldOut outputMagnitudes);
5    using ExecutionSignature = _2(_1);
6
7    using InputDomain = _1;
8
9    template<typename T, vtkm::IdComponent Size>
10   VTKM_EXEC T operator()(const vtkm::Vec<T, Size>& inVector) const
11   {
12     return vtkm::Magnitude(inVector);
13   }
14 };
```

Although simple, the `WorkletMapField` worklet type can be used (and abused) as a general parallel-for/scheduling mechanism. In particular, the `WorkIndex` execution signature tag can be used to get a unique index, the `WholeArray*` tags can be used to get random access to arrays, and the `ExecObject` control signature tag can be used to pass execution objects directly to the worklet. Whole arrays and execution objects are talked about in more detail in Chapters 28 and 29, respectively, in more detail, but here is a simple example that uses the random access of `WholeArrayOut` to make a worklet that copies an array in reverse order.

Example 21.2: Leveraging field maps and field maps for general processing.

```
namespace vtkm
{
namespace worklet
{

struct ReverseArrayCopyWorklet : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn inputArray, WholeArrayOut outputArray);
  using ExecutionSignature = void(_1, _2, WorkIndex);
  using InputDomain = _1;

  template<typename InputType, typename OutputArrayPortalType>
  VTKM_EXEC void operator()(const InputType& inputValue,
                            const OutputArrayPortalType& outputArrayPortal,
                            vtkm::Id workIndex) const
  {
    vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
    if (outIndex >= 0)
    {
      outputArrayPortal.Set(outIndex, inputValue);
    }
    else
    {
      this->RaiseError("Output array not sized correctly.");
    }
  }
};

} // namespace worklet
} // namespace vtkm
```

21.2 Topology Map

A topology map performs a mapping that it applies a function (the operator in the worklet) on all the elements of a `DataSet` of a particular type (i.e. point, edge, face, or cell). While operating on the element, the worklet has access to data from all incident elements of another type.

There are several versions of topology maps that differ in what type of element being mapped from and what type of element being mapped to. The subsequent sections describe these different variations of the topology maps.

21.2.1 Visit Cells with Points

A worklet deriving `vtkm::worklet::WorkletVisitCellsWithPoints` performs a mapping operation that applies a function (the operator in the worklet) on all the cells of a `DataSet`. While operating on the cell, the worklet has access to fields associated both with the cell and with all incident points. Additionally, the worklet can get information about the structure of the cell and can perform operations like interpolation on it.

A visit cells with points worklet supports the following tags in the parameters of its `ControlSignature`.

`CellSetIn` This tag represents the cell set that defines the collection of cells the map will operate on. A `CellSetIn` argument expects a `CellSet` subclass or an `UnknownCellSet` in the associated parameter of the `Invoker`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.)

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of points.

Each invocation of the worklet gets a `Vec`-like object containing the field values for all the points incident with the cell being visited. The order of the entries is consistent with the defined order of the vertices for the visited cell's shape. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

`FieldOutCell` This tag represents an output field, which is necessarily associated with cells. A `FieldOutCell` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` is an alias for `FieldOutCell` (since output arrays can only be defined on cells).

`FieldInOutCell` This tag represents field that is both an input and an output, which is necessarily associated with cells. A `FieldInOutCell` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`FieldInOut` is an alias for `FieldInOutCell` (since output arrays can only be defined on cells).

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`AtomicArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `AtomicArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

`WholeCellSetIn` This tag represents the connectivity of a cell set. A `WholeCellSetIn` argument expects a `vtkm::cont::CellSet` in the associated parameter of the `Invoker`. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A visit cells with points worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`CellShape` This tag produces a shape tag corresponding to the shape of the visited cell. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.) This is the same value that gets provided if you reference the `CellSetIn` parameter.

`PointCount` This tag produces a `vtkm::IdComponent` equal to the number of points incident on the cell being visited. The Vecs provided from a `FieldInPoint` parameter will be the same size as `PointCount`.

`PointIndices` This tag produces a `Vec`-like object of `vtkm::Id` s giving the indices for all incident points. Like values from a `FieldInPoint` parameter, the order of the entries is consistent with the defined order of the vertices for the visited cell's shape.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

Point to cell field maps are a powerful construct that allow you to interpolate point fields throughout the space of the data set. See Chapter 25 for a description on how to work with the cell information provided to the worklet. The following example provides a simple demonstration that finds the geometric center of each cell by interpolating the point coordinates to the cell centers.

Example 21.3: Implementation and use of a visit cells with points worklet.

```
1  namespace vtkm
2  {
3  namespace worklet
4  {
5
6  struct CellCenter : public vtkm::worklet::WorkletVisitCellsWithPoints
7  {
8  public:
9    using ControlSignature = void(CellSetIn cellSet,
10                                  FieldInPoint inputPointField,
11                                  FieldOut outputCellField);
12   using ExecutionSignature = void(_1, PointCount, _2, _3);
```

```
13
14    using InputDomain = _1;
15
16    template<typename CellShape, typename InputPointFieldType, typename OutputType>
17    VTKM_EXEC void operator()(CellShape shape,
18                              vtkm::IdComponent numPoints,
19                              const InputPointFieldType& inputPointField,
20                              OutputType& centerOut) const
21    {
22      vtkm::Vec3f parametricCenter;
23      vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, parametricCenter);
24      vtkm::exec::CellInterpolate(inputPointField, parametricCenter, shape, centerOut);
25    }
26  };
27
28  } // namespace worklet
29  } // namespace vtkm
```

21.2.2 Visit Points with Cells

A worklet deriving `vtkm::worklet::WorkletVisitPointsWithCells` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `DataSet`. While operating on the point, the worklet has access to fields associated both with the point and with all incident cells.

A visit points with cells worklet supports the following tags in the parameters of its `ControlSignature`.

`CellSetIn` This tag represents the cell set that defines the collection of points the map will operate on. A `CellSetIn` argument expects a `CellSet` subclass or an `UnknownCellSet` in the associated parameter of the `Invoker`.

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of cells.

Each invocation of the worklet gets a `Vec`-like object containing the field values for all the cells incident with the point being visited. The order of the entries is arbitrary but will be consistent with the values of all other `FieldInCell` arguments for the same worklet invocation. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of points. Each invocation of the worklet gets a single value out of this array.

`FieldOutPoint` This tag represents an output field, which is necessarily associated with points. A `FieldOutPoint` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` is an alias for `FieldOutPoint` (since output arrays can only be defined on points).

`FieldInOutPoint` This tag represents field that is both an input and an output, which is necessarily associated with points. A `FieldInOutPoint` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the

associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

FieldInOut is an alias for FieldInOutPoint (since output arrays can only be defined on points).

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

WholeArrayOut This tag represents an array where any entry can be written by any worklet invocation. A WholeArrayOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

WholeArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A WholeArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

AtomicArrayInOut This tag represents an array where any entry can be read or written by any worklet invocation. A AtomicArrayInOut argument expects an ArrayHandle in the associated parameter of the Invoker. A vtkm::exec::AtomicArray object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

WholeCellSetIn This tag represents the connectivity of a cell set. A WholeCellSetIn argument expects a vtkm::cont::CellSet in the associated parameter of the Invoker. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

ExecObject This tag represents an execution object that is passed directly from the control environment to the worklet. A ExecObject argument expects a subclass of vtkm::exec::ExecutionObjectBase. Subclasses of ExecutionObjectBase behave like a factory for objects that work on particular devices. They do this by implementing a PrepareForExecution method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A visit points with cells worklet supports the following tags in the parameters of its ExecutionSignature.

_1, _2,... These reference the corresponding parameter in the ControlSignature.

CellCount This tag produces a vtkm::IdComponent equal to the number of cells incident on the point being visited. The Vecs provided from a FieldInCell parameter will be the same size as CellCount.

CellIndices This tag produces a Vec-like object of vtkm::Id s giving the indices for all incident cells. The order of the entries is arbitrary but will be consistent with the values of all other FieldInCell arguments for the same worklet invocation.

WorkIndex This tag produces a vtkm::Id that uniquely identifies the invocation of the worklet.

VisitIndex This tag produces a vtkm::IdComponent that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

Cell to point field maps are typically used for converting fields associated with cells to points so that they can be interpolated. The following example does a simple averaging, but you can also implement other strategies such as a volume weighted average.

Example 21.4: Implementation and use of a visit points with cells worklet.

```
class AverageCellField : public vtkm::worklet::WorkletVisitPointsWithCells
{
public:
  using ControlSignature = void(CellSetIn cellSet,
                                FieldInCell inputCellField,
                                FieldOut outputPointField);
  using ExecutionSignature = void(CellCount, _2, _3);

  using InputDomain = _1;

  template<typename InputCellFieldType, typename OutputFieldType>
  VTKM_EXEC void operator()(vtkm::IdComponent numCells,
                            const InputCellFieldType& inputCellField,
                            OutputFieldType& fieldAverage) const
  {
    fieldAverage = OutputFieldType(0);

    for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
    {
      fieldAverage = fieldAverage + inputCellField[cellIndex];
    }

    fieldAverage = fieldAverage / OutputFieldType(numCells);
  }
};

//
// Later in the associated Filter class...
//

        vtkm::cont::ArrayHandle<T> outFieldData;
        this->Invoke(AverageCellField{}, inCellSet, inFieldData, outFieldData);
```

21.2.3 General Topology Maps

A worklet deriving `vtkm::worklet::WorkletMapTopology` performs a mapping operation that applies a function (the operator in the worklet) on all the elements of a specified type from a `DataSet`. While operating on each element, the worklet has access to fields associated both with that element and with all incident elements of a different specified type.

The `WorkletMapTopology` class is a template with two template parameters. The first template parameter specifies the “visit” topology element, and the second parameter specifies the “incident” topology element. The

worklet is scheduled such that each instance is associated with a particular "visit" topology element and has access to "incident" topology elements.

These visit and incident topology elements are specified with topology element tags, which are defined in the vtkm/TopologyElementTag.h header file. The available topology element tags are vtkm::TopologyElementTagCell, vtkm::TopologyElementTagPoint, vtkm::TopologyElementTagEdge, and vtkm::TopologyElementTagFace, which represent the cell, point, edge, and face elements, respectively.

WorkletMapTopology is a generic form of a topology map, and it can perform identically to the aforementioned forms of topology map with the correct template parameters. For example,

```
vtkm::worklet::WorkletMapTopology <vtkm::TopologyElementTagCell, vtkm::TopologyEle-
mentTagPoint >
```

is equivalent to the vtkm::worklet::WorkletVisitCellsWithPoints class except the signature tags have different names. The names used in the specific topology map superclasses (such as WorkletVisitCellsWithPoints) tend to be easier to read and are thus preferable. However, the generic WorkletMapTopology is available for topology combinations without a specific superclass or to support more general mappings in a worklet.

The general topology map worklet supports the following tags in the parameters of its ControlSignature, which are equivalent to tags in the other topology maps but with different (more general) names.

CellSetIn This tag represents the cell set that defines the collection of elements the map will operate on. A CellSetIn argument expects a CellSet subclass or an UnknownCellSet in the associated parameter of the Invoker. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.)

There must be exactly one CellSetIn argument, and the worklet's InputDomain must be set to this argument.

FieldInVisit This tag represents an input field that is associated with the "visit" element. A FieldInVisit argument expects an ArrayHandle or an UnknownArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of cells. Each invocation of the worklet gets a single value out of this array.

FieldInIncident This tag represents an input field that is associated with the "incident" elements. A FieldInIncident argument expects an ArrayHandle or an UnknownArrayHandle in the associated parameter of the Invoker. The size of the array must be exactly the number of "incident" elements.

Each invocation of the worklet gets a Vec-like object containing the field values for all the "incident" elements incident with the "visit" element being visited. If the field is a vector field, then the provided object is a Vec of Vecs.

FieldOut This tag represents an output field, which is necessarily associated with "visit" elements. A FieldOut argument expects an ArrayHandle or an UnknownArrayHandle in the associated parameter of the Invoker. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

FieldInOut This tag represents field that is both an input and an output, which is necessarily associated with "visit" elements. A FieldInOut argument expects an ArrayHandle or an UnknownArrayHandle in the associated parameter of the Invoker. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

WholeArrayIn This tag represents an array where all entries can be read by every worklet invocation. A WholeArrayIn argument expects an ArrayHandle in the associated parameter of the Invoker. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`AtomicArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `AtomicArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

`WholeCellSetIn` This tag represents the connectivity of a cell set. A `WholeCellSetIn` argument expects a `vtkm::cont::CellSet` in the associated parameter of the `Invoker`. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A general topology map worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`CellShape` This tag produces a shape tag corresponding to the shape of the visited element. (Cell shapes and the operations you can do with cells are discussed in Chapter 25.) This is the same value that gets provided if you reference the `CellSetIn` parameter.

If the "visit" element is cells, the `CellShape` clearly will match the shape of each cell. Other elements will have shapes to match their structures. Points have vertex shapes, edges have line shapes, and faces have some type of polygonal shape.

`IncidentElementCount` This tag produces a `vtkm::IdComponent` equal to the number of elements incident on the element being visited. The Vecs provided from a `FieldInIncident` parameter will be the same size as `IncidentElementCount`.

`IncidentElementIndices` This tag produces a `Vec`-like object of `vtkm::Id` s giving the indices for all incident elements. The order of the entries is consistent with the values of all other `FieldInIncident` arguments for the same worklet invocation.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

21.3 Point Neighborhood

A worklet deriving `vtkm::worklet::WorkletPointNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `DataSet`. While operating on the point, the worklet has access to field values on nearby points within a neighborhood.

A point neighborhood worklet supports the following tags in the parameters of its `ControlSignature`.

`CellSetIn` This tag represents the cell set that defines the collection of points the map will operate on. A `CellSetIn` argument expects a `vtkm::cont::CellSetStructured` object in the associated parameter of the `Invoker`. The object could also be stored in an `UnknownCellSet`, but it is an error to use any object other than `CellSetStructured`.

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldIn` This tag represents an input field that is associated with the points. A `FieldIn` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The size of the array must be exactly the number of points. Each invocation of the worklet gets a single value out of this array.

`FieldInNeighborhood` This tag represents an input field that is associated with the points. A `FieldInNeighborhood` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the `Invoker`. The size of the array must be exactly the number of points.

What differentiates `FieldInNeighborhood` from `FieldIn` is that `FieldInNeighborhood` allows the worklet function to access the field value at the point it is visiting and the field values in the neighborhood around it. Thus, instead of getting a single value out of the array, each invocation of the worklet gets a `vtkm::exec::FieldNeighborhood` object. `FieldNeighborhood` objects are described in the Neighborhood Information section starting on page 180.

`FieldOut` This tag represents an output field, which is necessarily associated with points. A `FieldOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldInOut` This tag represents field that is both an input and an output, which is necessarily associated with points. A `FieldInOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`AtomicArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `AtomicArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

`WholeCellSetIn` This tag represents the connectivity of a cell set. A `WholeCellSetIn` argument expects a `vtkm::cont::CellSet` in the associated parameter of the `Invoker`. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A point neighborhood worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`Boundary` This tag produces a `vtkm::exec::arg::BoundaryState` object, which provides information about where the local neighborhood is in relationship to the full mesh. `BoundaryState` objects are described in the Neighborhood Information section starting on page 180.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

21.3.1 Neighborhood Information

As stated earlier in this section, what makes a `WorkletPointNeighborhood` worklet special is its ability to get field information in a neighborhood surrounding a point rather than just the point itself. This is done using the special `FieldInNeighborhood ControlSignature` tag. When you use this tag, rather than getting the single field value for the point, you get a `vtkm::exec::FieldNeighborhood` object.

The `FieldNeighborhood` class (which has a single template argument of the array portal type values are stored in) contains a `Get` method that retrieves a field value relative to the local neighborhood. `FieldNeighborhood::Get` takes the i, j, k index of the point with respect to the local point. So, calling `FieldNeighborhood::Get(0,0,0)` retrieves at the point being visited. Likewise, `Get(-1,0,0)` gets the value to the "left" of the point visited and `Get(1,0,0)` gets the value to the "right." `FieldNeighborhood::Get` is overloaded to accept the index as either three separate `vtkm::IdComponent` values or a single `vtkm::Vec <vtkm::IdComponent,3>`.

Example 21.5: Retrieve neighborhood field value.

```
1 |           sum = sum + inputField.Get(i, j, k);
```

When performing operations on a neighborhood within the mesh, it is often important to know whether the expected neighborhood is contained completely within the mesh or whether the neighborhood extends beyond the borders of the mesh. This can be queried using a `vtkm::exec::BoundaryState` object, which is provided when a `Boundary` tag is listed in the `ExecutionSignature`.

Generally, `BoundaryState` allows you to specify the size of the neighborhood at runtime. The neighborhood size is specified by a *radius*. The radius specifies the number of items in each direction the neighborhood extends. So, for example, a point neighborhood with radius 1 would contain a $3 \times 3 \times 3$ neighborhood centered around the point. Likewise, a point neighborhood with radius 2 would contain a $5 \times 5 \times 5$ neighborhood centered around the point. `BoundaryState` provides several methods to determine if the neighborhood is contained in the mesh.

`MinNeighborIndices` Given a radius for the neighborhood, returns a `vtkm::Vec <vtkm::IdComponent,3>` for the "lower left" (minimum) index. If the visited point is in the middle of the mesh, the returned triplet is the negative radius for all components. But if the visited point is near the mesh boundary, then the minimum index will be clipped.

For example, if the visited point is at $[5,5,5]$ and `MinNeighborIndices(2)` is called, then $[-2,-2,-2]$ is returned. However, if the visited point is at $[0,1,2]$ and `MinNeighborIndices(2)` is called, then $[0,-1,-2]$ is returned.

`MaxNeighborIndices` Given a radius for the neighborhood, returns a `vtkm::Vec <vtkm::IdComponent,3>` for the "upper right" (maximum) index. If the visited point is in the middle of the mesh, the returned triplet is the negative radius for all components. But if the visited point is near the mesh boundary, then the maximum index will be clipped.

For example, if the visited point is at $[5,5,5]$ in a 10^3 mesh and `MaxNeighborIndices(2)` is called, then $[2,2,2]$ is returned. However, if the visited point is at $[7,8,9]$ in the same mesh and `MaxNeighborIndices(2)` is called, then $[2,1,0]$ is returned.

`InBoundary` Given a radius for the neighborhood, returns true if the neighborhood is contained completely within the boundary of the mesh, false otherwise.

`InXBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative x (I) direction, true otherwise.

`InYBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative y (J) direction, true otherwise.

`InZBoundary` Given a radius for the neighborhood, returns false if the neighborhood extends beyond the edge of the mesh in the positive or negative z (K) direction, true otherwise.

The `BoundaryState::MinNeighborIndices` and `BoundaryState::MaxNeighborIndices` are particularly useful for iterating over the valid portion of the neighborhood.

Example 21.6: Iterating over the valid portion of a neighborhood.

```
auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);

T sum = 0;
vtkm::IdComponent size = 0;
for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
{
  for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)
  {
    for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
    {
      sum = sum + inputField.Get(i, j, k);
      ++size;
    }
  }
}
```

21.3.2 Convolving Small Kernels

A common use case for point neighborhood worklets is to convolve a small kernel with a structured mesh. A very simple example of this is averaging out the values the values within some distance to the central point. This has the effect of smoothing out the field (although smoothing filters with better properties exist). The following example shows a worklet that applies this simple "box" averaging.

Example 21.7: Implementation and use of a point neighborhood worklet.

```
class ApplyBoxKernel : public vtkm::worklet::WorkletPointNeighborhood
{
private:
  vtkm::IdComponent NumberOfLayers;

public:
  using ControlSignature = void(CellSetIn cellSet,
                                FieldInNeighborhood inputField,
                                FieldOut outputField);
  using ExecutionSignature = _3(_2, Boundary);

  using InputDomain = _1;

  ApplyBoxKernel(vtkm::IdComponent kernelSize)
  {
    VTKM_ASSERT(kernelSize >= 3);
    VTKM_ASSERT((kernelSize % 2) == 1);

    this->NumberOfLayers = (kernelSize - 1) / 2;
  }

  template<typename InputFieldPortalType>
  VTKM_EXEC typename InputFieldPortalType::ValueType operator()(
    const vtkm::exec::FieldNeighborhood<InputFieldPortalType>& inputField,
    const vtkm::exec::BoundaryState& boundary) const
  {
    using T = typename InputFieldPortalType::ValueType;
```

```
28
29      auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
30      auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);
31
32      T sum = 0;
33      vtkm::IdComponent size = 0;
34      for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
35      {
36        for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)
37        {
38          for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
39          {
40            sum = sum + inputField.Get(i, j, k);
41            ++size;
42          }
43        }
44      }
45
46      return static_cast<T>(sum / size);
47    }
48  };
```

21.4 Reduce by Key

A worklet deriving `vtkm::worklet::WorkletReduceByKey` operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a `Vec`-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

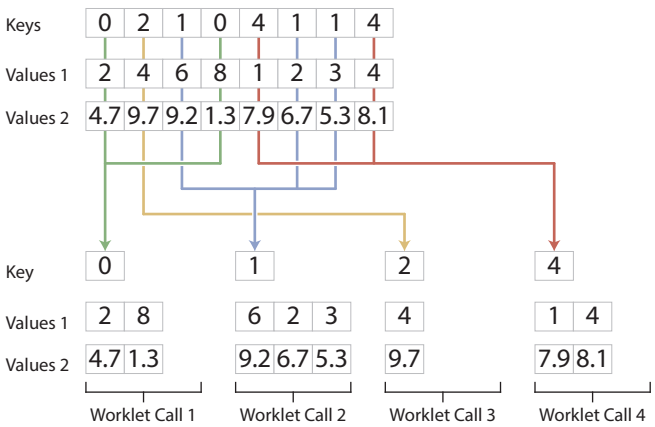


Figure 21.1: The collection of values for a reduce by key worklet.

Figure 21.1 show a pictorial representation of how VTK-m collects data for a reduce by key worklet. All calls to a reduce by key worklet has exactly one array of keys. The key array in this example has 4 unique keys: 0, 1, 2, 4. These 4 unique keys will result in 4 calls to the worklet function. This example also has 2 arrays of values associated with the keys. (A reduce by keys worklet can have any number of values arrays.)

When the worklet is invoked, all these common keys will be collected with their associated values. The parenthesis operator of the worklet will be called once per each unique key. The worklet call will be given a `Vec`-like containing

all values that have the key.

A reduce by key worklet supports the following tags in the parameters of its `ControlSignature`.

`KeysIn` This tag represents the input keys. A `KeysIn` argument expects a `vtkm::worklet::Keys` object in the associated parameter of the `Invoker`. The `Keys` object, which wraps around an `ArrayHandle` containing the keys and manages the auxiliary structures for collecting like keys, is described later in this section.

Each invocation of the worklet gets a single unique key.

A `WorkletReduceByKey` object must have exactly one `KeysIn` parameter in its `ControlSignature`, and the `InputDomain` must point to the `KeysIn` parameter.

`ValuesIn` This tag represents a set of input values that are associated with the keys. A `ValuesIn` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The number of values in this array must be equal to the size of the array used with the `KeysIn` argument. Each invocation of the worklet gets a `Vec`-like object containing all the values associated with the unique key.

`ValuesInOut` This tag behaves the same as `ValuesIn` except that the worklet may write values back into the `Vec`-like object, and these values will be placed back in their original locations in the array. Use of `ValuesInOut` is rare.

`ValuesOut` This tag behaves the same as `ValuesInOut` except that the array is resized appropriately and no input values are passed to the worklet. As with `ValuesInOut`, values the worklet writes to its `Vec`-like object get placed in the location of the original arrays. Use of `ValuesOut` is rare.

`ReducedValuesOut` This tag represents the resulting reduced values. A `ReducedValuesOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`ReducedValuesIn` This tag represents input values that come from (typically) from a previous invocation of a reduce by key. A `ReducedValuesOut` argument expects an `ArrayHandle` or an `UnknownArrayHandle` in the associated parameter of the `Invoker`. The number of values in the array must equal the number of *unique* keys.

A `ReducedValuesIn` argument is usually used to pass reduced values from one invocation of a reduce by key worklet to another invocation of a reduced by key worklet such as in an algorithm that requires iterative steps.

`ReducedValuesInOut` This tag behaves the same as `ReducedValuesIn` except that the worklet may write values back into the array. Make sure that the associated parameter to the worklet operator is a reference so that the changed value gets written back to the array.

`WholeArrayIn` This tag represents an array where all entries can be read by every worklet invocation. A `WholeArrayIn` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of reading from any place in the array is given to the worklet. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayOut` This tag represents an array where any entry can be written by any worklet invocation. A `WholeArrayOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`WholeArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `WholeArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. An

array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible. Whole arrays are discussed in detail in Section 28.1 starting on page 241.

`AtomicArrayInOut` This tag represents an array where any entry can be read or written by any worklet invocation. A `AtomicArrayInOut` argument expects an `ArrayHandle` in the associated parameter of the `Invoker`. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm. Atomic arrays are discussed in detail in Section 28.2 starting on page 243.

`WholeCellSetIn` This tag represents the connectivity of a cell set. A `WholeCellSetIn` argument expects a `vtkm::cont::CellSet` in the associated parameter of the `Invoker`. A connectivity object capable of finding elements of one type that are incident on elements of a different type. Accessing whole cell set connectivity is discussed in detail in Section 28.3.

`ExecObject` This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet. Execution objects are discussed in detail in Section 29 starting on page 249.

A reduce by key worklet supports the following tags in the parameters of its `ExecutionSignature`.

`_1`, `_2`,... These reference the corresponding parameter in the `ControlSignature`.

`ValueCount` This tag produces a `vtkm::IdComponent` that is equal to the number of times the key associated with this call to the worklet occurs in the input. This is the same size as the `Vec`-like objects provided by `ValuesIn` arguments.

`WorkIndex` This tag produces a `vtkm::Id` that uniquely identifies the invocation of the worklet.

`VisitIndex` This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter (as described in Section 31.1).

`InputIndex` This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter (as described in Section 31.1).

`OutputIndex` This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

`ThreadIndices` This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects are described in Section 43.2, but most users can get the information they need through other signature tags.

As stated earlier, the reduce by key worklet is useful for collected like values. To demonstrate the reduce by key worklet, we will create a simple mechanism to generate a histogram in parallel. (VTK-m comes with its own histogram implementation, but we create our own version here for a simple example.) The way we can use the reduce by key worklet to compute a histogram is to first identify which bin of the histogram each value is in, and then use the bin identifiers as the keys to collect the information. To help with this example, we will first create a helper class named `BinScalars` that helps us manage the bins.

Example 21.8: A helper class to manage histogram bins.

```
class BinScalars
{
public:
  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Id numBins)
    : Range(range)
    , NumBins(numBins)
  {
  }

  VTKM_EXEC_CONT
  BinScalars(const vtkm::Range& range, vtkm::Float64 tolerance)
    : Range(range)
  {
    this->NumBins = vtkm::Id(this->Range.Length() / tolerance) + 1;
  }

  VTKM_EXEC_CONT
  vtkm::Id GetBin(vtkm::Float64 value) const
  {
    vtkm::Float64 ratio = (value - this->Range.Min) / this->Range.Length();
    vtkm::Id bin = vtkm::Id(ratio * this->NumBins);
    bin = vtkm::Max(bin, vtkm::Id(0));
    bin = vtkm::Min(bin, this->NumBins - 1);
    return bin;
  }

private:
  vtkm::Range Range;
  vtkm::Id NumBins;
};
```

Using this helper class, we can easily create a simple map worklet that takes values, identifies a bin, and writes that result out to an array that can be used as keys.

Example 21.9: A simple map worklet to identify histogram bins, which will be used as keys.

```
  struct IdentifyBins : vtkm::worklet::WorkletMapField
  {
    using ControlSignature = void(FieldIn data, FieldOut bins);
    using ExecutionSignature = _2(_1);
    using InputDomain = _1;

    BinScalars Bins;

    VTKM_CONT
    IdentifyBins(const BinScalars& bins)
      : Bins(bins)
    {
    }

    VTKM_EXEC
    vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
  };
```

Once you generate an array to be used as keys, you need to make a `vtkm::worklet::Keys` object. The `Keys` object is what will be passed to the `Invoker` for the argument associated with the `KeysIn ControlSignature` tag. This of course happens in the control environment after calling the `Invoker` for our worklet for generating the keys.

Example 21.10: Creating a `vtkm::worklet::Keys` object.

```
    vtkm::cont::ArrayHandle<vtkm::Id> binIds;
```

```
  this->Invoke(IdentifyBins(bins), valuesArray, binIds);

  vtkm::worklet::Keys<vtkm::Id> keys(binIds);
```

Now that we have our keys, we are finally ready for our reduce by key worklet. A histogram is simply a count of the number of elements in a bin. In this case, we do not really need any values for the keys. We just need the size of the bin, which can be identified with the internally calculated `ValueCount`.

A complication we run into with this histogram filter is that it is possible for a bin to be empty. If a bin is empty, there will be no key associated with that bin, and the `Invoker` will not call the worklet for that bin/key. To manage this case, we have to initialize an array with 0's and then fill in the non-zero entities with our reduce by key worklet. We can find the appropriate entry into the array by using the key, which is actually the bin identifier, which doubles as an index into the histogram. The following example gives the implementation for the reduce by key worklet that fills in positive values of the histogram.

Example 21.11: A reduce by key worklet to write histogram bin counts.

```
struct CountBins : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys, WholeArrayOut binCounts);
  using ExecutionSignature = void(_1, ValueCount, _2);
  using InputDomain = _1;

  template<typename BinCountsPortalType>
  VTKM_EXEC void operator()(vtkm::Id binId,
                            vtkm::IdComponent numValuesInBin,
                            BinCountsPortalType& binCounts) const
  {
    binCounts.Set(binId, numValuesInBin);
  }
};
```

The previous example demonstrates the basic usage of the reduce by key worklet to count common keys. A more common use case is to collect values associated with those keys, do an operation on those values, and provide a "reduced" value for each unique key. The following example demonstrates such an operation by providing a worklet that finds the average of all values in a particular bin rather than counting them.

Example 21.12: A worklet that averages all values with a common key.

```
struct BinAverage : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys,
                                ValuesIn originalValues,
                                ReducedValuesOut averages);
  using ExecutionSignature = _3(_2);
  using InputDomain = _1;

  template<typename OriginalValuesVecType>
  VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
    const OriginalValuesVecType& originalValues) const
  {
    typename OriginalValuesVecType::ComponentType sum = 0;
    for (vtkm::IdComponent index = 0;
         index < originalValues.GetNumberOfComponents();
         index++)
    {
      sum = sum + originalValues[index];
    }
    return sum / originalValues.GetNumberOfComponents();
  }
};
```

To complete the code required to average all values that fall into the same bin, the following example shows the full code required to invoke such a worklet. Note that this example repeats much of the previous examples, but shows it in a more complete context.

Example 21.13: Using a reduce by key worklet to average values falling into the same bin.

```
struct IdentifyBins : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn data, FieldOut bins);
  using ExecutionSignature = _2(_1);
  using InputDomain = _1;

  BinScalars Bins;

  VTKM_CONT
  IdentifyBins(const BinScalars& bins)
    : Bins(bins)
  {
  }

  VTKM_EXEC
  vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
};

struct BinAverage : vtkm::worklet::WorkletReduceByKey
{
  using ControlSignature = void(KeysIn keys,
                                ValuesIn originalValues,
                                ReducedValuesOut averages);
  using ExecutionSignature = _3(_2);
  using InputDomain = _1;

  template<typename OriginalValuesVecType>
  VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
    const OriginalValuesVecType& originalValues) const
  {
    typename OriginalValuesVecType::ComponentType sum = 0;
    for (vtkm::IdComponent index = 0;
         index < originalValues.GetNumberOfComponents();
         index++)
    {
      sum = sum + originalValues[index];
    }
    return sum / originalValues.GetNumberOfComponents();
  }
};

//
// Later in the associated Filter class...
//

  vtkm::Range range = vtkm::cont::ArrayRangeCompute(inField).ReadPortal().Get(0);
  BinScalars bins(range, numBins);

  vtkm::cont::ArrayHandle<vtkm::Id> binIds;
  this->Invoke(IdentifyBins(bins), inField, binIds);

  vtkm::worklet::Keys<vtkm::Id> keys(binIds);

  vtkm::cont::ArrayHandle<T> combinedValues;

  this->Invoke(BinAverage{}, keys, inField, combinedValues);
```

CHAPTER

TWENTYTWO

FILTER TYPE REFERENCE

In Chapters 17 and 21 we discuss how to implement an algorithm in the VTK-m framework by creating a worklet. For simplicity, worklet algorithms are wrapped in what are called filter objects for general usage. Chapter 9 introduces the concept of filters and documents those that come with the VTK-m library. Chapter 18 gives a brief introduction on implementing filters. This chapter elaborates on building new filter objects by introducing new filter types. These will be used to wrap filters around the extended worklet examples in Chapter 21.

Unsurprisingly, the base filter objects are contained in the `vtkm::filter` package. In particular, all filter objects inherit from `vtkm::filter::Filter`, either directly or indirectly. The filter implementation must override the protected pure virtual method `Filter::DoExecute`. The base class will call this method to run the operation of the filter.

The `DoExecute` method has a single argument that is a `vtkm::cont::DataSet`. The `DataSet` contains the data on which the filter will operate. `DoExecute` must then return a new `DataSet` containing the derived data. The `DataSet` should be created with one of the `Filter::CreateResult` methods.

A filter implementation may also optionally override the `DoExecutePartitions`. This method is similar to `DoExecute` except that it takes and returns a `vtkm::cont::PartitionedDataSet` object. If a filter does not provide a `DoExecutePartitions` method, then if given a `PartitionedDataSet`, the base class will call `DoExecute` on each of the partitions and build a `PartitionedDataSet` with the results.

In addition to (or instead of) operating on the geometric structure of a `DataSet`, a filter will commonly take one or more fields from the input `DataSet` and write one or more fields to the result. For this common occurrence, `vtkm::filter::Filter` provides convenience methods to select input fields and output field names. It also provides a method named `GetFieldFromDataSet` that can be used to get the input fields from the `DataSet` passed to `DoExecute`.

Because `Filter` subclasses must read fields from the input and/or write fields to the output, `Filter` provides some convenience methods on top of those provided by `Filter`. When getting a field with `GetFieldFromDataSet`, you get a `vtkm::cont::Field` object. Before you can operate on the `Field`, you have to convert it to a `vtkm::cont::ArrayHandle`. `Filter::CastAndCallScalarField` can be used to do this conversion. It takes the field object as the first argument and attempts to convert it to an `ArrayHandle` of different types. When it finds the correct type, it calls the provided functor with the appropriate `ArrayHandle`. The similar `Filter::CastAndCallVecField` does the same thing to find an `ArrayHandle` with `vtkm::Vec` s of a selected length.

`Filter` also provides a `CreateResultField` in addition to the `CreateResult` provided by its superclass. `CreateResultField` takes the `DataSet` provided to `DoExecute` and the specification for a field that has been generated and produces a resulting `DataSet` with the same structure as the input with the provided field added.

The remainder of this chapter will provide some common patterns of filter operation based on the data they use and generate.

22.1 Deriving Fields from other Fields

A common type of filter is one that generates a new field that is derived from one or more existing fields or point coordinates on the data set. For example, mass, volume, and density are interrelated, and any one can be derived from the other two.

Filters of this nature should be implemented in classes that derive the `vtkm::filter::Filter` base class. As described previously, `Filter` provides facilities to manage input and output fields. Typically, you would use `Filter::GetFieldFromDataSet` to retrieve the input fields, one of the `Filter::CastAndCall` methods to resolve the array type of the field, and finally use `Filter::CreateResultField` to produce the output.

In this section we provide an example implementation of a field filter that wraps the "magnitude" worklet provided in Example 21.1 (listed on page 169). By C++ convention, object implementations are split into two files. The first file is a standard header file with a .h extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named FieldMagnitude.h.

Example 22.1: Header declaration for a field filter.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5  namespace vector_calculus
6  {
7
8  class VTKM_FILTER_VECTOR_CALCULUS_EXPORT FieldMagnitude : public vtkm::filter::Filter
9  {
10 public:
11   VTKM_CONT FieldMagnitude();
12
13   VTKM_CONT vtkm::cont::DataSet DoExecute(
14     const vtkm::cont::DataSet& inDataSet) override;
15 };
16
17 } // namespace vector_calculus
18 } // namespace filter
19 } // namespace vtkm
```

You may notice in Example 22.1 line 8 there is a special macro names `VTKM_FILTER_VECTOR_CALCULUS_EXPORT`. This macro tells the C++ compiler that the class `FieldMagnitude` is going to be exported from a library. More specifically, the CMake for VTK-m's build will generate a header file containing this export macro for the associated library. By VTK-m's convention, a filter in the `vtkm::filter::vector_calculus` will be defined in the vtkm/filter/vector_calculus directory and placed in a library named vtkm_filter_vector_calculus. When defining the targets for this library, CMake will create a header file named vtkm_filter_vector_calculus.h that contains the macro named `VTKM_FILTER_VECTOR_CALCULUS_EXPORT`. This macro will provide the correct modifiers for the particular C++ compiler being used to export the class from the library. If this macro is left out, then the library will work on some platforms, but on other platforms will produce a linker error for missing symbols.

Once the filter class is declared in the .h file, the implementation filter is by convention given in a separate .cxx file. So the continuation of our example that follows would be expected in a file named FieldMagnitude.cxx.

Example 22.2: Implementation of a field filter.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5  namespace vector_calculus
6  {
```

```
7
8  VTKM_CONT
9  FieldMagnitude::FieldMagnitude()
10 {
11   this->SetOutputFieldName("");
12 }
13
14 VTKM_CONT vtkm::cont::DataSet FieldMagnitude::DoExecute(
15   const vtkm::cont::DataSet& inDataSet)
16 {
17   vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
18
19   vtkm::cont::UnknownArrayHandle outField;
20
21   // Use a C++ lambda expression to provide a callback for CastAndCall. The lambda
22   // will capture references to local variables like outFieldArray (using `[&]`)
23   // that it can read and write.
24   auto resolveType = [&](const auto& inFieldArray)
25   {
26     using InArrayHandleType = std::decay_t<decltype(inFieldArray)>;
27     using ComponentType =
28       typename vtkm::VecTraits<typename InArrayHandleType::ValueType>::ComponentType;
29
30     vtkm::cont::ArrayHandle<ComponentType> outFieldArray;
31
32     this->Invoke(ComputeMagnitude{}, inFieldArray, outFieldArray);
33     outField = outFieldArray;
34   };
35
36   this->CastAndCallVecField<3>(inField, resolveType);
37
38   std::string outFieldName = this->GetOutputFieldName();
39   if (outFieldName == "")
40   {
41     outFieldName = inField.GetName() + "_magnitude";
42   }
43
44   return this->CreateResultField(
45     inDataSet, outFieldName, inField.GetAssociation(), outField);
46 }
47
48 } // namespace vector_calculus
49 } // namespace filter
50 } // namespace vtkm
```

The implementation of `DoExecute` first pulls the input field from the provided `DataSet` using `Filter::GetFieldFromDataSet`. It then uses `Filter::CastAndCallVecField` to determine what type of `ArrayHandle` is contained in the input field. That calls a lambda function that invokes a worklet to create the output field. Finally, `Filter::CreateResultField` generates the output of the filter.

Did you know?

The filter implemented in Example 22.2 is limited to only find the magnitude of `vtkm::Vec` *s with 3 components. It may be the case you wish to implement a filter that operates on* `vtkm::Vec` *s of multiple sizes (or perhaps even any size). Chapter 33 discusses how you can use the* `vtkm::cont::UnknownArrayHandle` *contained in the* `Field` *to more expressively decide what types to check for.*

Note that all fields need a unique name, which is the reason for the second argument to `CreateResult`. The `vtkm::filter::Filter` base class contains a pair of methods named `SetOutputFieldName` and `GetOutput-`

`FieldName` to allow users to specify the name of output fields. The `DoExecute` method should respect the given output field name. However, it is also good practice for the filter to have a default name if none is given. This might be simply specifying a name in the constructor, but it is worthwhile for many filters to derive a name based on the name of the input field.

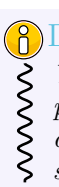
22.2 Deriving Fields from Topology

The previous example performed a simple operation on each element of a field independently. However, it is also common for a "field" filter to take into account the topology of a data set. In this case, the implementation involves pulling a `vtkm::cont::CellSet` from the input `vtkm::cont::DataSet` and performing operations on fields associated with different topological elements. The steps involve calling `DataSet::GetCellSet` to get access to the `CellSet` object and then using topology-based worklets, described in Section 21.2, to operate on them.

In this section we provide an example implementation of a field filter on cells that wraps the "cell center" worklet provided in Example 21.3 (listed on page 172).

Example 22.3: Header declaration for a field filter using cell topology.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5  namespace field_conversion
6  {
7
8  class VTKM_FILTER_FIELD_CONVERSION_EXPORT CellCenters : public vtkm::filter::Filter
9  {
10 public:
11   VTKM_CONT CellCenters();
12
13   VTKM_CONT vtkm::cont::DataSet DoExecute(
14     const vtkm::cont::DataSet& inDataSet) override;
15 };
16
17 } // namespace field_conversion
18 } // namespace filter
19 } // namespace vtkm
```



Did you know?

You may have noticed that Example 22.1 provided a specification for `SupportedTypes` but Example 22.3 provides no such specification. This demonstrates that declaring `SupportedTypes` is optional. If a filter only works on some limited number of types, then it can use `SupportedTypes` to specify the specific types it supports. But if a filter is generally applicable to many field types, it can simply use the default filter types.

As with any subclass of `Filter`, the filter implements `DoExecute`, which in this case invokes a worklet to compute a new field array and then return a newly constructed `vtkm::cont::DataSet` object.

Example 22.4: Implementation of a field filter using cell topology.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
```

```
5  namespace field_conversion
6  {
7
8  VTKM_CONT
9  CellCenters::CellCenters()
10 {
11   this->SetOutputFieldName("");
12 }
13
14 VTKM_CONT cont::DataSet CellCenters::DoExecute(const vtkm::cont::DataSet& inDataSet)
15 {
16   vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
17
18   if (!inField.IsPointField())
19   {
20     throw vtkm::cont::ErrorBadType("Cell Centers filter operates on point data.");
21   }
22
23   vtkm::cont::UnknownArrayHandle outUnknownArray;
24
25   auto resolveType = [&](const auto& inArray)
26   {
27     using InArrayHandleType = std::decay_t<decltype(inArray)>;
28     using ValueType = typename InArrayHandleType::ValueType;
29     vtkm::cont::ArrayHandle<ValueType> outArray;
30
31     this->Invoke(
32       vtkm::worklet::CellCenter{}, inDataSet.GetCellSet(), inArray, outArray);
33
34     outUnknownArray = outArray;
35   };
36
37   vtkm::cont::UnknownArrayHandle inUnknownArray = inField.GetData();
38   inUnknownArray.CastAndCallForTypesWithFloatFallback<VTKM_DEFAULT_TYPE_LIST,
39                                                       VTKM_DEFAULT_STORAGE_LIST>(
40     resolveType);
41
42   std::string outFieldName = this->GetOutputFieldName();
43   if (outFieldName == "")
44   {
45     outFieldName = inField.GetName() + "_center";
46   }
47
48   return this->CreateResultFieldCell(inDataSet, outFieldName, outUnknownArray);
49 }
50
51 } // namespace field_conversion
52 } // namespace filter
53 } // namespace vtkm
```

22.3 Data Set Filters

Sometimes, a filter will generate a data set with a new cell set based off the cells of an input data set. For example, a data set can be significantly altered by adding, removing, or replacing cells.

As with any filter, data set filters can be implemented in classes that derive the `vtkm::filter::Filter` base class and implement its `DoExecute` method.

In this section we provide an example implementation of a data set filter that wraps the functionality of extracting the edges from a data set as line elements. Many variations of implementing this functionality are given in

Chapter 32. Suffice it to say that a pair of worklets will be used to create a new `vtkm::cont::CellSet`, and this `CellSet` will be used to create the result `DataSet`. Details on how the worklets work are given in Section 32.1.

Because the operation of this edge extraction depends only on `CellSet` in a provided `DataSet`, the filter class is a simple subclass of `vtkm::filter::Filter`.

Example 22.5: Header declaration for a data set filter.

```
namespace vtkm
{
namespace filter
{
namespace entity_extraction
{

class VTKM_FILTER_ENTITY_EXTRACTION_EXPORT ExtractEdges : public vtkm::filter::Filter
{
public:
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inData) override;
};

} // namespace entity_extraction
} // namespace filter
} // namespace vtkm
```

The implementation of `DoExecute` first gets the `CellSet` and calls the worklet methods to generate a new `CellSet` class. It then uses a form of `Filter::CreateResult` to generate the resulting `DataSet`.

Example 22.6: Implementation of the `DoExecute` method of a data set filter.

```
inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
  const vtkm::cont::DataSet& inData)
{
  auto inCellSet = inData.GetCellSet();

  // Count number of edges in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);

  // Build the scatter object (for non 1-to-1 mapping of input to output)
  vtkm::worklet::ScatterCounting scatter(edgeCounts);
  auto outputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());

  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
               scatter,
               inCellSet,
               vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  vtkm::cont::CellSetSingleType<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  // This lambda function maps an input field to the output data set. It is
  // used with the CreateResult method.
  auto fieldMapper =
    [&](vtkm::cont::DataSet& outData, const vtkm::cont::Field& inputField)
  {
    if (inputField.IsCellField())
    {
      vtkm::filter::MapFieldPermutation(inputField, outputToInputCellMap, outData);
    }
    else
```

```
35      {
36        outData.AddField(inputField); // pass through
37      }
38    };
39
40    return this->CreateResult(inData, outCellSet, fieldMapper);
41  }
```

The form of `CreateResult` used (line 40) takes as input a `CellSet` to use in the generated data. In forms of `CreateResult` used in previous examples of this chapter, the cell structure of the output was created from the cell structure of the input. Because these cell structures were the same, coordinate systems and fields needed to be changed. However, because we are providing a new `CellSet`, we need to also specify how the coordinate systems and fields change.

The last two arguments to `CreateResult` are providing this information. The second-to-last argument is a `std::vector` of the `CoordinateSystem`s to use. Because this filter does not actually change the points in the data set, the `CoordinateSystem`s can just be copied over. The last argument provides a functor that maps a field from the input to the output. The functor takes two arguments: the output `DataSet` to modify and the input `Field` to map. In this example, the functor is defined as a lambda function (line 27).

Did you know?

The field mapper in Example 22.5 uses a helper function named `vtkm::filter::MapFieldPermutation`*. In the case of this example, every cell in the output comes from one cell in the input. For this common case, the values in the field arrays just need to be permuted so that each input value gets to the right output value.* `MapFieldPermutation` *will do this shuffling for you.*

VTK-m also comes with a similar helper function `vtkm::filter::MapFieldAverage` *that can be used when each output cell (or point) was constructed from multiple inputs. In this case,* `MapFieldAverage` *can do a simple average for each output value of all input values that contributed.*

Did you know?

Although not the case in this example, sometimes a filter creating a new cell set changes the points of the cells. As long as the field mapper you provide to `CreateResult` *properly converts points from the input to the output, all fields and coordinate systems will be automatically filled in the output. Sometimes when creating this new cell set you also create new point coordinates for it. This might be because the point coordinates are necessary for the computation or might be due to a faster way of computing the point coordinates. In either case, if the filter already has point coordinates computed, it can use* `CreateResultCoordinateSystem` *to use the precomputed point coordinates.*

22.4 Data Set with Field Filters

Sometimes, a filter will generate a data set with a new cell set based off the cells of an input data set along with the data in at least one field. For example, a field might determine how each cell is culled, clipped, or sliced.

In this section we provide an example implementation of a data set with field filter that blanks the cells in a data set based on a field that acts as a mask (or stencil). Any cell associated with a mask value of zero will be removed. For simplicity of this example, we will use the `Threshold` filter internally for the implementation.

Example 22.7: Header declaration for a data set with field filter.

```
namespace vtkm
{
namespace filter
{
namespace entity_extraction
{

class VTKM_FILTER_ENTITY_EXTRACTION_EXPORT BlankCells : public vtkm::filter::Filter
{
public:
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet) override;
};


} // namespace entity_extraction
} // namespace filter
} // namespace vtkm
```

The implementation of `DoExecute` first derives an array that contains a flag whether the input array value is zero or non-zero. This is simply to guarantee the range for the threshold filter. After that a threshold filter is set up and run to generate the result.

Example 22.8: Implementation of the `DoExecute` method of a data set with field filter.

```
VTKM_CONT vtkm::cont::DataSet BlankCells::DoExecute(
  const vtkm::cont::DataSet& inData)
{
  vtkm::cont::Field inField = this->GetFieldFromDataSet(inData);
  if (!inField.IsCellField())
  {
    throw vtkm::cont::ErrorBadValue("Blanking field must be a cell field.");
  }

  // Set up this array to have a 0 for any cell to be removed and
  // a 1 for any cell to keep.
  vtkm::cont::ArrayHandle<vtkm::FloatDefault> blankingArray;

  auto resolveType = [&](const auto& inFieldArray)
  {
    auto transformArray = vtkm::cont::make_ArrayHandleTransform(
      inFieldArray, vtkm::NotZeroInitialized{});
    vtkm::cont::ArrayCopyDevice(transformArray, blankingArray);
  };

  this->CastAndCallScalarField(inField, resolveType);

  // Make a temporary DataSet (shallow copy of the input) to pass blankingArray
  // to threshold.
  vtkm::cont::DataSet tempData = inData;
  tempData.AddCellField("vtkm-blanking-array", blankingArray);

  // Just use the Threshold filter to implement the actual cell removal.
  vtkm::filter::entity_extraction::Threshold thresholdFilter;
  thresholdFilter.SetLowerThreshold(0.5);
  thresholdFilter.SetUpperThreshold(2.0);
  thresholdFilter.SetActiveField("vtkm-blanking-array",
                                 vtkm::cont::Field::Association::Cells);

  // Make sure threshold filter passes all the fields requested, but not the
  // blanking array.
  thresholdFilter.SetFieldsToPass(this->GetFieldsToPass());
  thresholdFilter.SetFieldsToPass("vtkm-blanking-array",
```

```
39                                    vtkm::cont::Field::Association::Cells,
40                                    vtkm::filter::FieldSelection::Mode::Exclude);
41
42    // Use the threshold filter to generate the actual output.
43    return thresholdFilter.Execute(tempData);
44  }
```

CHAPTER

TWENTYTHREE

WORKLET ERROR HANDLING

It is sometimes the case during the execution of an algorithm that an error condition can occur that causes the computation to become invalid. At such a time, it is important to raise an error to alert the calling code of the problem. Since VTK-m uses an exception mechanism to raise errors, we want an error in the execution environment to throw an exception.

However, throwing exceptions in a parallel algorithm is problematic. Some accelerator architectures, like CUDA, do not even support throwing exceptions. Even on architectures that do support exceptions, throwing them in a thread block can cause problems. An exception raised in one thread may or may not be thrown in another, which increases the potential for deadlocks, and it is unclear how uncaught exceptions progress through thread blocks.

VTK-m handles this problem by using a flag and check mechanism. When a worklet (or other subclass of `vtkm::exec::FunctorBase`) encounters an error, it can call its `RaiseError` method to flag the problem and record a message for the error. Once all the threads terminate, the scheduler checks for the error, and if one exists it throws a `vtkm::cont::ErrorExecution` exception in the control environment. Thus, calling `RaiseError` looks like an exception was thrown from the perspective of the control environment code that invoked it.

Example 23.1: Raising an error in the execution environment.

```
1  struct SquareRoot : vtkm::worklet::WorkletMapField
2  {
3  public:
4    using ControlSignature = void(FieldIn, FieldOut);
5    using ExecutionSignature = _2(_1);
6
7    template<typename T>
8    VTKM_EXEC T operator()(T x) const
9    {
10     if (x < 0)
11     {
12       this->RaiseError("Cannot take the square root of a negative number.");
13     }
14     return vtkm::Sqrt(x);
15   }
16 };
```

It is also worth noting that the `VTKM_ASSERT` macro described in Section 11.2 also works within worklets and other code running in the execution environment. Of course, a failed assert will terminate execution rather than just raise an error so is best for checking invalid conditions for debugging purposes.

CHAPTER

TWENTYFOUR

MATH

VTK-m comes with several math functions that tend to be useful for visualization algorithms. The implementation of basic math operations can vary subtly on different accelerators, and these functions provide cross platform support.

All math functions are located in the `vtkm` package. The functions are most useful in the execution environment, but they can also be used in the control environment when needed.

24.1 Basic Math

The vtkm/Math.h header file contains several math functions that replicate the behavior of the basic POSIX math functions as well as related functionality.

Did you know?

When writing worklets, you should favor using these math functions provided by VTK-m over the standard math functions in math.h. VTK-m's implementation manages several compiling and efficiency issues when porting.

`vtkm::Abs` Returns the absolute value of the single argument. If given a vector, performs a component-wise operation.

`vtkm::ACos` Returns the arccosine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ACosH` Returns the hyperbolic arccossine. If given a vector, performs a component-wise operation.

`vtkm::ASin` Returns the arcsine of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ASinH` Returns the hyperbolic arcsine. If given a vector, performs a component-wise operation.

`vtkm::ATan` Returns the arctangent of a ratio in radians. If given a vector, performs a component-wise operation.

`vtkm::ATan2` Computes the arctangent of y/x where y is the first argument and x is the second argument. `ATan2` uses the signs of both arguments to determine the quadrant of the return value. `ATan2` is only defined for floating point types (no vectors).

`vtkm::ATanH` Returns the hyperbolic arctangent. If given a vector, performs a component-wise operation.

`vtkm::Cbrt` Takes one argument and returns the cube root of that argument. If called with a vector type, returns a component-wise cube root.

`vtkm::Ceil` Rounds and returns the smallest integer not less than the single argument. If given a vector, performs a component-wise operation.

`vtkm::CopySign` Copies the sign of the second argument onto the first argument and returns that. If the second argument is positive, returns the absolute value of the first argument. If the second argument is negative, returns the negative absolute value of the first argument.

`vtkm::Cos` Returns the cosine of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::CosH` Returns the hyperbolic cosine. If given a vector, performs a component-wise operation.

`vtkm::Epsilon` Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Epsilon32` and `Epsilon64` are non-templated versions that return the precision for a particular precision.

`vtkm::Exp` Computes e^x where x is the argument to the function and e is Euler's number (approximately 2.71828). If called with a vector type, returns a component-wise exponent.

`vtkm::Exp10` Computes 10^x where x is the argument. If called with a vector type, returns a component-wise exponent.

`vtkm::Exp2` Computes 2^x where x is the argument. If called with a vector type, returns a component-wise exponent.

`vtkm::ExpM1` Computes $e^x - 1$ where x is the argument to the function and e is Euler's number (approximately 2.71828). The accuracy of this function is good even for very small values of x. If called with a vector type, returns a component-wise exponent.

`vtkm::FloatDistance` Computes the number of representables between two floating point numbers. This function is non-negative and symmetric in its arguments. If either argument is non-finite, the value returned is the maximum value allowed by 64-bit unsigned integers: $2^{64} - 1$.

`vtkm::Floor` Rounds and returns the largest integer not greater than the single argument. If given a vector, performs a component-wise operation.

`vtkm::FMod` Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards zero to an integer. For example, `FMod(6.5,2.3)` returns 1.9, which is $6.5 - 2 \cdot 4.6$. If given vectors, `FMod` performs a component-wise operation. `FMod` is similar to `Remainder` except that the quotient is rounded toward 0 instead of the nearest integer.

`vtkm::Infinity` Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity` method is templated to specify either a 32 or 64 bit floating point number. The convenience methods `Infinity32` and `Infinity64` are non-templated versions that return the precision for a particular precision.

vtkm::IsFinite Returns true if the argument is a normal number (neither a NaN nor an infinite).

vtkm::IsInf Returns true if the argument is either positive infinity or negative infinity.

vtkm::IsNan Returns true if the argument is not a number (NaN).

vtkm::IsNegative Returns true if the single argument is less than zero, false otherwise.

vtkm::Log Computes the natural logarithm (i.e. logarithm to the base e) of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log10 Computes the logarithm to the base 10 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Log1P Computes $\ln(1+x)$ where x is the single argument and ln is the natural logarithm (i.e. logarithm to the base e). The accuracy of this function is good for very small values. If called with a vector type, returns a component-wise logarithm.

vtkm::Log2 Computes the logarithm to the base 2 of the single argument. If called with a vector type, returns a component-wise logarithm.

vtkm::Max Takes two arguments and returns the argument that is greater. If called with a vector type, returns a component-wise maximum.

vtkm::Min Takes two arguments and returns the argument that is lesser. If called with a vector type, returns a component-wise minimum.

vtkm::ModF Returns the integral and fractional parts of the first argument. The second argument is a reference in which the integral part is stored. The return value is the fractional part. If given vectors, ModF performs a component-wise operation.

vtkm::Nan Returns the representation for not-a-number (NaN). A NaN represents an invalid value or the result of an invalid operation such as 0/0. A NaN is neither greater than nor less than nor equal to any other number including other NaNs. The NaN method is templated to specify either a 32 or 64 bit floating point number. The convenience methods Nan32 and NaN64 are non-templated versions that return the precision for a particular precision.

vtkm::NegativeInfinity Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The NegativeInfinity method is templated to specify either a 32 or 64 bit floating point number. The convenience methods NegativeInfinity32 and NegativeInfinity64 are non-templated versions that return the precision for a particular precision.

vtkm::Pi Returns the constant π (about 3.14159).

vtkm::Pi_2 Returns the constant $\pi/2$ (about 1.570796).

vtkm::Pi_3 Returns the constant $\pi/3$ (about 1.047197).

vtkm::Pi_4 Returns the constant $\pi/4$ (about 0.785398).

vtkm::Pow Takes two arguments and returns the first argument raised to the power of the second argument. This function is only defined for vtkm::Float32 and vtkm::Float64.

`vtkm::QuadraticRoots` Takes the coefficients a, b, c of the quadratic equation $ax^2 + bx + c = 0$ and returns the real roots in a `vtkm::Vec` of size 2. If there are no roots, then both returned values are `NaN`s. If there are two real roots, the first element is less than or equal to the second. If compiled with FMA support, each root is accurate to 3 ulps; otherwise the discriminant is prone to catastrophic subtractive cancellation and no accuracy guarantees can be provided.

`vtkm::RCbrt` Takes one argument and returns the cube root of that argument. The result of this function is equivalent to `1/Cbrt(x)`. However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

`vtkm::Remainder` Computes the remainder on the division of 2 floating point numbers. The return value is $numerator - n \cdot denominator$, where *numerator* is the first argument, *denominator* is the second argument, and n is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer. For example, `FMod(6.5,2.3)` returns -0.4, which is $6.5 - 3 \cdot 2.3$. If given vectors, `Remainder` performs a component-wise operation. `Remainder` is similar to `FMod` except that the quotient is rounded toward the nearest integer instead of toward 0.

`vtkm::RemainderQuotient` Performs an operation identical to `Reminder`. In addition, this function takes a third argument that is a reference in which the quotient is given.

`vtkm::Round` Rounds and returns the integer nearest the single argument. If given a vector, performs a component-wise operation.

`vtkm::RSqrt` Takes one argument and returns the square root of that argument. The result of this function is equivalent to `1/Sqrt(x)`. However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

`vtkm::SignBit` Returns a nonzero value if the single argument is negative.

`vtkm::Sin` Returns the sine of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::SinH` Returns the hyperbolic sine. If given a vector, performs a component-wise operation.

`vtkm::Sqrt` Takes one argument and returns the square root of that argument. If called with a vector type, returns a component-wise square root. On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide by the square root.

`vtkm::Tan` Returns the tangent of an angle given in radians. If given a vector, performs a component-wise operation.

`vtkm::TanH` Returns the hyperbolic tangent. If given a vector, performs a component-wise operation.

`vtkm::TwoPi` Returns the constant 2π (about 6.283185).

24.2 Vector Analysis

Visualization and computational geometry algorithms often perform vector analysis operations. The vtkm/VectorAnalysis.h header file provides functions that perform the basic common vector analysis operations.

`vtkm::Cross` Returns the cross product of two `vtkm::Vec` of size 3. If compiled with FMA support, it uses Kahan's difference of products algorithm to achieve a maximum error of 1.5 ulps in each component.

vtkm::Lerp Given two values x and y in the first two parameters and a weight w as the third parameter, interpolates between x and y. Specifically, the linear interpolation is $(y-x)w+x$ although Lerp might compute the interpolation faster than using the independent arithmetic operations. The two values may be scalars or equal sized vectors. If the two values are vectors and the weight is a scalar, all components of the vector are interpolated with the same weight. If the weight is also a vector, then each component of the value vectors are interpolated with the respective weight component.

vtkm::Magnitude Returns the magnitude of a vector. This function works on scalars as well as vectors, in which case it just returns the scalar. It is usually much faster to compute MagnitudeSquared, so that should be substituted when possible (unless you are just going to take the square root, which would be besides the point). On some hardware it is also faster to find the reciprocal magnitude, so RMagnitude should be used if you actually plan to divide by the magnitude.

vtkm::MagnitudeSquared Returns the square of the magnitude of a vector. It is usually much faster to compute the square of the magnitude than the length, so you should use this function in place of Magnitude or RMagnitude when needing the square of the magnitude or any monotonically increasing function of a magnitude or distance. This function works on scalars as well as vectors, in which case it just returns the square of the scalar.

vtkm::Normal Returns a normalized version of the given vector. The resulting vector points in the same direction as the argument but has unit length.

vtkm::Normalize Takes a reference to a vector and modifies it to be of unit length. Normalize(v) is functionally equivalent to v *= RMagnitude(v).

vtkm::RMagnitude Returns the reciprocal magnitude of a vector. On some hardware RMagnitude is faster than Magnitude, but neither is as fast as MagnitudeSquared. This function works on scalars as well as vectors, in which case it just returns the reciprocal of the scalar.

vtkm::TriangleNormal Given three points in space (contained in vtkm::Vec s of size 3) that compose a triangle return a vector that is perpendicular to the triangle. The magnitude of the result is equal to twice the area of the triangle. The result points away from the "front" of the triangle as defined by the standard counter-clockwise ordering of the points.

24.3 Matrices

Linear algebra operations on small matrices that are done on a single thread are located in vtkm/Matrix.h.

This header defines the vtkm::Matrix templated class. The template parameters are first the type of component, then the number of rows, then the number of columns. The overloaded parentheses operator can be used to retrieve values based on row and column indices. Likewise, the bracket operators can be used to reference the Matrix as a 2D array (indexed by row first). The following example builds a Matrix that contains the values

$$\begin{vmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \end{vmatrix}$$

Example 24.1: Creating a Matrix.

```
1  vtkm::Matrix<vtkm::Float32, 2, 3> matrix;
2
3  // Using parenthesis notation.
4  matrix(0, 0) = 0.0f;
5  matrix(0, 1) = 1.0f;
```

```
6   matrix(0, 2) = 2.0f;
7
8   // Using bracket notation.
9   matrix[1][0] = 10.0f;
10  matrix[1][1] = 11.0f;
11  matrix[1][2] = 12.0f;
```

The vtkm/Matrix.h header also defines the following functions that operate on matrices.

`vtkm::MatrixDeterminant` Takes a square `Matrix` as its single argument and returns the determinant of that matrix.

`vtkm::MatrixGetColumn` Given a `Matrix` and a column index, returns a `vtkm::Vec` of that column. This function might not be as efficient as `vtkm::MatrixRow`. (It performs a copy of the column).

`vtkm::MatrixGetRow` Given a `Matrix` and a row index, returns a `vtkm::Vec` of that row.

`vtkm::MatrixIdentity` Returns the identity matrix. If given no arguments, it creates an identity matrix and returns it. (In this form, the component type and size must be explicitly set.) If given a single square matrix argument, fills that matrix with the identity.

`vtkm::MatrixInverse` Finds and returns the inverse of a given matrix. The function takes two arguments. The first argument is the matrix to invert. The second argument is a reference to a Boolean that is set to true if the inverse is found or false if the matrix is singular and the returned matrix is incorrect.

`vtkm::MatrixMultiply` Performs a matrix-multiply on its two arguments. Overloaded to work for matrix-matrix, vector-matrix, or matrix-vector multiply.

`vtkm::MatrixSetColumn` Given a `Matrix`, a column index, and a `vtkm::Vec`, sets the column of that index to the values of the `Tuple`.

`vtkm::MatrixSetRow` Given a `Matrix`, a row index, and a `vtkm::Vec`, sets the row of that index to the values of the `Tuple`.

`vtkm::MatrixTranspose` Takes a `Matrix` and returns its transpose.

`vtkm::SolveLinearSystem` Solves the linear system $Ax = b$ and returns x. The function takes three arguments. The first two arguments are the matrix A and the vector b, respectively. The third argument is a reference to a Boolean that is set to true if a single solution is found, false otherwise.

24.4 Newton's Method

VTK-m's matrix methods (documented in Section 24.3) provide a method to solve a small linear system of equations. However, sometimes it is necessary to solve a small nonlinear system of equations. This can be done with the `vtkm::NewtonsMethod` function defined in the vtkm/NewtonsMethod.h header.

The `NewtonsMethod` function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the `NewtonsMethod`. The function takes the following 6 parameters (three of which are optional).

1. A functor whose operation takes a `vtkm::Vec` and returns a `vtkm::Matrix` containing the math function's Jacobian vector at that point.

2. A functor whose operation takes a `vtkm::Vec` and returns the evaluation of the math function at that point as another `vtkm::Vec`.
3. The `vtkm::Vec` that represents the desired output of the function.
4. A `vtkm::Vec` to use as the initial guess. If not specified, the origin is used.
5. The convergence distance. If the iterative method changes all values less than this amount, then it considers the solution found. If not specified, set to 10^{-3}.
6. The maximum amount of iterations to run before giving up and returning the best solution. If not specified, set to 10.

The `NewtonsMethod` function returns a `vtkm::NewtonsMethodResult` object. `NewtonsMethodResult` is a `struct` templated on the type and number of input values of the nonlinear system. `NewtonsMethodResult` contains the following items.

`Valid` A `bool` that is set to false if the solution runs into a singularity so that no possible solution is found.

`Converged` A `bool` that is set to true if a solution is found that is within the convergence distance specified. It is set to false if the method did not convert in the specified number of iterations.

`Solution` A `vtkm::Vec` containing the solution to the nonlinear system. If `Converged` is false, then this value is likely inaccurate. If `Valid` is false, then this value is undefined.

Example 24.2: Using `NewtonsMethod` to solve a small system of nonlinear equations.

```
// A functor for the mathematical function f(x) = [dot(x,x),x[0]*x[1]]
struct FunctionFunctor
{
  template<typename T>
  VTKM_EXEC_CONT vtkm::Vec<T, 2> operator()(const vtkm::Vec<T, 2>& x) const
  {
    return vtkm::make_Vec(vtkm::Dot(x, x), x[0] * x[1]);
  }
};

// A functor for the Jacobian of the mathematical function
// f(x) = [dot(x,x),x[0]*x[1]], which is
//   | 2*x[0] 2*x[1] |
//   |   x[1]   x[0] |
struct JacobianFunctor
{
  template<typename T>
  VTKM_EXEC_CONT vtkm::Matrix<T, 2, 2> operator()(const vtkm::Vec<T, 2>& x) const
  {
    vtkm::Matrix<T, 2, 2> jacobian;
    jacobian(0, 0) = 2 * x[0];
    jacobian(0, 1) = 2 * x[1];
    jacobian(1, 0) = x[1];
    jacobian(1, 1) = x[0];

    return jacobian;
  }
};

VTKM_EXEC
void SolveNonlinear()
{
  // Use Newton's method to solve the nonlinear system of equations:
```

```
34    //
35    //    x^2 + y^2 = 2
36    //    x*y = 1
37    //
38    // There are two possible solutions, which are (x=1,y=1) and (x=-1,y=-1).
39    // The one found depends on the starting value.
40    vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer1 =
41      vtkm::NewtonsMethod(JacobianFunctor(),
42                          FunctionFunctor(),
43                          vtkm::make_Vec(2.0f, 1.0f),
44                          vtkm::make_Vec(1.0f, 0.0f));
45    if (!answer1.Valid || !answer1.Converged)
46    {
47      // Failed to find solution
48    }
49    // answer1.Solution is [1,1]
50
51    vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer2 =
52      vtkm::NewtonsMethod(JacobianFunctor(),
53                          FunctionFunctor(),
54                          vtkm::make_Vec(2.0f, 1.0f),
55                          vtkm::make_Vec(0.0f, -2.0f));
56    if (!answer2.Valid || !answer2.Converged)
57    {
58      // Failed to find solution
59    }
60    // answer2 is [-1,-1]
61  }
```

CHAPTER

TWENTYFIVE

WORKING WITH CELLS

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See Section 7.2 starting on page 36 for information on defining cell sets in the control environment.) When worklets that operate on cells are scheduled, these grid structures are broken into their independent cells, and that data is handed to the worklet. Thus, cell-based operations in the execution environment exclusively operate on independent cells.

Unlike some other libraries such as VTK, VTK-m does not have a cell class that holds all the information pertaining to a cell of a particular type. Instead, VTK-m provides tags or identifiers defining the cell shape, and companion data like coordinate and field information are held in separate structures. This organization is designed so a worklet may specify exactly what information it needs, and only that information will be loaded.

25.1 Cell Shape Tags and Ids

Cell shapes can be specified with either a tag (defined with a struct with a name like `CellShapeTag*`) or an enumerated identifier (defined with a constant number with a name like `CELL_SHAPE_*`). These shape tags and identifiers are defined in vtkm/CellShape.h and declared in the `vtkm` namespace (because they can be used in either the control or the execution environment). Figure 25.1 gives both the identifier and the tag names.

In addition to the basic cell shapes, there is a special "empty" cell with the identifier `vtkm::CELL_SHAPE_EMPTY` and tag `vtkm::CellShapeTagEmpty`. This type of cell has no points, edges, or faces and can be thought of as a placeholder for a null or void cell.

There is also a special cell shape "tag" named `vtkm::CellShapeTagGeneric` that is used when the actual cell shape is not known at compile time. `CellShapeTagGeneric` actually has a member variable named `Id` that stores the identifier for the cell shape. There is no equivalent identifier for a generic cell; cell shape identifiers can be placed in a `vtkm::IdComponent` at runtime.

When using cell shapes in templated classes and functions, you can use the `VTKM_IS_CELL_SHAPE_TAG` to ensure a type is a valid cell shape tag. This macro takes one argument and will produce a compile error if the argument is not a cell shape tag type.

25.1.1 Converting Between Tags and Identifiers

Every cell shape tag has a member variable named `Id` that contains the identifier for the cell shape. This provides a convenient mechanism for converting a cell shape tag to an identifier. Most cell shape tags have their `Id` member as a compile-time constant, but `CellShapeTagGeneric` is set at run time.

vtkm/CellShape.h also declares a templated class named `vtkm::CellShapeIdToTag` that converts a cell shape

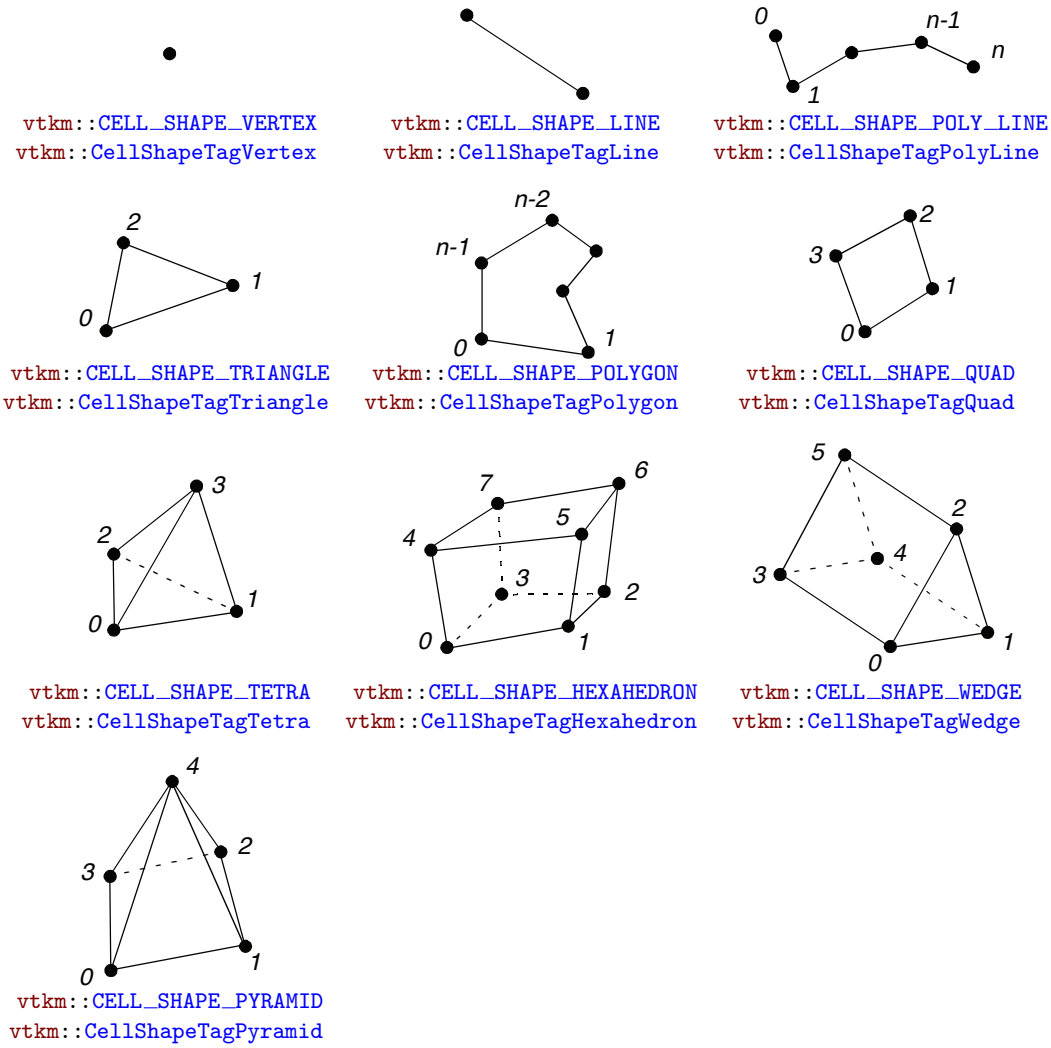


Figure 25.1: Basic Cell Shapes

identifier to a cell shape tag. `CellShapeIdToTag` has a single template argument that is the identifier. Inside the class is a type named `Tag` that is the type of the correct tag.

Example 25.1: Using `CellShapeIdToTag`.

```
1  void CellFunction(vtkm::CellShapeTagTriangle)
2  {
3    std::cout << "In CellFunction for triangles." << std::endl;
4  }
5
6  void DoSomethingWithACell()
7  {
8    // Calls CellFunction overloaded with a vtkm::CellShapeTagTriangle.
9    CellFunction(vtkm::CellShapeIdToTag<vtkm::CELL_SHAPE_TRIANGLE>::Tag());
10 }
```

However, `CellShapeIdToTag` is only viable if the identifier can be resolved at compile time. In the case where a cell identifier is stored in a variable or an array or the code is using a `CellShapeTagGeneric`, the correct cell

shape is not known at run time. In this case, vtkmGenericCellShapeMacro can be used to check all possible conditions. This macro is embedded in a switch statement where the condition is the cell shape identifier. vtkmGenericCellShapeMacro has a single argument, which is an expression to be executed. Before the expression is executed, a type named CellShapeTag is defined as the type of the appropriate cell shape tag. Often this method is used to implement the condition for a CellShapeTagGeneric in a function overloaded for cell types. A demonstration of vtkmGenericCellShapeMacro is given in Example 25.2.

25.1.2 Cell Traits

The vtkm/CellTraits.h header file contains a traits class named vtkm::CellTraits that provides information about a cell. Each specialization of CellTraits contains the following members.

TOPOLOGICAL_DIMENSIONS Defines the topological dimensions of the cell type. This is 3 for polyhedra, 2 for polygons, 1 for lines, and 0 for points.

TopologicalDimensionsTag A type set to either vtkm::CellTopologicalDimensionsTag <3>, CellTopologicalDimensionsTag<2>, CellTopologicalDimensionsTag<1>, or CellTopologicalDimensionsTag<0>. The number is consistent with TOPOLOGICAL_DIMENSIONS. This tag is provided for convenience when specializing functions.

IsSizeFixed Set to either vtkm::CellTraitsTagSizeFixed for cell types with a fixed number of points (for example, triangle) or vtkm::CellTraitsTagSizeVariable for cell types with a variable number of points (for example, polygon).

NUM_POINTS A vtkm::IdComponent set to the number of points in the cell. This member is only defined when there is a constant number of points (i.e. IsSizeFixed is set to vtkm::CellTraitsTagSizeFixed).

Example 25.2: Using CellTraits to implement a polygon normal estimator.

```
namespace detail
{

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector, typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
  const PointCoordinatesVector& pointCoordinates,
  vtkm::CellTopologicalDimensionsTag<2>,
  const WorkletType& worklet)
{
  if (pointCoordinates.GetNumberOfComponents() >= 3)
  {
    return vtkm::TriangleNormal(
      pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
  }
  else
  {
    worklet.RaiseError("Degenerate polygon.");
    return typename PointCoordinatesVector::ComponentType();
  }
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PointCoordinatesVector,
         vtkm::IdComponent Dimensions,
         typename WorkletType>
VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
  const PointCoordinatesVector&,
```

```
29    vtkm::CellTopologicalDimensionsTag<Dimensions>,
30    const WorkletType& worklet)
31  {
32    worklet.RaiseError("Only polygons supported for cell normals.");
33    return typename PointCoordinatesVector::ComponentType();
34  }
35
36  } // namespace detail
37
38  VTKM_SUPPRESS_EXEC_WARNINGS
39  template<typename CellShape, typename PointCoordinatesVector, typename WorkletType>
40  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
41    CellShape,
42    const PointCoordinatesVector& pointCoordinates,
43    const WorkletType& worklet)
44  {
45    return detail::CellNormalImpl(
46      pointCoordinates,
47      typename vtkm::CellTraits<CellShape>::TopologicalDimensionsTag(),
48      worklet);
49  }
50
51  VTKM_SUPPRESS_EXEC_WARNINGS
52  template<typename PointCoordinatesVector, typename WorkletType>
53  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
54    vtkm::CellShapeTagGeneric shape,
55    const PointCoordinatesVector& pointCoordinates,
56    const WorkletType& worklet)
57  {
58    switch (shape.Id)
59    {
60      vtkmGenericCellShapeMacro(
61        return CellNormal(CellShapeTag(), pointCoordinates, worklet));
62      default:
63        worklet.RaiseError("Unknown cell type.");
64        return typename PointCoordinatesVector::ComponentType();
65    }
66  }
```

25.2 Parametric and World Coordinates

Each cell type supports a one-to-one mapping between a set of parametric coordinates in the unit cube (or some subset of it) and the points in 3D space that are the locus contained in the cell. Parametric coordinates are useful because certain features of the cell, such as vertex location and center, are at a consistent location in parametric space irrespective of the location and distortion of the cell in world space. Also, many field operations are much easier with parametric coordinates.

The vtkm/exec/ParametricCoordinates.h header file contains the following functions for working with parametric coordinates.

`vtkm::exec::ParametricCoordinatesCenter` Returns the parametric coordinates for the center of a given shape. It takes 3 arguments: the number of points in the cell, a shape tag, and a `vtkm::Vec` of size 3 to store the results. An `vtkm::ErrorCode` is returned (see Section 19.9).

`vtkm::exec::ParametricCoordinatesPoint` Returns the parametric coordinates for a given point of a given shape. It takes 4 arguments: the number of points in the cell, the index of the point to query, a shape tag, and a `vtkm::Vec` of size 3 to store the results. An `vtkm::ErrorCode` is returned (see Section 19.9).

`vtkm::exec::ParametricCoordinatesToWorldCoordinates` Converts parametric coordinates (coordinates relative to the cell) to world coordinates (coordinates in the global system). It takes a vector of point coordinates (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag, and a `vtkm::Vec` of size 3 to store the resulting world coordinates. An `vtkm::ErrorCode` is returned (see Section 19.9).

`vtkm::exec::WorldCoordinatesToParametricCoordinates` Converts world coordinates (coordinates in the global system) to parametric coordinates (coordinates relative to the cell). It takes a vector of point coordinates (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing world coordinates, a shape tag, and a `vtkm::Vec` of size 3 to store the resulting parametric coordinates. An `vtkm::ErrorCode` is returned (see Section 19.9). This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

25.3 Interpolation

The shape of every cell is defined by the connections of some finite set of points. Field values defined on those points can be interpolated to any point within the cell to estimate a continuous field.

The vtkm/exec/CellInterpolate.h header contains the function `vtkm::exec::CellInterpolate` that takes a vector of point field values (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag. The result of the interpolation is placed in the reference passed as the fourth argument. `CellInterpolate` returns an `vtkm::ErrorCode` for the status of the operation.

Example 25.3: Interpolating field values to a cell's center.

```
1  struct CellCenters : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn,
4                                  FieldInPoint inputField,
5                                  FieldOutCell outputField);
6    using ExecutionSignature = void(CellShape, PointCount, _2, _3);
7    using InputDomain = _1;
8
9    template<typename CellShapeTag, typename FieldInVecType, typename FieldOutType>
10   VTKM_EXEC void operator()(CellShapeTag shape,
11                             vtkm::IdComponent pointCount,
12                             const FieldInVecType& inputField,
13                             FieldOutType& outputField) const
14   {
15     vtkm::Vec3f center;
16     vtkm::ErrorCode status =
17       vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, center);
18     if (status != vtkm::ErrorCode::Success)
19     {
20       this->RaiseError(vtkm::ErrorString(status));
21       return;
22     }
23     vtkm::exec::CellInterpolate(inputField, center, shape, outputField);
24   }
25 };
```

25.4 Derivatives

Since interpolations provide a continuous field function over a cell, it is reasonable to consider the derivative of this function. The vtkm/exec/CellDerivative.h header contains the function `vtkm::exec::CellDerivative` that

takes a vector of scalar point field values (usually given by a `FieldPointIn` worklet argument), a `vtkm::Vec` of size 3 containing parametric coordinates, a shape tag, and a worklet object (for raising errors). It returns the field derivative at the location represented by the given parametric coordinates. The derivative is return in a `vtkm::Vec` of size 3 corresponding to the partial derivatives in the x, y, and z directions. This derivative is equivalent to the gradient of the field.

Example 25.4: Computing the derivative of the field at cell centers.

```
struct CellDerivatives : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn,
                                FieldInPoint inputField,
                                FieldInPoint pointCoordinates,
                                FieldOutCell outputField);
  using ExecutionSignature = void(CellShape, PointCount, _2, _3, _4);
  using InputDomain = _1;

  template<typename CellShapeTag,
           typename FieldInVecType,
           typename PointCoordVecType,
           typename FieldOutType>
  VTKM_EXEC void operator()(CellShapeTag shape,
                            vtkm::IdComponent pointCount,
                            const FieldInVecType& inputField,
                            const PointCoordVecType& pointCoordinates,
                            FieldOutType& outputField) const
  {
    vtkm::Vec3f center;
    vtkm::ErrorCode status =
      vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, center);
    if (status != vtkm::ErrorCode::Success)
    {
      this->RaiseError(vtkm::ErrorString(status));
      return;
    }
    vtkm::exec::CellDerivative(
      inputField, pointCoordinates, center, shape, outputField);
  }
};
```

25.5 Edges and Faces

As explained earlier in this chapter, a cell is defined by a collection of points and a shape identifier that describes how the points come together to form the structure of the cell. The cell shapes supported by VTK-m are documented in Section 25.1. It contains Figure 25.1 on page 210, which shows how the points for each shape form the structure of the cell.

Most cell shapes can be broken into subelements. 2D and 3D cells have pairs of points that form *edges* at the boundaries of the cell. Likewise, 3D cells have loops of edges that form *faces* that encase the cell. Figure 25.2 demonstrates the relationship of these constituent elements for some example cell shapes.

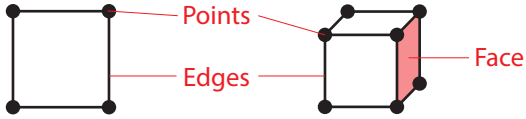


Figure 25.2: The constituent elements (points, edges, and faces) of cells.

The header file vtkm/exec/CellEdge.h contains a collection of functions to help identify the edges of a cell. The first such function is vtkm::exec::CellEdgeNumberOfEdges. This function takes the number of points in the cell, the shape of the cell, and places the number of edges the cell as in its third argument (which is a reference to a vtkm::IdComponent). The function returns a vtkm::ErrorCode (section 19.9) for the status of the operation.

The second function is vtkm::exec::CellEdgeLocalIndex. This function takes, respectively, the number of points, the local index of the point in the edge (0 or 1), the local index of the edge (0 to the number of edges in the cell), the shape of the cell, and a reference to a vtkm::IdComponent to put the result. The result is the local index (between 0 and the number of points in the cell) of the requested point in the edge. This local point index is consistent with the point labels in Figure 25.2. To get the point indices relative to the data set, the edge indices should be used to reference a PointIndices list. The function returns a vtkm::ErrorCode (section 19.9) for the status of the operation.

The third function is vtkm::exec::CellEdgeCanonicalId. This function takes the number of points, the local index of the edge, the shape of the cell, a Vec-like containing the global id of each cell point, and a reference to a vtkm::Id2 to put the result. The result is a pair of numbers that is globally unique to that edge. If CellEdgeCanonicalId is called on an edge for a different cell, the two will be the same if and only if the two cells share that edge. CellEdgeCanonicalId is useful for finding coincident components of topology.

The following example demonstrates a pair of worklets that use the cell edge functions. As is typical for operations of this nature, one worklet counts the number of edges in each cell and another uses this count to generate the data.

Did you know?

Example 25.5 demonstrates one of many techniques for creating cell sets in a worklet. Chapter 32 describes this and many more such techniques.

Example 25.5: Using cell edge functions.

```
1  struct EdgesCount : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn, FieldOutCell numEdgesInCell);
4    using ExecutionSignature = void(CellShape, PointCount, _2);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC void operator()(CellShapeTag cellShape,
9                              vtkm::IdComponent numPointsInCell,
10                             vtkm::IdComponent& numEdges) const
11   {
12     vtkm::ErrorCode status =
13       vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
14     if (status != vtkm::ErrorCode::Success)
15     {
16       this->RaiseError(vtkm::ErrorString(status));
17     }
18   }
19 };
20
21 struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
22 {
23   using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
24   using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
25   using InputDomain = _1;
26
27   using ScatterType = vtkm::worklet::ScatterCounting;
28
```

```
29    template<typename CellShapeTag,
30             typename PointIndexVecType,
31             typename EdgeIndexVecType>
32    VTKM_EXEC void operator()(CellShapeTag cellShape,
33                              const PointIndexVecType& globalPointIndicesForCell,
34                              vtkm::IdComponent edgeIndex,
35                              EdgeIndexVecType& edgeIndices) const
36    {
37      vtkm::IdComponent numPointsInCell =
38        globalPointIndicesForCell.GetNumberOfComponents();
39
40      vtkm::IdComponent pointInCellIndex0;
41      vtkm::exec::CellEdgeLocalIndex(
42        numPointsInCell, 0, edgeIndex, cellShape, pointInCellIndex0);
43      vtkm::IdComponent pointInCellIndex1;
44      vtkm::exec::CellEdgeLocalIndex(
45        numPointsInCell, 1, edgeIndex, cellShape, pointInCellIndex1);
46
47      edgeIndices[0] = globalPointIndicesForCell[pointInCellIndex0];
48      edgeIndices[1] = globalPointIndicesForCell[pointInCellIndex1];
49    }
50  };
```

The header file vtkm/exec/CellFace.h contains a collection of functions to help identify the faces of a cell. The first such function is `vtkm::exec::CellFaceNumberOfFaces`. This function takes the shape of the cell and places the number of faces in the cell as its second argument (which is a reference to a `vtkm::IdComponent`. The function returns a `vtkm::ErrorCode` (section 19.9) for the status of the operation.

The second function is `vtkm::exec::CellFaceNumberOfPoints`. This function takes the local index of the face (0 to the number of faces in the cell), the shape of the cell, and a reference to a `vtkm::IdComponent` to put the result. The result is the number of points the specified face has. The function returns a `vtkm::ErrorCode` (section 19.9) for the status of the operation.

The third function is `vtkm::exec::CellFaceLocalIndex`. This function takes, respectively, the local index of the point in the face (0 to the number of points in the face), the local index of the face (0 to the number of faces in the cell), the shape of the cell, and a reference to a `vtkm::IdComponent` to put the result. The result is the local index (between 0 and the number of points in the cell) of the requested point in the face. The points are indexed in counterclockwise order when viewing the face from the outside of the cell. This local point index is consistent with the point labels in Figure 25.2. To get the point indices relative to the data set, the face indices should be used to reference a `PointIndices` list. The function returns a `vtkm::ErrorCode` (section 19.9) for the status of the operation.

The fourth function is `vtkm::exec::CellFaceCanonicalId`. This function takes the local index of the face, the shape of the cell, a `Vec`-like containing the global id of each cell point, and a reference to a `vtkm::Id3` to put the result. The result is a triplet of numbers that is globally unique to that face. If `CellFaceCanonicalId` is called on a face for a different cell, the two will be the same if and only if the two cells share that face. `CellFaceCanonicalId` is useful for finding coincident components of topology.

The following example demonstrates a triple of worklets that use the cell face functions. As is typical for operations of this nature, the worklets are used in steps to first count entities and then generate new entities. In this case, the first worklet counts the number of faces and the second worklet counts the points in each face. The third worklet generates cells for each face.

Example 25.6: Using cell face functions.

```
1   struct FacesCount : vtkm::worklet::WorkletVisitCellsWithPoints
2   {
3     using ControlSignature = void(CellSetIn, FieldOutCell numFacesInCell);
4     using ExecutionSignature = void(CellShape, _2);
5     using InputDomain = _1;
```

```
6
7       template<typename CellShapeTag>
8       VTKM_EXEC void operator()(CellShapeTag cellShape,
9                                 vtkm::IdComponent& numFaces) const
10      {
11        vtkm::ErrorCode status =
12          vtkm::exec::CellFaceNumberOfFaces(cellShape, numFaces);
13        if (status != vtkm::ErrorCode::Success)
14        {
15          this->RaiseError(vtkm::ErrorString(status));
16        }
17      }
18    };
19
20    struct FacesCountPoints : vtkm::worklet::WorkletVisitCellsWithPoints
21    {
22      using ControlSignature = void(CellSetIn,
23                                    FieldOutCell numPointsInFace,
24                                    FieldOutCell faceShape);
25      using ExecutionSignature = void(CellShape, VisitIndex, _2, _3);
26      using InputDomain = _1;
27
28      using ScatterType = vtkm::worklet::ScatterCounting;
29
30      template<typename CellShapeTag>
31      VTKM_EXEC void operator()(CellShapeTag cellShape,
32                                vtkm::IdComponent faceIndex,
33                                vtkm::IdComponent& numPointsInFace,
34                                vtkm::UInt8& faceShape) const
35      {
36        vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, numPointsInFace);
37        switch (numPointsInFace)
38        {
39          case 3:
40            faceShape = vtkm::CELL_SHAPE_TRIANGLE;
41            break;
42          case 4:
43            faceShape = vtkm::CELL_SHAPE_QUAD;
44            break;
45          default:
46            faceShape = vtkm::CELL_SHAPE_POLYGON;
47            break;
48        }
49      }
50    };
51
52    struct FacesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
53    {
54      using ControlSignature = void(CellSetIn, FieldOutCell faceIndices);
55      using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
56      using InputDomain = _1;
57
58      using ScatterType = vtkm::worklet::ScatterCounting;
59
60      template<typename CellShapeTag,
61               typename PointIndexVecType,
62               typename FaceIndexVecType>
63      VTKM_EXEC void operator()(CellShapeTag cellShape,
64                                const PointIndexVecType& globalPointIndicesForCell,
65                                vtkm::IdComponent faceIndex,
66                                FaceIndexVecType& faceIndices) const
67      {
68        vtkm::IdComponent numPointsInFace = faceIndices.GetNumberOfComponents();
69        for (vtkm::IdComponent pointInFaceIndex = 0;
```

```
70            pointInFaceIndex < numPointsInFace;
71            pointInFaceIndex++)
72        {
73          vtkm::IdComponent pointInCellIndex;
74          vtkm::exec::CellFaceLocalIndex(
75            pointInFaceIndex, faceIndex, cellShape, pointInCellIndex);
76          faceIndices[pointInFaceIndex] = globalPointIndicesForCell[pointInCellIndex];
77        }
78      }
79    };
```

CHAPTER

TWENTYSIX

FANCY ARRAY HANDLES

One of the features of using `ArrayHandle`s is that they hide the implementation and layout of the array behind a generic interface. This gives us the opportunity to replace a simple C array with some custom definition of the data and the code using the `ArrayHandle` is none the wiser.

This gives us the opportunity to implement *fancy* arrays that do more than simply look up a value in an array. For example, arrays can be augmented on the fly by mutating their indices or values. Or values could be computed directly from the index so that no storage is required for the array at all.

VTK-m provides many of the fancy arrays, which we explore in this section. Later in Chapter 36 we explore how to create custom arrays that adapt new memory layouts or augment other types of arrays.

Did you know?

One of the advantages of VTK-m's implementation of fancy arrays is that they can define whole arrays without actually storing and values. For example, `ArrayHandleConstant`*,* `ArrayHandleCounting`*, and* `ArrayHandleIndex` *do not store data in any array in memory. Rather, they construct the value for an index at runtime. Likewise, arrays like* `ArrayHandlePermutation` *construct new arrays from the values of other arrays without having to create a copy of the data.*

26.1 Constant Arrays

A constant array is a fancy array handle that has the same value in all of its entries. The constant array provides this array without actually using any memory.

Specifying a constant array in VTK-m is straightforward. VTK-m has a class named `vtkm::cont::ArrayHandleConstant`. `ArrayHandleConstant` is a templated class with a single template argument that is the type of value for each element in the array. The constructor for `ArrayHandleConstant` takes the value to provide by the array and the number of values the array should present. The following example is a simple demonstration of the constant array handle.

Example 26.1: Using `ArrayHandleConstant`.

```
1  // Create an array of 50 entries, all containing the number 3. This could be
2  // used, for example, to represent the sizes of all the polygons in a set
3  // where we know all the polygons are triangles.
4  vtkm::cont::ArrayHandleConstant<vtkm::Id> constantArray(3, 50);
```

The vtkm/cont/ArrayHandleConstant.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleConstant that takes a value and a size for the array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.2: Using make_ArrayHandleConstant.

```
1   // Create an array of 50 entries, all containing the number 3.
2   vtkm::cont::make_ArrayHandleConstant(3, 50)
```

26.2 ArrayHandleView

An array handle view is a fancy array handle that returns a subset of an already existing array handle. The array handle view uses the same memory as the existing array handle the view was created from. This means that changes to the data in the array handle view will also change the data in the original array handle.

To use the ArrayHandleView you must supply an ArrayHandle to the vtkm::cont::ArrayHandleView class constructor. ArrayHandleView is a templated class with a single template argument that is the ArrayHandle type of the array that the view is being created from. The constructor for ArrayHandleView takes a target array, starting index, and length. The following example shows a simple usage of the array handle view.

Example 26.3: Using ArrayHandleView.

```
1  vtkm::cont::ArrayHandle<vtkm::Id> sourceArray;
2  vtkm::cont::ArrayCopy(vtkm::cont::ArrayHandleIndex(10), sourceArray);
3  // sourceArray has [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4
5  vtkm::cont::ArrayHandleView<vtkm::cont::ArrayHandle<vtkm::Id>> viewArray(
6    sourceArray, 3, 5);
7  // viewArray has [3, 4, 5, 6, 7]
```

The vtkm/cont/ArrayHandleView.h header contains a templated convenience function vtkm::cont::make_ArrayHandleView that takes a target array, index, and length.

Example 26.4: Using make_ArrayHandleView.

```
1   vtkm::cont::make_ArrayHandleView(sourceArray, 3, 5)
```

26.3 Uniform Random Bits Array

An uniform random bits array is a fancy array handle that generates pseudo random bits as vtkm::Unit64 in its entries. The uniform random bits array provides this array without actually using any memory.

The constructor for ArrayHandleRandomUniformBits takes two arguments: the first argument is the length of the array handle, the second is a seed of type vtkm::Vec<Uint32, 1>. If the seed is not specified, the C++11 std::random_device is used as default.

Example 26.5: Using ArrayHandleRandomUniformBits.

```
1  // Create an array containing a sequence of random bits seeded
2  // by std::random_device.
3  vtkm::cont::ArrayHandleRandomUniformBits randomArray(50);
4  // Create an array containing a sequence of random bits with
5  // a user supplied seed.
6  vtkm::cont::ArrayHandleRandomUniformBits randomArraySeeded(50, { 123 });
```

ArrayHandleRandomUniformBits is functional, in the sense that once an instance of ArrayHandleRandomUniformBits is created, its content does not change and always returns the same vtkm::UInt64 value given the same index.

Example 26.6: ArrayHandleRandomUniformBits is functional

```
// ArrayHandleRandomUniformBits is functional, it returns
// the same value for the same entry is accessed.
auto r0 = randomArray.ReadPortal().Get(5);
auto r1 = randomArray.ReadPortal().Get(5);
assert(r0 == r1);
```

To generate a new set of random bits, we need to create another instance of ArrayHandleRandomUniformBits with a different seed, we can either let std::random_device provide a unique seed or use some unique identifier such as iteration number as the seed.

Example 26.7: Independent ArrayHandleRandomUniformBits.

```
// Create a new insance of ArrayHandleRandomUniformBits
// for each set of random bits.
vtkm::cont::ArrayHandleRandomUniformBits randomArray0(50, { 0 });
vtkm::cont::ArrayHandleRandomUniformBits randomArray1(50, { 1 });
assert(randomArray0.ReadPortal().Get(5) != randomArray1.ReadPortal().Get(5));
```

26.4 Counting Arrays

A counting array is a fancy array handle that provides a sequence of numbers. These fancy arrays can represent the data without actually using any memory.

VTK-m provides two versions of a counting array. The first version is an index array that provides a specialized but common form of a counting array called an index array. An index array has values of type vtkm::Id that start at 0 and count up by 1 (i.e. $0, 1, 2, 3, \ldots$). The index array mirrors the array's index.

Specifying an index array in VTK-m is done with a class named vtkm::cont::ArrayHandleIndex. The constructor for ArrayHandleIndex takes the size of the array to create. The following example is a simple demonstration of the index array handle.

Example 26.8: Using ArrayHandleIndex.

```
// Create an array containing [0, 1, 2, 3, ..., 49].
vtkm::cont::ArrayHandleIndex indexArray(50);
```

The vtkm::cont::ArrayHandleCounting class provides a more general form of counting. ArrayHandleCounting is a templated class with a single template argument that is the type of value for each element in the array. The constructor for ArrayHandleCounting takes three arguments: the start value (used at index 0), the step from one value to the next, and the length of the array. The following example is a simple demonstration of the counting array handle.

Example 26.9: Using ArrayHandleCounting.

```
// Create an array containing [-1.0, -0.9, -0.8, ..., 0.9, 1.0]
vtkm::cont::ArrayHandleCounting<vtkm::Float32> sampleArray(-1.0f, 0.1f, 21);
```

Did you know?

In addition to being simpler to declare, `ArrayHandleIndex` *is slightly faster than* `ArrayHandleCounting`*. Thus, when applicable, you should prefer using* `ArrayHandleIndex`*.*

The vtkm/cont/ArrayHandleCounting.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandleCounting` that also takes the start value, step, and length as arguments. This function can sometimes be used to avoid having to declare the full array type.

Example 26.10: Using `make_ArrayHandleCounting`.

```
// Create an array containing [-1.0, -0.9, -0.8, ..., 0.9, 1.0]
vtkm::cont::make_ArrayHandleCounting(-1.0f, 0.1f, 21)
```

There are no fundamental limits on how `ArrayHandleCounting` counts. For example, it is possible to count backwards.

Example 26.11: Counting backwards with `ArrayHandleCounting`.

```
// Create an array containing [49, 48, 47, 46, ..., 0].
vtkm::cont::ArrayHandleCounting<vtkm::Id> backwardIndexArray(49, -1, 50);
```

It is also possible to use `ArrayHandleCounting` to make sequences of `vtkm::Vec` values with piece-wise counting in each of the components.

Example 26.12: Using `ArrayHandleCounting` with `vtkm::Vec` objects.

```
// Create an array containg [(0,-3,75), (1,2,25), (3,7,-25)]
vtkm::cont::make_ArrayHandleCounting(
  vtkm::make_Vec(0, -3, 75), vtkm::make_Vec(1, 5, -50), 3)
```

26.5 Cast Arrays

A cast array is a fancy array that changes the type of the elements in an array. The cast array provides this re-typed array without actually copying or generating any data. Instead, casts are performed as the array is accessed.

VTK-m has a class named `vtkm::cont::ArrayHandleCast` to perform this implicit casting. `ArrayHandleCast` is a templated class with two template arguments. The first argument is the type to cast values to. The second argument is the type of the original `ArrayHandle`. The constructor to `ArrayHandleCast` takes the `ArrayHandle` to modify by casting.

Example 26.13: Using `ArrayHandleCast`.

```
template<typename T>
VTKM_CONT void Foo(const std::vector<T>& inputData)
{
  vtkm::cont::ArrayHandle<T> originalArray =
    vtkm::cont::make_ArrayHandle(inputData, vtkm::CopyFlag::On);

  vtkm::cont::ArrayHandleCast<vtkm::Float64, vtkm::cont::ArrayHandle<T>> castArray(
    originalArray);
```

The vtkm/cont/ArrayHandleCast.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandleCast` that constructs the cast array. The first argument is the original `ArrayHandle` original array to cast. The optional second argument is of the type to cast to (or you can optionally specify the cast-to type as a template argument.

Example 26.14: Using make_ArrayHandleCast.

```
1 |   vtkm::cont::make_ArrayHandleCast<vtkm::Float64>(originalArray)
```

26.6 Discard Arrays

It is sometimes the case where you will want to run an operation in VTK-m that fills values in two (or more) arrays, but you only want the values that are stored in one of the arrays. It is possible to allocate space for both arrays and then throw away the values that you do not want, but that is a waste of memory. It is also possible to rewrite the functionality to output only what you want, but that is a poor use of developer time.

To solve this problem easily, VTK-m provides vtkm::cont::ArrayHandleDiscard. This array behaves similar to a regular ArrayHandle in that it can be "allocated" and has size, but any values that are written to it are immediately discarded. ArrayHandleDiscard takes up no memory.

Example 26.15: Using ArrayHandleDiscard.

```
1  template<typename InputArrayType,
2           typename OutputArrayType1,
3           typename OutputArrayType2>
4  VTKM_CONT void DoFoo(InputArrayType input,
5                       OutputArrayType1 output1,
6                       OutputArrayType2 output2);
7
8  template<typename InputArrayType>
9  VTKM_CONT inline vtkm::cont::ArrayHandle<vtkm::FloatDefault> DoBar(
10   InputArrayType input)
11 {
12   VTKM_IS_ARRAY_HANDLE(InputArrayType);
13
14   vtkm::cont::ArrayHandle<vtkm::FloatDefault> keepOutput;
15
16   vtkm::cont::ArrayHandleDiscard<vtkm::FloatDefault> discardOutput;
17
18   DoFoo(input, keepOutput, discardOutput);
19
20   return keepOutput;
21 }
```

26.7 Permuted Arrays

A permutation array is a fancy array handle that reorders the elements in an array. Elements in the array can be skipped over or replicated. The permutation array provides this reordered array without actually coping any data. Instead, indices are adjusted as the array is accessed.

Specifying a permutation array in VTK-m is straightforward. VTK-m has a class named vtkm::cont::ArrayHandlePermutation that takes two arrays: an array of values and an array of indices that maps an index in the permutation to an index of the original values. The index array is specified first. The following example is a simple demonstration of the permutation array handle.

Example 26.16: Using ArrayHandlePermutation.

```
1  using IdArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
2  using IdPortalType = IdArrayType::WritePortalType;
3
4  using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Float64>;
5  using ValuePortalType = ValueArrayType::WritePortalType;
```

```
6
7   // Create array with values [0.0, 0.1, 0.2, 0.3]
8   ValueArrayType valueArray;
9   valueArray.Allocate(4);
10  ValuePortalType valuePortal = valueArray.WritePortal();
11  valuePortal.Set(0, 0.0);
12  valuePortal.Set(1, 0.1);
13  valuePortal.Set(2, 0.2);
14  valuePortal.Set(3, 0.3);
15
16  // Use ArrayHandlePermutation to make an array = [0.3, 0.0, 0.1].
17  IdArrayType idArray1;
18  idArray1.Allocate(3);
19  IdPortalType idPortal1 = idArray1.WritePortal();
20  idPortal1.Set(0, 3);
21  idPortal1.Set(1, 0);
22  idPortal1.Set(2, 1);
23  vtkm::cont::ArrayHandlePermutation<IdArrayType, ValueArrayType> permutedArray1(
24    idArray1, valueArray);
25
26  // Use ArrayHandlePermutation to make an array = [0.1, 0.2, 0.2, 0.3, 0.0]
27  IdArrayType idArray2;
28  idArray2.Allocate(5);
29  IdPortalType idPortal2 = idArray2.WritePortal();
30  idPortal2.Set(0, 1);
31  idPortal2.Set(1, 2);
32  idPortal2.Set(2, 2);
33  idPortal2.Set(3, 3);
34  idPortal2.Set(4, 0);
35  vtkm::cont::ArrayHandlePermutation<IdArrayType, ValueArrayType> permutedArray2(
36    idArray2, valueArray);
```

The vtkm/cont/ArrayHandlePermutation.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandlePermutation` that takes instances of the index and value array handles and returns a permutation array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.17: Using `make_ArrayHandlePermutation`.

```
1   vtkm::cont::make_ArrayHandlePermutation(idArray, valueArray)
```

Common Errors

When using an `ArrayHandlePermutation`, take care that all the provided indices in the index array point to valid locations in the values array. Bad indices can cause reading from or writing to invalid memory locations, which can be difficult to debug. Also, be wary about having duplicate indices, which means that multiple array entries point to the same memory location. This will work fine when using the array as input, but will cause a dangerous race condition if used as an output.

Did you know?

You can write to a `ArrayHandlePermutation` by, for example, using it as an output array. Writes to the `ArrayHandlePermutation` will go to the respective location in the source array. However, `ArrayHandlePermutation` cannot be resized.

26.8 Zipped Arrays

A zip array is a fancy array handle that combines two arrays of the same size to pair up the corresponding values. Each element in the zipped array is a `vtkm::Pair` containing the values of the two respective arrays. These pairs are not stored in their own memory space. Rather, the pairs are generated as the array is used. Writing a pair to the zipped array writes the values in the two source arrays.

Specifying a zipped array in VTK-m is straightforward. VTK-m has a class named `vtkm::cont::ArrayHandleZip` that takes the two arrays providing values for the first and second entries in the pairs. The following example is a simple demonstration of creating a zip array handle.

Example 26.18: Using `ArrayHandleZip`.

```
using ArrayType1 = vtkm::cont::ArrayHandle<vtkm::Id>;
using PortalType1 = ArrayType1::WritePortalType;

using ArrayType2 = vtkm::cont::ArrayHandle<vtkm::Float64>;
using PortalType2 = ArrayType2::WritePortalType;

// Create an array of vtkm::Id with values [3, 0, 1]
ArrayType1 array1;
array1.Allocate(3);
PortalType1 portal1 = array1.WritePortal();
portal1.Set(0, 3);
portal1.Set(1, 0);
portal1.Set(2, 1);

// Create a second array of vtkm::Float32 with values [0.0, 0.1, 0.2]
ArrayType2 array2;
array2.Allocate(3);
PortalType2 portal2 = array2.WritePortal();
portal2.Set(0, 0.0);
portal2.Set(1, 0.1);
portal2.Set(2, 0.2);

// Zip the two arrays together to create an array of
// vtkm::Pair<vtkm::Id, vtkm::Float64> with values [(3,0.0), (0,0.1), (1,0.2)]
vtkm::cont::ArrayHandleZip<ArrayType1, ArrayType2> zipArray(array1, array2);
```

The vtkm/cont/ArrayHandleZip.h header also contains the templated convenience function `vtkm::cont::make_ArrayHandleZip` that takes instances of the two array handles and returns a zip array. This function can sometimes be used to avoid having to declare the full array type.

Example 26.19: Using `make_ArrayHandleZip`.

```
vtkm::cont::make_ArrayHandleZip(array1, array2)
```

26.9 Coordinate System Arrays

Many of the data structures we use in VTK-m are described in a 3D coordinate system. Although, as we will see in Chapter 7, we can use any `ArrayHandle` to store point coordinates, including a raw array of 3D vectors, there are some common patterns for point coordinates that we can use specialized arrays to better represent the data.

There are two fancy array handles that each handle a special form of coordinate system. The first such array handle is `vtkm::cont::ArrayHandleUniformPointCoordinates`, which represents a uniform sampling of space. The constructor for `ArrayHandleUniformPointCoordinates` takes three arguments. The first argument is a `vtkm::Id3` that specifies the number of samples in the x, y, and z directions. The second argument, which is

optional, specifies the origin (the location of the first point at the lower left corner). If not specified, the origin is set to $[0,0,0]$. The third argument, which is also optional, specifies the distance between samples in the x, y, and z directions. If not specified, the spacing is set to 1 in each direction.

Example 26.20: Using `ArrayHandleUniformPointCoordinates`.

```
// Create a set of point coordinates for a uniform grid in the space between
// -5 and 5 in the x direction and -3 and 3 in the y and z directions. The
// uniform sampling is spaced in 0.08 unit increments in the x direction (for
// 126 samples), 0.08 unit increments in the y direction (for 76 samples) and
// 0.24 unit increments in the z direction (for 26 samples). That makes
// 248,976 values in the array total.
vtkm::cont::ArrayHandleUniformPointCoordinates uniformCoordinates(
  vtkm::Id3(126, 76, 26),
  vtkm::Vec3f{ -5.0f, -3.0f, -3.0f },
  vtkm::Vec3f{ 0.08f, 0.08f, 0.24f });
```

The second fancy array handle for special coordinate systems is `vtkm::cont::ArrayHandleCartesianProduct`, which represents a rectilinear sampling of space where the samples are axis aligned but have variable spacing. Sets of coordinates of this type are most efficiently represented by having a separate array for each component of the axis, and then for each $[i,j,k]$ index of the array take the value for each component from each array using the respective index. This is equivalent to performing a Cartesian product on the arrays.

`ArrayHandleCartesianProduct` is a templated class. It has three template parameters, which are the types of the arrays used for the x, y, and z axes. The constructor for `ArrayHandleCartesianProduct` takes the three arrays.

Example 26.21: Using a `ArrayHandleCartesianProduct`.

```
using AxisArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
using AxisPortalType = AxisArrayType::WritePortalType;

// Create array for x axis coordinates with values [0.0, 1.1, 5.0]
AxisArrayType xAxisArray;
xAxisArray.Allocate(3);
AxisPortalType xAxisPortal = xAxisArray.WritePortal();
xAxisPortal.Set(0, 0.0f);
xAxisPortal.Set(1, 1.1f);
xAxisPortal.Set(2, 5.0f);

// Create array for y axis coordinates with values [0.0, 2.0]
AxisArrayType yAxisArray;
yAxisArray.Allocate(2);
AxisPortalType yAxisPortal = yAxisArray.WritePortal();
yAxisPortal.Set(0, 0.0f);
yAxisPortal.Set(1, 2.0f);

// Create array for z axis coordinates with values [0.0, 0.5]
AxisArrayType zAxisArray;
zAxisArray.Allocate(2);
AxisPortalType zAxisPortal = zAxisArray.WritePortal();
zAxisPortal.Set(0, 0.0f);
zAxisPortal.Set(1, 0.5f);

// Create point coordinates for a "rectilinear grid" with axis-aligned points
// with variable spacing by taking the Cartesian product of the three
// previously defined arrays. This generates the following 3x2x2 = 12 values:
//
// [0.0, 0.0, 0.0], [1.1, 0.0, 0.0], [5.0, 0.0, 0.0],
// [0.0, 2.0, 0.0], [1.1, 2.0, 0.0], [5.0, 2.0, 0.0],
// [0.0, 0.0, 0.5], [1.1, 0.0, 0.5], [5.0, 0.0, 0.5],
// [0.0, 2.0, 0.5], [1.1, 2.0, 0.5], [5.0, 2.0, 0.5]
vtkm::cont::
```

```
35    ArrayHandleCartesianProduct<AxisArrayType, AxisArrayType, AxisArrayType>
36      rectilinearCoordinates(xAxisArray, yAxisArray, zAxisArray);
```

The vtkm/cont/ArrayHandleCartesianProduct.h/header also contains the templated convenience function vtkm::cont::make_ArrayHandleCartesianProduct that takes the three axis arrays and returns an array of the Cartesian product. This function can sometimes be used to avoid having to declare the full array type.

Example 26.22: Using make_ArrayHandleCartesianProduct.

```
1    vtkm::cont::make_ArrayHandleCartesianProduct(xAxisArray, yAxisArray, zAxisArray)
```

Did you know?

These specialized arrays for coordinate systems greatly reduce the code duplication in VTK-m. Most scientific visualization systems need separate implementations of algorithms for uniform, rectilinear, and unstructured grids. But in VTK-m an algorithm can be written once and then applied to all these different grid structures by using these specialized array handles and letting the compiler's templates optimize the code.

26.10 Composite Vector Arrays

A composite vector array is a fancy array handle that combines two to four arrays of the same size and value type and combines their corresponding values to form a vtkm::Vec. A composite vector array is similar in nature to a zipped array (described in Section 26.8) except that values are combined into vtkm::Vec s instead of vtkm::Pair s. The created vtkm::Vec s are not stored in their own memory space. Rather, the Vecs are generated as the array is used. Writing Vecs to the composite vector array writes values into the components of the source arrays.

A composite vector array can be created using the vtkm::cont::ArrayHandleCompositeVector class. This class has a variadic template argument that is a "signature" for the arrays to be combined. The constructor for ArrayHandleCompositeVector takes instances of the array handles to combine.

Example 26.23: Using ArrayHandleCompositeVector.

```
1   // Create an array with [0, 1, 2, 3, 4]
2   using ArrayType1 = vtkm::cont::ArrayHandleIndex;
3   ArrayType1 array1(5);
4
5   // Create an array with [3, 1, 4, 1, 5]
6   using ArrayType2 = vtkm::cont::ArrayHandle<vtkm::Id>;
7   ArrayType2 array2;
8   array2.Allocate(5);
9   ArrayType2::WritePortalType arrayPortal2 = array2.WritePortal();
10  arrayPortal2.Set(0, 3);
11  arrayPortal2.Set(1, 1);
12  arrayPortal2.Set(2, 4);
13  arrayPortal2.Set(3, 1);
14  arrayPortal2.Set(4, 5);
15
16  // Create an array with [2, 7, 1, 8, 2]
17  using ArrayType3 = vtkm::cont::ArrayHandle<vtkm::Id>;
18  ArrayType3 array3;
19  array3.Allocate(5);
20  ArrayType2::WritePortalType arrayPortal3 = array3.WritePortal();
```

```
21  arrayPortal3.Set(0, 2);
22  arrayPortal3.Set(1, 7);
23  arrayPortal3.Set(2, 1);
24  arrayPortal3.Set(3, 8);
25  arrayPortal3.Set(4, 2);
26
27  // Create an array with [0, 0, 0, 0]
28  using ArrayType4 = vtkm::cont::ArrayHandleConstant<vtkm::Id>;
29  ArrayType4 array4(0, 5);
30
31  // Use ArrayhandleCompositeVector to create the array
32  // [(0,3,2,0), (1,1,7,0), (2,4,1,0), (3,1,8,0), (4,5,2,0)].
33  using CompositeArrayType = vtkm::cont::
34    ArrayHandleCompositeVector<ArrayType1, ArrayType2, ArrayType3, ArrayType4>;
35  CompositeArrayType compositeArray(array1, array2, array3, array4);
```

The vtkm/cont/ArrayHandleCompositeVector.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleCompositeVector which takes a variable number of array handles and returns an ArrayHandleCompositeVector. This function can sometimes be used to avoid having to declare the full array type. ArrayHandleCompositeVector is also often used to combine scalar arrays into vector arrays.

Example 26.24: Using make_ArrayHandleCompositeVector.

```
1  vtkm::cont::make_ArrayHandleCompositeVector(array1, array2, array3, array4)
```

26.11 Extract Component Arrays

Component extraction allows access to a single component of an ArrayHandle with a vtkm::Vec ValueType. vtkm::cont::ArrayHandleExtractComponent allows one component of a vector array to be extracted without creating a copy of the data. ArrayHandleExtractComponent can also be combined with ArrayHandleCompositeVector (described in Section 26.10) to arbitrarily stitch several components from multiple arrays together.

As a simple example, consider an ArrayHandle containing 3D coordinates for a collection of points and a filter that only operates on the points' elevations (Z, in this example). We can easily create the elevation array on-the-fly without allocating a new array as in the following example.

Example 26.25: Extracting components of Vecs in an array with ArrayHandleExtractComponent.

```
1  using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec3f_64>;
2
3  // Create array with values [ (0.0, 0.1, 0.2), (1.0, 1.1, 1.2), (2.0, 2.1, 2.2) ]
4  ValueArrayType valueArray;
5  valueArray.Allocate(3);
6  auto valuePortal = valueArray.WritePortal();
7  valuePortal.Set(0, vtkm::make_Vec(0.0, 0.1, 0.2));
8  valuePortal.Set(1, vtkm::make_Vec(1.0, 1.1, 1.2));
9  valuePortal.Set(2, vtkm::make_Vec(2.0, 2.1, 2.2));
10
11 // Use ArrayHandleExtractComponent to make an array = [1.3, 2.3, 3.3].
12 vtkm::cont::ArrayHandleExtractComponent<ValueArrayType> extractedComponentArray(
13   valueArray, 2);
```

The vtkm/cont/ArrayHandleExtractComponent.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleExtractComponent that takes an ArrayHandle of Vecs and vtkm::IdComponent which returns an appropriately typed ArrayHandleExtractComponent containing the values for a specified component. The index of the component to extract is provided as an argument to make_ArrayHandleExtractComponent, which is required. The use of make_ArrayHandleExtractComponent can be used to avoid having to declare the full array type.

Example 26.26: Using make_ArrayHandleExtractComponent.

```
vtkm::cont::make_ArrayHandleExtractComponent(valueArray, 2)
```

26.12 Swizzle Arrays

It is often useful to reorder or remove specific components from an ArrayHandle with a vtkm::Vec ValueType. vtkm::cont::ArrayHandleSwizzle provides an easy way to accomplish this.

The template parameters of ArrayHandleSwizzle specify a "component map," which defines the swizzle operation. This map consists of the components from the input ArrayHandle, which will be exposed in the ArrayHandleSwizzle. For instance, vtkm::cont::ArrayHandleSwizzle <Some3DArrayType, 3> with Some3DArray and vtkm::Vec <vtkm::IdComponent, 3>(0, 2, 1) as constructor arguments will allow access to a 3D array, but with the Y and Z components exchanged. This rearrangement does not create a copy, and occurs on-the-fly as data are accessed through the ArrayHandleSwizzle's portal. This fancy array handle can also be used to eliminate unnecessary components from an ArrayHandle's data, as shown below.

Example 26.27: Swizzling components of Vecs in an array with ArrayHandleSwizzle.

```
using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec4f_64>;

// Create array with values
// [ (0.0, 0.1, 0.2, 0.3), (1.0, 1.1, 1.2, 1.3), (2.0, 2.1, 2.2, 2.3) ]
ValueArrayType valueArray;
valueArray.Allocate(3);
auto valuePortal = valueArray.WritePortal();
valuePortal.Set(0, vtkm::make_Vec(0.0, 0.1, 0.2, 0.3));
valuePortal.Set(1, vtkm::make_Vec(1.0, 1.1, 1.2, 1.3));
valuePortal.Set(2, vtkm::make_Vec(2.0, 2.1, 2.2, 2.3));

// Use ArrayHandleSwizzle to make an array of Vec-3 with x,y,z,w swizzled to z,x,w
// [ (0.2, 0.0, 0.3), (1.2, 1.0, 1.3), (2.2, 2.0, 2.3) ]
vtkm::cont::ArrayHandleSwizzle<ValueArrayType, 3> swizzledArray(
  valueArray, vtkm::IdComponent3(2, 0, 3));
```

The vtkm/cont/ArrayHandleSwizzle.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleSwizzle that takes an ArrayHandle of Vecs and returns an appropriately typed ArrayHandleSwizzle containing swizzled vectors. The indices of the swizzled components are provided as arguments to make_ArrayHandleSwizzle after the ArrayHandle. The use of make_ArrayHandleSwizzle can be used to avoid having to declare the full array type.

Example 26.28: Using make_ArrayHandleSwizzle.

```
vtkm::cont::make_ArrayHandleSwizzle(valueArray, 2, 0, 3)
```

26.13 Grouped Vector Arrays

A grouped vector array is a fancy array handle that groups consecutive values of an array together to form a vtkm::Vec. The source array must be of a length that is divisible by the requested Vec size. The created vtkm::Vec s are not stored in their own memory space. Rather, the Vecs are generated as the array is used. Writing Vecs to the grouped vector array writes values into the the source array.

A grouped vector array is created using the vtkm::cont::ArrayHandleGroupVec class. This templated class has two template arguments. The first argument is the type of array being grouped and the second argument is an integer specifying the size of the Vecs to create (the number of values to group together).

Example 26.29: Using ArrayHandleGroupVec.

```
// Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
using ArrayType = vtkm::cont::ArrayHandleIndex;
ArrayType sourceArray(12);

// Create an array containing [(0,1), (2,3), (4,5), (6,7), (8,9), (10,11)]
vtkm::cont::ArrayHandleGroupVec<ArrayType, 2> vec2Array(sourceArray);

// Create an array containing [(0,1,2), (3,4,5), (6,7,8), (9,10,11)]
vtkm::cont::ArrayHandleGroupVec<ArrayType, 3> vec3Array(sourceArray);
```

The vtkm/cont/ArrayHandleGroupVec.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleGroupVec that takes an instance of the array to group into Vecs. You must specify the size of the Vecs as a template parameter when using vtkm::cont::make_ArrayHandleGroupVec.

Example 26.30: Using make_ArrayHandleGroupVec.

```
  // Create an array containing [(0,1,2,3), (4,5,6,7), (8,9,10,11)]
  vtkm::cont::make_ArrayHandleGroupVec<4>(sourceArray)
```

ArrayHandleGroupVec is handy when you need to build an array of vectors that are all of the same length, but what about when you need an array of vectors of different lengths? One common use case for this is if you are defining a collection of polygons of different sizes (triangles, quadrilaterals, pentagons, and so on). We would like to define an array such that the data for each polygon were stored in its own Vec (or, rather, Vec-like) object. vtkm::cont::ArrayHandleGroupVecVariable does just that.

ArrayHandleGroupVecVariable takes two arrays. The first array, identified as the "source" array, is a flat representation of the values (much like the array used with ArrayHandleGroupVec). The second array, identified as the "offsets" array, provides for each vector the index into the source array where the start of the vector is. The offsets array must be monotonically increasing. The size of the offsets array is one greater than the number of vectors in the resulting array. The first offset is always 0 and the last offset is always the size of the input source array. The first and second template parameters to ArrayHandleGroupVecVariable are the types for the source and offset arrays, respectively.

It is often the case that you will start with a group of vector lengths rather than offsets into the source array. If this is the case, then the vtkm::cont::ConvertNumComponentsToOffsets helper function can convert an array of vector lengths to an array of offsets. The first argument to this function is always the array of vector lengths. The second argument, which is optional, is a reference to a ArrayHandle into which the offsets should be stored. If this offset array is not specified, an ArrayHandle will be returned from the function instead. The third argument, which is also optional, is a reference to a vtkm::Id into which the expected size of the source array is put. Having the size of the source array is often helpful, as it can be used to allocate data for the source array or check the source array's size. It is also OK to give the expected size reference but not the offset array reference.

Example 26.31: Using ArrayHandleGroupVecVariable.

```
// Create an array of counts containing [4, 2, 3, 3]
vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray =
  vtkm::cont::make_ArrayHandle<vtkm::IdComponent>({ 4, 2, 3, 3 });

// Convert the count array to an offset array [0, 4, 6, 9, 12]
// Returns the number of total components: 12
vtkm::Id sourceArraySize;
using OffsetArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
OffsetArrayType offsetArray =
  vtkm::cont::ConvertNumComponentsToOffsets(countArray, sourceArraySize);

// Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
using SourceArrayType = vtkm::cont::ArrayHandleIndex;
SourceArrayType sourceArray(sourceArraySize);
```

```
15
16  // Create an array containing [(0,1,2,3), (4,5), (6,7,8), (9,10,11)]
17  vtkm::cont::ArrayHandleGroupVecVariable<SourceArrayType, OffsetArrayType>
18    vecVariableArray(sourceArray, offsetArray);
```

The vtkm/cont/ArrayHandleGroupVecVariable.h header also contains the templated convenience function vtkm::cont::make_ArrayHandleGroupVecVariable that takes an instance of the source array to group into Vec-like objects and the offset array.

Example 26.32: Using MakeArrayHandleGroupVecVariable.

```
1  // Create an array containing [(0,1,2,3), (4,5), (6,7,8), (9,10,11)]
2  vtkm::cont::make_ArrayHandleGroupVecVariable(sourceArray, offsetArray)
```

Did you know?

You can write to ArrayHandleGroupVec *and* ArrayHandleGroupVecVariable *by, for example, using it as an output array. Writes to these arrays will go to the respective location in the source array.* ArrayHandleGroupVec *can also be allocated and resized (which in turn causes the source array to be allocated). However,* ArrayHandleGroupVecVariable *cannot be resized and the source array must be pre-allocated. You can use the source array size value returned from* ConvertNumComponentsToOffsets *to allocate source arrays.*

Common Errors

Keep in mind that the values stored in a ArrayHandleGroupVecVariable *are not actually* vtkm::Vec *objects. Rather, they are "*Vec*-like" objects, which has some subtle but important ramifications. First, the type will not match the* vtkm::Vec *template, and there is no automatic conversion to* vtkm::Vec *objects. Thus, many functions that accept* vtkm::Vec *objects as parameters will not accept the* Vec*-like object. Second, the size of* Vec*-like objects are not known until runtime. See Sections 4.3 and 19.5.2 for more information on the difference between* vtkm::Vec *and* Vec*-like objects.*

CHAPTER

TWENTYSEVEN

ACCESSING AND ALLOCATING ARRAY HANDLES

So far we have seen examples of creating `vtkm::cont::ArrayHandle` s from normal C++ arrays (Chapter 16) and creating some special arrays (Chapter 26). However, we have so far avoided discussing how to access the actual data in `ArrayHandle`s. So far we have only accessed `ArrayHandle` data indirectly through other VTK-m features such as worklets.

In this chapter we describe how to more directly access the data in an `ArrayHandle`, how to allocate space for data in an `ArrayHandle`, and how data is transferred to the execution environment to be used in a worklet.

27.1 Array Portals

An array handle defines auxiliary structures called *array portals* that provide direct access into its data. An array portal is a simple object that is somewhat functionally equivalent to an STL-type iterator, but with a much simpler interface. Array portals can be read-only or read-write and they can be accessible from either the control environment or the execution environment. All these variants have similar interfaces although some features that are not applicable can be left out.

An array portal object contains each of the following:

`ValueType` The type for each item in the array.

`GetNumberOfValues` A method that returns the number of entries in the array.

`Get` A method that returns the value at a given index.

`Set` A method that changes the value at a given index. This method does not need to exist for read-only array portals.

The following code example defines an array portal for a simple C array of scalar values. This definition has no practical value (it is covered by the more general `vtkm::cont::internal::ArrayPortalFromIterators`), but demonstrates the function of each component.

Example 27.1: A simple array portal implementation.

```
1 template<typename T>
2 class SimpleScalarArrayPortal
3 {
4 public:
```

```
  using ValueType = T;

  // There is no specification for creating array portals, but they generally
  // need a constructor like this to be practical.
  VTKM_EXEC_CONT
  SimpleScalarArrayPortal(ValueType* array, vtkm::Id numberOfValues)
    : Array(array)
    , NumberOfValues(numberOfValues)
  {
  }

  VTKM_EXEC_CONT
  SimpleScalarArrayPortal()
    : Array(NULL)
    , NumberOfValues(0)
  {
  }

  VTKM_EXEC_CONT
  vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }

  VTKM_EXEC_CONT
  ValueType Get(vtkm::Id index) const { return this->Array[index]; }

  VTKM_EXEC_CONT
  void Set(vtkm::Id index, ValueType value) const { this->Array[index] = value; }

private:
  ValueType* Array;
  vtkm::Id NumberOfValues;
};
```

Although array portals are simple to implement and use, and array portals' functionality is similar to iterators, there exists a great deal of code already based on STL iterators and it is often convenient to interface with an array through an iterator rather than an array portal. The `vtkm::cont::ArrayPortalToIterators` class can be used to convert an array portal to an STL-compatible iterator. The class is templated on the array portal type and has a constructor that accepts an instance of the array portal. It contains the following features.

`IteratorType` The type of an STL-compatible random-access iterator that can provide the same access as the array portal.

`GetBegin` A method that returns an STL-compatible iterator of type `IteratorType` that points to the beginning of the array.

`GetEnd` A method that returns an STL-compatible iterator of type `IteratorType` that points to the end of the array.

Example 27.2: Using `ArrayPortalToIterators`.

```
template<typename PortalType>
VTKM_CONT std::vector<typename PortalType::ValueType> CopyArrayPortalToVector(
  const PortalType& portal)
{
  using ValueType = typename PortalType::ValueType;
  std::vector<ValueType> result(
    static_cast<std::size_t>(portal.GetNumberOfValues()));

  vtkm::cont::ArrayPortalToIterators<PortalType> iterators(portal);

  std::copy(iterators.GetBegin(), iterators.GetEnd(), result.begin());

```

```
13   return result;
14 }
```

As a convenience, vtkm/cont/ArrayPortalToIterators.h also defines a pair of functions named `vtkm::cont::ArrayPortalToIteratorBegin()` and `vtkm::cont::ArrayPortalToIteratorEnd()` that each take an array portal as an argument and return a begin and end iterator, respectively.

Example 27.3: Using `ArrayPortalToIteratorBegin` and `ArrayPortalToIteratorEnd`.

```
1  std::vector<vtkm::Float32> myContainer(
2    static_cast<std::size_t>(portal.GetNumberOfValues()));
3
4  std::copy(vtkm::cont::ArrayPortalToIteratorBegin(portal),
5            vtkm::cont::ArrayPortalToIteratorEnd(portal),
6            myContainer.begin());
```

`ArrayHandle` contains two internal type definitions for array portal types that are capable of interfacing with the underlying data in the control environment. These are `WritePortalType` and `ReadPortalType`, which define read-write and read-only array portals, respectively.

`ArrayHandle` provides the methods `ReadPortal` and `WritePortal` to get the associated array portal objects to access the data in the control environment. These methods also have the side effect of refreshing the control environment copy of the data as if you called `SyncControlArray`. Be aware that calling `WritePortal` will invalidate any copy in the execution environment, meaning that any subsequent use will cause the data to be copied back again.

Example 27.4: Using portals from an `ArrayHandle`.

```
1  template<typename T, typename Storage>
2  void SortCheckArrayHandle(vtkm::cont::ArrayHandle<T, Storage> arrayHandle)
3  {
4    using WritePortalType =
5      typename vtkm::cont::ArrayHandle<T, Storage>::WritePortalType;
6    using ReadPortalType =
7      typename vtkm::cont::ArrayHandle<T, Storage>::ReadPortalType;
8
9    WritePortalType readwritePortal = arrayHandle.WritePortal();
10   // This is actually pretty dumb. Sorting would be generally faster in
11   // parallel in the execution environment using the device adapter algorithms.
12   std::sort(vtkm::cont::ArrayPortalToIteratorBegin(readwritePortal),
13             vtkm::cont::ArrayPortalToIteratorEnd(readwritePortal));
14
15   ReadPortalType readPortal = arrayHandle.ReadPortal();
16   for (vtkm::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
17   {
18     if (readPortal.Get(index - 1) > readPortal.Get(index))
19     {
20       std::cout << "Sorting is wrong!" << std::endl;
21       break;
22     }
23   }
24 }
```

Did you know?

Most operations on arrays in VTK-m should really be done in the execution environment. Keep in mind that whenever doing an operation using a control array portal, that operation will likely be slow for large arrays. However, some operations, like performing file I/O, make sense in the control environment.

Common Errors

The portal returned from `ReadPortal` *or* `WritePortal` *is only good as long as the data in the* `ArrayHandle` *are not moved or reallocated. For example, if you call* `ArrayHandle::Allocate`*, any previously created array portals are likely to become invalid, and using them will result in undefined behavior. Thus, you should keep portals only as long as is necessary to complete an operation.*

27.2 Allocating and Populating Array Handles

`vtkm::cont::ArrayHandle` is capable of allocating its own memory. The most straightforward way to allocate memory is to call the `ArrayHandle::Allocate` method. The `Allocate` method takes a single argument, which is the number of elements to make the array.

Example 27.5: Allocating an `ArrayHandle`.

```
1  vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;
2
3  const vtkm::Id ARRAY_SIZE = 50;
4  arrayHandle.Allocate(ARRAY_SIZE);
```

By default when you `Allocate` an array, it potentially destroys any existing data in it. However, there are cases where you wish to grow or shrink an array while preserving the existing data. To preserve the existing data when allocating an array, pass `vtkm::CopyFlag::On` as an optional second argument.

Example 27.6: Resizing an `ArrayHandle`.

```
1  // Add space for 10 more values at the end of the array.
2  arrayHandle.Allocate(arrayHandle.GetNumberOfValues() + 10, vtkm::CopyFlag::On);
```

Did you know?

The ability to allocate memory is a key difference between `ArrayHandle` *and many other common forms of smart pointers. When one* `ArrayHandle` *allocates new memory, all other* `ArrayHandle`*s pointing to the same managed memory get the newly allocated memory. This feature makes it possible to pass an* `ArrayHandle` *to a method to be reallocated and filled without worrying about C++ details on how to reference the* `ArrayHandle` *object itself.*

Once an `ArrayHandle` is allocated, it can be populated by using the portal returned from `ArrayHandle::WritePortal`, as described in Section 27.1. This is roughly the method used by the readers in the I/O package (Chapter 8).

Example 27.7: Populating a newly allocated `ArrayHandle`.

```
1  vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;
2
3  const vtkm::Id ARRAY_SIZE = 50;
4  arrayHandle.Allocate(ARRAY_SIZE);
5
6  // Usually it is easier to just use the auto keyword.
7  using PortalType = vtkm::cont::ArrayHandle<vtkm::Float32>::WritePortalType;
8  PortalType portal = arrayHandle.WritePortal();
```

```
9
10  for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
11  {
12    portal.Set(index, GetValueForArray(index));
13  }
```

27.3 Compute Array Range

It is common to need to know the minimum and/or maximum values in an array. To help find these values, VTK-m provides the `vtkm::cont::ArrayRangeCompute` convenience function defined in vtkm/cont/ArrayRangeCompute.h. `ArrayRangeCompute` simply takes an `ArrayHandle` on which to find the range of values.

If given an array with `vtkm::Vec` values, `ArrayRangeCompute` computes the range separately for each component of the `Vec`. The return value for `ArrayRangeCompute` is `vtkm::cont::ArrayHandle <vtkm::Range >`. This returned array will have one value for each component of the input array's type. So for example if you call `ArrayRangeCompute` on a `vtkm::cont::ArrayHandle <vtkm::Id3 >`, the returned array of `Range`s will have 3 values in it. Of course, when `ArrayRangeCompute` is run on an array of scalar types, you get an array with a single value in it.

Each value of `vtkm::Range` holds the minimum and maximum value for that component. The `Range` object is documented in Section 19.3.

Example 27.8: Using `ArrayRangeCompute`.

```
1  vtkm::cont::ArrayHandle<vtkm::Range> rangeArray =
2    vtkm::cont::ArrayRangeCompute(arrayHandle);
3  auto rangePortal = rangeArray.ReadPortal();
4  for (vtkm::Id index = 0; index < rangePortal.GetNumberOfValues(); ++index)
5  {
6    vtkm::Range componentRange = rangePortal.Get(index);
7    std::cout << "Values for component " << index << " go from "
8              << componentRange.Min << " to " << componentRange.Max << std::endl;
9  }
```

Did you know?

`ArrayRangeCompute` *will compute the minimum and maximum values in parallel. If desired, you can specify the parallel hardware device used for the computation as an optional second argument to* `ArrayRangeCompute`*. You can specify the device using a runtime device tracker, which is documented in Section 12.3.*

27.4 Interface to Execution Environment

One of the main functions of the array handle is to allow an array to be defined in the control environment and then be used in the execution environment. When using an `ArrayHandle` with filters, worklets, or algorithms, this transition is handled automatically. However, it is also possible to invoke the transfer for a known device. This is most useful when creating execution objects, as discussed in Chapter 29.

The `ArrayHandle` class manages the transition from control to execution with a set of three methods that allocate, transfer, and ready the data in one operation. These methods all start with the prefix `Prepare` and are meant to be called before some operation happens in the execution environment. The methods are as follows.

`ArrayHandle::PrepareForInput` Copies data from the control to the execution environment, if necessary, and readies the data for read-only access.

`ArrayHandle::PrepareForInPlace` Copies the data from the control to the execution environment, if necessary, and readies the data for both reading and writing.

`ArrayHandle::PrepareForOutput` Allocates space (the size of which is given as a parameter) in the execution environment, if necessary, and readies the space for writing.

The `PrepareForInput` and `PrepareForInPlace` methods each take two arguments. The first argument is the device adapter tag where execution will take place (see Section 12.1 for more information on device adapter tags). The second argument is a reference to a `vtkm::cont::Token`, which scopes the returned array portal. While the given `Token` exists, the returned portal is guaranteed to be valid and any conflicting operations on the `ArrayHandle` will block. Once the `Token` is destroyed, the associated array portal becomes invalid.

`PrepareForOutput` takes three arguments: the size of the space to allocate, the device adapter tag, and a reference to a `Token` object.

Each of these `Prepare` methods returns an array portal that can be used in the execution environment. `PrepareForInput` returns an object of type `ArrayHandle::ReadPortalType` whereas `PrepareForInPlace` and `PrepareForOutput` each return an object of type `ArrayHandle::WritePortalType`.

Although these `Prepare` methods are called in the control environment, the returned array portal can only be used in the execution environment. Thus, the portal must be passed to an invocation of the execution environment.

Most of the time, the passing of `ArrayHandle` data to the execution environment is handled automatically by VTK-m. The most common need to call one of these `Prepare` methods is to build execution objects (Chapter 29) or to construct derived array types (Section 36.4).

The following example is a contrived example for preparing arrays for the execution environment. It is contrived because it would be easier to create a worklet or transform array handle to have the same effect, and in those cases VTK-m would take of the transfers internally. More realistic examples can be found in Chapter 29 and Section 36.4.

Example 27.9: Using an execution array portal from an `ArrayHandle`.

```
1  template<typename InputPortalType, typename OutputPortalType>
2  struct DoubleFunctor : public vtkm::exec::FunctorBase
3  {
4    InputPortalType InputPortal;
5    OutputPortalType OutputPortal;
6
7    VTKM_CONT
8    DoubleFunctor(InputPortalType inputPortal, OutputPortalType outputPortal)
9      : InputPortal(inputPortal)
10     , OutputPortal(outputPortal)
11   {
12   }
13
14   VTKM_EXEC
15   void operator()(vtkm::Id index) const
16   {
17     this->OutputPortal.Set(index, 2 * this->InputPortal.Get(index));
18   }
19 };
20
21 template<typename T, typename Device>
22 void DoubleArray(vtkm::cont::ArrayHandle<T> inputArray,
23                  vtkm::cont::ArrayHandle<T> outputArray,
```

```
                  Device)
{
  vtkm::Id numValues = inputArray.GetNumberOfValues();

  vtkm::cont::Token token;
  auto inputPortal = inputArray.PrepareForInput(Device{}, token);
  auto outputPortal = outputArray.PrepareForOutput(numValues, Device{}, token);
  // Token is now attached to inputPortal and outputPortal. Those two portals
  // are guaranteed to be valid until token goes out of scope at the end of
  // this function.

  DoubleFunctor<decltype(inputPortal), decltype(outputPortal)> functor(inputPortal,
                                                                       outputPortal);

  vtkm::cont::DeviceAdapterAlgorithm<Device>::Schedule(functor, numValues);
}
```

Common Errors

Once one of the `Prepare` *methods have been called, further operations on the* `ArrayHandle` *that might cause access hazards will block. This opens the possibility of deadlock. To help prevent deadlock, the attached* `Token` *object should be scoped to last only as long as necessary.*

CHAPTER

TWENTYEIGHT

GLOBAL ARRAYS AND TOPOLOGY

When writing an algorithm in VTK-m by creating a worklet, the data each instance of the worklet has access to is intentionally limited. This allows VTK-m to provide safety from race conditions and other parallel programming difficulties. However, there are times when the complexity of an algorithm requires all threads to have shared global access to a global structure. This chapter describes worklet tags that can be used to pass data globally to all instances of a worklet.

28.1 Whole Arrays

A *whole array* argument to a worklet allows you to pass in an `ArrayHandle`. All instances of the worklet will have access to all the data in the `ArrayHandle`.

Common Errors

The VTK-m worklet invoking mechanism performs many safety checks to prevent race conditions across concurrently running worklets. Using a whole array within a worklet circumvents this guarantee of safety, so be careful when using whole arrays, especially when writing to whole arrays.

A whole array is declared by adding a `WholeArrayIn`, a `WholeArrayInOut`, or a `WholeArrayOut` to the `ControlSignature` of a worklet. The corresponding argument to the `Invoker` should be an `ArrayHandle`. The `ArrayHandle` must already be allocated in all cases, including when using `WholeArrayOut`. When the data are passed to the operator of the worklet, it is passed as an array portal object. (Array portals are discussed in Section 27.1.) This means that the worklet can access any entry in the array with `Get` and/or `Set` methods.

We have already seen a demonstration of using a whole array in Example 21.2 to perform a simple array copy. Here we will construct a more thorough example of building functionality that requires random array access.

Let's say we want to measure the quality of triangles in a mesh. A common method for doing this is using the equation

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where a is the area of the triangle and h_1, h_2, and h_3 are the lengths of the sides. We can easily compute this in a cell to point map, but what if we want to speed up the computations by reducing precision? After all, we probably only care if the triangle is good, reasonable, or bad. So instead, let's build a lookup table and then retrieve the triangle quality from that lookup table based on its sides.

The following example demonstrates creating such a table lookup in an array and using a worklet argument tagged with `WholeArrayIn` to make it accessible.

Example 28.1: Using `WholeArrayIn` to access a lookup table in a worklet.

```
namespace detail
{

static const vtkm::Id TRIANGLE_QUALITY_TABLE_DIMENSION = 8;
static const vtkm::Id TRIANGLE_QUALITY_TABLE_SIZE =
  TRIANGLE_QUALITY_TABLE_DIMENSION * TRIANGLE_QUALITY_TABLE_DIMENSION;

VTKM_CONT
vtkm::cont::ArrayHandle<vtkm::Float32> GetTriangleQualityTable()
{
  // Use these precomputed values for the array. A real application would
  // probably use a larger array, but we are keeping it small for demonstration
  // purposes.
  static vtkm::Float32 triangleQualityBuffer[TRIANGLE_QUALITY_TABLE_SIZE] = {
    0, 0,        0,        0,        0,        0,        0,        0,
    0, 0,        0,        0,        0,        0,        0,        0.24431f,
    0, 0,        0,        0,        0,        0,        0.43298f, 0.47059f,
    0, 0,        0,        0,        0,        0.54217f, 0.65923f, 0.66408f,
    0, 0,        0,        0,        0.57972f, 0.75425f, 0.82154f, 0.81536f,
    0, 0,        0,        0.54217f, 0.75425f, 0.87460f, 0.92567f, 0.92071f,
    0, 0,        0.43298f, 0.65923f, 0.82154f, 0.92567f, 0.97664f, 0.98100f,
    0, 0.24431f, 0.47059f, 0.66408f, 0.81536f, 0.92071f, 0.98100f, 1
  };

  return vtkm::cont::make_ArrayHandle(
    triangleQualityBuffer, TRIANGLE_QUALITY_TABLE_SIZE, vtkm::CopyFlag::Off);
}

template<typename T>
VTKM_EXEC_CONT vtkm::Vec<T, 3> TriangleEdgeLengths(const vtkm::Vec<T, 3>& point1,
                                                   const vtkm::Vec<T, 3>& point2,
                                                   const vtkm::Vec<T, 3>& point3)
{
  return vtkm::make_Vec(vtkm::Magnitude(point1 - point2),
                        vtkm::Magnitude(point2 - point3),
                        vtkm::Magnitude(point3 - point1));
}

VTKM_SUPPRESS_EXEC_WARNINGS
template<typename PortalType, typename T>
VTKM_EXEC_CONT static vtkm::Float32 LookupTriangleQuality(
  const PortalType& triangleQualityPortal,
  const vtkm::Vec<T, 3>& point1,
  const vtkm::Vec<T, 3>& point2,
  const vtkm::Vec<T, 3>& point3)
{
  vtkm::Vec<T, 3> edgeLengths = TriangleEdgeLengths(point1, point2, point3);

  // To reduce the size of the table, we just store the quality of triangles
  // with the longest edge of size 1. The table is 2D indexed by the length
  // of the other two edges. Thus, to use the table we have to identify the
  // longest edge and scale appropriately.
  T smallEdge1 = vtkm::Min(edgeLengths[0], edgeLengths[1]);
  T tmpEdge = vtkm::Max(edgeLengths[0], edgeLengths[1]);
  T smallEdge2 = vtkm::Min(edgeLengths[2], tmpEdge);
  T largeEdge = vtkm::Max(edgeLengths[2], tmpEdge);

  smallEdge1 /= largeEdge;
  smallEdge2 /= largeEdge;

```

```
61    // Find index into array.
62    vtkm::Id index1 = static_cast<vtkm::Id>(
63      vtkm::Floor(smallEdge1 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
64    vtkm::Id index2 = static_cast<vtkm::Id>(
65      vtkm::Floor(smallEdge2 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
66    vtkm::Id totalIndex = index1 + index2 * TRIANGLE_QUALITY_TABLE_DIMENSION;
67
68    return triangleQualityPortal.Get(totalIndex);
69  }
70
71  } // namespace detail
72
73  struct TriangleQualityWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
74  {
75    using ControlSignature = void(CellSetIn cells,
76                                  FieldInPoint pointCoordinates,
77                                  WholeArrayIn triangleQualityTable,
78                                  FieldOutCell triangleQuality);
79    using ExecutionSignature = _4(CellShape, _2, _3);
80    using InputDomain = _1;
81
82    template<typename CellShape,
83             typename PointCoordinatesType,
84             typename TriangleQualityTablePortalType>
85    VTKM_EXEC vtkm::Float32 operator()(
86      CellShape shape,
87      const PointCoordinatesType& pointCoordinates,
88      const TriangleQualityTablePortalType& triangleQualityTable) const
89    {
90      if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
91      {
92        this->RaiseError("Only triangles are supported for triangle quality.");
93        return vtkm::Nan32();
94      }
95
96      return detail::LookupTriangleQuality(triangleQualityTable,
97                                           pointCoordinates[0],
98                                           pointCoordinates[1],
99                                           pointCoordinates[2]);
100   }
101 };
102
103 //
104 // Later in the associated Filter class...
105 //
106
107     vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualityTable =
108       detail::GetTriangleQualityTable();
109
110     vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
111
112     this->Invoke(TriangleQualityWorklet{},
113                  inputDataSet.GetCellSet(),
114                  inputPointCoordinatesField,
115                  triangleQualityTable,
116                  triangleQualities);
```

28.2 Atomic Arrays

One of the problems with writing to whole arrays is that it is difficult to coordinate the access to an array from multiple threads. If multiple threads are going to write to a common index of an array, then you will probably

need to use an *atomic array*.

An atomic array allows random access into an array of data, similar to a whole array. However, the operations on the values in the atomic array allow you to perform an operation that modifies its value that is guaranteed complete without being interrupted and potentially corrupted.

Common Errors

Due to limitations in available atomic operations, atomic arrays can currently only contain `vtkm::Int32` *or* `vtkm::Int64` *values.*

To use an array as an atomic array, first add the `AtomicArrayInOut` tag to the worklet's `ControlSignature`. The corresponding argument to the `Invoker` should be an `ArrayHandle`, which must already be allocated and initialized with values.

When the data are passed to the operator of the worklet, it is passed in a `vtkm::exec::AtomicArrayExecutionObject` structure. `AtomicArrayExecutionObject` has two important methods:

`Add` Takes as arguments an index and a value. The entry in the array corresponding to the index will have the value added to it. If multiple threads attempt to add to the same index in the array, the requests will be serialized so that the final result is the sum of all the additions. `AtomicArrayExecutionObject::Add` returns the value that was replaced. That is, it returns the value right *before* the addition.

`CompareAndSwap` Takes as arguments an index, a new value, and an old value. If the entry in the array corresponding to the index has the same value as the "old value," then it is changed to the "new value" and the original value is return from the method. If the entry in the array is not the same as the "old value," then nothing happens to the array and the value that is actually stored in the array is returned. If multiple threads attempt to compare and swap to the same index in the array, the requests are serialized.

Common Errors

Atomic arrays help resolve hazards in parallel algorithms, but they come at a cost. Atomic operations are more costly than non-thread-safe ones, and they can slow a parallel program immensely if used incorrectly.

The following example uses an atomic array to count the bins in a histogram. It does this by making the array of histogram bins an atomic array and then using an atomic add. Note that this is not the fastest way to create a histogram. We gave an implementation in Section 21.4 that is generally faster (unless your histogram happens to be very sparse). VTK-m also comes with a histogram worklet that uses a similar approach.

Example 28.2: Using `AtomicArrayInOut` to count histogram bins in a worklet.

```
1  struct CountBins : vtkm::worklet::WorkletMapField
2  {
3    using ControlSignature = void(FieldIn data, AtomicArrayInOut histogramBins);
4    using ExecutionSignature = void(_1, _2);
5    using InputDomain = _1;
6
7    vtkm::Range HistogramRange;
8    vtkm::Id NumberOfBins;
```

```
9
10      VTKM_CONT
11      CountBins(const vtkm::Range& histogramRange, vtkm::Id& numBins)
12        : HistogramRange(histogramRange)
13        , NumberOfBins(numBins)
14      {
15      }
16
17      template<typename T, typename AtomicArrayType>
18      VTKM_EXEC void operator()(T value, const AtomicArrayType& histogramBins) const
19      {
20        vtkm::Float64 interp =
21          (value - this->HistogramRange.Min) / this->HistogramRange.Length();
22        vtkm::Id bin = static_cast<vtkm::Id>(interp * this->NumberOfBins);
23        if (bin < 0)
24        {
25          bin = 0;
26        }
27        if (bin >= this->NumberOfBins)
28        {
29          bin = this->NumberOfBins - 1;
30        }
31
32        histogramBins.Add(bin, 1);
33      }
34    };
```

28.3 Whole Cell Sets

Section 21.2 describes how to make a topology map filter that performs an operation on cell sets. The worklet has access to a single cell element (such as point or cell) and its immediate connections. But there are cases when you need more general queries on a topology. For example, you might need more detailed information than the topology map gives or you might need to trace connections from one cell to the next. To do this VTK-m allows you to provide a *whole cell set* argument to a worklet that provides random access to the entire topology.

A whole cell set is declared by adding a `WholeCellSetIn` to the worklet's `ControlSignature`. The corresponding argument to the `Invoker` should be a `CellSet` subclass or an `UnknownCellSet` (both of which are described in Section 7.2).

The `WholeCellSetIn` is templated and takes two arguments: the "visit" topology type and the "incident" topology type, respectively. These template arguments must be one of the topology element tags, but for convenience you can use `Point` and `Cell` in lieu of `vtkm::TopologyElementTagPoint` and `vtkm::TopologyElementTagCell`, respectively. The "visit" and "incident" topology types define which topological elements can be queried (visited) and which incident elements are returned. The semantics of the "visit" and "incident" topology is the same as that for the general topology maps described in Section 21.2.3. You can look up an element of the "visit" topology by index and then get all of the "incident" elements from it.

For example, a `WholeCellSetIn<Cell, Point>` allows you to find all the points that are incident on each cell (as well as querying the cell shape). Likewise, a `WholeCellSetIn<Point, Cell>` allows you to find all the cells that are incident on each point. The default parameters of `WholeCellSetIn` are visiting cells with incident points. That is, `WholeCellSetIn<>` is equivalent to `WholeCellSetIn<Cell, Point>`.

When the cell set is passed to the operator of the worklet, it is passed in a special connectivity object. The actual object type depends on the cell set, but `vtkm::exec::CellSetStructured` and are two common examples `vtkm::exec::CellSetExplicit`. All these connectivity objects share a common interface. First, they all declare the following public types.

`CellShapeTag` The tag for the cell shapes of the cell set. (Cell shape tags are described in Section 25.1.) If the connectivity potentially contains more than one type of cell shape, then this type will be `vtkm::CellShapeTagGeneric`.

`IndicesType` A `Vec`-like type that stores all the incident indices.

Second they all provide the following methods.

`GetNumberOfElements` Get the number of "to" topology elements in the cell set. All the other methods require an element index, and this represents the range of valid indices. The return type is `vtkm::Id`.

`GetCellShape` Takes an index for an element and returns a `CellShapeTag` object of the corresponding cell shape. If the "to" topology elements are not strictly cell, then a reasonably close shape is returned. For example, if the "to" topology elements are points, then the shape is returned as a vertex.

`GetNumberOfIndices` Takes an index for an element and returns the number of incident "from" elements are connected to it. The returned type is `vtkm::IdComponent`.

`GetIndices` Takes an index for an element and returns a `Vec`-like object of type `IndicesType` containing the indices of all incident "from" elements. The size of the `Vec`-like object is the same as that returned from `GetNumberOfIndicices`.

VTK-m comes with several functions to work with the shape and index information returned from these connectivity objects. Most of these methods are documented in Chapter 25.

Let us use the whole cell set feature to help us determine the "flatness" of a polygonal mesh. We will do this by summing up all the angles incident on each on each point. That is, for each point, we will find each incident polygon, then find the part of that polygon using the given point, then computing the angle at that point, and then summing for all such angles. So, for example, in the mesh fragment shown in Figure 28.1 one of the angles attached to the middle point is labeled θ_j.

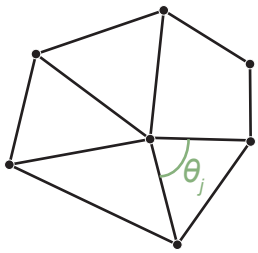


Figure 28.1: The angles incident around a point in a mesh.

We want a worklet to compute $\sum_j \theta$ for all such attached angles. This measure is related (but not the same as) the curvature of the surface. A flat surface will have a sum of 2π. Convex and concave surfaces have a value less than 2π, and saddle surfaces have a value greater than 2π.

To do this, we create a visit points with cells worklet (Section 21.2.2) that visits every point and gives the index of every incident cell. The worklet then uses a whole cell set to inspect each incident cell to measure the attached angle and sum them together.

Example 28.3: Using `WholeCellSetIn` to sum the angles around each point.

```
1 struct SumOfAngles : vtkm::worklet::WorkletVisitPointsWithCells
2 {
3   using ControlSignature = void(CellSetIn inputCells,
```

```
4                                 WholeCellSetIn<>, // Same as inputCells
5                                 WholeArrayIn pointCoords,
6                                 FieldOutPoint angleSum);
7   using ExecutionSignature = void(CellIndices incidentCells,
8                                   InputIndex pointIndex,
9                                   _2 cellSet,
10                                  _3 pointCoordsPortal,
11                                  _4 outSum);
12  using InputDomain = _1;
13
14  template<typename IncidentCellVecType,
15           typename CellSetType,
16           typename PointCoordsPortalType,
17           typename SumType>
18  VTKM_EXEC void operator()(const IncidentCellVecType& incidentCells,
19                            vtkm::Id pointIndex,
20                            const CellSetType& cellSet,
21                            const PointCoordsPortalType& pointCoordsPortal,
22                            SumType& outSum) const
23  {
24    using CoordType = typename PointCoordsPortalType::ValueType;
25
26    CoordType thisPoint = pointCoordsPortal.Get(pointIndex);
27
28    outSum = 0;
29    for (vtkm::IdComponent incidentCellIndex = 0;
30         incidentCellIndex < incidentCells.GetNumberOfComponents();
31         ++incidentCellIndex)
32    {
33      // Get information about incident cell.
34      vtkm::Id cellIndex = incidentCells[incidentCellIndex];
35      typename CellSetType::CellShapeTag cellShape = cellSet.GetCellShape(cellIndex);
36      typename CellSetType::IndicesType cellConnections =
37        cellSet.GetIndices(cellIndex);
38      vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
39      vtkm::IdComponent numEdges;
40      vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
41
42      // Iterate over all edges and find the first one with pointIndex.
43      // Use that to find the first vector.
44      vtkm::IdComponent edgeIndex = -1;
45      CoordType vec1;
46      while (true)
47      {
48        ++edgeIndex;
49        if (edgeIndex >= numEdges)
50        {
51          this->RaiseError("Bad cell. Could not find two incident edges.");
52          return;
53        }
54        vtkm::IdComponent2 edge;
55        vtkm::exec::CellEdgeLocalIndex(
56          numPointsInCell, 0, edgeIndex, cellShape, edge[0]);
57        vtkm::exec::CellEdgeLocalIndex(
58          numPointsInCell, 1, edgeIndex, cellShape, edge[1]);
59        if (cellConnections[edge[0]] == pointIndex)
60        {
61          vec1 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
62          break;
63        }
64        else if (cellConnections[edge[1]] == pointIndex)
65        {
66          vec1 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
67          break;
```

```
68          }
69          else
70          {
71            // Continue to next iteration of loop.
72          }
73        }
74
75        // Continue iteration over remaining edges and find the second one with
76        // pointIndex. Use that to find the second vector.
77        CoordType vec2;
78        while (true)
79        {
80          ++edgeIndex;
81          if (edgeIndex >= numEdges)
82          {
83            this->RaiseError("Bad cell. Could not find two incident edges.");
84            return;
85          }
86          vtkm::IdComponent2 edge;
87          vtkm::exec::CellEdgeLocalIndex(
88            numPointsInCell, 0, edgeIndex, cellShape, edge[0]);
89          vtkm::exec::CellEdgeLocalIndex(
90            numPointsInCell, 1, edgeIndex, cellShape, edge[1]);
91          if (cellConnections[edge[0]] == pointIndex)
92          {
93            vec2 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
94            break;
95          }
96          else if (cellConnections[edge[1]] == pointIndex)
97          {
98            vec2 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
99            break;
100         }
101         else
102         {
103           // Continue to next iteration of loop.
104         }
105       }
106
107       // The dot product of two unit vectors is equal to the cosine of the
108       // angle between them.
109       vtkm::Normalize(vec1);
110       vtkm::Normalize(vec2);
111       SumType cosine = static_cast<SumType>(vtkm::Dot(vec1, vec2));
112
113       outSum += vtkm::ACos(cosine);
114     }
115   }
116 };
```

CHAPTER

TWENTYNINE

EXECUTION OBJECTS

Although passing whole arrays and cell sets into a worklet is a convenient way to provide data to a worklet that is not divided by the input or output domain, they are sometimes not the best structures to represent data. Thus, all worklets support a another type of argument called an *execution object*, or exec object for short, that provides a user-defined object directly to each invocation of the worklet. This is defined by an `ExecObject` tag in the `ControlSignature`.

The execution object must be a subclass of `vtkm::cont::ExecutionObjectBase`. Also, it must implement a `PrepareForExecution` method declared with `VTKM_CONT`. `PrepareForExecution` should take two arguments. The first argument is the device adapter tag. The second argument is a `vtkm::cont::Token` object that should be used to scope any execution objects created internally.

The `PrepareForExecution` function creates an execution object that can be passed from the control environment to the execution environment and be usable in the execution environment, and any method of the produced object used within the worklet must be declared with `VTKM_EXEC` or `VTKM_EXEC_CONT`.

An execution object can refer to an array, but the array reference must be through an array portal for the execution environment. This can be retrieved from the `ArrayHandle::PrepareForInput` method as described in Section 27.4. Other VTK-m data objects, such as the subclasses of `vtkm::cont::CellSet`, have similar methods.

Returning to the example we have in Section 28.1, we are computing triangle quality quickly by looking up the value in a table. In Example 28.1 (page 242) the table is passed directly to the worklet as a whole array. However, there is some additional code involved to get the appropriate index into the table for a given triangle. Let us say that we want to have the ability to compute triangle quality in many different worklets. Rather than pass in a raw array, it would be better to encapsulate the functionality in an object.

We can do that by creating an execution object with a `PrepareForExecution` method that creates an object that has the table stored inside and methods to compute the triangle quality. The following example uses the table built in Example 28.1 to create such an object.

Example 29.1: Using `ExecObject` to access a lookup table in a worklet.

```
1  template<typename Device>
2  class TriangleQualityTableExecutionObject
3  {
4    using TableArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
5    using TablePortalType = typename TableArrayType::ReadPortalType;
6    TablePortalType TablePortal;
7
8  public:
9    VTKM_CONT
10   TriangleQualityTableExecutionObject(const TablePortalType& tablePortal)
11     : TablePortal(tablePortal)
12   {
```

```
  }

  template<typename T>
  VTKM_EXEC vtkm::Float32 GetQuality(const vtkm::Vec<T, 3>& point1,
                                     const vtkm::Vec<T, 3>& point2,
                                     const vtkm::Vec<T, 3>& point3) const
  {
    return detail::LookupTriangleQuality(this->TablePortal, point1, point2, point3);
  }
};

class TriangleQualityTable : public vtkm::cont::ExecutionObjectBase
{
public:
  template<typename Device>
  VTKM_CONT TriangleQualityTableExecutionObject<Device> PrepareForExecution(
    Device,
    vtkm::cont::Token& token) const
  {
    return TriangleQualityTableExecutionObject<Device>(
      detail::GetTriangleQualityTable().PrepareForInput(Device{}, token));
  }
};

struct TriangleQualityWorklet2 : vtkm::worklet::WorkletVisitCellsWithPoints
{
  using ControlSignature = void(CellSetIn cells,
                                FieldInPoint pointCoordinates,
                                ExecObject triangleQualityTable,
                                FieldOutCell triangleQuality);
  using ExecutionSignature = _4(CellShape, _2, _3);
  using InputDomain = _1;

  template<typename CellShape,
           typename PointCoordinatesType,
           typename TriangleQualityTableType>
  VTKM_EXEC vtkm::Float32 operator()(
    CellShape shape,
    const PointCoordinatesType& pointCoordinates,
    const TriangleQualityTableType& triangleQualityTable) const
  {
    if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
    {
      this->RaiseError("Only triangles are supported for triangle quality.");
      return vtkm::Nan32();
    }

    return triangleQualityTable.GetQuality(
      pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
  }
};

//
// Later in the associated Filter class...
//

    TriangleQualityTable triangleQualityTable;

    vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;

    this->Invoke(TriangleQualityWorklet2{},
                 inputDataSet.GetCellSet(),
                 inputPointCoordinatesField,
                 triangleQualityTable,
```

```
77 |                   triangleQualities);
```

CHAPTER

THIRTY

LOCATORS

Locators are a special type of structure that allows you to take a point coordinate in space and then find a topological element that contains or is near that coordinate. VTK-m comes with multiple types of locators, which are categorized by the type of topological element that they find. For example, a *cell locator* takes a coordinate in world space and finds the cell in a `vtkm::cont::DataSet` that contains that cell. Likewise, a *point locator* takes a coordinate in world space and finds a point from a `vtkm::cont::CoordinateSystem` nearby.

Different locators differ in their interface slightly, but they all follow the same basic operation. First, they are constructed and provided with one or more elements of a `vtkm::cont::DataSet`. Then they are built with a call to an `Update` method. The locator can then be passed to a worklet as an `ExecObject`, which will cause the worklet to get a special execution version of the locator that can do the queries.

Did you know?

Other visualization libraries, like VTK-m's big sister toolkit VTK, provide similar locator structures that allow iterative building by adding one element at a time. VTK-m explicitly disallows this use case. Although iteratively adding elements to a locator is undoubtedly useful, such an operation will inevitably bottleneck a highly threaded algorithm in critical sections. This makes iterative additions to locators too costly to support in VTK-m.

30.1 Cell Locators

Cell Locators in VTK-m provide a means of building spatial search structures that can later be used to find a cell containing a certain point. This could be useful in scenarios where the application demands the cell to which a point belongs to to achieve a certain functionality. For example, while tracing a particle's path through a vector field, after every step we lookup which cell the particle has entered to interpolate the velocity at the new location to take the next step.

Using cell locators is a two step process. The first step is to build the search structure. This is done by instantiating one of the `CellLocator` classes, providing a cell set and coordinate system (usually from a `vtkm::cont::DataSet`), and then updating the structure. Once the cell locator is built, it can be used in the execution environment within a filter or worklet.

30.1.1 Building a Cell Locator

All cell locators in VTK-m share the same basic interface for the required features of cell locators. This generic interface provides methods to set the cell set (with `SetCellSet` and `GetCellSet`) and to set the coordinate system (with `SetCoordinates` and `GetCoordinates`). Once the cell set and coordinates are provided, you may call `Update` to construct the search structures. Although `Update` is called from the control environment, the search structure will be built on parallel devices.

Example 30.1: Constructing a `CellLocator`.

```
1  vtkm::cont::CellLocatorGeneral cellLocator;
2  cellLocator.SetCellSet(inDataSet.GetCellSet());
3  cellLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
4  cellLocator.Update();
```

VTK-m currently exposes the implementations of the following Cell Locators.

`vtkm::cont::CellLocatorGeneral` This locator will automatically select another locator to use as its implementation. `CellLocatorGeneral` allows you to automatically select cell locators optimized for certain cell structures without knowing the cell set type. You can change how `CellLocatorGeneral` selects a `CellLocator` by providing a function to the `SetConfigurator` method. If no configurator is set, then a default one is used.

`vtkm::cont::CellLocatorUniformGrid` This locator is optimized for structured data that has uniform axis-aligned spacing. For this cell locator to work, it has to be given a cell set of type `CellSetStructured` and a coordinate system using an `ArrayHandleUniformPointCoordinates` for its data.

`vtkm::cont::CellLocatorRectilinearGrid` This locator is optimized for structured data that has nonuniform axis-aligned spacing. For this cell locator to work, it has to be given a cell set of type `CellSetStructured` and a coordinate system using an `ArrayHandleCartesianProduct` for its data.

`vtkm::cont::CellLocatorTwoLevel` This locator builds a 2-level hierarchy of uniform bins. The first level is a coarse partitioning of the space. Each bin in the first level has a second grid who's size depends on the number of cells in the first level. The density (number of cells expected in each bin) for each level can be set with `SetDensityL1` and `SetDensityL2`. Their default values are 32 and 2, respectively.

`vtkm::cont::CellLocatorBoundingIntervalHierarchy` This locator is based on the bounding interval hierarchy spatial search structure. `CellLocatorBoundingIntervalHierarchy` takes two parameters: the number of splitting planes used to split the cells uniformly along an axis at each level and the maximum leaf size, which determines if a node needs to be split further. These parameters can set through the `SetNumberOfPlanes` and `SetMaxLeafSize` methods.

`vtkm::cont::CellLocatorChooser` This locator is similar to `CellLocatorGeneral` in that it automatically selects an appropriate locator based on the type of cell structure being used. However, unlike the other locator, `CellLocatorChooser` is a templated class that chooses the correct locator based on the cell set type and the coordinates array type provided as template arguments. This means that you have to know the data types at compile time. `CellLocatorChooser` is a good choice in applications where you know your `DataSet` is of a particular type.

30.1.2 Using Cell Locators in a Worklet

The `CellLocator` interface implements `vtkm::cont::ExecutionObjectBase`. This means that any `CellLocator` can be used in worklets as an `ExecObject` argument (as defined in the ControlSignature). See Chapter 29 for information on `ExecObject` arguments to worklets.

When a `vtkm::cont::CellLocator` class is passed as an `ExecObject` argument to a worklet `Invoke`, the worklet receives a different object defined in the `vtkm::exec` namespace. This `vtkm::exec::CellLocator` object provides a `FindCell` method that identifies a containing cell given a point location in space.

Common Errors

Note that the `vtkm::cont::CellLocator` and `vtkm::exec::CellLocator` classes are different objects with different interfaces despite the similar names.

The `CellLocator::FindCell` method takes 3 arguments. The first argument is an input query point. The second argument is used to return the id of the cell containing this point (or -1 if the point is not found in any cell). The third argument is used to return the parametric coordinates for the point within the cell (assuming it is found in any cell). `FindCell` returns an `ErrorCode` to indicate the status of the query. If the cell and the location within the cell are found, `ErrorCode::Success` is returned. If the point is not inside any cell, `ErrorCode::CellNotFound` is likely to be returned.

The following example defines a simple worklet to get the value of a point field interpolated to a group of query point coordinates provided.

Example 30.2: Using a `CellLocator` in a worklet.

```
1  struct QueryCellsWorklet : public vtkm::worklet::WorkletMapField
2  {
3    using ControlSignature =
4      void(FieldIn, ExecObject, WholeCellSetIn<Cell, Point>, WholeArrayIn, FieldOut);
5    using ExecutionSignature = void(_1, _2, _3, _4, _5);
6
7    template<typename Point,
8             typename CellLocatorExecObject,
9             typename CellSet,
10            typename FieldPortal,
11            typename OutType>
12   VTKM_EXEC void operator()(const Point& point,
13                             const CellLocatorExecObject& cellLocator,
14                             const CellSet& cellSet,
15                             const FieldPortal& field,
16                             OutType& out) const
17   {
18     // Use the cell locator to find the cell containing the point and the parametric
19     // coordinates within that cell.
20     vtkm::Id cellId;
21     vtkm::Vec3f parametric;
22     vtkm::ErrorCode status = cellLocator.FindCell(point, cellId, parametric);
23     if (status != vtkm::ErrorCode::Success)
24     {
25       this->RaiseError(vtkm::ErrorString(status));
26     }
27
28     // Use this information to interpolate the point field to the given location.
29     if (cellId >= 0)
30     {
31       // Get shape information about the cell containing the point coordinate
32       auto cellShape = cellSet.GetCellShape(cellId);
33       auto indices = cellSet.GetIndices(cellId);
34
35       // Make a Vec-like containing the field data at the cell's points
36       auto fieldValues = vtkm::make_VecFromPortalPermute(&indices, &field);
37
```

```
      // Do the interpolation
      vtkm::exec::CellInterpolate(fieldValues, parametric, cellShape, out);
    }
    else
    {
      this->RaiseError("Given point outside of the cell set.");
    }
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::CellLocatorGeneral cellLocator;
    cellLocator.SetCellSet(inDataSet.GetCellSet());
    cellLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    cellLocator.Update();

    vtkm::cont::ArrayHandle<FieldType> interpolatedField;

    this->Invoke(QueryCellsWorklet{},
                 this->QueryPoints,
                 &cellLocator,
                 inDataSet.GetCellSet(),
                 inputField,
                 interpolatedField);
```

30.2 Point Locators

Point Locators in VTK-m provide a means of building spatial search structures that can later be used to find the nearest neighbor a certain point. This could be useful in scenarios where the closest pairs of points are needed. For example, during halo finding of particles in cosmology simulations, pairs of nearest neighbors within certain linking length are used to form clusters of particles.

Using point locators is a two step process. The first step is to build the search structure. This is done by instantiating one of the `vtkm::cont::PointLocator` classes, providing a coordinate system (usually from a `vtkm::cont::DataSet`) representing the location of points that can later be found through queries, and then updating the structure. Once the point locator is built, it can be used in the execution environment within a filter or worklet.

30.2.1 Building Point Locators

All point locators in VTK-m share the same basic interface for the required features of point locators. This generic interface provides methods to set the coordinate system (with `SetCoordinates` and `GetCoordinates`) of training points. Once the coordinates are provided, you may call `Update` to construct the search structures. Although `Update` is called from the control environment, the search structure will be built on parallel devices

Example 30.3: Constructing a `PointLocator`.

```
vtkm::cont::PointLocatorSparseGrid pointLocator;
pointLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
pointLocator.Update();
```

VTK-m currently exposes the implementations of the following Point Locators.

`vtkm::cont::PointLocatorSparseGrid` This point locator is based on the uniform grid search structure. It divides the search space into a uniform grid of bins. A search for a point near a given coordinate starts in the bin containing the search coordinates. If a candidate point is not found in that bin, points are searched in an expanding neighborhood of grid bins. The size of the grid used by the locator to partition the space can be set with `SetNumberOfBins`. By default, `PointLocatorSparseGrid` uses a 32^3 grid. It is also possible to set the physical space over which the search space is constructed with the `SetRange` method. If the range is not set, it will automatically be set to the space of the coordinates.

30.2.2 Using Point Locators in a Worklet

The `PointLocator` interface implements `vtkm::cont::ExecutionObjectBase`. This means that any `PointLocator` can be used in worklets as an `ExecObject` argument (as defined in the `ControlSignature`). See Chapter 29 for information on `ExecObject` arguments to worklets.

When a `vtkm::cont::PointLocator` class is passed as an `ExecObject` argument to a worklet `Invoke`, the worklet receives a different object defined in the `vtkm::exec` namespace. This `vtkm::exec::PointLocator` object provides a `FindNearestNeighbor` method that identifies the nearest neighbor point given a coordinate in space.

Common Errors

Note that `vtkm::cont::PointLocator` and `vtkm::exec::PointLocator` are different objects with different interfaces despite the similar names.

The `FindNearestNeighbor` method takes 3 arguments. The first argument is an input query point. The second argument is used to return the id of the nearest neighbor point (or -1 if no nearby point is found, for example, in the case of an empty set of data set points). The third argument is used to return the squared distance for the query point to its nearest neighbor.

Example 30.4: Using a `PointLocator` in a worklet.

```
1  /// Worklet that generates for each input coordinate a unit vector that points
2  /// to the closest point in a locator.
3  struct PointToClosestWorklet : public vtkm::worklet::WorkletMapField
4  {
5    using ControlSignature = void(FieldIn, ExecObject, WholeArrayIn, FieldOut);
6    using ExecutionSignature = void(_1, _2, _3, _4);
7
8    template<typename Point,
9             typename PointLocatorExecObject,
10            typename CoordinateSystemPortal,
11            typename OutType>
12   VTKM_EXEC void operator()(const Point& queryPoint,
13                             const PointLocatorExecObject& pointLocator,
14                             const CoordinateSystemPortal& coordinateSystem,
15                             OutType& out) const
16   {
17     // Use the point locator to find the point in the locator closest to the point
18     // given.
19     vtkm::Id pointId;
20     vtkm::FloatDefault distanceSquared;
21     pointLocator.FindNearestNeighbor(queryPoint, pointId, distanceSquared);
22
23     // Use this information to find the nearest point and create a unit vector
```

```
    // pointing to it.
    if (pointId >= 0)
    {
      // Get nearest point coordinate.
      auto point = coordinateSystem.Get(pointId);

      // Get the vector pointing to this point
      out = point - queryPoint;

      // Convert to unit vector (if possible)
      if (distanceSquared > vtkm::Epsilon<vtkm::FloatDefault>())
      {
        out = vtkm::RSqrt(distanceSquared) * out;
      }
    }
    else
    {
      this->RaiseError("Locator could not find closest point.");
    }
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::PointLocatorSparseGrid pointLocator;
    pointLocator.SetCoordinates(inDataSet.GetCoordinateSystem());
    pointLocator.Update();

    vtkm::cont::ArrayHandle<vtkm::Vec3f> pointDirections;

    this->Invoke(PointToClosestWorklet{},
                 this->QueryPoints,
                 &pointLocator,
                 pointLocator.GetCoordinates(),
                 pointDirections);
```

CHAPTER

THIRTYONE

WORKLET INPUT OUTPUT SEMANTICS

The default scheduling of a worklet provides a 1 to 1 mapping from the input domain to the output domain. For example, a `vtkm::worklet::WorkletMapField` gets run once for every item of the input array and produces one item for the output array. Likewise, `vtkm::worklet::WorkletVisitCellsWithPoints` gets run once for every cell in the input topology and produces one associated item for the output field.

However, there are many operations that do not fall well into this 1 to 1 mapping procedure. The operation might need to pass over elements that produce no value or the operation might need to produce multiple values for a single input element. Such non 1 to 1 mappings can be achieved by defining a scatter or a mask (or both) on a worklet.

31.1 Scatter

A *scatter* allows you to specify for each input element how many output elements should be created. For example, a scatter allows you to create two output elements for every input element. A scatter could also allow you to drop every other input element from the output. The following types of scatter are provided by VTK-m.

`vtkm::worklet::ScatterIdentity` Provides a basic 1 to 1 mapping from input to output. This is the default scatter used if none is specified.

`vtkm::worklet::ScatterUniform` Provides a 1 to many mapping from input to output with the same number of outputs for each input. A template parameter provides the number of output values to produce per input.

`vtkm::worklet::ScatterCounting` Provides a 1 to any mapping from input to output with different numbers of outputs for each input. The constructor takes an `ArrayHandle` that is the same size as the input containing the count of output values to produce for each input. Values can be zero, in which case that input will be skipped.

`vtkm::worklet::ScatterPermutation` Reorders the indices. The constructor takes a permutation `ArrayHandle` that is sized to the number of output values and maps output indices to input indices. For example, if index i of the permutation array contains j, then the worklet invocation for output i will get the j^{th} input values. The reordering does not have to be 1 to 1. Any input not referenced by the permutation array will be dropped, and any input referenced by the permutation array multiple times will be duplicated. However, unlike `ScatterCounting` VisitIndex is always 0 even if an input value happens to be duplicated.

Did you know?

Scatters are often used to create multiple outputs for a single input, but they can also be used to remove inputs from the output. In particular, if you provide a count of 0 in a `ScatterCounting` *count array, no outputs will be created for the associated input. To simply mask out some elements from the input, provide* `ScatterCounting` *with a stencil array of 0's and 1's with a 0 for every element you want to remove and a 1 for every element you want to pass. You can also mix 0's with counts larger than 1 to drop some elements and add multiple results for other elements.* `ScatterPermutation` *can similarly be used to remove input values by leaving them out of the permutation.*

To define a scatter procedure, the worklet must provide a type definition named `ScatterType`. The `ScatterType` must be set to one of the aforementioned `Scatter*` classes. It is common, but optional, to also provide a static method named `MakeScatter` that generates an appropriate scatter object for the worklet if you cannot use the default constructor for the scatter. This static method can be used by users of the worklet to set up the scatter for the `Invoker`.

Example 31.1: Declaration of a scatter type in a worklet.

```
1  using ScatterType = vtkm::worklet::ScatterCounting;
2
3  template<typename CountArrayType>
4  VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray)
5  {
6    VTKM_IS_ARRAY_HANDLE(CountArrayType);
7    return ScatterType(countArray);
8  }
```

When using a scatter that produces multiple outputs for a single input, the worklet is invoked multiple times with the same input values. In such an event the worklet operator needs to distinguish these calls to produce the correct associated output. This is done by declaring one of the `ExecutionSignature` arguments as `VisitIndex`. This tag will pass a `vtkm::IdComponent` to the worklet that identifies which invocation is being called.

It is also the case that the when a scatter can produce multiple outputs for some input that the index of the input element is not the same as the `WorkIndex`. If the index to the input element is needed, you can use the `InputIndex` tag in the `ExecutionSignature`. It is also good practice to use the `OutputIndex` tag if the index to the output element is needed.

Most `Scatter` objects have a state, and this state must be passed to the `vtkm::cont::Invoker` when invoking the worklet. In this case, the `Scatter` object should be passed as the second object to the call to the `Invoker` (after the worklet object).

Example 31.2: Invoking with a custom scatter.

```
1  vtkm::worklet::ScatterCounting generateScatter =
2    ClipPoints::Generate::MakeScatter(countArray);
3  this->Invoke(
4    ClipPoints::Generate{}, generateScatter, inField, clippedPointsArray);
```

Did you know?

A scatter object does not have to be tied to a single worklet/invoker instance. In some cases it makes sense to use the same scatter object multiple times for worklets that have the same input to output mapping. Although this is not common, it can save time by reusing the set up computations of `ScatterCounting`.

To demonstrate using scatters with worklets, we provide some contrived but illustrative examples. The first example is a worklet that takes a pair of input arrays and interleaves them so that the first, third, fifth, and so on entries come from the first array and the second, fourth, sixth, and so on entries come from the second array. We achieve this by using a `vtkm::worklet::ScatterUniform` of size 2 and using the `VisitIndex` to determine from which array to pull a value.

Example 31.3: Using `ScatterUniform`.

```
struct InterleaveArrays : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn, FieldIn, FieldOut);
  using ExecutionSignature = void(_1, _2, _3, VisitIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterUniform<2>;

  template<typename T>
  VTKM_EXEC void operator()(const T& input0,
                            const T& input1,
                            T& output,
                            vtkm::IdComponent visitIndex) const
  {
    if (visitIndex == 0)
    {
      output = input0;
    }
    else // visitIndex == 1
    {
      output = input1;
    }
  }
};
```

The second example takes a collection of point coordinates and clips them by an axis-aligned bounding box. It does this using a `vtkm::worklet::ScatterCounting` with an array containing 0 for all points outside the bounds and 1 for all points inside the bounds. As is typical with this type of operation, we use another worklet with a default identity scatter to build the count array.

Example 31.4: Using `ScatterCounting`.

```
struct ClipPoints
{
  class Count : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn points, FieldOut count);
    using ExecutionSignature = _2(_1);
    using InputDomain = _1;

    VTKM_CONT Count(const vtkm::Bounds& bounds)
      : Bounds(bounds)
    {
    }

    template<typename T>
    VTKM_EXEC vtkm::IdComponent operator()(const vtkm::Vec<T, 3>& point) const
    {
      return (this->Bounds.Contains(point) ? 1 : 0);
    }

  private:
    vtkm::Bounds Bounds;
  };
```

```
24
25    class Generate : public vtkm::worklet::WorkletMapField
26    {
27    public:
28      using ControlSignature = void(FieldIn inPoints, FieldOut outPoints);
29      using ExecutionSignature = void(_1, _2);
30      using InputDomain = _1;
31
32      using ScatterType = vtkm::worklet::ScatterCounting;
33
34      template<typename CountArrayType>
35      VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray)
36      {
37        VTKM_IS_ARRAY_HANDLE(CountArrayType);
38        return ScatterType(countArray);
39      }
40
41      template<typename InType, typename OutType>
42      VTKM_EXEC void operator()(const vtkm::Vec<InType, 3>& inPoint,
43                                vtkm::Vec<OutType, 3>& outPoint) const
44      {
45        // The scatter ensures that this method is only called for input points
46        // that are passed to the output (where the count was 1). Thus, in this
47        // case we know that we just need to copy the input to the output.
48        outPoint = vtkm::Vec<OutType, 3>(inPoint[0], inPoint[1], inPoint[2]);
49      }
50    };
51  };
52
53  //
54  // Later in the associated Filter class...
55  //
56
57      vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray;
58
59      this->Invoke(ClipPoints::Count(this->Bounds), inField, countArray);
60
61      vtkm::cont::ArrayHandle<T> clippedPointsArray;
62
63      vtkm::worklet::ScatterCounting generateScatter =
64        ClipPoints::Generate::MakeScatter(countArray);
65      this->Invoke(
66        ClipPoints::Generate{}, generateScatter, inField, clippedPointsArray);
```

The third example takes an input array and reverses the ordering. It does this using a `vtkm::worklet::ScatterPermutation` with a permutation array generated from a `vtkm::cont::ArrayHandleCounting` counting down from the input array size to 0.

Example 31.5: Using `ScatterPermutation`.

```
1  struct ReverseArrayWorklet : vtkm::worklet::WorkletMapField
2  {
3    using ControlSignature = void(FieldIn inputArray, FieldOut outputArray);
4    using ExecutionSignature = void(_1, _2);
5    using InputDomain = _1;
6
7    using ArrayStorageTag =
8      typename vtkm::cont::ArrayHandleCounting<vtkm::Id>::StorageTag;
9    using ScatterType = vtkm::worklet::ScatterPermutation<ArrayStorageTag>;
10
11   VTKM_CONT
12   static ScatterType MakeScatter(vtkm::Id arraySize)
13   {
14     return ScatterType(
```

```
      vtkm::cont::ArrayHandleCounting<vtkm::Id>(arraySize - 1, -1, arraySize));
  }

  template<typename FieldType>
  VTKM_EXEC void operator()(FieldType inputArrayField,
                            FieldType& outputArrayField) const
  {
    outputArrayField = inputArrayField;
  }
};

//
// Later in the associated Filter class...
//

    vtkm::cont::ArrayHandle<T> outputField;
    this->Invoke(ReverseArrayWorklet{},
                 ReverseArrayWorklet::MakeScatter(inputField.GetNumberOfValues()),
                 inputField,
                 outputField);
```

Did you know?

A `vtkm::worklet::ScatterPermutation` *can have less memory usage than a* `vtkm::worklet::ScatterCounting` *when zeroing indices. By default, a* `vtkm::worklet::ScatterPermutation` *will omit all fields that are not specified in the input permutation, whereas* `vtkm::worklet::ScatterCounting` *requires 0 values. If mapping an input to an output that omits fields, consider using a* `vtkm::worklet::ScatterPermutation` *to save memory.*

Common Errors

A permutation array provided to `vtkm::worklet::ScatterPermutation` *can be filled with arbitrary id values. If an input permutation id exceeds the bounds of an input provided to a* `worklet`*, an out of bounds error will occur in the worklet functor. To prevent this kind of error, you should ensure that ids in the* `vtkm::worklet::ScatterPermutation` *do not exceed the bounds of provided inputs.*

CHAPTER

THIRTYTWO

GENERATING CELL SETS

This chapter describes techniques for designing algorithms in VTK-m that generate cell sets to be inserted in a `vtkm::cont::DataSet`. Although Chapter 7 on data sets describes how to create a data set, including defining its set of cells, these are serial functions run in the control environment that are not designed for computing geometric structures. Rather, they are designed for specifying data sets built from existing data arrays, from inherently slow processes (such as file I/O), or for small test data. In this chapter we discuss how to write worklets that create new mesh topologies by writing data that can be incorporated into a `vtkm::cont::CellSet`.

This chapter is constructed as a set of patterns that are commonly employed to build cell sets. These techniques apply the worklet structures documented in Chapter 21. Although it is possible for these worklets to generate data of its own, the algorithms described here follow the more common use case of deriving one topology from another input data set. This chapter is not (and cannot be) completely comprehensive by covering every possible mechanism for building cell sets. Instead, we provide the basic and common patterns used in scientific visualization.

32.1 Single Cell Type

For our first example of algorithms that generate cell sets is one that creates a set of cells in which all the cells are of the same shape and have the same number of points. Our motivating example is an algorithm that will extract all the edges from a cell set. The resulting cell set will comprise a collection of line cells that represent the edges from the original cell set. Since all cell edges can be represented as lines with two endpoints, we know all the output cells will be of the same type. As we will see later in the example, we can use a `vtkm::cont::CellSetSingleType` to represent the data.

It is rare that an algorithm generating a cell set will generate exactly one output cell for each input cell. Thus, the first step in an algorithm generating a cell set is to count the number of cells each input item will create. In our motivating example, this is the the number of edges for each input cell.

Example 32.1: A simple worklet to count the number of edges on each cell.

```
1  struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
4    using ExecutionSignature = _2(CellShape, PointCount);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC_CONT vtkm::IdComponent operator()(
9      CellShapeTag cellShape,
10     vtkm::IdComponent numPointsInCell) const
11   {
```

```
    vtkm::IdComponent numEdges;
    vtkm::ErrorCode status =
      vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
    if (status != vtkm::ErrorCode::Success)
    {
      // There is an error in the cell. As good as it would be to return an
      // error, we probably don't want to invalidate the entire run if there
      // is just one malformed cell. Instead, ignore the cell.
      return 0;
    }
    return numEdges;
  }
};
```

This count array generated in Example 32.1 can be used in a `vtkm::worklet::ScatterCounting` of a subsequent worklet that generates the output cells. (See Section 31.1 for information on using a scatter with a worklet.) We will see this momentarily.

Did you know?

If you happen to have an operation that you know will have the same count for every input cell, then you can skip the count step and use a `vtkm::worklet::ScatterUniform` *instead of* `ScatterCount`*. Doing so will simplify the code and skip some computation. We cannot use* `ScatterUniform` *in this example because different cell shapes have different numbers of edges and therefore different counts. However, if we were theoretically to make an optimization for 3D structured grids, we know that each cell is a hexahedron with 12 edges and could use a* `ScatterUniform<12>` *for that.*

The second and final worklet we need to generate our wireframe cells is one that outputs the indices of an edge. The worklet parenthesis' operator takes information about the input cell (shape and point indices) and an index of which edge to output. The aforementioned `ScatterCounting` provides a `VisitIndex` that signals which edge to output. The worklet parenthesis operator returns the two indices for the line in, naturally enough, a `vtkm::Id2`.

Example 32.2: A worklet to generate indices for line cells.

```
class EdgeIndicesWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut connectivityOut);
  using ExecutionSignature = void(CellShape, PointIndices, _2, VisitIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::Id2& connectivityOut,
                            vtkm::IdComponent edgeIndex) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();

    vtkm::IdComponent pointInCellIndex0;
    vtkm::exec::CellEdgeLocalIndex(
      numPointsInCell, 0, edgeIndex, cellShape, pointInCellIndex0);
    vtkm::IdComponent pointInCellIndex1;
    vtkm::exec::CellEdgeLocalIndex(
```

```
24        numPointsInCell, 1, edgeIndex, cellShape, pointInCellIndex1);
25
26      connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
27      connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
28    }
29  };
```

Our ultimate goal is to fill a `vtkm::cont::CellSetSingleType` object with the generated line cells. A `CellSetSingleType` requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively. The last item, the array of connection indices, is what we are creating with the worklet in Example 32.2.

However, there is a complication. The connectivity array for `CellSetSingleType` is expected to be a flat array of `vtkm::Id` indices, not an array of `Vec` objects. We could jump through some hoops adjusting the `ScatterCounting` to allow the worklet to output only one index of one cell rather than all indices of one cell. But that would be overly complicated and inefficient.

A simpler approach is to use the `vtkm::cont::ArrayHandleGroupVec` fancy array handle (described in Section 26.13) to make a flat array of indices look like an array of `Vec` objects. The following example shows what the `DoExecute` method in the associated filter would look like. Note the use `make_ArrayHandleGroupVec` when calling `Invoke` on line 16 to make this conversion.

Example 32.3: Invoking worklets to extract edges from a cell set.

```
1  inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
2    const vtkm::cont::DataSet& inData)
3  {
4    auto inCellSet = inData.GetCellSet();
5
6    // Count number of edges in each cell.
7    vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
8    this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);
9
10   // Build the scatter object (for non 1-to-1 mapping of input to output)
11   vtkm::worklet::ScatterCounting scatter(edgeCounts);
12   auto outputToInputCellMap =
13     scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
14
15   vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
16   this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
17                scatter,
18                inCellSet,
19                vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));
20
21   vtkm::cont::CellSetSingleType<> outCellSet;
22   outCellSet.Fill(
23     inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);
24
25   // This lambda function maps an input field to the output data set. It is
26   // used with the CreateResult method.
27   auto fieldMapper =
28     [&](vtkm::cont::DataSet& outData, const vtkm::cont::Field& inputField)
29   {
30     if (inputField.IsCellField())
31     {
32       vtkm::filter::MapFieldPermutation(inputField, outputToInputCellMap, outData);
33     }
34     else
35     {
36       outData.AddField(inputField); // pass through
```

```
37      }
38    };
39
40    return this->CreateResult(inData, outCellSet, fieldMapper);
41  }
```

Another feature to note in Example 32.3 is that the method calls `GetOutputToInputMap` on the `Scatter` object it creates and squirrels the map array away for later use (line 12). The reason for this behavior is to implement mapping fields that are attached on the input cells to the indices of the output. In practice, `DoExecute` is called on `DataSet` objects to create new `DataSet` objects. The method in Example 32.3 creates a new `CellSet`, but we also need a method to transform the `Field`s on the data set. The saved `outputToInputCellMap` array allows us to transform input fields to output fields.

The lambda function in Example 32.3 starting on line 27 uses this saved `outputToInputCellMap` array and converts an array from an input cell field to an output cell field array. It does this using the `vtkm::filter::MapFieldPermutation` helper function while using the `outputToInputCellMap` as the permutation array.

32.2 Combining Like Elements

Our motivating example in Section 32.1 created a cell set with a line element representing each edge in some input data set. However, on close inspection there is a problem with our algorithm: it is generating a lot of duplicate elements. The cells in a typical mesh are connected to each other. As such, they share edges with each other. That is, the edge of one cell is likely to also be part of one or more other cells. When multiple cells contain the same edge, the algorithm we created in Section 32.1 will create multiple overlapping lines, one for each cell using the edge, as demonstrated in Figure 32.1. What we really want is to have one line for every edge in the mesh rather than many overlapping lines.

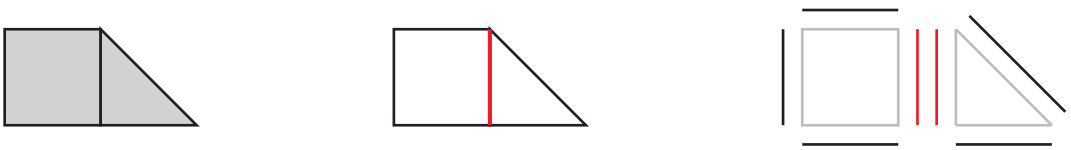


Figure 32.1: Duplicate lines from extracted edges. Consider the small mesh at the left comprising a square and a triangle. If we count the edges in this mesh, we would expect to get 6. However, our naïve implementation in Section 32.1 generates 7 because the shared edge (highlighted in red in the wireframe in the middle) is duplicated. As seen in the exploded view at right, one line is created for the square and one for the triangle.

In this section we will re-implement the algorithm to generate a wireframe by creating a line for each edge, but this time we will merge duplicate edges together. Our first step is the same as before. We need to count the number of edges in each input cell and use those counts to create a `vtkm::worklet::ScatterCounting` for subsequent worklets. Counting the edges is a simple worklet.

Example 32.4: A simple worklet to count the number of edges on each cell.

```
1  struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
4    using ExecutionSignature = _2(CellShape, PointCount);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC_CONT vtkm::IdComponent operator()(
9      CellShapeTag cellShape,
10     vtkm::IdComponent numPointsInCell) const
```

```
  {
    vtkm::IdComponent numEdges;
    vtkm::ErrorCode status =
      vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
    if (status != vtkm::ErrorCode::Success)
    {
      // There is an error in the cell. As good as it would be to return an
      // error, we probably don't want to invalidate the entire run if there
      // is just one malformed cell. Instead, ignore the cell.
      return 0;
    }
    return numEdges;
  }
};
```

In our previous version, we used the count to directly write out the lines. However, before we do that, we want to identify all the unique edges and identify which cells share this edge. This grouping is exactly the function that the reduce by key worklet type (described in Section 21.4) is designed to accomplish. The principal idea is to write a "key" that uniquely identifies the edge. The reduce by key worklet can then group the edges by the key and allow you to combine the data for the edge.

Thus, our goal of finding duplicate edges hinges on producing a key where two keys are identical if and only if the edges are the same. One straightforward key is to use the coordinates in 3D space by, say, computing the midpoint of the edge. The main problem with using this point coordinates approach is that a computer can hold a point coordinate only with floating point numbers of limited precision. Computer floating point computations are notorious for providing slightly different answers when the results should be the same. For example, if an edge has endpoints at p_1 and p_2 and two different cells compute the midpoint as $(p_1 + p_2)/2$ and $(p_2 + p_1)/2$, respectively, the answer is likely to be slightly different. When this happens, the keys will not be the same and we will still produce 2 edges in the output.

Fortunately, there is a better choice for keys based on the observation that in the original cell set each edge is specified by endpoints that each have unique indices. We can combine these 2 point indices to form a "canonical" descriptor of an edge (correcting for order).[1] VTK-m comes with a helper function, `vtkm::exec::CellEdgeCanonicalId`, defined in `vtkm/exec/CellEdge.h`, to produce these unique edge keys as `vtkm::Id2` s. Our second worklet produces these canonical edge identifiers.

Example 32.5: Worklet generating canonical edge identifiers.

```
class EdgeIdsWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut canonicalIds);
  using ExecutionSignature = void(CellShape cellShape,
                                  PointIndices globalPointIndices,
                                  VisitIndex localEdgeIndex,
                                  _2 canonicalIdOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC void operator()(CellShapeTag cellShape,
                            const PointIndexVecType& globalPointIndicesForCell,
                            vtkm::IdComponent localEdgeIndex,
                            vtkm::Id2& canonicalIdOut) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();
```

[1] Using indices to find common mesh elements is described by Miller et al. in "Finely-Threaded History-Based Topology Computation" (in *Eurographics Symposium on Parallel Graphics and Visualization*, June 2014).

```
21
22      vtkm::ErrorCode status =
23        vtkm::exec::CellEdgeCanonicalId(numPointsInCell,
24                                        localEdgeIndex,
25                                        cellShape,
26                                        globalPointIndicesForCell,
27                                        canonicalIdOut);
28      if (status != vtkm::ErrorCode::Success)
29      {
30        this->RaiseError(vtkm::ErrorString(status));
31      }
32    }
33  };
```

Our third and final worklet generates the line cells by outputting the indices of each edge. As hinted at earlier, this worklet is a reduce by key worklet (inheriting from `vtkm::worklet::WorkletReduceByKey`). When the worklet is invoked, VTK-m will collect the unique keys and call the worklet once for each unique edge. Because there is no longer a consistent mapping from the generated lines to the elements of the input cell set, we need pairs of indices identifying the cells/edges from which the edge information comes. We use these indices along with a connectivity structure produced by a `WholeCellSetIn` to find the information about the edge. As shown later, these indices of cells and edges can be extracted from the `ScatterCounting` used to execute the worklet back in Example 32.5.

As we did in Section 32.1, this worklet writes out the edge information in a `vtkm::Id2` (which in some following code will be created with an `ArrayHandleGroupVec`).

Example 32.6: A worklet to generate indices for line cells from combined edges.

```
1  class EdgeIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
2  {
3  public:
4    using ControlSignature = void(KeysIn keys,
5                                  WholeCellSetIn<> inputCells,
6                                  ValuesIn originCells,
7                                  ValuesIn originEdges,
8                                  ReducedValuesOut connectivityOut);
9    using ExecutionSignature = void(_2 inputCells,
10                                   _3 originCell,
11                                   _4 originEdge,
12                                   _5 connectivityOut);
13   using InputDomain = _1;
14
15   template<typename CellSetType, typename OriginCellsType, typename OriginEdgesType>
16   VTKM_EXEC void operator()(const CellSetType& cellSet,
17                             const OriginCellsType& originCells,
18                             const OriginEdgesType& originEdges,
19                             vtkm::Id2& connectivityOut) const
20   {
21     // Regardless of how many cells are sharing the edge we are generating, we
22     // know that each cell/edge given to us by the reduce-by-key refers to the
23     // same edge, so we can just look at the first cell to get the edge.
24     vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(originCells[0]);
25     vtkm::IdComponent edgeIndex = originEdges[0];
26     auto cellShape = cellSet.GetCellShape(originCells[0]);
27
28     vtkm::IdComponent pointInCellIndex0;
29     vtkm::exec::CellEdgeLocalIndex(
30       numPointsInCell, 0, edgeIndex, cellShape, pointInCellIndex0);
31     vtkm::IdComponent pointInCellIndex1;
32     vtkm::exec::CellEdgeLocalIndex(
33       numPointsInCell, 1, edgeIndex, cellShape, pointInCellIndex1);
34
35     auto globalPointIndicesForCell = cellSet.GetIndices(originCells[0]);
```

```
36      connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
37      connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
38    }
39  };
```

Did you know?

It so happens that the `vtkm::Id2` *s generated by* `CellEdgeCanonicalId` *contain the point indices of the two endpoints, which is enough information to create the edge. Thus, in this example it would be possible to forgo the steps of looking up indices through the cell set. That said, this is more often not the case, so for the purposes of this example we show how to construct cells without depending on the structure of the keys.*

With these 3 worklets, it is now possible to generate all the information we need to fill a `vtkm::cont::CellSetSingleType` object. A `CellSetSingleType` requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively.

The last item, the array of connection indices, is what we are creating with the worklet in Example 32.6. The connectivity array for `CellSetSingleType` is expected to be a flat array of `vtkm::Id` indices, but the worklet needs to provide groups of indices for each cell (in this case as a `Vec` object). To reconcile what the worklet provides and what the connectivity array must look like, we use the `vtkm::cont::ArrayHandleGroupVec` fancy array handle (described in Section 26.13) to make a flat array of indices look like an array of `Vec` objects. The following example shows what the `DoExecute` method in the associated filter would look like. Note the use of `make_ArrayHandleGroupVec` when calling `Invoke` on line 25 to make this conversion.

Example 32.7: Invoking worklets to extract unique edges from a cell set.

```
1  inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
2    const vtkm::cont::DataSet& inData)
3  {
4    auto inCellSet = inData.GetCellSet();
5
6    // First, count the edges in each cell.
7    vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
8    this->Invoke(CountEdgesWorklet{}, inCellSet, edgeCounts);
9
10   // Second, using these counts build a scatter that repeats a cell's visit
11   // for each edge in the cell.
12   vtkm::worklet::ScatterCounting scatter(edgeCounts);
13   vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
14     scatter.GetVisitArray(inCellSet.GetNumberOfCells());
15
16   // Third, for each edge, extract a canonical id.
17   vtkm::cont::ArrayHandle<vtkm::Id2> canonicalIds;
18   this->Invoke(EdgeIdsWorklet{}, scatter, inCellSet, canonicalIds);
19
20   // Fourth, construct a Keys object to combine all like edge ids.
21   vtkm::worklet::Keys<vtkm::Id2> cellToEdgeKeys(canonicalIds);
22
23   // Fifth, use a reduce-by-key to extract indices for each unique edge.
24   vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
25   this->Invoke(EdgeIndicesWorklet{},
26                cellToEdgeKeys,
27                inCellSet,
28                scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells()),
```

```
29                outputToInputEdgeMap,
30                vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));
31
32    // Sixth, use the created connectivity array to build a cell set.
33    vtkm::cont::CellSetSingleType<> outCellSet;
34    outCellSet.Fill(
35      inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);
36
37    // Finally, we need to create an output data set. A lambda expression is one
38    // of the easiest ways to map fields from the input to the output with the
39    // CreateResult method.
40    auto fieldMapper =
41      [&](vtkm::cont::DataSet& outData, const vtkm::cont::Field& inField)
42    {
43      if (inField.IsCellField())
44      {
45        // New cells were created. Need to find cells that created the output.
46        // First, the cells were subselected with a scatter. Use the
47        // output-to-input array from the scatter to permute the array.
48        vtkm::cont::Field subselectionField;
49        vtkm::filter::MapFieldPermutation(
50          inField,
51          scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells()),
52          subselectionField);
53        // Next, coicident edges are combined together. Use the keys object
54        // for combining the cells to average out the cell values.
55        vtkm::filter::MapFieldMergeAverage(subselectionField, cellToEdgeKeys, outData);
56      }
57      else
58      {
59        outData.AddField(inField); // Pass through
60      }
61    };
62
63    return this->CreateResult(inData, outCellSet, fieldMapper);
64  }
```

Another feature to note in Example 32.7 is that because the cells returned in the output data are not the same as the input, the output cell fields must be similarly converted. This is done by creating a lambda function (lines 37–61) to convert the fields that is then passed to `CreateResult` (line 63). The mapping process reuses the object from before to extract the edges from the cells. It first uses `GetOutputToInputMap` on the `Scatter` object it creates with a convenience function named `vtkm::filter::MapFieldPermutation` that duplicates the cell values for each edge. It then uses the `vtkm::worklet::Keys` object from the duplicate edge removal with a convenience function named `vtkm::filter::MapFieldMergeAverage` that averages cell values for edges of adjacent cells.

Did you know?

For simplicity, Example 32.7 is creating an intermediate array to hold the permutation. It would be possible to remove this temporary array for saved performance and memory, but this requires building a custom mapping function, which adds complexity. We will show an example of such a function in the following section.

32.3 Faster Combining Like Elements with Hashes

In the previous two sections we constructed worklets that took a cell set and created a new set of cells that represented the edges of the original cell set, which can provide a wireframe of the mesh. In Section 32.1 we provided a pair of worklets that generate one line per edge per cell. In Section 32.2 we improved on this behavior by using a reduce by key worklet to find and merge shared edges.

If we were to time all the operations run in the later implementation to generate the wireframe (i.e. the operations in Example 32.7), we would find that the vast majority of the time is not spent in the actual worklets. Rather, the majority of the time is spent in collecting the like keys, which happens in the constructor of the `vtkm::worklet::Keys` object. Internally, keys are collected by sorting them. The most fruitful way to improve the performance of this algorithm is to improve the sorting behavior.

The details of how the sort works is dependent on the inner workings of the device adapter. It turns out that the performance of the sort of the keys is highly dependent on the data type of the keys. For example, sorting numbers stored in a 32-bit integer is often much faster than sorting groups of 2 or 3 64-bit integer. This is particularly true when the sort is capable of performing a radix-based sort.

An easy way to convert collections of indices like those returned from `vtkm::exec::CellEdgeCanonicalId` to a 32-bit integer is to use a hash function. To facilitate the creation of hash values, VTK-m comes with a simple `vtkm::Hash` function (in the vtkm/Hash.h header file). `Hash` takes a `Vec` or `Vec`-like object of integers and returns a value of type `vtkm::HashType` (an alias for a 32-bit integer). This hash function uses the FNV-1a algorithm that is designed to create hash values that are quasi-random but deterministic. This means that hash values of two different identifiers are unlikely to be the same.

That said, hash collisions can happen and become increasingly likely on larger data sets. Therefore, if we wish to use hash values, we also have to add conditions that manage collisions when they happen. Resolving hash value collisions adds overhead, but the time saved in faster sorting of hash values generally outweighs the overhead added by resolving collisions.[2] In this section we will improve on the implementation given in Section 32.2 by using hash values for keys and resolving for collisions.

As always, our first step is to count the number of edges in each input cell. These counts are used to create a `vtkm::worklet::ScatterCounting` for subsequent worklets.

Example 32.8: A simple worklet to count the number of edges on each cell.

```
1  struct CountEdgesWorklet : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3    using ControlSignature = void(CellSetIn cellSet, FieldOut numEdges);
4    using ExecutionSignature = _2(CellShape, PointCount);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC_CONT vtkm::IdComponent operator()(
9      CellShapeTag cellShape,
10     vtkm::IdComponent numPointsInCell) const
11   {
12     vtkm::IdComponent numEdges;
13     vtkm::ErrorCode status =
14       vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
15     if (status != vtkm::ErrorCode::Success)
16     {
17       // There is an error in the cell. As good as it would be to return an
18       // error, we probably don't want to invalidate the entire run if there
19       // is just one malformed cell. Instead, ignore the cell.
20       return 0;
```

[2] A comparison of the time required for completely unique keys and hash keys with collisions is studied by Lessley, et al. in "Techniques for Data-Parallel Searching for Duplicate Elements" (in *IEEE Symposium on Large Data Analysis and Visualization*, October 2017).

```
    }
    return numEdges;
  }
};
```

Our next step is to generate keys that can be used to find like elements. As before, we will use the `vtkm::exec::CellEdgeCanonicalId` function to create a unique representation for each edge. However, rather than directly use the value from `CellEdgeCanonicalId`, which is a `vtkm::Id2`, we will instead use that to generate a hash value.

Example 32.9: Worklet generating hash values.

```
class EdgeHashesWorklet : public vtkm::worklet::WorkletVisitCellsWithPoints
{
public:
  using ControlSignature = void(CellSetIn cellSet, FieldOut hashValues);
  using ExecutionSignature = _2(CellShape cellShape,
                                PointIndices globalPointIndices,
                                VisitIndex localEdgeIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag, typename PointIndexVecType>
  VTKM_EXEC vtkm::HashType operator()(
    CellShapeTag cellShape,
    const PointIndexVecType& globalPointIndicesForCell,
    vtkm::IdComponent localEdgeIndex) const
  {
    vtkm::IdComponent numPointsInCell =
      globalPointIndicesForCell.GetNumberOfComponents();
    vtkm::Id2 canonicalId;
    vtkm::ErrorCode status =
      vtkm::exec::CellEdgeCanonicalId(numPointsInCell,
                                      localEdgeIndex,
                                      cellShape,
                                      globalPointIndicesForCell,
                                      canonicalId);
    if (status != vtkm::ErrorCode::Success)
    {
      this->RaiseError(vtkm::ErrorString(status));
      return vtkm::HashType(-1);
    }
    return vtkm::Hash(canonicalId);
  }
};
```

The hash values generated by the worklet in Example 32.9 will be the same for two identical edges. However, it is no longer guaranteed that two distinct edges will have different keys, and collisions of this nature become increasingly common for larger cell sets. Thus, our next step is to resolve any such collisions.

The following example provides a worklet that goes through each group of edges associated with the same hash value (using a reduce by key worklet). It identifies which edges are actually the same as which other edges, marks a local identifier for each unique edge group, and returns the number of unique edges associated with the hash value.

Example 32.10: Worklet to resolve hash collisions occurring on edge identifiers.

```
class EdgeHashCollisionsWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
```

```
6                                 ValuesIn originCells,
7                                 ValuesIn originEdges,
8                                 ValuesOut localEdgeIndices,
9                                 ReducedValuesOut numEdges);
10  using ExecutionSignature = _6(_2 inputCells,
11                                _3 originCells,
12                                _4 originEdges,
13                                _5 localEdgeIndices);
14  using InputDomain = _1;
15
16  template<typename CellSetType,
17           typename OriginCellsType,
18           typename OriginEdgesType,
19           typename localEdgeIndicesType>
20  VTKM_EXEC vtkm::IdComponent operator()(
21    const CellSetType& cellSet,
22    const OriginCellsType& originCells,
23    const OriginEdgesType& originEdges,
24    localEdgeIndicesType& localEdgeIndices) const
25  {
26    vtkm::IdComponent numEdgesInHash = localEdgeIndices.GetNumberOfComponents();
27
28    // Sanity checks.
29    VTKM_ASSERT(originCells.GetNumberOfComponents() == numEdgesInHash);
30    VTKM_ASSERT(originEdges.GetNumberOfComponents() == numEdgesInHash);
31
32    // Clear out localEdgeIndices
33    for (vtkm::IdComponent index = 0; index < numEdgesInHash; ++index)
34    {
35      localEdgeIndices[index] = -1;
36    }
37
38    // Count how many unique edges there are and create an id for each;
39    vtkm::IdComponent numUniqueEdges = 0;
40    for (vtkm::IdComponent firstEdgeIndex = 0; firstEdgeIndex < numEdgesInHash;
41         ++firstEdgeIndex)
42    {
43      if (localEdgeIndices[firstEdgeIndex] == -1)
44      {
45        vtkm::IdComponent edgeId = numUniqueEdges;
46        localEdgeIndices[firstEdgeIndex] = edgeId;
47        // Find all matching edges.
48        vtkm::Id firstCellIndex = originCells[firstEdgeIndex];
49        vtkm::Id2 canonicalEdgeId;
50        vtkm::exec::CellEdgeCanonicalId(cellSet.GetNumberOfIndices(firstCellIndex),
51                                        originEdges[firstEdgeIndex],
52                                        cellSet.GetCellShape(firstCellIndex),
53                                        cellSet.GetIndices(firstCellIndex),
54                                        canonicalEdgeId);
55        for (vtkm::IdComponent laterEdgeIndex = firstEdgeIndex + 1;
56             laterEdgeIndex < numEdgesInHash;
57             ++laterEdgeIndex)
58        {
59          vtkm::Id laterCellIndex = originCells[laterEdgeIndex];
60          vtkm::Id2 otherCanonicalEdgeId;
61          vtkm::exec::CellEdgeCanonicalId(cellSet.GetNumberOfIndices(laterCellIndex),
62                                          originEdges[laterEdgeIndex],
63                                          cellSet.GetCellShape(laterCellIndex),
64                                          cellSet.GetIndices(laterCellIndex),
65                                          otherCanonicalEdgeId);
66          if (canonicalEdgeId == otherCanonicalEdgeId)
67          {
68            localEdgeIndices[laterEdgeIndex] = edgeId;
69          }
```

```
        }
        ++numUniqueEdges;
      }
    }

    return numUniqueEdges;
  }
};
```

With all hash collisions correctly identified, we are ready to generate the connectivity array for the line elements. This worklet uses a reduce by key worklet like the previous example, but this time we use a `ScatterCounting` to run the worklet multiple times for hash values that contain multiple unique edges. The worklet takes all the information it needs to reference back to the edges in the original mesh including a `WholeCellSetIn`, look back indices for the cells and respective edges, and the unique edge group indicators produced by Example 32.9.

As in the previous sections, this worklet writes out the edge information in a `vtkm::Id2` (which in some following code will be created with an `ArrayHandleGroupVec`).

Example 32.11: A worklet to generate indices for line cells from combined edges and potential collisions.

```
class EdgeIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originEdges,
                                ValuesIn localEdgeIndices,
                                ReducedValuesOut connectivityOut);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originEdge,
                                  _5 localEdgeIndices,
                                  VisitIndex localEdgeIndex,
                                  _6 connectivityOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginEdgesType,
           typename LocalEdgeIndicesType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginEdgesType& originEdges,
                            const LocalEdgeIndicesType& localEdgeIndices,
                            vtkm::IdComponent localEdgeIndex,
                            vtkm::Id2& connectivityOut) const
  {
    // Find the first edge that matches the index given and return it.
    for (vtkm::IdComponent edgeIndex = 0;; ++edgeIndex)
    {
      if (localEdgeIndices[edgeIndex] == localEdgeIndex)
      {
        vtkm::Id cellIndex = originCells[edgeIndex];
        vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
        vtkm::IdComponent edgeInCellIndex = originEdges[edgeIndex];
        auto cellShape = cellSet.GetCellShape(cellIndex);

        vtkm::IdComponent pointInCellIndex0;
        vtkm::exec::CellEdgeLocalIndex(
          numPointsInCell, 0, edgeInCellIndex, cellShape, pointInCellIndex0);
        vtkm::IdComponent pointInCellIndex1;
```

```
45          vtkm::exec::CellEdgeLocalIndex(
46            numPointsInCell, 1, edgeInCellIndex, cellShape, pointInCellIndex1);
47
48          auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
49          connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
50          connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
51
52          break;
53        }
54      }
55    }
56  };
```

With these 3 worklets, it is now possible to generate all the information we need to fill a `vtkm::cont::CellSetSingleType` object. A `CellSetSingleType` requires 4 items: the number of points, the constant cell shape, the constant number of points in each cell, and an array of connection indices. The first 3 items are trivial. The number of points can be taken from the input cell set as they are the same. The cell shape and number of points are predetermined to be line and 2, respectively.

The last item, the array of connection indices, is what we are creating with the worklet in Example 32.11. The connectivity array for `CellSetSingleType` is expected to be a flat array of `vtkm::Id` indices, but the worklet needs to provide groups of indices for each cell (in this case as a `Vec` object). To reconcile what the worklet provides and what the connectivity array must look like, we use the `vtkm::cont::ArrayHandleGroupVec` fancy array handle (described in Section 26.13) to make a flat array of indices look like an array of `Vec` objects.

The following example shows what the `DoExecute` method in the associated filter would look like.

Example 32.12: Invoking worklets to extract unique edges from a cell set using hash values.

```
1  inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
2    const vtkm::cont::DataSet& inData)
3  {
4    auto inCellSet = inData.GetCellSet();
5
6    // First, count the edges in each cell.
7    vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
8    this->Invoke(CountEdgesWorklet{}, inCellSet, edgeCounts);
9
10   // Second, using these counts build a scatter that repeats a cell's visit
11   // for each edge in the cell.
12   vtkm::worklet::ScatterCounting scatter(edgeCounts);
13   vtkm::worklet::ScatterCounting::OutputToInputMapType outputToInputCellMap =
14     scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
15   vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
16     scatter.GetVisitArray(inCellSet.GetNumberOfCells());
17
18   // Third, for each edge, extract a hash.
19   vtkm::cont::ArrayHandle<vtkm::HashType> hashValues;
20   this->Invoke(EdgeHashesWorklet{}, scatter, inCellSet, hashValues);
21
22   // Fourth, use a Keys object to combine all like hashes.
23   vtkm::worklet::Keys<vtkm::HashType> cellToEdgeKeys(hashValues);
24
25   // Fifth, use a reduce-by-key to collect like hash values, resolve collisions,
26   // and count the number of unique edges associated with each hash.
27   vtkm::cont::ArrayHandle<vtkm::IdComponent> numUniqueEdgesInEachHash;
28   vtkm::cont::ArrayHandle<vtkm::IdComponent> localEdgeIndices;
29   this->Invoke(EdgeHashCollisionsWorklet{},
30                cellToEdgeKeys,
31                inCellSet,
32                outputToInputCellMap,
33                outputToInputEdgeMap,
34                localEdgeIndices,
```

```
                numUniqueEdgesInEachHash);

  // Sixth, use a reduce-by-key to extract indices for each unique edge.
  vtkm::worklet::ScatterCounting hashCollisionScatter(numUniqueEdgesInEachHash);

  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  this->Invoke(EdgeIndicesWorklet{},
               hashCollisionScatter,
               cellToEdgeKeys,
               inCellSet,
               outputToInputCellMap,
               outputToInputEdgeMap,
               localEdgeIndices,
               vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  // Seventh, use the created connectivity array to build a cell set.
  vtkm::cont::CellSetSingleType<> outCellSet;
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  auto fieldMapper =
    [&](vtkm::cont::DataSet& dataset, const vtkm::cont::Field& inField)
  {
    MapCellEdgesField(dataset,
                      inField,
                      outputToInputCellMap,
                      cellToEdgeKeys,
                      localEdgeIndices,
                      hashCollisionScatter);
  };
  return this->CreateResult(inData, outCellSet, fieldMapper);
}
```

As noted in Section 32.2, in practice `DoExecute` is called on `DataSet` objects to create new `DataSet` objects. Because Example 32.12 creates a new `CellSet`, it also needs a mechanism to transform the `Field`s on the data set. To do this, we need to repurpose some of the data generated earlier in the algorithm. This includes the `outputToInputCellMap` retrieved from the `Scatter` object to replicate the cells for each edge, the `cellToEdgeKeys` `Keys` object to find like hash values, the `localEdgeIndices` array used to identify edges in colliding hashes, and the `hashCollisionScatter` `ScatterCounting` object used to separate edges from colliding hashes.

In Section 32.2 we used a convenience method to average a field attached to cells on the input to each unique edge in the output. Unfortunately, that function does not take into account the collisions that can occur on the keys. Instead we need a custom worklet to average those values that match the same unique edge.

Example 32.13: A worklet and helper function to average values with the same key, resolving for collisions.

```
class AverageCellEdgesFieldWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                ValuesIn inFieldValues,
                                ValuesIn localEdgeIndices,
                                ReducedValuesOut averagedField);
  using ExecutionSignature = void(_2 inFieldValues,
                                  _3 localEdgeIndices,
                                  VisitIndex localEdgeIndex,
                                  _4 averagedField);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename InFieldValuesType,
           typename LocalEdgeIndicesType,
```

```
18            typename OutFieldValuesType>
19  VTKM_EXEC void operator()(const InFieldValuesType& inFieldValues,
20                            const LocalEdgeIndicesType& localEdgeIndices,
21                            vtkm::IdComponent localEdgeIndex,
22                            OutFieldValuesType& averageField) const
23  {
24    using FieldType = typename InFieldValuesType::ComponentType;
25
26    vtkm::IdComponent numValues = 0;
27    for (vtkm::IdComponent reduceIndex = 0;
28         reduceIndex < inFieldValues.GetNumberOfComponents();
29         ++reduceIndex)
30    {
31      if (localEdgeIndices[reduceIndex] == localEdgeIndex)
32      {
33        FieldType fieldValue = inFieldValues[reduceIndex];
34        if (numValues == 0)
35        {
36          averageField = fieldValue;
37        }
38        else
39        {
40          averageField = averageField + fieldValue;
41        }
42        ++numValues;
43      }
44    }
45    VTKM_ASSERT(numValues > 0);
46    averageField = averageField / numValues;
47  }
48};
49
50void MapCellEdgesField(
51  vtkm::cont::DataSet& dataset,
52  const vtkm::cont::Field& inField,
53  const vtkm::worklet::ScatterCounting::OutputToInputMapType& cellPermutationMap,
54  const vtkm::worklet::Keys<vtkm::HashType>& cellToEdgeKeys,
55  const vtkm::cont::ArrayHandle<vtkm::IdComponent>& localEdgeIndices,
56  const vtkm::worklet::ScatterCounting& hashCollisionScatter)
57{
58  if (inField.IsCellField())
59  {
60    vtkm::cont::Invoker invoke;
61    vtkm::cont::UnknownArrayHandle inArray = inField.GetData();
62    vtkm::cont::UnknownArrayHandle outArray = inArray.NewInstanceBasic();
63
64    // Need to pre-allocate outArray because the way it is accessed in
65    // doMap it cannot be resized.
66    outArray.Allocate(hashCollisionScatter.GetOutputRange(
67      cellToEdgeKeys.GetUniqueKeys().GetNumberOfValues()));
68
69    auto doMap = [&](auto& concreteInput)
70    {
71      using T =
72        typename std::decay_t<decltype(concreteInput)>::ValueType::ComponentType;
73      auto concreteOutput =
74        outArray.ExtractArrayFromComponents<T>(vtkm::CopyFlag::Off);
75      invoke(
76        AverageCellEdgesFieldWorklet{},
77        hashCollisionScatter,
78        cellToEdgeKeys,
79        vtkm::cont::make_ArrayHandlePermutation(cellPermutationMap, concreteInput),
80        localEdgeIndices,
81        concreteOutput);
```

```
82      };
83      inArray.CastAndCallWithExtractedArray(doMap);
84
85      dataset.AddCellField(inField.GetName(), outArray);
86    }
87    else
88    {
89      dataset.AddField(inField); // pass through
90    }
91  }
```

With this helper function, it is straightforward to process cell fields (as demonstrated in lines 55–64 in Example 32.12.

32.4 Variable Cell Types

So far in our previous examples we have demonstrated creating a cell set where every cell is the same shape and number of points (i.e. a `CellSetSingleType`). However, it can also be the case where an algorithm must create cells of a different type (into a `vtkm::cont::CellSetExplicit`). The procedure for generating cells of different shapes is similar to that of creating a single shape. There is, however, an added step of counting the size (in number of points) of each shape to build the appropriate structure for storing the cell connectivity.

Our motivating example is a filter that extracts all the unique faces in a cell set and stores them in a cell set of polygons. This problem is similar to the one addressed in Sections 32.1, 32.2, and 32.3. In both cases it is necessary to find all subelements of each cell (in this case the faces instead of the edges). It is also the case that we expect many faces to be shared among cells in the same way edges are shared among cells. We will use the hash-based approach demonstrated in Section 32.3 except this time applied to faces instead of edges.

The main difference between the two extraction tasks is that whereas all edges are lines with two points, faces can come in different sizes. A tetrahedron has triangular faces whereas a hexahedron has quadrilateral faces. Pyramid and wedge cells have both triangular and quadrilateral faces. Thus, in general the algorithm must be capable of outputting multiple cell types.

Our algorithm for extracting unique cell faces follows the same algorithm as that in Section 32.3. We first need three worklets (used in succession) to count the number of faces in each cell, to generate a hash value for each face, and to resolve hash collisions. These are essentially the same as Examples 32.8, 32.9, and 32.10, respectively, with superficial changes made (like changing `Edge` to `Face`). To make it simpler to follow the discussion, the code is not repeated here.

When extracting edges, these worklets provide everything necessary to write out line elements. However, before we can write out polygons of different sizes, we first need to count the number of points in each polygon. The following example does just that. This worklet also writes out the identifier for the shape of the face, which we will eventually require to build a `CellSetExplicit`. Also recall that we have to work with the information returned from the collision resolution to report on the appropriate unique cell face.

Example 32.14: A worklet to count the points in the final cells of extracted faces

```
1  class CountPointsInFaceWorklet : public vtkm::worklet::WorkletReduceByKey
2  {
3  public:
4    using ControlSignature = void(KeysIn keys,
5                                  WholeCellSetIn<> inputCells,
6                                  ValuesIn originCells,
7                                  ValuesIn originFaces,
8                                  ValuesIn localFaceIndices,
9                                  ReducedValuesOut faceShape,
10                                 ReducedValuesOut numPointsInEachFace);
```

```
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originFace,
                                  _5 localFaceIndices,
                                  VisitIndex localFaceIndex,
                                  _6 faceShape,
                                  _7 numPointsInFace);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginFacesType,
           typename LocalFaceIndicesType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginFacesType& originFaces,
                            const LocalFaceIndicesType& localFaceIndices,
                            vtkm::IdComponent localFaceIndex,
                            vtkm::UInt8& faceShape,
                            vtkm::IdComponent& numPointsInFace) const
  {
    // Find the first face that matches the index given.
    for (vtkm::IdComponent faceIndex = 0;; ++faceIndex)
    {
      if (localFaceIndices[faceIndex] == localFaceIndex)
      {
        vtkm::Id cellIndex = originCells[faceIndex];
        vtkm::exec::CellFaceShape(
          originFaces[faceIndex], cellSet.GetCellShape(cellIndex), faceShape);
        vtkm::exec::CellFaceNumberOfPoints(
          originFaces[faceIndex], cellSet.GetCellShape(cellIndex), numPointsInFace);
        break;
      }
    }
  }
};
```

When extracting edges, we converted a flat array of connectivity information to an array of `Vec`s using an `ArrayHandleGroupVec`. However, `ArrayHandleGroupVec` can only create `Vec`s of a constant size. Instead, for this use case we need to use `vtkm::cont::ArrayHandleGroupVecVariable`. As described in Section 26.13, `ArrayHandleGroupVecVariable` takes a flat array of values and an index array of offsets that points to the beginning of each group to represent as a `Vec`-like. The worklet in Example 32.14 does not actually give us the array of offsets we need. Rather, it gives us the count of each group. We can get the offsets from the counts by using the `vtkm::cont::ConvertNumComponentsToOffsets` convenience function.

Example 32.15: Converting counts of connectivity groups to offsets for `ArrayHandleGroupVecVariable`.

```
  vtkm::cont::ArrayHandle<vtkm::Id> offsets;
  vtkm::Id connectivityArraySize;
  vtkm::cont::ConvertNumComponentsToOffsets(
    numPointsInEachFace, offsets, connectivityArraySize);

  vtkm::cont::CellSetExplicit<>::ConnectivityArrayType connectivityArray;
  connectivityArray.Allocate(connectivityArraySize);
  auto connectivityArrayVecs =
    vtkm::cont::make_ArrayHandleGroupVecVariable(connectivityArray, offsets);
```

Once we have created an `ArrayHandleGroupVecVariable`, we can pass that to a worklet that produces the point connections for each output polygon. The worklet is very similar to the one for creating edge lines (shown in Example 32.11), but we have to correctly handle the `Vec`-like of unknown type and size.

Example 32.16: A worklet to generate indices for polygon cells of different sizes from combined edges and potential collisions.

```
class FaceIndicesWorklet : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn originCells,
                                ValuesIn originFaces,
                                ValuesIn localFaceIndices,
                                ReducedValuesOut connectivityOut);
  using ExecutionSignature = void(_2 inputCells,
                                  _3 originCell,
                                  _4 originFace,
                                  _5 localFaceIndices,
                                  VisitIndex localFaceIndex,
                                  _6 connectivityOut);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginFacesType,
           typename LocalFaceIndicesType,
           typename ConnectivityVecType>
  VTKM_EXEC void operator()(const CellSetType& cellSet,
                            const OriginCellsType& originCells,
                            const OriginFacesType& originFaces,
                            const LocalFaceIndicesType& localFaceIndices,
                            vtkm::IdComponent localFaceIndex,
                            ConnectivityVecType& connectivityOut) const
  {
    // Find the first face that matches the index given and return it.
    for (vtkm::IdComponent faceIndex = 0;; ++faceIndex)
    {
      if (localFaceIndices[faceIndex] == localFaceIndex)
      {
        vtkm::Id cellIndex = originCells[faceIndex];
        vtkm::IdComponent faceInCellIndex = originFaces[faceIndex];
        auto cellShape = cellSet.GetCellShape(cellIndex);
        vtkm::IdComponent numPointsInFace = connectivityOut.GetNumberOfComponents();

        auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
        for (vtkm::IdComponent localPointI = 0; localPointI < numPointsInFace;
             ++localPointI)
        {
          vtkm::IdComponent pointInCellIndex;
          vtkm::exec::CellFaceLocalIndex(
            localPointI, faceInCellIndex, cellShape, pointInCellIndex);
          connectivityOut[localPointI] = globalPointIndicesForCell[pointInCellIndex];
        }

        break;
      }
    }
  }
};
```

With these worklets in place, we can implement a filter `DoExecute` as follows.

Example 32.17: Invoking worklets to extract unique faces from a cell set.

```
inline VTKM_CONT vtkm::cont::DataSet ExtractFaces::DoExecute(
```

```
  const vtkm::cont::DataSet& inData)
{

  auto inCellSet = inData.GetCellSet();

  // First, count the faces in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> faceCounts;
  this->Invoke(CountFacesWorklet{}, inCellSet, faceCounts);

  // Second, using these counts build a scatter that repeats a cell's visit
  // for each edge in the cell.
  vtkm::worklet::ScatterCounting scatter(faceCounts);
  vtkm::worklet::ScatterCounting::OutputToInputMapType outputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
  vtkm::worklet::ScatterCounting::VisitArrayType outputToInputFaceMap =
    scatter.GetVisitArray(inCellSet.GetNumberOfCells());

  // Third, for each face, extract a hash.
  vtkm::cont::ArrayHandle<vtkm::HashType> hashValues;
  this->Invoke(FaceHashesWorklet{}, scatter, inCellSet, hashValues);

  // Fourth, use a Keys object to combine all like hashes.
  vtkm::worklet::Keys<vtkm::HashType> cellToFaceKeys(hashValues);

  // Fifth, use a reduce-by-key to collect like hash values, resolve collisions,
  // and count the number of unique faces associated with each hash.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> localFaceIndices;
  vtkm::cont::ArrayHandle<vtkm::IdComponent> numUniqueFacesInEachHash;
  this->Invoke(FaceHashCollisionsWorklet{},
               cellToFaceKeys,
               inCellSet,
               outputToInputCellMap,
               outputToInputFaceMap,
               localFaceIndices,
               numUniqueFacesInEachHash);

  // Sixth, use a reduce-by-key to count the number of points in each unique face.
  // Also identify the shape of each face.
  vtkm::worklet::ScatterCounting hashCollisionScatter(numUniqueFacesInEachHash);

  vtkm::cont::CellSetExplicit<>::ShapesArrayType shapeArray;
  vtkm::cont::ArrayHandle<vtkm::IdComponent> numPointsInEachFace;

  this->Invoke(CountPointsInFaceWorklet{},
               hashCollisionScatter,
               cellToFaceKeys,
               inCellSet,
               outputToInputCellMap,
               outputToInputFaceMap,
               localFaceIndices,
               shapeArray,
               numPointsInEachFace);

  // Seventh, convert the numPointsInEachFace array to an offsets array and use that
  // to create an ArrayHandleGroupVecVariable.
  vtkm::cont::ArrayHandle<vtkm::Id> offsets;
  vtkm::Id connectivityArraySize;
  vtkm::cont::ConvertNumComponentsToOffsets(
    numPointsInEachFace, offsets, connectivityArraySize);

  vtkm::cont::CellSetExplicit<>::ConnectivityArrayType connectivityArray;
  connectivityArray.Allocate(connectivityArraySize);
  auto connectivityArrayVecs =
    vtkm::cont::make_ArrayHandleGroupVecVariable(connectivityArray, offsets);
```

```
66
67    // Eigth, use a reduce-by-key to extract indices for each unique face.
68    this->Invoke(FaceIndicesWorklet{},
69                 hashCollisionScatter,
70                 cellToFaceKeys,
71                 inCellSet,
72                 outputToInputCellMap,
73                 outputToInputFaceMap,
74                 localFaceIndices,
75                 connectivityArrayVecs);
76
77    // Ninth, use the created connectivity array and others to build a cell set.
78    vtkm::cont::CellSetExplicit<> outCellSet;
79    outCellSet.Fill(
80      inCellSet.GetNumberOfPoints(), shapeArray, connectivityArray, offsets);
81
82    auto fieldMapper =
83      [&](vtkm::cont::DataSet& dataset, const vtkm::cont::Field& inField)
84    {
85      MapCellEdgesField(dataset,
86                        inField,
87                        outputToInputCellMap,
88                        cellToFaceKeys,
89                        localFaceIndices,
90                        hashCollisionScatter);
91    };
92    return this->CreateResult(inData, outCellSet, fieldMapper);
93  }
```

As noted previously, in practice `DoExecute` is called on `DataSet` objects to create new `DataSet` objects. The process for doing so is no different from our previous algorithm as described at the end of Section 32.3 (Example 32.13).

CHAPTER

THIRTYTHREE

UNKNOWN ARRAY HANDLES

The ArrayHandle class uses templating to make very efficient and type-safe access to data. However, it is sometimes inconvenient or impossible to specify the element type and storage at run-time. The UnknownArrayHandle class provides a mechanism to manage arrays of data with unspecified types.

vtkm::cont::UnknownArrayHandle holds a reference to an array. Unlike ArrayHandle, UnknownArrayHandle is *not* templated. Instead, it uses C++ run-type type information to store the array without type and cast it when appropriate.

An UnknownArrayHandle can be established by constructing it with or assigning it to an ArrayHandle. The following example demonstrates how an UnknownArrayHandle might be used to load an array whose type is not known until run-time.

Example 33.1: Creating an UnknownArrayHandle.

```
1  VTKM_CONT
2  vtkm::cont::UnknownArrayHandle LoadUnknownArray(const void* buffer,
3                                                  vtkm::Id length,
4                                                  std::string type)
5  {
6    vtkm::cont::UnknownArrayHandle handle;
7    if (type == "float")
8    {
9      vtkm::cont::ArrayHandle<vtkm::Float32> concreteArray =
10       vtkm::cont::make_ArrayHandle(
11         reinterpret_cast<const vtkm::Float32*>(buffer), length, vtkm::CopyFlag::On);
12     handle = concreteArray;
13   }
14   else if (type == "int")
15   {
16     vtkm::cont::ArrayHandle<vtkm::Int32> concreteArray =
17       vtkm::cont::make_ArrayHandle(
18         reinterpret_cast<const vtkm::Int32*>(buffer), length, vtkm::CopyFlag::On);
19     handle = concreteArray;
20   }
21   return handle;
22 }
```

33.1 Allocation

Data pointed to by an UnknownArrayHandle is not directly accessible. However, it is still possible to do some type-agnostic manipulation of the array allocations.

First, it is always possible to call UnknownArrayHandle::GetNumberOfValues to retrieve the current size of

the array. It is also possible to call `UnknownArrayHandle::Allocate` to change the size of an unknown array. `UnknownArrayHandle`'s `Allocate` works exactly the same as the `Allocate` in the basic `ArrayHandle`.

Example 33.2: Checking the size of an `ArrayHandle` and resizing it.

```
vtkm::cont::UnknownArrayHandle unknownHandle = // ... some valid array

// Double the size of the array while preserving all the initial values.
vtkm::Id originalArraySize = unknownHandle.GetNumberOfValues();
unknownHandle.Allocate(originalArraySize * 2, vtkm::CopyFlag::On);
```

It is often the case where you have an `UnknownArrayHandle` as the input to an operation and you want to generate an output of the same type. To handle this case, use the `NewInstance` method to create a new array of the same type (without having to determine the type).

Example 33.3: Creating a new instance of an unknown array handle.

```
vtkm::cont::UnknownArrayHandle unknownHandle = // ... some valid array

// Double the size of the array while preserving all the initial values.
vtkm::Id originalArraySize = unknownHandle.GetNumberOfValues();
unknownHandle.Allocate(originalArraySize * 2, vtkm::CopyFlag::On);

// Create a new array of the same type as the original.
vtkm::cont::UnknownArrayHandle newArray = unknownHandle.NewInstance();

newArray.Allocate(originalArraySize);
```

That said, there are many fancy array types (described in Chapter 26) that cannot be used as outputs. Thus, if you do not know the storage of the array, the similar array returned by `NewInstance` could be infeasible for use as an output. Thus, `UnknownArrayHandle` also contains the `NewInstanceBasic` method to create a new array with the same value type but using the basic array storage, which can always be resized and written to.

Example 33.4: Creating a new basic instance of an unknown array handle.

```
vtkm::cont::UnknownArrayHandle indexArray = vtkm::cont::ArrayHandleIndex();
// Returns an array of type ArrayHandleBasic<vtkm::Id>
vtkm::cont::UnknownArrayHandle basicArray = indexArray.NewInstanceBasic();
```

It is occasionally the case that you need a new array of a similar type, but that type has to hold floating point values. For example, if you had an operation that computed a discrete cosine transform on an array, the result would be very inaccurate if stored as integers. In this case, you would actually want to store the result in an array of floating point values. For this case, you can use the `NewInstanceFloatBasic` to create a new basic `ArrayHandle` with the component type changed to `vtkm::FloatDefault`. For example, if the `UnknownArrayHandle` stores an `ArrayHandle` of type `vtkm::Id`, `NewInstanceFloatBasic` will create an `ArrayHandle` of type `vtkm::FloatDefault`. If the `UnknownArrayHandle` stores an `ArrayHandle` of type `vtkm::Id3`, `NewInstanceFloatBasic` will create an `ArrayHandle` of type `vtkm::Vec3f`.

Example 33.5: Creating a new array instance with floating point values.

```
vtkm::cont::UnknownArrayHandle intArray = vtkm::cont::ArrayHandleIndex();
// Returns an array of type ArrayHandleBasic<vtkm::FloatDefault>
vtkm::cont::UnknownArrayHandle floatArray = intArray.NewInstanceFloatBasic();

vtkm::cont::UnknownArrayHandle id3Array = vtkm::cont::ArrayHandle<vtkm::Id3>();
// Returns an array of type ArrayHandleBasic<vtkm::Vec3f>
vtkm::cont::UnknownArrayHandle float3Array = id3Array.NewInstanceFloatBasic();
```

33.2 Casting to Known Types

Data pointed to by an UnknownArrayHandle is not directly accessible. To access the data, you need to retrieve the data as an ArrayHandle. If you happen to know (or can guess) the type, you can use the AsArrayHandle method to retrieve the array as a specific type.

Example 33.6: Retrieving an array of a known type from UnknownArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32> knownArray =
  unknownArray.AsArrayHandle<vtkm::cont::ArrayHandle<vtkm::Float32>>();
```

AsArrayHandle actually has two forms. The first form, shown in the previous example, has no arguments and returns the ArrayHandle. This form requires you to specify the type of array as a template parameter. The alternate form has you pass a reference to a concrete ArrayHandle as an argument as shown in the following example. This form can imply the template parameter from the argument.

Example 33.7: Alternate form for retrieving an array of a known type from UnknownArrayHandle.

```
unknownArray.AsArrayHandle(knownArray);
```

AsArrayHandle treats ArrayHandleCast and ArrayHandleMultiplexer special. If the special ArrayHandle can hold the actual array stored, then AsArrayHandle will return successfully. In the following example, AsArrayHandle returns an array of type vtkm::Float32 as an ArrayHandleCast that converts the values to vtkm::Float64.

Example 33.8: Getting a cast array handle from an ArrayHandleCast.

```
vtkm::cont::ArrayHandle<vtkm::Float32> originalArray;
vtkm::cont::UnknownArrayHandle unknownArray = originalArray;

vtkm::cont::ArrayHandleCast<vtkm::Float64, decltype(originalArray)> castArray;
unknownArray.AsArrayHandle(castArray);
```

Did you know?

The inverse retrieval works as well. If you create an UnknownArrayHandle with an ArrayHandleCast or ArrayHandleMultiplexer, you can get the underlying array with AsArrayHandle. These relationships also work recursively (e.g. an array placed in a cast array which is placed in a multiplexer).

If the UnknownArrayHandle cannot store its array in the type given to AsArrayHandle, it will throw an exception. Thus, you should not use AsArrayHandle with types that you are not sure about. Use the CanConvert method to determine if a given ArrayHandle type will work with AsArrayHandle.

Example 33.9: Querying whether a given ArrayHandle can be retrieved from an UnknownArrayHandle.

```
VTKM_CONT vtkm::FloatDefault GetMiddleValue(
  const vtkm::cont::UnknownArrayHandle& unknownArray)
{
  if (unknownArray.CanConvert<vtkm::cont::ArrayHandleConstant<vtkm::FloatDefault>>())
  {
    // Fast path for known array
    vtkm::cont::ArrayHandleConstant<vtkm::FloatDefault> constantArray;
    unknownArray.AsArrayHandle(constantArray);
    return constantArray.GetValue();
  }
  else
  {
    // General path
```

```
14      auto ranges = vtkm::cont::ArrayRangeCompute(unknownArray);
15      vtkm::Range range = ranges.ReadPortal().Get(0);
16      return static_cast<vtkm::FloatDefault>((range.Min + range.Max) / 2);
17    }
18  }
```

By design, `CanConvert` will return true for types that are not actually stored in the `UnknownArrayHandle` but can be retrieved. If you need to know specifically what type is stored in the `UnknownArrayHandle`, you can use the `IsType` method instead. If you need to query either the value type or the storage, you can use `IsValueType` and `IsStorageType`, respectively. `UnknownArrayHandle` also provides `GetValueTypeName` and `GetStorageTypeName` for debugging purposes.

Common Errors

`CanConvert` *is almost always safer to use than* `IsType` *or its similar methods. Even though* `IsType` *reflects the actual array type,* `CanConvert` *better describes how* `UnknownArrayHandle` *will behave.*

If you do not know the exact type of the array contained in an `UnknownArrayHandle`, a brute force method to get the data out is to copy it to an array of a known type. This can be done with the `UnknownArrayHandle::DeepCopyFrom` method, which will copy the contents of a target array into an existing array of a (potentially) different type.

Example 33.10: Deep copy arrays of unknown types.

```
1  VTKM_CONT vtkm::cont::ArrayHandle<vtkm::FloatDefault> CopyToDefaultArray(
2    const vtkm::cont::UnknownArrayHandle& unknownArray)
3  {
4    // Initialize the output UnknownArrayHandle with the array type we want to copy to.
5    vtkm::cont::UnknownArrayHandle output =
6      vtkm::cont::ArrayHandle<vtkm::FloatDefault>{};
7    output.DeepCopyFrom(unknownArray);
8    return output.AsArrayHandle<vtkm::cont::ArrayHandle<vtkm::FloatDefault>>();
9  }
```

It is often the case that you have good reason to believe that an array is of an expected type, but you have no way to be sure. To simplify code, the most rational thing to do is to get the array as the expected type if that is indeed what it is, or to copy it to an array of that type otherwise. The `UnknownArrayHandle::CopyShallowIfPossible` does just that.

Example 33.11: Using `ArrayCopyShallowIfPossible` to get an unknown array as a particular type.

```
1  VTKM_CONT vtkm::cont::ArrayHandle<vtkm::FloatDefault> GetAsDefaultArray(
2    const vtkm::cont::UnknownArrayHandle& unknownArray)
3  {
4    // Initialize the output UnknownArrayHandle with the array type we want to copy to.
5    vtkm::cont::UnknownArrayHandle output =
6      vtkm::cont::ArrayHandle<vtkm::FloatDefault>{};
7    output.CopyShallowIfPossible(unknownArray);
8    return output.AsArrayHandle<vtkm::cont::ArrayHandle<vtkm::FloatDefault>>();
9  }
```

Did you know?

The `UnknownArrayHandle` *copy methods behave similarly to the* `vtkm::cont::ArrayCopy` *functions. One advantage of using the* `UnknownArrayHandle` *methods is that they do not require using a device compiler (such as nvcc). Both versions will (potentially) perform the copy on a device, but the methods for* `UnknownArrayHandle` *are sufficiently hidden in a library to avoid calling code needing to compile device instructions.*

33.3 Casting to a List of Potential Types

Using `AsArrayHandle` is fine as long as the correct types are known, but often times they are not. For this use case `UnknownArrayHandle` has a method named `CastAndCallForTypes` that attempts to cast the array to some set of types.

The `CastAndCallForTypes` method accepts a functor to run on the appropriately cast array. The functor must have an overloaded const parentheses operator that accepts an `ArrayHandle` of the appropriate type. You also have to specify two template parameters that specify a `vtkm::List` of value types to try and a `vtkm::List` of storage types to try, respectively. The macros `VTKM_DEFAULT_TYPE_LIST` and `VTKM_DEFAULT_STORAGE_LIST` are often used when nothing more specific is known.

Example 33.12: Operating on an `UnknownArrayHandle` with `CastAndCallForTypes`.

```
struct PrintArrayContentsFunctor
{
  template<typename T, typename S>
  VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<T, S>& array) const
  {
    this->PrintArrayPortal(array.ReadPortal());
  }

private:
  template<typename PortalType>
  VTKM_CONT void PrintArrayPortal(const PortalType& portal) const
  {
    for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
    {
      // All ArrayPortal objects have ValueType for the type of each value.
      using ValueType = typename PortalType::ValueType;
      using VTraits = vtkm::VecTraits<ValueType>;

      ValueType value = portal.Get(index);

      vtkm::IdComponent numComponents = VTraits::GetNumberOfComponents(value);
      for (vtkm::IdComponent componentIndex = 0; componentIndex < numComponents;
           componentIndex++)
      {
        std::cout << " " << VTraits::GetComponent(value, componentIndex);
      }
      std::cout << std::endl;
    }
  }
};

void PrintArrayContents(const vtkm::cont::UnknownArrayHandle& array)
{
  array.CastAndCallForTypes<VTKM_DEFAULT_TYPE_LIST, VTKM_DEFAULT_STORAGE_LIST>(
    PrintArrayContentsFunctor{});
```

```
36 }
```

Did you know?

The first (required) argument to `CastAndCallForTypes` *is the functor to call with the array. You can supply any number of optional arguments after that. Those arguments will be passed directly to the functor. This makes it easy to pass state to the functor.*

Did you know?

When an `UnknownArrayHandle` *is used in place of an* `ArrayHandle` *as an argument to a worklet invocation, it will internally use* `CastAndCallForTypes` *to attempt to call the worklet with an* `ArrayHandle` *of the correct type.*

`UnknownArrayHandle` has a simple subclass named `vtkm::cont::UncertainArrayHandle` for use when you can narrow the array to a finite set of types. `UncertainArrayHandle` has two template parameters that must be specified: a `vtkm::List` of value types and a `vtkm::List` of storage types. `UncertainArrayHandle` has a method named `CastAndCall` that behaves the same as `CastAndCallForTypes` except that you do not have to specify the types to try. Instead, the types are taken from the template parameters of the `UncertainArrayHandle` itself.

Example 33.13: Using `UncertainArrayHandle` to cast and call a functor.

```
vtkm::cont::UncertainArrayHandle<vtkm::TypeListScalarAll,
                                 vtkm::cont::StorageListBasic>
  uncertainArray(unknownArray);
uncertainArray.CastAndCall(PrintArrayContentsFunctor{});
```

Did you know?

Like with `UnknownArrayHandle`*, if an* `UncertainArrayHandle` *is used in a worklet invocation, it will internally use* `CastAndCall`*. This provides a convenient way to specify what array types the invoker should try.*

Both `UnknownArrayHandle` and `UncertainArrayHandle` provide a method named `ResetTypes` to redefine the types to try. `ResetTypes` has two template parameters that are the `vtkm::List`s of value and storage types. `ResetTypes` returns a new `UncertainArrayHandle` with the given types. This is a convenient way to pass these types to functions.

Example 33.14: Resetting the types of an `UnknownArrayHandle`.

```
vtkm::cont::Invoker invoke;
invoke(
  MyWorklet{},
  unknownArray.ResetTypes<vtkm::TypeListScalarAll, vtkm::cont::StorageListBasic>(),
  outArray);
```

Common Errors

Because it returns an `UncertainArrayHandle`*, you need to include* `vtkm/cont/UncertainArrayHandle.h` *if you use* `UnknownArrayHandle::ResetTypes`*. This is true even if you do not directly use the returned object.*

33.4 Accessing Truly Unknown Arrays

So far in Sections 33.2 and 33.3 we explored how to access the data in an `UnknownArrayHandle` when you actually know the array type or can narrow down the array type to some finite number of candidates. But what happens if you cannot practically narrow down the types in the `UnknownArrayHandle`? For this case, `UnknownArrayHandle` provides mechanisms for extracting data knowing little or nothing about the types.

33.4.1 Cast with Floating Point Fallback

The problem with `UnknownArrayHandle::CastAndCallForTypes` and `UncertainArrayHandle::CastAndCall` is that you can only list a finite amount of value types and storage types to try. If you encounter an `UnknownArrayHandle` containing a different `ArrayHandle` type, the cast and call will simply fail. Since the compiler must create a code path for each possible `ArrayHandle` type, it may not even be feasible to list all known types.

`UnknownArrayHandle::CastAndCallForTypesWithFloatFallback` works around this problem by providing a fallback in case the contained `ArrayHandle` does not match any of the types tried. If none of the types match, then `CastAndCallForTypesWithFloatFallback` will copy the data to an `ArrayHandle` with `vtkm::FloatDefault` values (or some compatible `vtkm::Vec` with `vtkm::FloatDefault` components) and basic storage. It will then attempt to match again with this copied array.

Example 33.15: Cast and call a functor from an `UnknownArrayHandle` with a float fallback.

```
1 unknownArray.CastAndCallForTypesWithFloatFallback<vtkm::TypeListField,
2                                                   VTKM_DEFAULT_STORAGE_LIST>(
3   PrintArrayContentsFunctor{});
```

In this case, we do not have to list every possible type because the array will be copied to a known type if nothing matches. Note that when using `CastAndCallForTypesWithFloatFallback`, you still need to include an appropriate type based on `vtkm::FloatDefault` in the value type list and `vtkm::cont::StorageTagBasic` in the storage list so that the copied array can match.

`UncertainArrayHandle` has a matching method named `CastAndCallWithFloatFallback` that does the same operation using the types specified in the `UncertainArrayHandle`.

Example 33.16: Cast and call a functor from an `UncertainArrayHandle` with a float fallback.

```
1 uncertainArray.CastAndCall(PrintArrayContentsFunctor{});
```

33.4.2 Extracting Components

Using a floating point fallback allows you to use arrays of unknown types in most circumstances, but it does have a few drawbacks. First, and most obvious, is that you may not operate on the data in its native format. If you want to preserve the integer format of data, this may not be the method. Second, the fallback requires a copy of

the data. If `CastAndCallForTypesWithFloatFallback` does not match the type of the array, it copies the array to a new type that (hopefully) can be matched. Third, `CastAndCallForTypesWithFloatFallback` still needs to match the number of components in each array value. If the contained `ArrayHandle` contains values that are `Vec`s of length 2, then the data will be copied to an array of `Vec2f`s. If `Vec2f` is not included in the types to try, the cast and call will still fail.

A way to get around these problems is to extract a single component from the array. You can use the `UnknownArrayHandle::ExtractComponent` method to return an `ArrayHandle` with the values for a given component for each value in the array. `ExtractComponent` must be given a template argument for the base component type. The following example extracts the first component of all `vtkm::Vec` values in an `UnknownArrayHandle` assuming that the component is of type `vtkm::FloatDefault` (line 11).

Example 33.17: Extracting the first component of every value in an `UnknownArrayHandle`.

```
1  vtkm::cont::ArrayHandleBasic<vtkm::Vec3f> concreteArray =
2    vtkm::cont::make_ArrayHandle<vtkm::Vec3f>({ { 0, 1, 2 },
3                                                { 3, 4, 5 },
4                                                { 6, 7, 8 },
5                                                { 9, 10, 11 },
6                                                { 12, 13, 14 },
7                                                { 15, 16, 17 } });
8
9  vtkm::cont::UnknownArrayHandle unknownArray(concreteArray);
10
11 auto componentArray = unknownArray.ExtractComponent<vtkm::FloatDefault>(0);
12 // componentArray contains [ 0, 3, 6, 9, 12, 15 ].
```

The code in Example 33.17 works with any array with values based on the default floating point type. If the `UnknownArrayHandle` has an array containing `vtkm::FloatDefault`, then the returned array has all the same values. If the `UnknownArrayHandle` contains values of type `vtkm::Vec3f`, then each value in the returned array will be the first component of this array.

If the `UnknownArrayHandle` really contains an array with incompatible value types (such as `ArrayHandle<vtkm::Id>`), then an `vtkm::cont::ErrorBadType` will be thrown. To check if the `UnknownArrayHandle` contains an array of a compatible type, use the `IsBaseComponentType` method.

Example 33.18: Checking the base component type in an `UnknownArrayHandle`.

```
1  unknownArray.IsBaseComponentType<vtkm::FloatDefault>()
```

This section started with the motivation of getting data from an `UnknownArrayHandle` without knowing anything about the type, yet `ExtractComponent` still requires a type parameter. However, by limiting the type needed to the base component type, you only need to check the base C types (standard integers and floating points) available in C++. You do not need to know whether these components are arranged in `Vec`s or the size of the `vtkm::Vec`. A general implementation of an algorithm might have to deal with scalars as well as `Vec`s of size 2, 3, and 4. If we consider operations on tensors, `Vec`s of size 6 and 9 can be common as well. But when using `ExtractComponent`, a single condition can handle any potential `Vec` size.

Another advantage of `ExtractComponent` is that the type of storage does not need to be specified. `ExtractComponent` works with any type of `ArrayHandle` storage (with some caveats). So, Example 33.17 works equally as well with `ArrayHandleBasic`, `ArrayHandleSOA`, `ArrayHandleUniformPointCoordinates`, `ArrayHandleCartesianProduct`, and many others. Trying to capture all reasonable types of arrays could easily require hundreds of conditions, all of which and more can be captured with `ExtractComponent` and the roughly 12 basic C data types. In practice, you often only really have to worry about floating point components, which further reduces the cases down to (usually) 2.

`UnknownArrayHandle::ExtractComponent` works by returning an `ArrayHandleStride`. This is a special `ArrayHandle` that can access data buffers by skipping values at regular intervals. This allows it to access data packed in

different ways such as ArrayHandleBasic, ArrayHandleSOA, and many others. That said, ArrayHandleStride is not magic, so if cannot directly access memory, some or all of it may be copied. If you are attempting to use the array from ExtractComponent as an output array, pass vtkm::CopyFlag::Off as a second argument. This will ensure that data are not copied so that any data written will go to the original array (or throw an exception if this cannot be done).

Common Errors

Although UnknownArrayHandle::ExtractComponent *will technically work with any* ArrayHandle *(of simple* Vec *types), it may require a very inefficient memory copy. Pay attention if* ExtractComponent *issues a warning about an inefficient memory copy. This is likely a serious performance issue, and the data should be retrieved in a different way (or better yet stored in a different way).*

Example 33.17 access the first component of each Vec in an array. But in practice you usually want to operate on all components stored in the array. A simple solution is to iterate over each component.

Example 33.19: Extracting each component from an UnknownArrayHandle.

```
1  std::vector<vtkm::cont::ArrayHandle<vtkm::FloatDefault>> outputArrays(
2    static_cast<std::size_t>(unknownArray.GetNumberOfComponentsFlat()));
3  for (vtkm::IdComponent componentIndex = 0;
4       componentIndex < unknownArray.GetNumberOfComponentsFlat();
5       ++componentIndex)
6  {
7    invoke(MyWorklet{},
8           unknownArray.ExtractComponent<vtkm::FloatDefault>(componentIndex),
9           outputArrays[static_cast<std::size_t>(componentIndex)]);
10 }
```

To ensure that the type of the extracted component is a basic C type, the vtkm::Vec values are "flattened." That is, they are treated as if they are a single level vtkm::Vec. For example, if you have a value type of vtkm::Vec<vtkm::Id3, 2>, ExtractComponent treats this type as vtkm::Vec<vtkm::Id, 6>. This allows you to extract the components as type vtkm::Id rather than having a special case for vtkm::Id3.

Although iterating over components works fine, it can be inconvenient. An alternate mechanism is to use UnknownArrayHandle::ExtractArrayFromComponents to get all the components at once. ExtractArrayFromComponents works like ExtractComponent except that instead of returning an ArrayHandleStride, it returns a special vtkm::cont::ArrayHandleRecombineVec that behaves like an ArrayHandle to reference all component arrays at once.

Example 33.20: Extracting all components from an UnknownArrayHandle at once.

```
1  invoke(MyWorklet{},
2         unknownArray.ExtractArrayFromComponents<vtkm::FloatDefault>(),
3         outArray);
```

Common Errors

Although it has the same interface as other ArrayHandles, ArrayHandleRecombineVec *has a special value type that breaks some conventions. For example, when used in a worklet, the value type passed from this*

array to the worklet cannot be replicated. That is, you cannot create a temporary stack value of the same type.

Because you still need to specify a base component type, you will likely still need to check several types to safely extract data from an `UnknownArrayHandle` by component. To do this automatically, you can use the `CastAndCallWithExtractedArray`. This method behaves similarly to `CastAndCall` except that it internally uses `ExtractArrayFromComponents`.

Example 33.21: Calling a functor for nearly any type of array stored in an `UnknownArrayHandle`.

```
unknownArray.CastAndCallWithExtractedArray(PrintArrayContentsFunctor{});
```

33.5 Mutability

One subtle feature of `UnknownArrayHandle` is that the class is, in principle, a pointer to an array pointer. This means that the data in an `UnknownArrayHandle` is always mutable even if the class is declared `const`. The upshot is that you can pass output arrays as constant `UnknownArrayHandle` references.

Example 33.22: Using a `const UnknownArrayHandle` for a function output.

```
void IndexInitialize(vtkm::Id size, const vtkm::cont::UnknownArrayHandle& output)
{
  vtkm::cont::ArrayHandleIndex input(size);
  output.DeepCopyFrom(input);
}
```

Although it seems strange, there is a good reason to allow output `UnknownArrayHandle`s to be `const`. It allows a typed `ArrayHandle` to be used as the argument to the function. In this case, the compiler will automatically convert the `ArrayHandle` to a `UnknownArrayHandle`. When C++ creates objects like this, they can only be passed as constant references or by value. So, declaring the output parameter as `const UnknownArrayHandle` allows it to be used for code like this.

Example 33.23: Passing an `ArrayHandle` as an output `UnknownArrayHandle`.

```
template<typename T>
void Foo(const vtkm::cont::ArrayHandle<T>& input, vtkm::cont::ArrayHandle<T>& output)
{
  IndexInitialize(input.GetNumberOfValues(), output);
  // ...
```

Of course, you could also declare the output by value instead of by reference, but this has the same semantics with extra internal pointer management.

Did you know?

When possible, it is better to pass `UnknownArrayHandle`s as constant references (or by value) rather than a mutable reference, even if the array contents are going to be modified. This allows the function to support automatic conversion of output `ArrayHandle`s.

So if a constant `UnknownArrayHandle` can have its contents modified, what is the difference between a constant reference and a non-constant reference? The difference is that the constant reference can change the array's content, but not the array itself. If you want to do operations like doing a shallow copy or changing the underlying type of the array, a non-constant reference is needed.

CHAPTER

THIRTYFOUR

UNKNOWN CELL SETS

vtkm::cont::DataSet must hold a vtkm::cont::CellSet object, but it cannot know its specific type at compile time. To manage storing CellSets without knowing their types, DataSet actually holds a reference using vtkm::cont::UnknownCellSet. UnknownCellSet is a simple polymorphic container that stores a reference to a vtkm::cont::CellSet of unknown type.

It is possible to create an empty UnknownCellSet. You can use the `IsValid` function to query whether an UnknownCellSet holds a valid CellSet. Performing operations on an invalid UnknownCellSet can lead to unexpected behavior.

34.1 Generic Operations

Some cell set operations in VTK-m require a specific, concrete class of CellSet. But UnknownCellSet provides several functions that allow you to operate on a cell set without knowing the exact type.

`IsValid` Returns true if the UnknownCellSet holds a legitimate CellSet. Other operations on the UnknownCellSet may have undefined behavior if it is not valid.

`GetCellSetBase` All cell set classes inherit from the vtkm::cont::CellSet base class. This method returns a pointer to the contained cell set object as this base type.

`NewInstance` Creates a new cell set object of the same type as that stored in the UnknownCellSet and returns the new instance in another UnknownCellSet.

`GetCellSetName` Return a `std::string` containing the specific class name of the contained cell set.

`GetNumberOfCells` Returns the number of cells in the cell set.

`GetNumberOfPoints` Returns the number of points in the cell set.

`GetCellShape` Given the index of a cell, returns the identifier for the cell shape.

`GetNumberOfPointsInCell` Given the index of a cell, returns the number of points incident on that cell.

`GetCellPointIds` Given the index of a cell and an array of vtkm::Id s, returns the indices for that the cell is incident to. The number of indices put in the array is equal to the value returned from `GetNumberOfPointsInCell`, and the array should be at least that long.

`DeepCopyFrom` Will copy the connectivity arrays from the provided UnknownCellSet to this one.

`PrintSummary` Prints to the provided `std::ostream` (such as `std::cout`) a summary of the contents of the cell set.

`ReleaseResourcesExecution` Removes any data stored on any device associated with the cell set. The data for the cell set will still be available, but may need to be loaded back on a device before an operation. This method has no effect if called on an invalid `UnknownCellSet`.

34.2 Casting to Known Types

There are many operations in VTK-m that need to know the specific type of cell set. To perform one of these types of operation, you need to retrieve the data as a `CellSet` concrete subclass. If you happen to know (or can guess) the type, you can use the `AsCellSet` method to retrieve the cell set as a specific type. You can pass in a reference to a cell set object of the desired type to `AsCellSet`. You can also call `AsCellSet` with no arguments and the cast cell set will be returned, but in this case you must specify the desired type with a template argument.

Example 34.1: Retrieving a cell set of a known type from `UnknownCellSet`.

```
1  vtkm::cont::CellSetExplicit<> cellSet;
2  unknownCells.AsCellSet(cellSet);
3
4  // This is an equivalent way to get the cell set.
5  auto cellSet2 = unknownCells.AsCellSet<vtkm::cont::CellSetExplicit<>>();
```

If the `UnknownCellSet` cannot store its cell set in the type given to `AsCellSet`, it will throw an exception. Thus, you should not use `AsCellSet` with types that you are not sure about. Use the `CanConvert` method to determine if a given `CellSet` type will work with `AsCellSet`.

Example 34.2: Querying whether a given `CellSet` can be retrieved from an `UnknownCellSet`.

```
1  VTKM_CONT vtkm::Id3 Get3DPointDimensions(
2    const vtkm::cont::UnknownCellSet& unknownCellSet)
3  {
4    if (unknownCellSet.CanConvert<vtkm::cont::CellSetStructured<3>>())
5    {
6      vtkm::cont::CellSetStructured<3> cellSet;
7      unknownCellSet.AsCellSet(cellSet);
8      return cellSet.GetPointDimensions();
9    }
10   else if (unknownCellSet.CanConvert<vtkm::cont::CellSetStructured<2>>())
11   {
12     vtkm::cont::CellSetStructured<2> cellSet;
13     unknownCellSet.AsCellSet(cellSet);
14     vtkm::Id2 dims = cellSet.GetPointDimensions();
15     return vtkm::Id3{ dims[0], dims[1], 1 };
16   }
17   else
18   {
19     return vtkm::Id3{ unknownCellSet.GetNumberOfPoints(), 1, 1 };
20   }
21 }
```

By design, `CanConvert` will return true for types that are not actually stored in the `UnknownCellSet` but can be retrieved. If you need to know specifically what type is stored in the `UnknownCellSet`, you can use the `IsType` method instead. `UnknownCellSet` also provides `GetCellSetName` for debugging purposes.

Common Errors

`CanConvert` *is almost always safer to use than* `IsType` *or its similar methods. Even though* `IsType` *reflects the actual cell set type,* `CanConvert` *better describes how* `UnknownCellSet` *will behave.*

34.3 Casting to a List of Potential Types

Using `AsCellSet` is fine as long as the correct types are known, but often times they are not. For this use case `UnknownCellSet` has a method named `CastAndCallForTypes` that attempts to cast the cell set to some set of types.

The `CastAndCallForTypes` method accepts a functor to run on the appropriately cast cell set. The functor must have an overloaded const parentheses operator that accepts a `CellSet` of the appropriate type. You also have to specify a template parameter that specifies a `vtkm::List` of cell set types to. The macro `VTKM_DEFAULT_CELL_SET_LIST` is often used when nothing more specific is known. The macros `VTKM_DEFAULT_CELL_SET_LIST_STRUCTURED` and `VTKM_DEFAULT_CELL_SET_LIST_UNSTRUCTURED` are also useful when you want to operate on only structured or unstructured cell sets.

Example 34.3: Operating on an `UnknownCellSet` with `CastAndCallForTypes`.

```
struct Get3DPointDimensionsFunctor
{
  template<vtkm::IdComponent Dims>
  VTKM_CONT void operator()(const vtkm::cont::CellSetStructured<Dims>& cellSet,
                            vtkm::Id3& outDims) const
  {
    vtkm::Vec<vtkm::Id, Dims> pointDims = cellSet.GetPointDimensions();
    for (vtkm::IdComponent d = 0; d < Dims; ++d)
    {
      outDims[d] = pointDims[d];
    }
  }

  VTKM_CONT void operator()(const vtkm::cont::CellSet& cellSet,
                            vtkm::Id3& outDims) const
  {
    outDims[0] = cellSet.GetNumberOfPoints();
  }
};

VTKM_CONT vtkm::Id3 Get3DPointDimensions(
  const vtkm::cont::UnknownCellSet& unknownCellSet)
{
  vtkm::Id3 dims(1);
  unknownCellSet.CastAndCallForTypes<VTKM_DEFAULT_CELL_SET_LIST>(
    Get3DPointDimensionsFunctor{}, dims);
  return dims;
}
```

Did you know?

The first (required) argument to `CastAndCallForTypes` *is the functor to call with the cell set. You can supply any number of optional arguments after that. Those arguments will be passed directly to the functor.*

. *This makes it easy to pass state to the functor.*

Did you know?

When an `UnknownCellSet` *is used in place of an* `CellSet` *as an argument to a worklet invocation, it will internally use* `CastAndCallForTypes` *to attempt to call the worklet with an* `CellSet` *of the correct type.*

`UnknownCellSet` has a simple subclass named `vtkm::cont::UncertainCellSet` for use when you can narrow the cell set to a finite set of types. `UncertainCellSet` has a template parameter that must be specified: a `vtkm::List` of cell set types. `UncertainCellSet` has a method named `CastAndCall` that behaves the same as `CastAndCallForTypes` except that you do not have to specify the types to try. Instead, the types are taken from the template parameters of the `UncertainCellSet` itself.

Example 34.4: Using `UncertainCellSet` to cast and call a functor.

```
using StructuredCellSetList = vtkm::List<vtkm::cont::CellSetStructured<1>,
                                         vtkm::cont::CellSetStructured<2>,
                                         vtkm::cont::CellSetStructured<3>>;
vtkm::cont::UncertainCellSet<StructuredCellSetList> uncertainCellSet(
  unknownCellSet);
uncertainCellSet.CastAndCall(Get3DPointDimensionsFunctor{}, dims);
```

Did you know?

Like with `UnknownCellSet`*, if an* `UncertainCellSet` *is used in a worklet invocation, it will internally use* `CastAndCall`*. This provides a convenient way to specify what cell set types the invoker should try.*

Both `UnknownCellSet` and `UncertainCellSet` provide a method named `ResetCellSetList` to redefine the types to try. `ResetCellSetList` has a template parameter that is the `vtkm::List` of cell sets. `ResetCellSetList` returns a new `UncertainCellSet` with the given types. This is a convenient way to pass these types to functions.

Example 34.5: Resetting the types of an `UnknownCellSet`.

```
using StructuredCellSetList = vtkm::List<vtkm::cont::CellSetStructured<1>,
                                         vtkm::cont::CellSetStructured<2>,
                                         vtkm::cont::CellSetStructured<3>>;
vtkm::cont::Invoker invoke;
invoke(
  MyWorklet{}, unknownCellSet.ResetCellSetList<StructuredCellSetList>(), outArray);
```

Common Errors

Because it returns an `UncertainCellSet`*, you need to include* `vtkm/cont/UncertainCellSet.h` *if you use* `UnknownCellSet::ResetCellSetList`*. This is true even if you do not directly use the returned object.*

CHAPTER

THIRTYFIVE

DEVICE ALGORITHMS

As described in Chapter 15, VTK-m is built around the concept of a *device adapter* that encapsulates the necessary features of each device on which VTK-m can run. At the core of the device adapter is a collection of basic algorithms optimized for the specific device. Many features of VTK-m, such as worklets, are built on top of these device algorithms. Using these higher level structures simplifies programming.

However, it is sometimes desirable to run directly run these algorithms provided by the device adapter. VTK-m comes with the templated class `vtkm::cont::Algorithm` that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. All algorithms also accept an optional device adapter argument.

`Algorithm` contains no state. It only has a set of static methods that implement its algorithms. The following methods are available.

Did you know?

Many of the following device adapter algorithms take input and output `ArrayHandles`*, and these functions will handle their own memory management. This means that it is unnecessary to allocate output arrays. For example, it is unnecessary to call* `ArrayHandle::Allocate` *for the output array passed to the* `Algorithm::Copy` *method.*

35.1 BitFieldToUnorderedSet

The `Algorithm::BitFieldToUnorderedSet` method creates a unique, unsorted list of indices denoting which bits are set in a bitfield. For example, running `BitFieldToUnorderedSet` on an input of [0,0,1,0,1,0,1,1,0,1,1,1] would return an array containing [2,4,6,7,9,10,11] or those numbers in some other order.

Example 35.1: Using the `BitFieldToUnorderedSet` algorithm.

```
1  vtkm::cont::BitField bits;
2  bits.Allocate(32);
3
4  auto fillPortal = bits.WritePortal();
5  fillPortal.SetWord(0, vtkm::UInt32(0xaa770011));
6
7  vtkm::cont::ArrayHandle<vtkm::Id> output;
8  auto setBits = vtkm::cont::Algorithm::BitFieldToUnorderedSet(bits, output);
```

35.2 Copy

The `Algorithm::Copy` method copies data from an input array to an output array. The copy takes place in the execution environment.

Example 35.2: Using the `Copy` algorithm.

```
vtkm::cont::ArrayHandleIndex input(12);

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::Algorithm::Copy(input, output);

// output has { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 }
```

35.3 CopyIf

The `Algorithm::CopyIf` method selectively removes values from an array. The *copy if* algorithm is also sometimes referred to as *stream compact*. The first argument, the input, is an `ArrayHandle` to be compacted (by removing elements). The second argument, the stencil, is an `ArrayHandle` of equal size with flags indicating whether the corresponding input value is to be copied to the output. The third argument is an output `ArrayHandle` whose length is set to the number of true flags in the stencil and the passed values are put in order to the output array.

`Algorithm::CopyIf` also accepts an optional fourth argument that is a unary predicate to determine what values in the stencil (second argument) should be considered true. See Section 35.23 for more information on unary predicates. The unary predicate determines the true/false value of the stencil that determines whether a given entry is copied. If no unary predicate is given, then `CopyIf` will copy all values whose stencil value is not equal to 0 (or the closest equivalent to it). More specifically, it copies values not equal to `vtkm::TypeTraits::ZeroInitialization`.

Example 35.3: Using the `CopyIf` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>(
    { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
vtkm::cont::ArrayHandle<vtkm::UInt8> stencil =
  vtkm::cont::make_ArrayHandle<vtkm::UInt8>(
    { 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1 });

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::Algorithm::CopyIf(input, stencil, output);


// output has { 0, 5, 3, 8, 3 }

struct LessThan5
{
  VTKM_EXEC_CONT bool operator()(vtkm::Int32 x) const { return x < 5; }
};

vtkm::cont::Algorithm::CopyIf(input, input, output, LessThan5());

// output has { 0, 1, 1, 4, 3, 3 }
```

35.4 CopySubRange

The `Algorithm::CopySubRange` method copies the contents of a section of one `ArrayHandle` to another. The first argument is the input `ArrayHandle`. The second argument is the index from which to start copying data. The third argument is the number of values to copy from the input to the output. The fourth argument is the output `ArrayHandle`, which will be grown if it is not large enough. The fifth argument, which is optional, is the index in the output array to start copying data to. If the output index is not specified, data are copied to the beginning of the output array.

Example 35.4: Using the `CopySubRange` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>(
    { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::Algorithm::CopySubRange(input, 1, 7, output);

// output has { 0, 1, 1, 5, 5, 4, 3 }
```

35.5 CountSetBits

The `Algorithm::CountSetBits` method returns the total number of set bits in a BitField. For example, running `BitFieldToUnorderedSet` on an input of [0,0,1,0,1,0,1,1,0,1,1,1] would return 7.

Example 35.5: Using the `CountSetBits` algorithm.

```
vtkm::cont::BitField bits;
bits.Allocate(32);

auto fillPortal = bits.WritePortal();
fillPortal.SetWord(0, vtkm::UInt32(0xaa770011));

vtkm::cont::ArrayHandle<vtkm::Id> output;
auto setBits = vtkm::cont::Algorithm::CountSetBits(bits);

// Will return that there are 12 set bits
```

35.6 Fill

The `Algorithm::Fill` methods fill a `BitField` or `ArrayHandle` with a specific pattern of bits/values. For a `BitField`, it is possible to supply a boolean value or a `WordType`. For boolean values, all bits are set to 1 if the value is true, 0 if the value is false. For word masks, the `WordType` must be an unsigned integral type; this value is stamped across the `BitField`. For a `ArrayHandle`, the entire array is filled with the provided value. For both types, if a `numValues` argument is provided the array is resized appropriately and filled with the given value.

Example 35.6: Using the `Fill` algorithm.

```
// Fill a BitField
vtkm::cont::BitField bits;
bits.Allocate(32);
vtkm::cont::Algorithm::Fill(bits, true);
// Will stamp the 8 bit word across 32 bits to result in bits = 0xf0f0f0f0
vtkm::cont::Algorithm::Fill(bits, vtkm::UInt8(0xf0));
vtkm::cont::Algorithm::Fill(bits, vtkm::UInt8(0xf0), 16);
```

```
8
9     // Fill an ArrayHandle
10    vtkm::cont::ArrayHandle<vtkm::Id> arrayHandle;
11    arrayHandle.Allocate(10);
12    vtkm::cont::Algorithm::Fill(arrayHandle, vtkm::Id(5));
13    vtkm::cont::Algorithm::Fill(arrayHandle, vtkm::Id(10), 5);
```

35.7 LowerBounds

The `Algorithm::LowerBounds` method takes three arguments. The first argument is an `ArrayHandle` of sorted values. The second argument is another `ArrayHandle` of items to find in the first array. `LowerBounds` find the index of the first item that is greater than or equal to the target value, much like the `std::lower_bound` STL algorithm. The results are returned in an `ArrayHandle` given in the third argument.

There are two specializations of `Algorithm::LowerBounds`. The first takes an additional comparison function that defines the less-than operation. The second specialization takes only two parameters. The first is an `ArrayHandle` of sorted `vtkm::Id` s and the second is an `ArrayHandle` of `vtkm::Id` to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

Example 35.7: Using the `LowerBounds` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
2       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
3         { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 });
4     vtkm::cont::ArrayHandle<vtkm::Int32> values =
5       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
6         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
7
8     vtkm::cont::ArrayHandle<vtkm::Id> output;
9
10    vtkm::cont::Algorithm::LowerBounds(sorted, values, output);
11
12    // output has { 8, 0, 1, 1, 6, 6, 5, 3, 8, 10, 11, 3 }
13
14    vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
15      vtkm::cont::make_ArrayHandle<vtkm::Int32>(
16        { 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 });
17
18    vtkm::cont::Algorithm::LowerBounds(
19      reverseSorted, values, output, vtkm::SortGreater());
20
21    // output has { 2, 11, 9, 9, 4, 4, 6, 7, 2, 1, 0, 7 }
```

35.8 Reduce

The `Algorithm::Reduce` method takes an input array, initial value, and a binary function and computes a "total" of applying the binary function to all entries in the array. The provided binary function must be associative (but it need not be commutative). There is a specialization of `Reduce` that does not take a binary function and computes the sum.

Example 35.8: Using the `Reduce` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Id> input =
2       vtkm::cont::make_ArrayHandle<vtkm::Id>({ 5, 1, 1, 6 });
3
4     vtkm::Id sum = vtkm::cont::Algorithm::Reduce(input, 0);
```

```
5
6     // sum is 13
7
8     vtkm::Id product = vtkm::cont::Algorithm::Reduce(input, 1, vtkm::Multiply());
9     // product is 30
```

35.9 ReduceByKey

The `Algorithm::ReduceByKey` method works similarly to the `Reduce` method except that it takes an additional array of keys, which must be the same length as the values being reduced. The arrays are partitioned into segments that have identical adjacent keys, and a separate reduction is performed on each partition. The unique keys and reduced values are returned in separate arrays.

Example 35.9: Using the `ReduceByKey` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Id> keys =
2       vtkm::cont::make_ArrayHandle<vtkm::Id>({ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 });
3     vtkm::cont::ArrayHandle<vtkm::Int32> input =
4       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
5         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
6
7     vtkm::cont::ArrayHandle<vtkm::Id> uniqueKeys;
8     vtkm::cont::ArrayHandle<vtkm::Int32> sums;
9
10    vtkm::cont::Algorithm::ReduceByKey(keys, input, uniqueKeys, sums, vtkm::Add());
11
12    // uniqueKeys is { 0, 3, 5, 6 }
13    // sums is { 7, 12, 4, 30 }
14
15    vtkm::cont::ArrayHandle<vtkm::Int32> products;
16
17    vtkm::cont::Algorithm::ReduceByKey(
18      keys, input, uniqueKeys, products, vtkm::Multiply());
19
20    // products is { 0, 25, 4, 4536 }
```

35.10 ScanInclusive

The `Algorithm::ScanInclusive` method takes an input and an output `ArrayHandle` and performs a running sum on the input array. For inclusive scans, the running sum value for position i in the input array *includes* the element at position i. The first value in the output is the same as the first value in the input. The second value in the output is the sum of the first two values in the input. The third value in the output is the sum of the first three values of the input, and so on. If the input and output array are the same, then the operation is done in place. `ScanInclusive` returns the sum of all values in the input. There are two forms of `ScanInclusive`: one performs the sum using addition whereas the other accepts a custom binary function to use as the sum operator.

Example 35.10: Using the `ScanInclusive` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Int32> input =
2       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
3         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
4
5     vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;
6
7     vtkm::cont::Algorithm::ScanInclusive(input, runningSum);
8
```

```
9     // runningSum is { 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50, 53 }
10
11    vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;
12
13    vtkm::cont::Algorithm::ScanInclusive(input, runningMax, vtkm::Maximum());
14
15    // runningMax is { 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9 }
```

35.11 ScanInclusiveByKey

The `Algorithm::ScanInclusiveByKey` method works similarly to the `ScanInclusive` method except that it takes an additional array of keys, which must be the same length as the values being scanned. The arrays are partitioned into segments that have identical adjacent keys, and a separate scan is performed on each partition. Only the scanned values are returned.

Example 35.11: Using the `ScanInclusiveByKey` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Id> keys =
2       vtkm::cont::make_ArrayHandle<vtkm::Id>({ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 });
3     vtkm::cont::ArrayHandle<vtkm::Int32> input =
4       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
5         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
6
7     vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;
8
9     vtkm::cont::Algorithm::ScanInclusiveByKey(keys, input, runningSums);
10
11    // runningSums is { 7, 7, 1, 2, 7, 12, 4, 3, 10, 18, 27, 30 }
12
13    vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;
14
15    vtkm::cont::Algorithm::ScanInclusiveByKey(
16      keys, input, runningMaxes, vtkm::Maximum());
17
18    // runningMax is { 7, 7, 1, 1, 5, 5, 4, 3, 7, 8, 9, 9 }
```

35.12 ScanExclusive

The `Algorithm::ScanExclusive` method takes an input and an output `ArrayHandle` and performs a running sum on the input array. For exclusive scans, the running sum value for position i in the input array *excludes* the element at position i. The first value in the output is always 0. The second value in the output is the same as the first value in the input. The third value in the output is the sum of the first two values in the input. The fourth value in the output is the sum of the first three values of the input, and so on. `ScanExclusive` returns the sum of all values in the input. If the input and output array are the same, then the operation is done in place. There are two forms of `ScanExclusive`. The first performs the sum using addition. The second form accepts a custom binary functor to use as the "sum" operator and a custom initial value (instead of 0).

Example 35.12: Using the `ScanExclusive` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Int32> input =
2       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
3         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
4
5     vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;
6
7     vtkm::cont::Algorithm::ScanExclusive(input, runningSum);
```

```
8
9     // runningSum is { 0, 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50 }
10
11    vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;
12
13    vtkm::cont::Algorithm::ScanExclusive(input, runningMax, vtkm::Maximum(), -1);
14
15    // runningMax is { -1, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9 }
```

35.13 ScanExclusiveByKey

The `Algorithm::ScanExclusiveByKey` method works similarly to the `ScanExclusive` method except that it takes an additional array of keys, which must be the same length as the values being scanned. The arrays are partitioned into segments that have identical adjacent keys, and a separate scan is performed on each partition. Only the scanned values are returned.

Example 35.13: Using `ScanExclusiveByKey` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Id> keys =
2       vtkm::cont::make_ArrayHandle<vtkm::Id>({ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 });
3     vtkm::cont::ArrayHandle<vtkm::Int32> input =
4       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
5         { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });
6
7     vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;
8
9     vtkm::cont::Algorithm::ScanExclusiveByKey(keys, input, runningSums);
10
11    // runningSums is { 0, 7, 0, 1, 2, 7, 0, 0, 3, 10, 18, 27 }
12
13    vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;
14
15    vtkm::cont::Algorithm::ScanExclusiveByKey(
16      keys, input, runningMaxes, -1, vtkm::Maximum());
17
18    // runningMax is { -1, 7, -1, 1, 1, 5, -1, -1, 3, 7, 8, 9 }
```

35.14 ScanExtended

The `Algorithm::ScanExtended` computes an extended prefix sum operation on the input `ArrayHandle` and stores it in a provided output `ArrayHandle`. The output array has length 1 greater than the input array. `Algorithm::ScanExtended` is a combination of the `Algorithm::ScanInclusive` and `Algorithm::ScanExclusive` methods. The exclusive scan values are stored in indices 0 through $size - 1$. The inclusive scan values are stored in indices 1 through $size$. The first entry in the resulting array is 0 (or the specified initial value) like with the exclusive scan. The last entry in the resulting array is the sum total like with the inclusive scan. Unlike the two referenced methods, `Algorithm::ScanExtended` does not return the total sum. By using an `ArrayHandleView`, arrays containing both inclusive and exclusive scans can be generated from an extended scan with minimal memory usage by referencing the correct indices in the extended scan output.

This algorithm may be more efficient than `Algorithm::ScanInclusive` and `Algorithm::ScanExclusive` on some devices; this algorithm may be able to avoid copying the total sum to the control environment to return. `Algorithm::ScanExtended` is similar to the STL partial sum function, with the exception that it does not perform a serial summation. This means that if you have defined a custom plus operator for T it must be associative, or you will get inconsistent results.

The first form performs the sum using addition. The second form accepts a custom binary functor to use as the operator and a custom initial value (instead of 0).

Example 35.14: Using `ScanExtended` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>(
    { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });

vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;
vtkm::cont::Algorithm::ScanExtended(input, runningSum);

// runningSum is { 0, 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50, 53 }

vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;
vtkm::cont::Algorithm::ScanExtended(input, runningMax, vtkm::Maximum(), -1);

// runningMax is { -1, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9 }
```

`ScanExtended` can be used to create a running sum that is quickly reversible. If you subtract to consecutive values of a scan you get back the original value. This is convenient if you need both the input and output of a scan; you can throw away the input and use differences of the output. However, `ScanInclusive` does not write out the initial value, so you cannot get back the original value at the beginning without a special condition. Likewise, `ScanExclusive` does not write out the total sum, so you cannot get back the original value at the end without a special condition. `ScanExtended` solves this problem by extending the array by 1. The original value for index i can be retrieved by subtracting scan value at index i from the value at index $i+1$ anywhere in the array, including the at the begin and end. This is particularly useful for storing packed arrays in structures like `vtkm::cont::CellSetExplicit` and `vtkm::cont::ArrayHandleGroupVecVariable`.

35.15 Schedule

The `Algorithm::Schedule` method takes a functor as its first argument and invokes it a number of times specified by the second argument. It should be assumed that each invocation of the functor occurs on a separate thread although in practice there could be some thread sharing.

There are two versions of the `Schedule` method. The first version takes a `vtkm::Id` and invokes the functor that number of times. The second version takes a `vtkm::Id3` and invokes the functor once for every entry in a 3D array of the given dimensions.

The functor is expected to be an object with a const overloaded parentheses operator. The operator takes as a parameter the index of the invocation, which is either a `vtkm::Id` or a `vtkm::Id3` depending on what version of `Schedule` is being used. The functor must also subclass `vtkm::exec::FunctorBase`, which provides the error handling facilities for the execution environment. `FunctorBase` contains a public method named `RaiseError` that takes a message and will cause a `vtkm::cont::ErrorExecution` exception to be thrown in the control environment.

35.16 Sort

The `Algorithm::Sort` method provides an unstable sort of an array. There are two forms of the `Sort` method. The first takes an `ArrayHandle` and sorts the values in place. The second takes an additional argument that is a functor that provides the comparison operation for the sort.

Example 35.15: Using the `Sort` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> array =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>(
    { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });

vtkm::cont::Algorithm::Sort(array);

// array has { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 }

vtkm::cont::Algorithm::Sort(array, vtkm::SortGreater());

// array has { 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 }
```

35.17 SortByKey

The `Algorithm::SortByKey` method works similarly to the `Sort` method except that it takes two `ArrayHandle`s: an array of keys and a corresponding array of values. The sort orders the array of keys in ascending values and also reorders the values so they remain paired with the same key. Like `Sort`, `SortByKey` has a version that sorts by the default less-than operator and a version that accepts a custom comparison functor.

Example 35.16: Using the `SortByKey` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> keys =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>({ 7, 0, 1, 5, 4, 8, 9, 3 });
vtkm::cont::ArrayHandle<vtkm::Id> values =
  vtkm::cont::make_ArrayHandle<vtkm::Id>({ 0, 1, 2, 3, 4, 5, 6, 7 });

vtkm::cont::Algorithm::SortByKey(keys, values);

// keys has   { 0, 1, 3, 4, 5, 7, 8, 9 }
// values has { 1, 2, 7, 4, 3, 0, 5, 6 }

vtkm::cont::Algorithm::SortByKey(keys, values, vtkm::SortGreater());

// keys has   { 9, 8, 7, 5, 4, 3, 1, 0 }
// values has { 6, 5, 0, 3, 4, 7, 2, 1 }
```

35.18 Synchronize

The `Synchronize` method waits for any asynchronous operations running on the device to complete and then returns.

35.19 Transform

The `Algorithm::Transform` method applies a given binary operation function element-wise on two input arrays, storing the result in a provided output array. The number of elements in the input arrays do not have to be the same; the output array will have the same number of elements as the smaller of the two input arrays.

Example 35.17: Using the `Transform` algorithm.

```
vtkm::cont::ArrayHandle<vtkm::Int32> input1 =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>({ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 });
vtkm::cont::ArrayHandle<vtkm::Int32> input2 =
  vtkm::cont::make_ArrayHandle<vtkm::Int32>({ 2, 3, 4, 5, 6, 7, 8, 9, 10 });

```

```
6     vtkm::cont::ArrayHandle<vtkm::Int32> output;
7     vtkm::cont::Algorithm::Transform(input1, input2, output, vtkm::Sum());
8
9     // output is { 3, 5, 7, 9, 11, 13, 15, 17, 19 }
```

35.20 Unique

The `Algorithm::Unique` method removes all duplicate values in an `ArrayHandle`. The method will only find duplicates if they are adjacent to each other in the array. The easiest way to ensure that duplicate values are adjacent is to sort the array first.

There are two versions of `Unique`. The first uses the equals operator to compare entries. The second accepts a binary functor to perform the comparisons.

Example 35.18: Using the `Unique` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Int32> values =
2       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
3         { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 7, 9 });
4
5     vtkm::cont::Algorithm::Unique(values);
6
7     // values has {0, 1, 3, 4, 5, 7, 9}
8
9     vtkm::cont::ArrayHandle<vtkm::Float64> fvalues =
10      vtkm::cont::make_ArrayHandle<vtkm::Float64>(
11        { 0.0, 0.001, 0.0, 1.5, 1.499, 2.0 });
12
13    struct AlmostEqualFunctor
14    {
15      VTKM_EXEC_CONT bool operator()(vtkm::Float64 x, vtkm::Float64 y) const
16      {
17        return (vtkm::Abs(x - y) < 0.1);
18      }
19    };
20
21    vtkm::cont::Algorithm::Unique(fvalues, AlmostEqualFunctor());
22
23    // values has {0.0, 1.5, 2.0}
```

35.21 UpperBounds

The `Algorithm::UpperBounds` method takes three arguments. The first argument is an `ArrayHandle` of sorted values. The second argument is another `ArrayHandle` of items to find in the first array. `UpperBounds` find the index of the first item that is greater than to the target value, much like the `std::upper_bound` STL algorithm. The results are returned in an `ArrayHandle` given in the third argument.

There are two specializations of `UpperBounds`. The first takes an additional comparison function that defines the less-than operation. The second takes only two parameters. The first is an `ArrayHandle` of sorted `vtkm::Id` s and the second is an `ArrayHandle` of `vtkm::Id` s to find in the first list. The results are written back out to the second array. This second specialization is useful for inverting index maps.

Example 35.19: Using the `UpperBounds` algorithm.

```
1     vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
2       vtkm::cont::make_ArrayHandle<vtkm::Int32>(
```

```
      { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 });
  vtkm::cont::ArrayHandle<vtkm::Int32> values =
    vtkm::cont::make_ArrayHandle<vtkm::Int32>(
      { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });

  vtkm::cont::ArrayHandle<vtkm::Id> output;

  vtkm::cont::Algorithm::UpperBounds(sorted, values, output);

  // output has { 10, 1, 3, 3, 8, 8, 6, 5, 10, 11, 12, 5 }

  vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
    vtkm::cont::make_ArrayHandle<vtkm::Int32>(
      { 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 });

  vtkm::cont::Algorithm::UpperBounds(
    reverseSorted, values, output, vtkm::SortGreater());

  // output has { 4, 12, 11, 11, 6, 6, 7, 9, 4, 2, 1, 9 }
```

35.22 Specifying the Device Adapter

When you call a method in `vtkm::cont::Algorithm`, a device is automatically specified based on available hardware and the VTK-m state. However, if you want to use a specific device, you can specify that device by passing either a `vtkm::cont::DeviceAdapterId` or a device adapter tag as the first argument to any of these methods.

Example 35.20: Using the `DeviceAdapter` with `vtkm::cont::Algorithm`.

```
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle<vtkm::Int32>(
      { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 });

  vtkm::cont::ArrayHandle<vtkm::Int32> output_no_device_specified;

  vtkm::cont::ArrayHandle<vtkm::Int32> output_device_specified;

  vtkm::cont::Algorithm::Copy(input, output_no_device_specified);

  //optional we can pass the device or int id number
  vtkm::cont::Algorithm::Copy(
    vtkm::cont::DeviceAdapterTagSerial(), input, output_device_specified);

  // output has { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 }
```

35.23 Predicates and Operators

VTK-m follows certain design philosophies consistent with the functional programming paradigm. This assists in making implementations device agnostic and ensuring that various functions operate correctly and efficiently in multiple environments. Many basic operations, such as binary and unary comparisons and predicates, are implemented as templated functors. These are mostly re-implementations of basic C++ STL functors that can be used in the VTK-m execution environment.

Strictly using a functor by itself adds little in the way of functionality to the code. Their use is demonstrated more when used as parameters to one of the `vtkm::cont::Algorithm` methods discussed earlier in this chapter.

Currently, VTK-m provides 3 categories of functors: `Unary Predicates`, `Binary Predicates`, and `Binary Operators`.

35.23.1 Unary Predicates

`Unary Predicates` are functors that take a single parameter and return a Boolean value. These types of functors are useful in determining if values have been initialized or zeroed out correctly.

`vtkm::IsZeroInitialized` Returns True if argument is the identity of its type.

`vtkm::NotZeroInitialized` Returns True if the argument is not the identify of its type.

`vtkm::LogicalNot` Returns True iff the argument is False. Requires that the argument type is convertible to a Boolean or implements the `!` operator.

Example 35.21: Basic Unary Predicate.

```
1  vtkm::IsZeroInitialized zero_initialized;
2  vtkm::NotZeroInitialized not_zero_initialized;
3  vtkm::LogicalNot logical_not;
4
5  bool zeroed = zero_initialized(vtkm::TypeTraits<vtkm::Id>::ZeroInitialization());
6  bool notZeroed = not_zero_initialized(vtkm::Id(1));
7  bool logicalNot = logical_not(false);
```

35.23.2 Binary Predicates

`Binary Predicates` take two parameters and return a single Boolean value. These types of functors are used when comparing two different parameters for some sort of condition.

`vtkm::Equal` Returns True iff the first argument is equal to the second argument. Requires that the argument type implements the `==` operator.

`vtkm::NotEqual` Returns True iff the first argument is not equal to the second argument. Requires that the argument type implements the `!=` operator.

`vtkm::SortLess` Returns True iff the first argument is less than the second argument. Requires that the argument type implements the `<` operator.

`vtkm::SortGreater` Returns True iff the first argument is greater than the second argument. Requires that the argument type implements the `<` operator (the comparison is inverted internally).

`vtkm::LogicalAnd` Returns True iff the first argument and the second argument are True. Requires that the argument type is convertible to a Boolean or implements the `&&` operator.

`vtkm::LogicalOr` Returns True iff the first argument or the second argument is True. Requires that the argument type is convertible to a Boolean or implements the `||` operator.

Example 35.22: Basic Binary Predicate.

```
1  vtkm::Equal equal_;
2  vtkm::NotEqual not_equal;
3  vtkm::SortLess sort_less;
4  vtkm::SortGreater sort_greater;
```

```
5   vtkm::LogicalAnd logical_and;
6   vtkm::LogicalOr logical_or;
7
8   bool equal = equal_(vtkm::Id(1), vtkm::Id(1));
9   bool notEqual = not_equal(vtkm::Id(1), vtkm::Id(2));
10  bool sortLess = sort_less(vtkm::Id(1), vtkm::Id(2));
11  bool sortGreater = sort_greater(vtkm::Id(2), vtkm::Id(1));
12  bool logicalAnd = logical_and(true, true);
13  bool logicalOr = logical_or(true, false);
```

35.23.3 Binary Operators

`Binary Operators` take two parameters and return a single value (usually of the same type as the input arguments). These types of functors are useful when performing reductions or transformations of a dataset.

`vtkm::Sum` Returns the sum of two arguments. Requires that the argument type implements the `+` operator.

`vtkm::Product` Returns the product (multiplication) of two arguments. Requires that the argument type implements the `*` operator.

`vtkm::Maximum` Returns the larger of two arguments. Requires that the argument type implements the `<` operator.

`vtkm::Minimum` Returns the smaller of two arguments. Requires that the argument type implements the `<` operator.

`vtkm::MinAndMax` Returns a `vtkm::Vec <T,2>` that represents the minimum and maximum values. Requires that the argument type implements the `vtkm::Min` and `vtkm::Max` functions.

`vtkm::BitwiseAnd` Returns the bitwise and of two arguments. Requires that the argument type implements the `&` operator.

`vtkm::BitwiseOr` Returns the bitwise or of two arguments. Requires that the argument type implements the `|` operator.

`vtkm::BitwiseXor` Returns the bitwise xor of two arguments. Requires that the argument type implements the `^` operator.

Example 35.23: Basic Binary Operator.

```
1   vtkm::Sum sum_;
2   vtkm::Product product_;
3   vtkm::Maximum maximum_;
4   vtkm::Minimum minimum_;
5   vtkm::MinAndMax<vtkm::Id> min_and_max;
6   vtkm::BitwiseAnd bitwise_and;
7   vtkm::BitwiseOr bitwise_or;
8   vtkm::BitwiseXor bitwise_xor;
9
10  vtkm::Id sum = sum_(vtkm::Id(1), vtkm::Id(1));
11  vtkm::Id product = product_(vtkm::Id(2), vtkm::Id(2));
12  vtkm::Id max = maximum_(vtkm::Id(1), vtkm::Id(2));
13  vtkm::Id min = minimum_(vtkm::Id(1), vtkm::Id(2));
14  vtkm::Id2 minAndMax = min_and_max(vtkm::Id(3), vtkm::Id(4));
15  vtkm::Id bitwiseAnd = bitwise_and(vtkm::Id(1), vtkm::Id(3));
16  vtkm::Id bitwiseOr = bitwise_or(vtkm::Id(1), vtkm::Id(2));
17  vtkm::Id bitwiseXor = bitwise_xor(vtkm::Id(7), vtkm::Id(4));
```

35.23.4 Creating Custom Comparators

In addition to using the built in operators and predicates, it is possible to create your own custom functors to be used in one of the vtkm::cont::Algorithm. Custom operator and predicate functors can be used to apply specific logic used to manipulate your data. The following example creates a unary predicate that checks if the input is a power of 2.

Example 35.24: Custom Unary Predicate Implementation.

```
struct PowerOfTwo
{
  VTKM_EXEC_CONT bool operator()(const vtkm::Id& x) const
  {
    if (x <= 0)
    {
      return false;
    }
    vtkm::BitwiseAnd bitwise_and;
    return bitwise_and(x, vtkm::Id(x - 1)) == 0;
  }
};
```

Example 35.25: Custom Unary Predicate Usage.

```
  PowerOfTwo power_of_two;

  bool powerOfTwo = power_of_two(vtkm::Id(4)); // returns true
  powerOfTwo = power_of_two(vtkm::Id(5)); // returns false
```

CHAPTER

THIRTYSIX

CUSTOM ARRAY STORAGE

Chapters 16, 26, and 27 introduce the `vtkm::cont::ArrayHandle` class. In them, we learned how an `ArrayHandle` manages the memory allocation of an array, provides access to the data via array portals, and supervises the movement of data between the control and execution environments.

In addition to these data management features, `ArrayHandle` also provides a configurable *storage* mechanism that allows you, through efficient template configuration, to redefine how data are stored and retrieved. The storage object provides an encapsulated interface around the data so that any necessary strides, offsets, or other access patterns may be handled internally. The relationship between array handles and their storage object is shown in Figure 36.1.

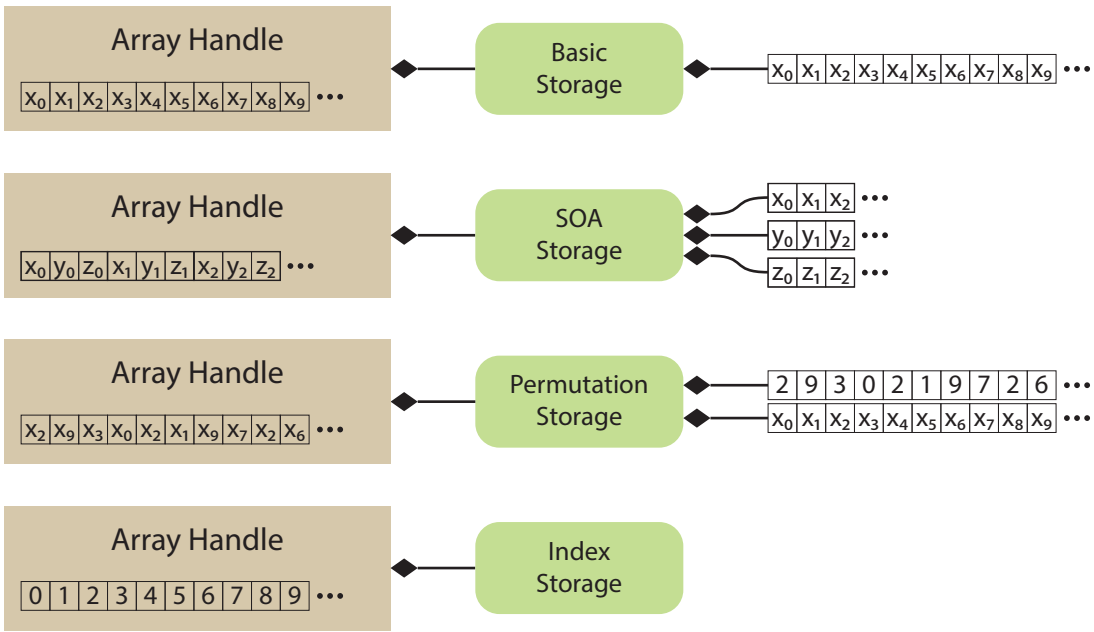


Figure 36.1: Array handles, storage objects, and the underlying data source.

As previously discussed in Chapter 16, `vtkm::cont::ArrayHandle` takes two template arguments.

Example 36.1: Declaration of the `vtkm::cont::ArrayHandle` templated class (again).

```
1 template<typename T, typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
2 class ArrayHandle;
```

The first argument is the only one required and has been demonstrated multiple times before. The second (optional) argument specifies something called a storage, which provides the interface between the generic `vtkm::cont::ArrayHandle` class and a specific storage mechanism in the control environment. If the storage parameter is not explicitly defined, it is set to `VTKM_DEFAULT_STORAGE_TAG`, which is a macro that resolves to `vtkm::cont::StorageTagBasic`.

The default storage can always be overridden by specifying an array storage tag. Here is an example of specifying the storage type when declaring an array handle.

Example 36.2: Specifying the storage type for an `ArrayHandle`.

```
1 | vtkm::cont::ArrayHandle<vtkm::Float32, vtkm::cont::StorageTagBasic> arrayHandle;
```

Although setting an `ArrayHandle`'s storage explicitly to `StorageTagBasic` as in Example 36.2 is seldom useful (since this is the default value), setting the storage is a good way to propagate the storage mechanism through template parameters. The remainder of this chapter uses the storage mechanism to customize the representation of arrays.

By replacing the storage template parameter for `ArrayHandle`, we can change how data are stored in memory. For example, when storing `vtkm::Vec` objects, the basic storage writes all `vtkm::Vec` s sequentially in a single array. This is known as an *array of structures*. An alternate representation would be to store each component of the `vtkm::Vec` s in a separate array. This alternate layout is known as a *structure of arrays*. There are reasons one might want to represent vector data in a structure of arrays, and VTK-m provides a separate storage to implement that: `vtkm::cont::StorageTagSOA`. From an interface perspective, the two `ArrayHandle`s behave the same, but they have very different implementations.

As is typical of different types of storage, the basic and SOA storage types have convenience `ArrayHandle` subclasses: `vtkm::cont::ArrayHandleBasic` and `vtkm::cont::ArrayHandleSOA`, respectively. These are trivial subclasses of `vtkm::cont::ArrayHandle` with the appropriate storage, but provide convenience methods for construction and data access.

One interesting consequence of using a generic storage object to manage data within an array handle is that the storage can be defined functionally rather than point to data stored in physical memory. Thus, implicit array handles are easily created by adapting to functional "storage." For example, the point coordinates of a uniform rectilinear grid are implicit based on the topological position of the point. Thus, the point coordinates for uniform rectilinear grids can be implemented as an implicit array with the same interface as explicit arrays (where unstructured grid points would be stored). Many examples of this are listed in Chapter 26.

In this chapter we explore the many ways you can manipulate the `ArrayHandle` storage. There are many ways to create custom storage for `ArrayHandle`. As we explore these different ways, we will start with the easiest but most restrictive ways and move to the most expressive ways.

36.1 Implicit Array Handles

The generic array handle and storage templating in VTK-m allows for any type of operations to retrieve a particular value. Typically this is used to convert an index to some location or locations in memory. However, it is also possible to compute a value directly from an index rather than look up some value in memory. Such an array is completely functional and requires no storage in memory at all. Such a functional array is called an *implicit array handle*. Implicit arrays are an example of *fancy array handles*, which are array handles that behave like regular arrays but do special processing under the covers to provide values.

Specifying a functional or implicit array in VTK-m is straightforward. VTK-m has a special class named `vtkm::cont::ArrayHandleImplicit` that makes an implicit array containing values generated by a user-specified *functor*. A functor is simply a C++ class or struct that contains an overloaded parenthesis operator so that it

can be used syntactically like a function.

To demonstrate the use of `ArrayHandleImplicit`, let us say we want an array of even numbers. The array has the values $[0, 2, 4, 6, \ldots]$ (double the index) up to some given size. Although we could easily create this array in memory, we can save space and possibly time by computing these values on demand.

Did you know?

VTK-m already comes with an implicit array handle named `vtkm::cont::ArrayHandleCounting` *that can make implicit even numbers as well as other more general counts. (See Section 26.4 for details.) So in practice you would not have to create a special implicit array, but we are doing so here for demonstrative purposes.*

The first step to using `ArrayHandleImplicit` is to declare a functor. The functor's parenthesis operator should accept a single argument of type `vtkm::Id` and return a value appropriate for that index. The parenthesis operator should also be declared `const` because it is not allowed to change the class' state.

Example 36.3: Functor that doubles an index.

```
1 struct DoubleIndexFunctor
2 {
3   VTKM_EXEC_CONT
4   vtkm::Id operator()(vtkm::Id index) const { return 2 * index; }
5 };
```

Common Errors

The functor used with `ArrayHandleImplicit` *may contain state, but it must be trivially copiable across memory spaces. That means it cannot contain any virtual methods nor hold any pointers or references. It also means it cannot hold objects like an* `ArrayHandle` *or* `ArrayPortal`*. Such behavior may seem to work at first, but will quickly break down on different devices. The point of an implicit array is that it completely computes each value without referencing any external data. Later in this chapter we will explore many ways to create storage that access data stored in memory or other* `ArrayHandle`*s.*

Once the functor is defined, an implicit array can be declared using the templated `vtkm::cont::ArrayHandleImplicit` class. The single template argument is the functor's type.

Example 36.4: Declaring a `ArrayHandleImplicit`.

```
1   vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor> implicitArray(
2     DoubleIndexFunctor(), 50);
```

For convenience, vtkm/cont/ArrayHandleImplicit.h also declares the `vtkm::cont::make_ArrayHandleImplicit` function. This function takes a functor and the size of the array and returns the implicit array.

Example 36.5: Using `make_ArrayHandleImplicit`.

```
1     vtkm::cont::make_ArrayHandleImplicit(DoubleIndexFunctor(), 50);
```

If the implicit array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of `vtkm::cont::ArrayHandleImplicit` for your array.

Example 36.6: Custom implicit array handle for even numbers.

```
#include <vtkm/cont/ArrayHandleImplicit.h>

class ArrayHandleDoubleIndex
  : public vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
    ArrayHandleDoubleIndex,
    (vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor>));

  VTKM_CONT
  ArrayHandleDoubleIndex(vtkm::Id numberOfValues)
    : Superclass(DoubleIndexFunctor(), numberOfValues)
  {
  }
};
```

Subclasses of `ArrayHandle` provide constructors that establish the state of the array handle. All array handle subclasses must also use either the `VTKM_ARRAY_HANDLE_SUBCLASS` macro or the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. Both of these macros define the types `Superclass`, `ValueType`, and `StorageTag` as well as a set of constructors and operators expected of all `ArrayHandle` classes. The difference between these two macros is that `VTKM_ARRAY_HANDLE_SUBCLASS` is used in templated classes whereas `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is used in non-templated classes.

The `ArrayHandle` subclass in Example 36.6 is not templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. This macro takes two parameters. The first parameter is the name of the subclass where the macro is defined and the second parameter is the immediate superclass including the full template specification. The second parameter of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification. (The other macro is described in Section 36.2 on page 318).

36.2 Transformed Arrays

Another type of fancy array handle is the transformed array. A transformed array takes another array and applies a function to all of the elements to produce a new array. A transformed array behaves much like a map operation except that a map operation writes its values to a new memory location whereas the transformed array handle produces its values on demand so that no additional storage is required.

Specifying a transformed array in VTK-m is straightforward. VTK-m has a special class named `vtkm::cont::ArrayHandleTransform` that takes an array handle and a functor and provides an interface to a new array comprising values of the functor applied the first array.

To demonstrate the use of `ArrayHandleTransform`, let us say that we want to scale and bias all of the values in a target array. That is, each value in the target array is going to be multiplied by a given scale and then offset by adding a bias value. (The scale and bias are uniform across all entries.) We could, of course, easily create a worklet to apply this scale and bias to each entry in the target array and save the result in a new array, but we can save space and possibly time by computing these values on demand.

The first step to using `ArrayHandleTransform` is to declare a functor. The functor's parenthesis operator should accept a single argument of the type of the target array and return the transformed value. For more generally applicable transform functors, it is often useful to make the parenthesis operator a template. The parenthesis operator should also be declared `const` because it is not allowed to change the class' state.

Example 36.7: Functor to scale and bias a value.

```
template<typename T>
```

```
struct ScaleBiasFunctor
{
  VTKM_EXEC_CONT
  ScaleBiasFunctor(T scale = T(1), T bias = T(0))
    : Scale(scale)
    , Bias(bias)
  {
  }

  VTKM_EXEC_CONT
  T operator()(T x) const { return this->Scale * x + this->Bias; }

  T Scale;
  T Bias;
};
```

Common Errors

As with functors for implicit arrays, a functor for `ArrayHandleTransform` *must be trivially copiable. It may not hold in its state reference to any other arrays. The functor may only access the one value from the input array that is passed as an argument to the parenthesis operator. Mechanisms to build arrays with more expressive access to one or more other arrays are described later in this chapter.*

Once the functor is defined, a transformed array can be declared using the templated `vtkm::cont::ArrayHandleTransform` class. The first template argument is the type of array being transformed. The second template argument is the type of functor used for the transformation. The third template argument, which is optional, is the type for an inverse functor that provides the inverse operation of the functor in the second argument. This inverse functor is used for writing values into the array. For arrays that will only be read from, there is no need to supply this inverse functor.

That said, it is generally easier to use the `vtkm::cont::make_ArrayHandleTransform` convenience function. This function takes an array and a functor (and optionally an inverse functor) and returns a transformed array.

Example 36.8: Using `make_ArrayHandleTransform`.

```
    vtkm::cont::make_ArrayHandleTransform(array,
                                          ScaleBiasFunctor<vtkm::Float32>(2, 3))
```

If the transformed array you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of `vtkm::cont::ArrayHandleTransform` or convenience `make_ArrayHandle*` function for your array.

Example 36.9: Custom transform array handle for scale and bias.

```
#include <vtkm/cont/ArrayHandleTransform.h>

template<typename ArrayHandleType>
class ArrayHandleScaleBias
  : public vtkm::cont::ArrayHandleTransform<
      ArrayHandleType,
      ScaleBiasFunctor<typename ArrayHandleType::ValueType>>
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);

public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleScaleBias,
```

```
14      (ArrayHandleScaleBias<ArrayHandleType>),
15      (vtkm::cont::ArrayHandleTransform<
16        ArrayHandleType,
17        ScaleBiasFunctor<typename ArrayHandleType::ValueType>>));
18
19    VTKM_CONT
20    ArrayHandleScaleBias(const ArrayHandleType& array, ValueType scale, ValueType bias)
21      : Superclass(array, ScaleBiasFunctor<ValueType>(scale, bias))
22    {
23    }
24  };
25
26  template<typename ArrayHandleType>
27  VTKM_CONT ArrayHandleScaleBias<ArrayHandleType> make_ArrayHandleScaleBias(
28    const ArrayHandleType& array,
29    typename ArrayHandleType::ValueType scale,
30    typename ArrayHandleType::ValueType bias)
31  {
32    return ArrayHandleScaleBias<ArrayHandleType>(array, scale, bias);
33  }
```

Subclasses of `ArrayHandle` provide constructors that establish the state of the array handle. All array handle subclasses must also use either the `VTKM_ARRAY_HANDLE_SUBCLASS` macro or the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. Both of these macros define the types `Superclass`, `ValueType`, and `StorageTag` as well as a set of constructors and operators expected of all `ArrayHandle` classes. The difference between these two macros is that `VTKM_ARRAY_HANDLE_SUBCLASS` is used in templated classes whereas `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is used in non-templated classes.

The `ArrayHandle` subclass in Example 36.9 is templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS` macro. This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification. (The other macro is described in Section 36.1 on page 316).

36.3 Decorated Arrays

In the previous section, we saw how to augment a single array to modify its values in place. However, `ArrayHandleTransform` is limited in that it only allows you to augment one array at a time, and it does not allow you to adjust the index lookups into the array.

If `ArrayHandleTransform` is not powerful enough, VTK-m provides `vtkm::cont::ArrayHandleDecorator` for more general augmentation of arrays. `ArrayHandleDecorator` provides a much more expressive (albeit complicated) interface.

In this section we will demonstrate the steps required to create a more general derived storage. For the purposes of the example in this section, let us say we want 2 array handles to behave as one array with the contents interlaced together. That is, the first w items are the same as the first w of the first array, the second w items are the same as the first w of the second array, the third w items are the same as the second w of the first array, the fourth w items are the same as the second w of the second array, and so on. We could of course actually copy the data, but we can also do it in place.

Using `ArrayHandleDecorator` requires creating functors for getting and setting data in the arrays plus creating another "interface" structure to manage the functors and data. As always, these can be used to define new `ArrayHandle` classes.

36.3.1 Functors

Like `ArrayHandleImplicit` and `ArrayHandleTransform`, you define the behavior of `ArrayHandleDecorator` by defining functors. However, unlike the functors for the other `ArrayHandle`s, we relax the restrictions on `ArrayHandleDecorator`'s functors and allow them to hold array portals as part of its state. Note that there are still restrictions on where these portals come from. They are generated internally by `ArrayHandleDecorator` and are passed to the functors as described in Section 36.3.2.

The decorator needs a functor with a parentheses operator that is given an index and returns a value for that index. The interface is the same as that for `ArrayHandleImplicit`, but the returned result can come from data in array portals.

Example 36.10: Functor that interlaces two array portals.

```
template<typename PortalType1, typename PortalType2>
class InterlaceFunctor
{
  PortalType1 Portal1;
  PortalType2 Portal2;
  vtkm::Id Width;

public:
  InterlaceFunctor(const PortalType1& portal1,
                   const PortalType2& portal2,
                   vtkm::Id width)
    : Portal1(portal1)
    , Portal2(portal2)
    , Width(width)
  {
  }

  VTKM_EXEC_CONT typename PortalType1::ValueType operator()(vtkm::Id index) const
  {
    vtkm::Id interleaveGroup = index / (this->Width * 2);
    vtkm::Id interleaveIndex = index % (this->Width * 2);
    if (interleaveIndex < this->Width)
    {
      return this->Portal1.Get(interleaveIndex + (interleaveGroup * this->Width));
    }
    else
    {
      return this->Portal2.Get((interleaveIndex - this->Width) +
                               (interleaveGroup * this->Width));
    }
  }
};
```

This functor will be used for retrieving data. If the array being generated is intended to be read-only, then this is all you need. However, if you wish to create a writable array, then you need a second "inverse" functor to set data to the array. Fundamentally, there is no reason why we should not support writing data to the interlaced arrays, so we define a secondary inverse functor.

Example 36.11: Inverse functor for writing data to interlaced array portals.

```
template<typename PortalType1, typename PortalType2>
class InterlaceInverseFunctor
{
  PortalType1 Portal1;
  PortalType2 Portal2;
  vtkm::Id Width;

public:
  InterlaceInverseFunctor(const PortalType1& portal1,
```

```
                          const PortalType2& portal2,
                          vtkm::Id width)
    : Portal1(portal1)
    , Portal2(portal2)
    , Width(width)
  {
  }

  template<typename T>
  VTKM_EXEC_CONT void operator()(vtkm::Id index, const T& value) const
  {
    vtkm::Id interleaveGroup = index / (this->Width * 2);
    vtkm::Id interleaveIndex = index % (this->Width * 2);
    if (interleaveIndex < this->Width)
    {
      this->Portal1.Set(interleaveIndex + (interleaveGroup * this->Width), value);
    }
    else
    {
      this->Portal2.Set(
        (interleaveIndex - this->Width) + (interleaveGroup * this->Width), value);
    }
  }
};
```

36.3.2 Interface

The next step in creating a decorator array is to define an "interface" class that tells `ArrayHandleDecorator` how to read, write, and resize the array. This is just a normal `class` or `struct` containing the following members.

`CreateFunctor` A method that takes one or more array portals and returns a functor used to get information from these portals. These portals will respectively come from the arrays being decorated. This method is required.

`CreateInverseFunctor` A method that takes one or more array portals and returns a functor used to set information in these portals. These portals will respectively come from the arrays being decorated. This method is optional. If it is omitted, then the decorated array will be read-only.

`AllocateSourceArrays` A method that resizes the arrays being decorated. The first three arguments are the new size of the array, a `vtkm::CopyFlag` indicating whether data should be preserved, and a `vtkm::cont::Token`, respectively. The remaining arguments are the arrays being decorated. This method is optional. If it is omitted, then the decorated array cannot be resized after creation.

Our implementation of interlaced array handles produces the functors presented in Exercises 36.10 and 36.11. It also provides a mechanism to resize the arrays.

Example 36.12: Decorator implementation class for interleaving `ArrayHandle`s.

```
struct InterlaceImplementation
{
  vtkm::Id Width;

  InterlaceImplementation(vtkm::Id width = 1)
    : Width(width)
  {
  }

  template<typename PortalType1, typename PortalType2>
```

```
  VTKM_CONT InterlaceFunctor<PortalType1, PortalType2> CreateFunctor(
    const PortalType1& portal1,
    const PortalType2 portal2) const
  {
    return InterlaceFunctor<PortalType1, PortalType2>(portal1, portal2, this->Width);
  }

  template<typename PortalType1, typename PortalType2>
  VTKM_CONT InterlaceInverseFunctor<PortalType1, PortalType2> CreateInverseFunctor(
    const PortalType1& portal1,
    const PortalType2 portal2) const
  {
    return InterlaceInverseFunctor<PortalType1, PortalType2>(
      portal1, portal2, this->Width);
  }

  template<typename ArrayType1, typename ArrayType2>
  VTKM_CONT void AllocateSourceArrays(vtkm::Id numValues,
                                      vtkm::CopyFlag preserve,
                                      vtkm::cont::Token& token,
                                      ArrayType1& array1,
                                      ArrayType2& array2) const
  {
    vtkm::Id numInterleaveGroups = (numValues / (this->Width * 2));
    vtkm::Id remainder = numValues - (numInterleaveGroups * this->Width * 2);
    if (remainder < this->Width)
    {
      array1.Allocate(
        (numInterleaveGroups * this->Width) + remainder, preserve, token);
      array2.Allocate((numInterleaveGroups * this->Width), preserve, token);
    }
    else
    {
      array1.Allocate((numInterleaveGroups + 1) * this->Width, preserve, token);
      array2.Allocate((numInterleaveGroups * this->Width) + remainder - this->Width,
                      preserve,
                      token);
    }
  }
};
```

36.3.3 Subclass

If the array decorator you are creating tends to be generally useful and is something you use multiple times, it might be worthwhile to make a convenience subclass of `vtkm::cont::ArrayHandleDecorator` or convenience `make_ArrayHandle*` function for your array.

Example 36.13: Custom decorator array handle for interleaving arrays.

```
template<typename ArrayType1, typename ArrayType2>
class ArrayHandleInterlace
  : public vtkm::cont::
      ArrayHandleDecorator<InterlaceImplementation, ArrayType1, ArrayType2>
{
  VTKM_IS_ARRAY_HANDLE(ArrayType1);
  VTKM_IS_ARRAY_HANDLE(ArrayType2);

public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleInterlace,
    (ArrayHandleInterlace<ArrayType1, ArrayType2>),
    (vtkm::cont::
```

```
14        ArrayHandleDecorator<InterlaceImplementation, ArrayType1, ArrayType2>));
15
16    VTKM_CONT ArrayHandleInterlace(const ArrayType1& array1,
17                                   const ArrayType2& array2,
18                                   vtkm::Id width = 1)
19      : Superclass(vtkm::cont::make_ArrayHandleDecorator(
20          array1.GetNumberOfValues() + array2.GetNumberOfValues(),
21          InterlaceImplementation(width),
22          array1,
23          array2))
24    {
25    }
26  };
27
28  template<typename ArrayType1, typename ArrayType2>
29  VTKM_CONT ArrayHandleInterlace<ArrayType1, ArrayType2> make_ArrayHandleInterlace(
30    const ArrayType1& array1,
31    const ArrayType2& array2,
32    vtkm::Id width = 1)
33  {
34    return ArrayHandleInterlace<ArrayType1, ArrayType2>(array1, array2, width);
35  }
```

Subclasses of `ArrayHandle` provide constructors that establish the state of the array handle. All array handle subclasses must also use either the `VTKM_ARRAY_HANDLE_SUBCLASS` macro or the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. Both of these macros define the types `Superclass`, `ValueType`, and `StorageTag` as well as a set of constructors and operators expected of all `ArrayHandle` classes. The difference between these two macros is that `VTKM_ARRAY_HANDLE_SUBCLASS` is used in templated classes whereas `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is used in non-templated classes.

The `ArrayHandle` subclass in Example 36.13 is templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS` macro. This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification. (The other macro is described in Section 36.1 on page 316).

36.4 Derived Storage

A *derived storage* is a type of fancy array that takes one or more other arrays and changes their behavior in some way. A transformed array (Section 36.2) is a specific type of derived array with a simple mapping. In this section we will demonstrate the steps required to create a more general derived storage. When applicable, it is much easier to create a derived array as a transformed array or using the other fancy arrays than to create your own derived storage. However, if these pre-existing fancy arrays do not work work, for example if your derivation uses multiple arrays or requires general lookups, you can do so by creating your own derived storage. For the purposes of the example in this section, let us say we want 2 array handles to behave as one array with the contents interlaced together. That is, the first w items are the same as the first w of the first array, the second w items are the same as the first w of the second array, the third w items are the same as the second w of the first array, the fourth w items are the same as the second w of the second array, and so on. We could of course actually copy the data, but we can also do it in place.

36.4.1 Array Portal

The first step to creating a derived storage is to build an array portal that will take portals from arrays being derived. The portal must work in both the control and execution environment.

Because the intention of our custom `ArrayHandle` is to augment the behavior of other `ArrayHandle`, it is typical for an `ArrayPortal` of this nature to reference other `ArrayPortal`s.

Example 36.14: Derived array portal for concatenated arrays.

```
template<typename PortalType1, typename PortalType2>
class ArrayPortalInterlace
{
  PortalType1 Portal1;
  PortalType2 Portal2;
  vtkm::Id Width;

public:
  using ValueType = typename PortalType1::ValueType;

  VTKM_EXEC_CONT ArrayPortalInterlace()
    : Portal1()
    , Portal2()
    , Width(1)
  {
  }

  VTKM_EXEC_CONT ArrayPortalInterlace(const PortalType1& portal1,
                                      const PortalType2& portal2,
                                      vtkm::Id width)
    : Portal1(portal1)
    , Portal2(portal2)
    , Width(width)
  {
  }

  /// Copy constructor for any other ArrayPortalInterlace with a portal type
  /// that can be copied to this portal type. This allows us to do any type
  /// casting that the portals do (like the non-const to const cast).
  template<typename OtherP1, typename OtherP2>
  VTKM_EXEC_CONT ArrayPortalInterlace(
    const ArrayPortalInterlace<OtherP1, OtherP2>& src)
    : Portal1(src.GetPortal1())
    , Portal2(src.GetPortal2())
    , Width(src.GetWidth())
  {
  }

  VTKM_EXEC_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
  }

  VTKM_EXEC_CONT
  ValueType Get(vtkm::Id index) const
  {
    vtkm::Id interleaveGroup = index / (this->Width * 2);
    vtkm::Id interleaveIndex = index % (this->Width * 2);
    if (interleaveIndex < this->Width)
    {
      return this->Portal1.Get(interleaveIndex + (interleaveGroup * this->Width));
    }
    else
```

```
    {
      return this->Portal2.Get((interleaveIndex - this->Width) +
                               (interleaveGroup * this->Width));
    }
  }

  // The template is a trick to use SFINAE semantics to only define this Set
  // method if both Portal1 and Portal2 define a Set method.
  template<
    typename P1 = PortalType1,
    typename P2 = PortalType2,
    typename =
      typename std::enable_if<vtkm::internal::PortalSupportsSets<P1>::value>::type,
    typename =
      typename std::enable_if<vtkm::internal::PortalSupportsSets<P2>::value>::type>
  VTKM_EXEC_CONT void Set(vtkm::Id index, const ValueType& value) const
  {
    vtkm::Id interleaveGroup = index / (this->Width * 2);
    vtkm::Id interleaveIndex = index % (this->Width * 2);
    if (interleaveIndex < this->Width)
    {
      this->Portal1.Set(interleaveIndex + (interleaveGroup * this->Width), value);
    }
    else
    {
      this->Portal2.Set(
        (interleaveIndex - this->Width) + (interleaveGroup * this->Width), value);
    }
  }

  VTKM_EXEC_CONT const PortalType1& GetPortal1() const { return this->Portal1; }
  VTKM_EXEC_CONT const PortalType2& GetPortal2() const { return this->Portal2; }

  VTKM_EXEC_CONT vtkm::Id GetWidth() const { return this->Width; }
};
```

36.4.2 Storage

The next step in creating a custom storage is to define a tag for the adapter. We shall call ours `StorageTagInterlace` and it will be templated on the two array handle types that we are deriving. Then, we need to create a specialization of the templated `vtkm::cont::internal::Storage` class. The `ArrayHandle` will instantiate an object using the array container tag we give it, and we define our own specialization so that it runs our interface into the code.

`vtkm::cont::internal::Storage` has two template arguments: the base type of the array and the storage tag.

Example 36.15: Prototype for `vtkm::cont::internal::Storage`.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, class StorageTag>
class Storage;

}
}
} // namespace vtkm::cont::internal
```

The vtkm::cont::internal::Storage must define the following items.

ReadPortalType The type of an array portal that can be used to access the underlying data. This array portal needs only be read-only. That is, the Set method is optional.

WritePortalType The type of an array portal that can be used to access the underlying data. This array portal should be both read and write capable. If the storage is intended to be read-only, then WritePortalType should be left out and VTKM_STORAGE_NO_WRITE_PORTAL should be declared in the storage class instead.

CreateBuffers The actual data of an ArrayHandle is stored in a std::vector of vtkm::cont::internal::Buffer objects. The Storage builds the array of Buffer objects in the CreateBuffers method. A derived array usually contains all the Buffers from the ArrayHandles it is referencing plus an additional Buffer for its own metadata. See Section 36.5.1 for details on Buffer.

GetNumberOfValues This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects and returns the number of values in the represented array. The number of values is either derived from the size of the Buffers, the size of the arrays being referenced, or stored in the metadata.

ResizeBuffers This is a static method that takes a number of values for the arrays, a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::CopyFlag, and a vtkm::cont::Token. It then resizes the memory in the Buffer objects to match the requested number of values. If the storage cannot be resized after it is created, then this method should be left out and the VTKM_STORAGE_NO_RESIZE macro should be declared in the storage class instead.

CreateReadPortal This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::cont::DeviceAdapterId, and a vtkm::cont::Token. It returns a ReadPortalType that can be used to read the data on the given device.

CreateWritePortal This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::cont::DeviceAdapterId, and a vtkm::cont::Token. It returns a WritePortalType that can be used to read the data on the given device. If the storage is intended to be read-only, then CreateWritePortal should be left out and VTKM_STORAGE_NO_WRITE_PORTAL should be declared in the storage class instead.

The CreateBuffers method must have a form that takes no arguments so that the ArrayHandle can initialize itself. Although it is not necessary, it is often convenient to have overloads of CreateBuffers to create the std::vector of vtkm::cont::internal::Buffer objects used for the storage based on some initial data (such as arrays it is based on). The vtkm::cont::internal::CreateBuffers function is often helpful for this task. It will push all of its arguments into a std::vector. Arguments that are ArrayHandles or std::vectors of Buffers will have their containing Buffers add. Arguments that do not contain any Buffer object will be added to the metadata of an empty Buffer object.

If the storage is working directly with data in memory, then the function of these items can be implemented through the vtkm::cont::internal::Buffer objects. However, if the storage is modifying the functionality of other array types, then the implementation is mostly performed using the storage for those respective array types. The following example shows the implementation of Storage for our example derived array type.

Example 36.16: Storage for derived container of interlaced arrays.

```
1 template<typename StorageType1, typename StorageType2>
2 struct StorageTagInterlace
3 {
4 };
5
6 namespace vtkm
7 {
```

```
8  namespace cont
9  {
10 namespace internal
11 {
12
13 template<typename T, typename StorageTag1, typename StorageTag2>
14 class Storage<T, StorageTagInterlace<StorageTag1, StorageTag2>>
15 {
16   // We will be deriving the behavior of two other arrays, so we will be
17   // using the Storage objects for those arrays to implement ours.
18   using SourceStorage1 = vtkm::cont::internal::Storage<T, StorageTag1>;
19   using SourceStorage2 = vtkm::cont::internal::Storage<T, StorageTag2>;
20
21   // Convenience aliases for the types of the ArrayHandles being modified.
22   using Array1 = vtkm::cont::ArrayHandle<T, StorageTag1>;
23   using Array2 = vtkm::cont::ArrayHandle<T, StorageTag2>;
24
25   // A structure holding the metadata needed to implement interlaced array
26   // storage.
27   struct Info
28   {
29     vtkm::Id Width;
30     std::size_t BufferOffset1;
31     std::size_t BufferOffset2;
32   };
33
34   // All storage objects store the actual data in a std::vector of Buffer objects.
35   // We will use the first object to store the metadata.
36   // The next objects will be for the first array, the remaining for the
37   // second array. These functions make it convenient to access these Buffers.
38   VTKM_CONT static std::vector<vtkm::cont::internal::Buffer> Buffers1(
39     const std::vector<vtkm::cont::internal::Buffer>& allbuffers)
40   {
41     Info info = allbuffers[0].GetMetaData<Info>();
42     return std::vector<vtkm::cont::internal::Buffer>(
43       allbuffers.begin() + info.BufferOffset1,
44       allbuffers.begin() + info.BufferOffset2);
45   }
46   VTKM_CONT static std::vector<vtkm::cont::internal::Buffer> Buffers2(
47     const std::vector<vtkm::cont::internal::Buffer>& allbuffers)
48   {
49     Info info = allbuffers[0].GetMetaData<Info>();
50     return std::vector<vtkm::cont::internal::Buffer>(
51       allbuffers.begin() + info.BufferOffset2, allbuffers.end());
52   }
53
54 public:
55   using ReadPortalType =
56     ArrayPortalInterlace<typename SourceStorage1::ReadPortalType,
57                          typename SourceStorage2::ReadPortalType>;
58   using WritePortalType =
59     ArrayPortalInterlace<typename SourceStorage1::WritePortalType,
60                          typename SourceStorage2::WritePortalType>;
61
62   // Not necessary for Storage, but useful.
63   VTKM_CONT static vtkm::Id GetWidth(
64     const std::vector<vtkm::cont::internal::Buffer>& buffers)
65   {
66     return buffers[0].GetMetaData<Info>().Width;
67   }
68
69   // Note that the default parameters create an overload that takes no arguments,
70   // which is necessary for all Storage objects.
71   VTKM_CONT static auto CreateBuffers(const Array1& array1 = Array1{},
```

```
72                                      const Array2& array2 = Array2{},
73                                      vtkm::Id width = 1)
74    -> decltype(vtkm::cont::internal::CreateBuffers())
75  {
76    Info info;
77    info.Width = width;
78    info.BufferOffset1 = 1;
79    info.BufferOffset2 = info.BufferOffset1 + array1.GetBuffers().size();
80    return vtkm::cont::internal::CreateBuffers(info, array1, array2);
81  }
82
83  VTKM_CONT static vtkm::Id GetNumberOfValues(
84    const std::vector<vtkm::cont::internal::Buffer>& buffers)
85  {
86    return (SourceStorage1::GetNumberOfValues(Buffers1(buffers)) +
87            SourceStorage2::GetNumberOfValues(Buffers2(buffers)));
88  }
89
90  VTKM_CONT static void ResizeBuffers(
91    vtkm::Id numValues,
92    const std::vector<vtkm::cont::internal::Buffer>& buffers,
93    vtkm::CopyFlag preserve,
94    vtkm::cont::Token& token)
95  {
96    vtkm::Id width = GetWidth(buffers);
97    vtkm::Id numInterleaveGroups = (numValues / (width * 2));
98    vtkm::Id remainder = numValues - (numInterleaveGroups * width * 2);
99    if (remainder < width)
100   {
101     SourceStorage1::ResizeBuffers((numInterleaveGroups * width) + remainder,
102                                   Buffers1(buffers),
103                                   preserve,
104                                   token);
105     SourceStorage2::ResizeBuffers(
106       (numInterleaveGroups * width), Buffers2(buffers), preserve, token);
107   }
108   else
109   {
110     SourceStorage1::ResizeBuffers(
111       (numInterleaveGroups + 1) * width, Buffers1(buffers), preserve, token);
112     SourceStorage2::ResizeBuffers((numInterleaveGroups * width) + remainder -
113                                     width,
114                                   Buffers2(buffers),
115                                   preserve,
116                                   token);
117   }
118 }
119
120 VTKM_CONT static ReadPortalType CreateReadPortal(
121   const std::vector<vtkm::cont::internal::Buffer>& buffers,
122   vtkm::cont::DeviceAdapterId device,
123   vtkm::cont::Token& token)
124 {
125   return ReadPortalType(
126     SourceStorage1::CreateReadPortal(Buffers1(buffers), device, token),
127     SourceStorage2::CreateReadPortal(Buffers2(buffers), device, token),
128     GetWidth(buffers));
129 }
130
131 VTKM_CONT static WritePortalType CreateWritePortal(
132   const std::vector<vtkm::cont::internal::Buffer>& buffers,
133   vtkm::cont::DeviceAdapterId device,
134   vtkm::cont::Token& token)
135 {
```

```
136       return WritePortalType(
137         SourceStorage1::CreateWritePortal(Buffers1(buffers), device, token),
138         SourceStorage2::CreateWritePortal(Buffers2(buffers), device, token),
139         GetWidth(buffers));
140     }
141
142     // These functions are not necessary for a Storage, but they are helpful for
143     // getting back the original arrays being derived.
144     VTKM_CONT static Array1 GetArray1(
145       const std::vector<vtkm::cont::internal::Buffer>& buffers)
146     {
147       return Array1(Buffers1(buffers));
148     }
149     VTKM_CONT static Array2 GetArray2(
150       const std::vector<vtkm::cont::internal::Buffer>& buffers)
151     {
152       return Array2(Buffers2(buffers));
153     }
154 };
155
156 } // namespace internal
157 } // namespace cont
158 } // namespace vtkm
```

36.4.3 Subclass

The final step to make a derived storage is to create a mechanism to construct an `ArrayHandle` with a storage derived from the desired arrays. This can be done by creating a trivial subclass of `vtkm::cont::ArrayHandle` that simply constructs the array handle to the state of an existing storage. It uses a protected constructor of `vtkm::cont::ArrayHandle` that accepts a constructed storage.

Example 36.17: `ArrayHandle` for derived storage of concatenated arrays.

```
1  template<typename ArrayHandleType1, typename ArrayHandleType2>
2  class ArrayHandleInterlace
3    : public vtkm::cont::ArrayHandle<
4        typename ArrayHandleType1::ValueType,
5        StorageTagInterlace<typename ArrayHandleType1::StorageTag,
6                            typename ArrayHandleType2::StorageTag>>
7  {
8    VTKM_IS_ARRAY_HANDLE(ArrayHandleType1);
9    VTKM_IS_ARRAY_HANDLE(ArrayHandleType2);
10
11 public:
12   VTKM_ARRAY_HANDLE_SUBCLASS(
13     ArrayHandleInterlace,
14     (ArrayHandleInterlace<ArrayHandleType1, ArrayHandleType2>),
15     (vtkm::cont::ArrayHandle<
16       typename ArrayHandleType1::ValueType,
17       StorageTagInterlace<typename ArrayHandleType1::StorageTag,
18                           typename ArrayHandleType2::StorageTag>>));
19
20 public:
21   VTKM_CONT
22   ArrayHandleInterlace(const ArrayHandleType1& array1,
23                        const ArrayHandleType2& array2,
24                        vtkm::Id width = 1)
25     : Superclass(StorageType::CreateBuffers(array1, array2, width))
26   {
27   }
28
29   // These extra methods may be appriciated by users.
```

```
30    ArrayHandleType1 GetArray1() const
31    {
32      return StorageType::GetArray1(this->GetBuffers());
33    }
34    ArrayHandleType2 GetArray2() const
35    {
36      return StorageType::GetArray2(this->GetBuffers());
37    }
38  };
```

Subclasses of `ArrayHandle` provide constructors that establish the state of the array handle. All array handle subclasses must also use either the `VTKM_ARRAY_HANDLE_SUBCLASS` macro or the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. The difference between these two macros is that `VTKM_ARRAY_HANDLE_SUBCLASS` is used in templated classes whereas `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is used in non-templated classes. `VTKM_ARRAY_HANDLE_SUBCLASS` takes three arguments. The first is the classname of the subclass being defined (without any template arguments). The second argument is the fully resolved template with the template arguments given. The third argument is the full name of the superclass, again with the template arguments specified. The second and third arguments *must* be encased in parentheses. The `VTKM_ARRAY_HANDLE_SUBCLASS_NT` is similar but only has 2 arguments: the derived class' name and the superclass.

Both of these macros define the following convenience types:

`Superclass` The fully resolved type of the superclass (given as the last argument to `VTKM_ARRAY_HANDLE_SUBCLASS` or `VTKM_ARRAY_HANDLE_SUBCLASS_NT`).

`ValueType` The value type of the `ArrayHandle` (same as `Superclass::ValueType`).

`StorageTag` The storage tag of the `ArrayHandle` (same as `Superclass::StorageTag`).

`StorageType` The resolved template of `vtkm::cont::internal::Storage`. This is useful for interacting with the `Storage` object appropriate for your `ArrayHandle`.

`ReadPortalType` The type of read portals for the `ArrayHandle` (same as `Superclass::ReadPortalType`).

`WritePortalType` The type of write portals for the `ArrayHandle` (same as `Superclass::WritePortalType`).

The macros also provide a set of constructors and operators expected of all `ArrayHandle` classes.

The `ArrayHandle` subclass in Example 36.17 is templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS` macro. (The other macro is described in Section 36.1 on page 316). This macro takes three parameters. The first parameter is the name of the subclass where the macro is defined, the second parameter is the type of the subclass including the full template specification, and the third parameter is the immediate superclass including the full template specification. The second and third parameters of the macro must be enclosed in parentheses so that the C pre-processor correctly handles commas in the template specification.

It is also customary to create helper functions for creating `ArrayHandle`s. This makes it simpler than creating matching template interfaces.

Example 36.18: Helper function for creating a custom derived `ArrayHandle`.

```
1  template<typename ArrayHandle1, typename ArrayHandle2>
2  VTKM_CONT ArrayHandleInterlace<ArrayHandle1, ArrayHandle2> make_ArrayHandleInterlace(
3    const ArrayHandle1& array1,
4    const ArrayHandle2& array2,
5    vtkm::Id width = 1)
6  {
7    VTKM_IS_ARRAY_HANDLE(ArrayHandle1);
8    VTKM_IS_ARRAY_HANDLE(ArrayHandle2);
9
```

```
10   return ArrayHandleInterlace<ArrayHandle1, ArrayHandle2>(array1, array2, width);
11 }
```

vtkm::cont::ArrayHandleCompositeVector is an example of a derived array handle provided by VTK-m. It references some fixed number of other arrays, pulls a specified component out of each, and produces a new component that is a tuple of these retrieved components.

36.5 Adapting Data Structures

The intention of the storage parameter for vtkm::cont::ArrayHandle is to implement the strategy design pattern to enable VTK-m to interface directly with the data of any third party code source. VTK-m is designed to work with data originating in other libraries or applications. By creating a new type of storage, VTK-m can be entirely adapted to new kinds of data structures.

Did you know?

VTK-m comes with several types of ArrayHandle *that can adapt memory in different ways. In practice, it is rare to have to write a custom* ArrayHandle *to adapt to a data structure, and this example is particularly contrived. However, we document it here for completeness.*

Common Errors

Keep in mind that memory layout used can have an effect on the running time of algorithms in VTK-m. Different data layouts and memory access can change cache performance and introduce memory affinity problems. The example code given in this section will likely have poorer cache performance than the basic storage provided by VTK-m. However, that might be an acceptable penalty to avoid data copies.

In this section we demonstrate the steps required to adapt the array handle to a data structure provided by a third party. For the purposes of the example, let us say that some fictitious library named "foo" has a simple structure named `FooAttributes` that holds the field values for a particular part of a mesh, and then maintain the field values for all locations in a mesh in a `FooFields` object.

Example 36.19: Fictitious field storage used in custom array storage examples.

```
1  struct FooAttributes
2  {
3    float Pressure;
4    float Temperature;
5    float Velocity[3];
6    // And so on...
7  };
8
9  class FooFields
10 {
11 public:
12   FooAttributes* GetAttributesArray();
13
14   std::size_t GetSize() const;
15
16   void Resize(std::size_t numberOfElements);
17 };
```

There are few restrictions on the structure of the data. The only real restriction is that you must be able to get the data in buffers of raw C pointers containing trivially copiable objects. In this example, we note that `FooFields` can return an array of `FooAttributes`, which satisfies this requirement.

VTK-m expects separate arrays for each of the fields rather than a single array containing a structure holding all of the fields. However, rather than copy each field to its own array, we can create a storage for each field that points directly to the data in a `FooFields` object.

36.5.1 Buffer Objects

VTK-m manages data across devices using the `vtkm::cont::internal::Buffer` object. As its name implies, `Buffer` manages a buffer in memory. This block of bytes can be allocated on different devices and the data it contains will be transferred across them. The `Buffer` object operates by specifying its size and then requesting pointers to the reserved memory on different devices.

`Buffer` contains the following methods.

`GetNumberOfBytes` Returns the number of bytes held by the buffer. `Buffer` actually allocates memory lazily, so there might not actually be any memory allocated anywhere. It is also possible that memory is simultaneously allocated on multiple devices. The number of bytes is returned as a `vtkm::BufferSizeType`, which might be a larger integer than `vtkm::Id`.

`SetNumberOfBytes` Changes the size of the buffer. `SetNumberOfBytes` has three arguments. The first argument is the number of bytes to allocate. The second argument is a `vtkm::CopyFlag` that indicates whether any existing data in the buffer should be preserved. The third argument is a `vtkm::cont::Token` that ensures that the resize will not interfere with other operations happening on the `Buffer`'s data.

`IsAllocatedOnHost` Returns true if the `Buffer` has memory allocated on the host (for the control environment).

`IsAllocatedOnDevice` Returns true if the `Buffer` has memory allocated on the device specified by a given device adapter tag. If `vtkm::cont::DeviceAdapterTagAny` is given as the device, then this returns true if the `Buffer` is allocated on any device. If `vtkm::cont::DeviceAdapterTagUnknown` is given as the device, then this returns true if the `Buffer` is allocated on the host (same as `IsAllocatedOnHost`).

`ReadPointerHost` Returns a readable host (control environment) pointer to the buffer. Memory will be allocated and data will be copied as necessary. A `vtkm::cont::Token` object is passed to `ReadPointerHost`, and the memory at the pointer will be valid as long as the `Token` is still in scope. Any write operation to this buffer will be blocked until then.

`ReadPointerDevice` Returns a readable device pointer to the buffer. The first argument to `ReadPointerDevice` is a `vtkm::cont::DeviceAdapterId`, and the pointer returned will only be valid for this device. If the device is `vtkm::cont::DeviceAdapterTagUnknown`, then this method has the same behavior as `ReadPointerHost`. Memory will be allocated and data will be copied as necessary. A `vtkm::cont::Token` object is passed to `ReadPointerDevice`, and the memory at the pointer will be valid as long as the `Token` is still in scope. Any write operation to this buffer will be blocked until then.

`WritePointerHost` Returns a writable host (control environment) pointer to the buffer. Memory will be allocated and data will be copied as necessary. A `vtkm::cont::Token` object is passed to `WritePointerHost`, and the memory at the pointer will be valid as long as the `Token` is still in scope. Any read or write operation to this buffer will be blocked until then.

`WritePointerDevice` Returns a writable device pointer to the buffer. The first argument to `WritePointerDevice` is a `vtkm::cont::DeviceAdapterId`, and the pointer returned will only be valid for this device. If the

device is vtkm::cont::DeviceAdapterTagUnknown, then this method has the same behavior as WritePointerHost. Memory will be allocated and data will be copied as necessary. A vtkm::cont::Token object is passed to WritePointerDevice, and the memory at the pointer will be valid as long as the Token is still in scope. Any read or write operation to this buffer will be blocked until then.

DeepCopyFrom Copies the data from the provided buffer into this buffer. If a device is given, then the copy will be preferred for that device.

ReleaseDeviceResources Unallocates the buffer from all devices. This method preserves the data on the host even if the data must be transferred there.

SetMetaData Takes an arbitrary object and copies it to the metadata of this buffer. Any existing metadata is deleted. Any object can be used as metadata as long as the object has a default constructor and is copiable. Holding metadata in a Buffer is optional, but it can be helpful for storing additional information or objects that cannot be implied by the buffer itself.

GetMetaData Gets the metadata for the buffer. When you call this method, you have to specify a template parameter for the type of the metadata. If the metadata has not yet been set in this buffer, a new metadata object is created, set to this buffer, and returned. If metadata of a different type has already been set, then an exception is thrown. The returned value is a reference that can be manipulated to alter the metadata of the buffer.

HasMetaData Returns whether the Buffer holds metadata.

MetaDataIsType Determines if the metadata for the buffer is set to a particular type. Specify the type of metadata as a template argument.

In addition to using Buffer to allocate data on devices and the host, you can wrap a Buffer around data that is already allocated. This is done by using the vtkm::cont::internal::MakeBuffer function. This method takes 6 arguments: the vtkm::cont::DeviceAdapterId of where the memory is allocated (vtkm::cont::DeviceAdapterTagUnknown if on the host), the pointer to the memory buffer, a pointer to a container managing the buffer, the size of the buffer in bytes, a function used to delete the buffer, and a function used to deallocate the buffer. An example of using MakeBuffer is given later.

36.5.2 Array Portal

The first step in creating an adapter storage is to create an array portal to the data. This is described in more detail in Section 27.1 and is generally straightforward for simple containers like this, which simple set and get values in a C array. Here is an example implementation for our FooFields container.

Example 36.20: Array portal to adapt a third-party container to VTK-m.

```
1  namespace vtkm
2  {
3  namespace internal
4  {
5
6  // Note: FooAttributesPointer expected to be either FooAttributes* or
7  // const FooAttributes*
8  template<typename FooAttributesPointer>
9  class ArrayPortalFooPressure
10 {
11   FooAttributesPointer AttributesArray = nullptr;
12   vtkm::Id NumberOfValues = 0;
13
14 public:
```

```
  using ValueType = float;

  ArrayPortalFooPressure() = default;

  VTKM_CONT ArrayPortalFooPressure(FooAttributesPointer array,
                                   vtkm::Id numberOfValues)
    : AttributesArray(array)
    , NumberOfValues(numberOfValues)
  {
  }

  VTKM_EXEC_CONT vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }

  VTKM_EXEC_CONT ValueType Get(vtkm::Id index) const
  {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    return this->AttributesArray[index].Pressure;
  }

  // This template is a trick to not define Set if FooAttributesPointer
  // is const. That saves us from having to create separate implementations
  // of this portal for the read and write versions.
  template<typename T = FooAttributesPointer,
           typename = typename std::enable_if<
             !std::is_const<std::remove_pointer<T>::value>::type>
  VTKM_EXEC_CONT void Set(vtkm::Id index, ValueType value) const
  {
    VTKM_ASSERT(index >= 0);
    VTKM_ASSERT(index < this->GetNumberOfValues());
    this->AttributesArray[index].Pressure = value;
  }
};

}
} // namespace vtkm::internal
```

36.5.3 Storage

The next step in creating an adapter storage is to define a tag for the adapter. We shall call ours `StorageTagFooPressure`. Then, we need to create a specialization of the templated `vtkm::cont::internal::Storage` class. The `ArrayHandle` will instantiate an object using the array container tag we give it, and we define our own specialization so that it runs our interface into the code.

`vtkm::cont::internal::Storage` has two template arguments: the base type of the array and the storage tag.

Example 36.21: Prototype for `vtkm::cont::internal::Storage`.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T, class StorageTag>
class Storage;

}
}
} // namespace vtkm::cont::internal
```

The vtkm::cont::internal::Storage must define the following items.

ReadPortalType The type of an array portal that can be used to access the underlying data. This array portal needs only be read-only. That is, the Set method is optional.

WritePortalType The type of an array portal that can be used to access the underlying data. This array portal should be both read and write capable. If the storage is intended to be read-only, then WritePortalType should be left out and VTKM_STORAGE_NO_WRITE_PORTAL should be declared in the storage class instead.

CreateBuffers The actual data of an ArrayHandle is stored in a std::vector of vtkm::cont::internal::Buffer objects. The Storage builds the array of Buffer objects in the CreateBuffers method. A derived array usually contains all the Buffers from the ArrayHandles it is referencing plus an additional Buffer for its own metadata. See Section 36.5.1 for details on Buffer.

GetNumberOfValues This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects and returns the number of values in the represented array. The number of values is either derived from the size of the Buffers, the size of the arrays being referenced, or stored in the metadata.

ResizeBuffers This is a static method that takes a number of values for the arrays, a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::CopyFlag, and a vtkm::cont::Token. It then resizes the memory in the Buffer objects to match the requested number of values. If the storage cannot be resized after it is created, then this method should be left out and the VTKM_STORAGE_NO_RESIZE macro should be declared in the storage class instead.

CreateReadPortal This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::cont::DeviceAdapterId, and a vtkm::cont::Token. It returns a ReadPortalType that can be used to read the data on the given device.

CreateWritePortal This is a static method that takes a std::vector of vtkm::cont::internal::Buffer objects, a vtkm::cont::DeviceAdapterId, and a vtkm::cont::Token. It returns a WritePortalType that can be used to read the data on the given device. If the storage is intended to be read-only, then CreateWritePortal should be left out and VTKM_STORAGE_NO_WRITE_PORTAL should be declared in the storage class instead.

The following provides an example implementation of our adapter to FooFields. It relies on the ArrayPortalFooPressure provided in Example 36.20.

Example 36.22: Storage to adapt a third-party container to VTK-m.

```
1  // Includes or definition for ArrayPortalFooPressure
2
3  namespace vtkm
4  {
5  namespace cont
6  {
7  namespace internal
8  {
9
10 struct StorageTagFooPressure
11 {
12 };
13
14 template<>
15 class Storage<float, StorageTagFooPressure>
16 {
17 public:
18   using ReadPortalType =
19     vtkm::internal::ArrayPortalFooPressure<const FooAttributes*>;
```

```
20    using WritePortalType = vtkm::internal::ArrayPortalFooPressure<FooAttributes*>;
21
22    // Note that the default parameters create an overload that takes no arguments,
23    // which is necessary for all Storage objects.
24    VTKM_CONT static std::vector<vtkm::cont::internal::Buffer> CreateBuffers(
25      const FooFields& fields = FooFields{})
26    {
27      FooFields* fieldsCopy = new FooFields(fields);
28      vtkm::cont::internal::Buffer memoryManager =
29        vtkm::cont::internal::MakeBuffer(
30          vtkm::cont::DeviceAdapterTagUndefined{},
31          fieldsCopy->GetAttributesArray(),
32          fieldsCopy,
33          static_cast<vtkm::BufferSizeType>(fieldsCopy->GetSize() *
34                                            sizeof(FooAttributes)),
35          FooFieldsDeleter,
36          FooFieldsReallocater);
37      return std::vector<vtkm::cont::internal::Buffer>(1, memoryManager);
38    }
39
40    VTKM_CONT static vtkm::Id GetNumberOfValues(
41      const std::vector<vtkm::cont::internal::Buffer>& buffers)
42    {
43      return static_cast<vtkm::Id>(
44        buffers[0].GetNumberOfBytes() /
45        static_cast<vtkm::BufferSizeType>(sizeof(FooAttributes)));
46    }
47
48    VTKM_CONT static void ResizeBuffers(
49      vtkm::Id numValues,
50      const std::vector<vtkm::cont::internal::Buffer>& buffers,
51      vtkm::CopyFlag preserve,
52      vtkm::cont::Token& token)
53    {
54      buffers[0].SetNumberOfBytes(
55        vtkm::internal::NumberOfValuesToNumberOfBytes<FooAttributes>(numValues),
56        preserve,
57        token);
58    }
59
60    VTKM_CONT static ReadPortalType CreateReadPortal(
61      const std::vector<vtkm::cont::internal::Buffer>& buffers,
62      vtkm::cont::DeviceAdapterId device,
63      vtkm::cont::Token& token)
64    {
65      return ReadPortalType(reinterpret_cast<const FooAttributes*>(
66                              buffers[0].ReadPointerDevice(device, token)),
67                            GetNumberOfValues(buffers));
68    }
69
70    VTKM_CONT static WritePortalType CreateWritePortal(
71      const std::vector<vtkm::cont::internal::Buffer>& buffers,
72      vtkm::cont::DeviceAdapterId device,
73      vtkm::cont::Token& token)
74    {
75      return WritePortalType(
76        reinterpret_cast<FooAttributes*>(buffers[0].WritePointerDevice(device, token)),
77        GetNumberOfValues(buffers));
78    }
79  };
80
81  } // namespace internal
82  } // namespace cont
83  } // namespace vtkm
```

One interesting feature of this example is the `CreateBuffers` method on line 24 that wraps the `Buffer` around an existing `FooFields` object. This allows VTK-m to read and write directly to and from the memory shared with the `FooFields`. To make this work, `CreateBuffers` creates a new `vtkm::cont::internal::Buffer` object using the `vtkm::cont::internal::MakeBuffer` function (lines 29–36). The first 4 arguments are straightforward: device holding the memory, the actual memory, the object containing the memory, and the number of bytes in the buffer. The fifth and sixth arguments are function pointers that handle deleting and reallocating the buffer (in the control environment). These functions can be defined as follows.

Example 36.23: Memory handling functions to adapt a third-party data structure to `ArrayHandle`.

```
1  VTKM_CONT void FooFieldsDeleter(void* container)
2  {
3    FooFields* fields = reinterpret_cast<FooFields*>(container);
4    delete fields;
5  }
6
7  VTKM_CONT void FooFieldsReallocater(void*& memory,
8                                      void*& container,
9                                      vtkm::BufferSizeType oldSize,
10                                     vtkm::BufferSizeType newSize)
11 {
12   FooFields* fields = reinterpret_cast<FooFields*>(container);
13   VTKM_ASSERT(static_cast<std::size_t>(oldSize) == fields->GetSize());
14   fields->Resize(static_cast<std::size_t>(newSize) / sizeof(FooAttributes));
15   memory = fields->GetAttributesArray();
16 }
```

36.5.4 Subclass

The final step to make a storage adapter is to make a mechanism to construct an `ArrayHandle` that points to a particular storage. This can be done by creating a trivial subclass of `vtkm::cont::ArrayHandle` that simply constructs the array handle to the state of an existing container.

Example 36.24: Array handle to adapt a third-party container to VTK-m.

```
1  class ArrayHandleFooPressure
2    : public vtkm::cont::ArrayHandle<float,
3                                     vtkm::cont::internal::StorageTagFooPressure>
4  {
5  public:
6    VTKM_ARRAY_HANDLE_SUBCLASS_NT(
7      ArrayHandleFooPressure,
8      (vtkm::cont::ArrayHandle<float, vtkm::cont::internal::StorageTagFooPressure>));
9
10   VTKM_CONT ArrayHandleFooPressure(const FooFields& fields)
11     : Superclass(StorageType::CreateBuffers(fields))
12   {
13   }
14 };
```

With this new version of `ArrayHandle`, VTK-m can now read to and write from the `FooFields` structure directly.

Example 36.25: Using an `ArrayHandle` with custom container.

```
1  struct AirPressureWorklet : vtkm::worklet::WorkletMapField
2  {
3    using ControlSignature = void(FieldIn elevation, FieldOut airPressure);
4
5    VTKM_EXEC void operator()(vtkm::Vec3f position, float& airPressure) const
6    {
7      // Use linear interpolation to estimate atmospheric pressure based on
```

```
8     // elevation in meters (0 = sea level). Atmospheric pressure is 101325 Pa
9     // at sea level and drops about 12 Pa per meter.
10    airPressure =
11      vtkm::Lerp(101325.0f, 77325.0f, static_cast<float>(position[2] / 2000.0f));
12  }
13 };
14
15 VTKM_CONT
16 void GetElevationAirPressure(vtkm::cont::DataSet grid, const FooFields& fields)
17 {
18   // Make an array handle that points to the pressure values in the fields.
19   ArrayHandleFooPressure pressureHandle(fields);
20
21   vtkm::cont::Invoker invoke;
22   invoke(AirPressureWorklet{},
23          grid.GetCoordinateSystem().GetDataAsMultiplexer(),
24          pressureHandle);
25
26   // Make sure the values are flushed back to the control environment.
27   pressureHandle.SyncControlArray();
28
29   // Now the pressure field is in the fields container.
30 }
```

Common Errors

When using an `ArrayHandle` *in VTK-m some code may be executed in an execution environment with a different memory space. In these cases data written to an* `ArrayHandle` *with a custom storage will not be written directly to the storage system you defined. Rather, they will be written to a separate array in the execution environment. If you need to access data in your custom data structure, make sure you call* `SyncControlArray` *on the* `ArrayHandle`*, as is demonstrated in Example 36.25.*

Common Errors

One of the challenges of introducing a new storage for `ArrayHandle` *is that if the rest of the VTK-m does not recognize the new* `ArrayHandle`*, they will not be built with compile support for it. Thus, if you do make a custom data adapter for* `ArrayHandle`*, you will likely need to define a new set of default types that adds the new storage to* `VTKM_DEFAULT_STORAGE_LIST`*.*

CHAPTER

THIRTYSEVEN

DISTRIBUTED SYSTEMS

37.1 Introduction

As an HPC visualization toolkit, VTK-m is designed to be executed in distributed systems. This is a key requirement for resource expensive applications where a computational job can be partitioned into different tasks which are then assigned to different processes located in potentially different nodes. Those nodes as a composite will normally form a cluster of computers or a supercomputer. These computing tasks can communicate and coordinate with each other by the use of a *middle-ware* like library which in the case of VTK-m corresponds to the DIY library. Furthermore, for both launching the jobs and ultimately to communicate between tasks we delegate into MPI (Message Passing Interface).

Despite of the fact that only some of the VTK-m filters can run out-of-the-box as a distributed job, many other VTK-m components can run as a distributed job by manually partitioning the computational problem data and assigning to each of the tasks one of those partitions.

37.2 DIY

DIY is a block-parallel library for implementing scalable algorithms that can execute both *in-core* and *out-of-core*. The same program can be executed with one or more threads per MPI process, seamlessly combining distributed-memory message passing with shared-memory thread parallelism. The abstraction enabling these capabilities is block parallelism: blocks and their message queues are mapped onto processing elements, consisting of either MPI processes or threads, and are migrated between memory and storage by the DIY runtime. Complex communication patterns, including neighbor exchange, merge reduction, swap reduction, and *all-to-all* exchange are possible in DIY both *in-core* and *out-of-core*.

A full description of using DIY to perform distributed visualization is beyond the scope of this guide. For a full description, reference the documentation provided by DIY. The basic procedure of any DIY algorithm is to first define a set of blocks, assign them to the ranks of an MPI job, and define neighborhood relationships between them. The following example demonstrates defining a set of blocks, one per MPI rank.

Example 37.1: Communication setup of an example DIY application.

```
1  vtkmdiy::mpi::communicator comm;
2  vtkm::cont::EnvironmentTracker::SetCommunicator(comm);
3
4  auto nblocks = comm.size();
5  std::vector<int> gids;
6
7  vtkmdiy::RoundRobinAssigner assigner(comm.size(), nblocks);
8  assigner.local_gids(comm.rank(), gids);
```

```
9
10  // In our example nblocks == num_ranks, thus gids.size() == 1
11  auto gid = gids[0];
12
13  // The link will be eventually freed by DIY.
14  auto link = new vtkmdiy::Link;
15
16  // Connect each blocks with itself.
17  vtkmdiy::BlockID neighbor;
18  neighbor.gid = gid;
19  neighbor.proc = assigner.rank(neighbor.gid);
20  link->add_neighbor(neighbor);
```

Did you know?

When using DIY objects from inside VTK-m, use the objects in the mangled `vtkmdiy` *rather than* `diy`*. VTK-m uses this mangled namespace to prevent conflicts if it is used with another library or executable that uses a different version of DIY.*

Communication in DIY is managed by the `vtkmdiy::Master` object. References to the defined blocks are added to the `Master` object. You can then run an operation on each of these blocks using the `foreach` method, which is given a function to execute on each block. This function is provided with a proxy that enables communicating data with other nodes using a variety of communication patterns. These communications do not happen right away but rather are queued for later exchange. This exchange is done by calling the free function `vtkm::cont::DIYMasterExchange`. The following example, which builds on the previous one, finds the median value of an array on each rank and then finds the maximum median value. The blocks and communication used by these examples are outlined in Figure 37.1.

Example 37.2: Example DIY application which finds the maximum of the medians of different `ArrayHandle`.

```
1   vtkmdiy::Master master(comm);
2
3   struct MyBlock
4   {
5     vtkm::cont::ArrayHandle<vtkm::Int32> in;
6   };
7
8   MyBlock block{ inputArrayHandle };
9   master.add(gid, &block, link);
10
11  master.foreach (
12    [&](MyBlock* b, const vtkmdiy::Master::ProxyWithLink& cp)
13    {
14      vtkm::cont::Algorithm::Sort(b->in);
15      cp.enqueue(cp.link()->target(0), b->in);
16    });
17  vtkm::cont::DIYMasterExchange(master);
18  master.foreach (
19    [&](MyBlock* b, const vtkmdiy::Master::ProxyWithLink& cp)
20    {
21      cp.dequeue(cp.link()->target(0).gid, b->in);
22
23      auto median_idx = (b->in.GetNumberOfValues() / 2) - 1;
24      auto median = vtkm::cont::ArrayGetValue(median_idx, b->in);
25
26      cp.all_reduce(median, vtkmdiy::mpi::maximum<vtkm::Int32>());
27    });
28  vtkm::cont::DIYMasterExchange(master);
```

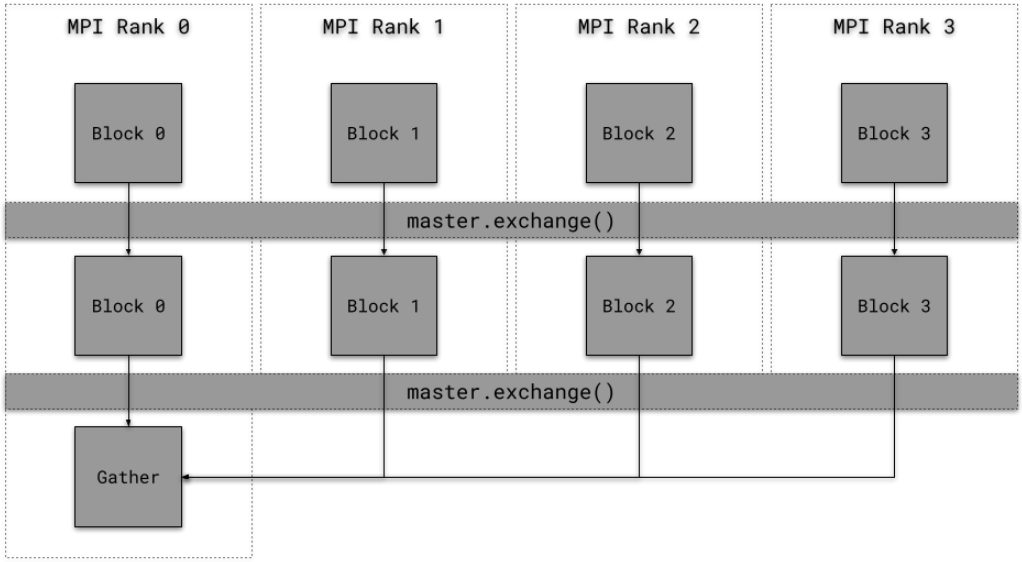


Figure 37.1: Communication topology of the example DIY application shown in the listings 37.1 and 37.2.

```
29
30   if (comm.rank() == 0)
31   {
32     std::cout << "Max(median): "
33               << master.proxy(master.loaded_block()).get<vtkm::Int32>() << std::endl;
34   }
```

Common Errors

Normally, the DIY exchange process is done by calling the `Master::exchange` *method. However, when using DIY with VTK-m, the exchange should be done instead by calling* `vtkm::cont::DIYMasterExchange`*. This function allows VTK-m to enable state to interface between DIY and VTK-m data (without otherwise affecting exchanges that happen outside of VTK-m).*

37.3 Object Serialization

When data are transferred among ranks, the format needs to be packed in a way the message layer understands. Objects that might have complex structure typically need to be converted to one or more buffers through serialization.

DIY provides *out-of-the-box* serialization of common C++ *stdlib* types such as `std::vector` and `std::string`, VTK-m also provides serialization for common VTK-m data types such as `vtkm::cont::ArrayHandle` and `vtkm::cont::DataSet`. This list is not exhaustive since VTK-m also provides DIY serialization to many other data types. For custom data types the user can specify how to serialize and deserialize the desired type by defining an additional template specialization for `struct vtkmdiy::Serialization`. An example of this can be found in the listing 37.3.

Example 37.3: Example DIY application which displays how to serialize custom data types in DIY.

```
struct TimedCoords
{
  vtkm::cont::ArrayHandle<vtkm::UInt64> TimeStamps;
  vtkm::cont::ArrayHandle<vtkm::Vec3i> Coordinates;
};

namespace vtkmdiy
{
template<>
struct Serialization<TimedCoords>
{
  static void save(BinaryBuffer& bb, const TimedCoords& p)
  {
    vtkmdiy::save(bb, p.TimeStamps);
    vtkmdiy::save(bb, p.Coordinates);
  }
  static void load(BinaryBuffer& bb, TimedCoords& p)
  {
    vtkmdiy::load(bb, p.TimeStamps);
    vtkmdiy::load(bb, p.Coordinates);
  }
};
}

void compute(vtkmdiy::Master& master, vtkmdiy::Link* link, int gid)
{
  TimedCoords timedCoords;
  master.add(gid, &timedCoords, link);

  master.foreach (
    [&](TimedCoords* tc, const vtkmdiy::Master::ProxyWithLink& cp)
    {
      *tc = ComputeLocalCoords(gid);
      cp.enqueue(cp.link()->target(0), *tc);
    });
  vtkm::cont::DIYMasterExchange(master);
  master.foreach (
    [&](TimedCoords* tc, const vtkmdiy::Master::ProxyWithLink& cp)
    {
      cp.dequeue(cp.link()->target(0).gid, *tc);
      auto expectedVec = ComputeLocalCoords(gid);
      if (*tc != expectedVec)
      {
        std::cerr << "ERROR: recieved incorrect vec values." << std::endl;
      }
    });
}
```

37.4 GPU-aware MPI

Modern HPC GPUs allow direct *GPU-to-GPU* communication. This provides GPUs with an efficient mechanism to directly send data stored in their device memory to the target GPU device memory. This is a significant departure from the traditional approach where the NIC is solely accessible from the CPU constraining us to a costly GPU communication pattern consisting in first copying the desired data from the device memory to host memory, transferring it over the network, and then again copying the received data from host memory to device memory.

Both major GPUs parallel platforms ROCM and CUDA provide an API which supports direct *GPU-to-GPU* communication. Nevertheless, to avoid vendor lock in VTK-m does not directly use these APIs. Instead, VTK-m delegates on MPI which implements an unified and standardized API for *GPU-to-GPU* communication. Consequently, VTK-m provides the user with the capability of using direct *GPU-to-GPU* communication during MPI (distributed) executions of VTK-m applications.

VTK-m can autonomously determine if *GPU-to-GPU* communication is possible. Consequently, it does not provide a specific API to control this type of communication. This decision on the type of communication is done at each call to the free function `vtkm::cont::DIYMasterExchange` (demonstrated in Example 37.2, line 17), which internally decorates the method `DIY::Master::exchange` so that it can perform this *GPU-to-GPU* communication if the situation allows it. Currently this is only possible with AMD GPUs that supports this feature such as the AMD MI250X which is used by both OLCF Crusher and OLCF Frontier.

This *GPU-aware* MPI feature can be enabled with the flag `VTKm_ENABLE_GPU_MPI=ON`. Lastly, enabling this feature in target supercomputers often requires additional setup which is dependent on the particular system, please refer to the target system documentation for further information.

CHAPTER

THIRTYEIGHT

REGRESSION TESTING

VTK-m has hundreds of regression tests built-in, to test the functionality of the entire VTK-m infrastructure on new platforms. In this chapter we will discuss how to run regression tests in VTK-m, as well as how to create new regression tests.

Did you know?

VTK-m's regression test infrastructure is enabled by default. If you don't need regression tests and are looking for a faster compile time, you can disable it using the CMake configuration variable described in Section 2.2.

38.1 Running Regression Testing

This section details how to run VTK-m's regression tests. First will explore how to use `ctest` to run these tests. `ctest` is the easiest option for running regression tests, as it sets a number of required arguments to the testing infrastructure automatically. Second, we will give an overview of how to run the regression tests without using ctest, and list the primary command line arguments for doing so.

38.1.1 Regression Testing Using ctest

The following code examples show how to run the regression tests in VTK-m using `ctest`. Example 38.1 shows how to run all of the enabled regression tests in VTK-m.

Example 38.1: Running all regression tests (Unix commands).

```
1 cd vtkm-build
2 ctest
```

You can get a list of all the available tests by giving `ctest` the `-N` option, which suppresses actually running the tests (see Example 38.2).

Example 38.2: List all available regression tests (Unix commands).

```
1 cd vtkm-build
2 ctest -N
```

Tests can be selected by using the `-R` option to `ctest`. The `-R` option is followed by a string or regular expression to match the names of tests to run (see Example 38.3).

Example 38.3: Running a single regression test (Unix commands).

```
cd vtkm-build
ctest -R SystemInformation
```

Verbose testing output can be selected by using the `-V` option to `ctest`. The `-V` options causes the tests to print the underlying run command used to launch each test, along with detailed test progression information (see Example 38.4).

Example 38.4: Running a single regression test with verbose output (Unix commands). The verbose output will first give the exact command used to run the regression test, along with detailed test progression information.

```
cd vtkm-build
ctest -R -V SystemInformation
```

Common Errors

Some of the regression tests in VTK-m use data files stored in git LFS. These files are automatically pulled when the VTK-m repository is cloned. However, if the device you are compiling on does not have git LFS installed, these unit tests will fail.

38.1.2 Regression Testing Without ctest

It is also possible to run VTK-m regression tests without using ctest. This can be accomplished by running individual unit test wrappers that are located in the <path/to/vtk-m/build>/bin directory. These tests require specific command line options in order for tests to run correctly.

Example 38.5 shows how to run a specific rendering test by passing in the location of the VTK-m data-dir and the baseline-dir

Example 38.5: Running a single regression test without calling ctest (Unix commands).

```
UnitTests_vtkm_rendering_testing \
  UnitTestMapperVolume \
  --data-dir=path/to/vtk-m/data \
  --baseline-dir=path/to/vtk-m/baseline
```

38.2 Creating Regression Tests

This section will detail the process and expectations for new regression tests in VTK-m.

38.2.1 How to Add Data to VTK-m

VTK-m uses Git LFS for all regression test data. In order to download or add test data to VTK-m you will need to have Git LFS installed. Once installed, you will add unit test data to the data directory in the VTK-m repository. Data in this directory is classified according to its type: structured or unstructured.

Example 38.6: Adding test data to the VTK-m repository (Unix commands).

```
cd vtkm-src-dir
cd data/data/<data type>
git add <file-name>
```

Part V

Core Development

CHAPTER

THIRTYNINE

TRY EXECUTE

Most operations in VTK-m do not require specifying on which device to run. For example, you may have noticed that when using `vtkm::cont::Invoker` to execute a worklet, you do not need to specify a device; it chooses a device for you. Internally, the `Invoker` has a mechanism to automatically select a device, try it, and fall back to other devices if the first one fails. We saw this at work in the implementation of filters in Chapter 22.

The `Invoker` is internally using a function named `vtkm::cont::TryExecute` to choose a device. This `TryExecute` function can be also be used in other instances where a specific device needs to be chosen.

`TryExecute` is a simple, generic mechanism to run an algorithm that requires a device adapter without directly specifying a device adapter. `vtkm::cont::TryExecute` is a templated function. The first argument is a functor object whose parenthesis operator takes a device adapter tag and returns a `bool` that is true if the call succeeds on the given device. If any further arguments are given to `TryExecute`, they are passed on to the functor. Thus, the parenthesis operator on the functor should take a device adapter tag as its first argument and any remaining arguments must match those passed to `TryExecute`.

To demonstrate the operation of `TryExecute`, consider an operation to find the average value of an array. Doing so with a given device adapter is a straightforward use of the reduction operator.

Example 39.1: A function to find the average value of an array in parallel.

```
1  template<typename T, typename Storage, typename Device>
2  VTKM_CONT T ArrayAverage(const vtkm::cont::ArrayHandle<T, Storage>& array, Device)
3  {
4    T sum = vtkm::cont::Algorithm::Reduce(array, T(0));
5    return sum / T(array.GetNumberOfValues());
6  }
```

The function in Example 39.1 requires a device adapter. We want to make an alternate version of this function that does not need a specific device adapter but rather finds one to use. To do this, we first make a functor as described earlier. It takes a device adapter tag as an argument, calls the version of the function shown in Example 39.1, and returns true when the operation succeeds. We then create a new version of the array average function that does not need a specific device adapter tag and calls `TryExecute` with the aforementioned functor.

Example 39.2: Using `TryExecute`.

```
1  namespace detail
2  {
3
4  struct ArrayAverageFunctor
5  {
6    template<typename Device, typename T, typename Storage>
7    VTKM_CONT bool operator()(Device,
8                              const vtkm::cont::ArrayHandle<T, Storage>& inArray,
9                              T& outValue) const
10   {
```

```
11      // Call the version of ArrayAverage that takes a DeviceAdapter.
12      outValue = ArrayAverage(inArray, Device());
13
14      return true;
15    }
16  };
17
18  } // namespace detail
19
20  template<typename T, typename Storage>
21  VTKM_CONT T ArrayAverage(const vtkm::cont::ArrayHandle<T, Storage>& array)
22  {
23    T outValue;
24
25    bool foundAverage =
26      vtkm::cont::TryExecute(detail::ArrayAverageFunctor{}, array, outValue);
27
28    if (!foundAverage)
29    {
30      throw vtkm::cont::ErrorExecution("Could not compute array average.");
31    }
32
33    return outValue;
34  }
```

Common Errors

When `TryExecute` *calls the operation of your functor, it will catch any exceptions that the functor might throw.* `TryExecute` *will interpret any thrown exception as a failure on that device and try another device. If all devices fail,* `TryExecute` *will return a false value rather than throw its own exception. This means if you want to have an exception thrown from a call to* `TryExecute`*, you will need to check the return value and throw the exception yourself.*

CHAPTER

FORTY

IMPLEMENTING DEVICE ADAPTERS

VTK-m comes with several implementations of device adapters so that it may be ported to a variety of platforms. It is also possible to provide new device adapters to support yet more devices, compilers, and libraries. A new device adapter provides a tag, a class to manage arrays in the execution environment, a collection of algorithms that run in the execution environment, and (optionally) a timer.

Most device adapters are associated with some type of device or library, and all source code related directly to that device is placed in a subdirectory of vtkm/cont. For example, files associated with CUDA are in vtkm/cont/cuda, files associated with the Intel Threading Building Blocks (TBB) are located in vtkm/cont/tbb, and files associated with OpenMP are in vtkm/cont/openmp. The documentation here assumes that you are adding a device adapter to the VTK-m source code and following these file conventions.

For the purposes of discussion in this section, we will give a simple example of implementing a device adapter using the `std::thread` class provided by C++11. We will call our device `Cxx11Thread` and place it in the directory vtkm/cont/cxx11.

By convention the implementation of device adapters within VTK-m are divided into 6 header files with the names DeviceAdapterTag*.h, DeviceAdapterRuntimeDetector*.h, DeviceAdapterMemoryManager*.h, RuntimeDeviceConfiguration*.h, and DeviceAdapterAlgorithm*.h, which are hidden in internal directories. The DeviceAdapter*.h that most code includes is a trivial header that simply includes these other 7 files. For our example `std::thread` device, we will create the base header at vtkm/cont/cxx11/DeviceAdapterCxx11Thread.h. The contents are the following (with minutia like include guards removed).

Example 40.1: Contents of the base header for a device adapter.

```
1 #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2 #include <vtkm/cont/cxx11/internal/DeviceAdapterRuntimeDetectorCxx11Thread.h>
3 #include <vtkm/cont/cxx11/internal/DeviceAdapterMemoryManagerCxx11Thread.h>
4 #include <vtkm/cont/cxx11/internal/RuntimeDeviceConfigurationCxx11Thread.h>
5 #include <vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h>
```

The reason VTK-m breaks up the code for its device adapters this way is that there is an interdependence between the implementation of each device adapter and the mechanism to pick a default device adapter. Breaking up the device adapter code in this way maintains an acyclic dependence among header files.

40.1 Tag

The device adapter tag, as described in Section 12.1 is a simple empty type that is used as a template parameter to identify the device adapter. Every device adapter implementation provides one. The device adapter tag is typically defined in an internal header file with a prefix of DeviceAdapterTag.

The device adapter tag should be created with the macro VTKM_VALID_DEVICE_ADAPTER. This adapter takes an abbreviated name that it will append to DeviceAdapterTag to make the tag structure. It will also create some support classes that allow VTK-m to introspect the device adapter. The macro also expects a unique integer identifier that is usually stored in a macro prefixed with VTKM_DEVICE_ADAPTER_. These identifiers for the device adapters provided by the core VTK-m are declared in vtkm/cont/internal/DeviceAdapterTag.h.

The following example gives the implementation of our custom device adapter, which by convention would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h header file.

Example 40.2: Implementation of a device adapter tag.

```
#include <vtkm/cont/DeviceAdapterTag.h>

// If this device adapter were to be contributed to VTK-m, then this macro
// declaration should be moved to DeviceAdapterTag.h and given a unique
// number. It also has te be less than VTK_MAX_DEVICE_ADAPTER_ID
#define VTKM_DEVICE_ADAPTER_CXX11_THREAD 6

VTKM_VALID_DEVICE_ADAPTER(Cxx11Thread, VTKM_DEVICE_ADAPTER_CXX11_THREAD);
```

This new device adapter tag needs to be added to vtkm::cont::DeviceAdapterListCommon, which is defined in vtkm/cont/DeviceAdapterList.h. Other components of VTK-m will use this list to write code for the device. If you do not add the device tag to this list, then the device will not be tried when things are invoked in the execution environment, and directly specifying execution on this device will likely fail.

Example 40.3: Modification of DeviceAdapterListCommon in DeviceAdapterList.h

```
using DeviceAdapterListCommon = vtkm::List<vtkm::cont::DeviceAdapterTagCuda,
                                            vtkm::cont::DeviceAdapterTagTBB,
                                            vtkm::cont::DeviceAdapterTagOpenMP,
                                            vtkm::cont::DeviceAdapterTagCxx11Thread,
                                            vtkm::cont::DeviceAdapterTagSerial>;
```

Did you know?

The order of device adapter tags in vtkm::cont::DeviceAdapterListCommon *matters. Devices will be tried in the order listed in this list. Thus, the most "preferred" devices should be listed first. In Example 40.3, our new C++11 thread device will be used before the serial device but after the other parallel devices.*

It is OK for vtkm::cont::DeviceAdapterListCommon *to contain device adapter tags for devices that are not being compiled for. These devices will be registered as inactive and be skipped.*

40.2 Runtime Detector

VTK-m defines a template named vtkm::cont::DeviceAdapterRuntimeDetector that provides the ability to detect whether a given device is available on the current system. DeviceAdapterRuntimeDetector has a single template argument that is the device adapter tag.

Example 40.4: Prototype for DeviceAdapterRuntimeDetector.

```
namespace vtkm
{
namespace cont
{

```

```
6 template<typename DeviceAdapterTag>
7 class DeviceAdapterRuntimeDetector;
8 }
9 } // namespace vtkm
```

All device adapter implementations must create a specialization of `DeviceAdapterRuntimeDetector`. They must contain a method named `DeviceAdapterRuntimeDetector::Exists` that returns a true or false value to indicate whether the device is available on the current runtime system. For our simple C++ threading example, the C++ threading is always available (even if only one such processing element exists) so our implementation simply returns true if the device has been compiled.

Example 40.5: Implementation of `DeviceAdapterRuntimeDetector` specialization

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5
6  template<>
7  class DeviceAdapterRuntimeDetector<vtkm::cont::DeviceAdapterTagCxx11Thread>
8  {
9  public:
10   VTKM_CONT bool Exists() const
11   {
12     return vtkm::cont::DeviceAdapterTagCxx11Thread::IsEnabled;
13   }
14 };
15
16 } // namespace cont
17 } // namespace vtkm
```

40.3 Memory Manager

VTK-m defines a template named `vtkm::cont::internal::DeviceAdapterMemoryManager` that provides the ability to allocate memory on the device and copy data. `DeviceAdapterMemoryManager` has a single template argument that is the device adapter tag.

Example 40.6: Prototype for `DeviceAdapterMemoryManager`.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename DeviceAdapterTag>
9  class DeviceAdapterMemoryManager;
10
11 }
12 }
13 } // namespace vtkm::cont::internal
```

All device adapter implementations must create a specialization of `DeviceAdapterMemoryManager`. This specialization of `DeviceAdapterMemoryManager` must inherit from `vtkm::cont::internal::DeviceAdapterMemoryManagerBase`. The `DeviceAdapterMemoryManager` allocates memory and returns it wrapped in a `vtkm::cont::internal::BufferInfo` object. The superclass provides the `DeviceAdapterMemoryManagerBase::ManageArray` method to take a raw pointer for the device (captured as a `void *`) along with some metadata and management functions and returns that pointer wrapped in a `BufferInfo` management object.

A specialization of DeviceAdapterMemoryManager must override the following pure virtual methods (which are defined in the DeviceAdapterMemoryManagerBase superclass).

`GetDevice` Return a `vtkm::cont::DeviceAdapterId` for the device that this memory manager allocates and deallocates for.

`Allocate` Given a buffer size in bytes, allocates the buffer on the device and returns it in a `BufferInfo` object.

`CopyHostToDevice` Copies a `BufferInfo` object for memory allocated on the host to the device. DeviceAdapterMemoryManager must implement two forms of `CopyHostToDevice`. The first form takes just a source `BufferInfo` and returns a new `BufferInfo` containing a copy of the data on the device. If the device supports shared or unified memory, this can be a shallow copy. The second form takes both a source `BufferInfo` and a pre-allocated destination `BufferInfo`.

`CopyDeviceToHost` Copies a `BufferInfo` object for memory allocated on the device to the host. DeviceAdapterMemoryManager must implement two forms of `CopyDeviceToHost`. The first form takes just a source `BufferInfo` and returns a new `BufferInfo` containing a copy of the data on the host. If the device supports shared or unified memory, this can be a shallow copy. The second form takes both a source `BufferInfo` and a pre-allocated destination `BufferInfo`.

`CopyDeviceToDevice` Copies a `BufferInfo` object for memory allocated on the device to another buffer on the device. DeviceAdapterMemoryManager must implement two forms of `CopyDeviceToDevice`. The first form takes just a source `BufferInfo` and returns a new `BufferInfo` containing a copy of the data on the device. The second form takes both a source `BufferInfo` and a pre-allocated destination `BufferInfo`.

If the control and execution environments share the same memory space, the execution array manager can, and should, share buffers among host and "device" and shallow copy data when possible. VTK-m comes with a class called `vtkm::cont::internal::DeviceAdapterMemoryManagerShared` that provides the implementation for a device memory manager that shares a memory space with the control environment. In this case, the DeviceAdapterMemoryManager specialization need only override the `GetDevice` method. (DeviceAdapterMemoryManagerShared will provide all other necessary overrides.)

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of DeviceAdapterMemoryManager, which by convention would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterMemoryManagerCxx11Thread.h header file.

Example 40.7: Specialization of DeviceAdapterMemoryManager.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/internal/DeviceAdapterMemoryManager.h>
4  #include <vtkm/cont/internal/DeviceAdapterMemoryManagerShared.h>
5
6  namespace vtkm
7  {
8  namespace cont
9  {
10 namespace internal
11 {
12
13 template<>
14 class DeviceAdapterMemoryManager<vtkm::cont::DeviceAdapterTagCxx11Thread>
15   : public vtkm::cont::internal::DeviceAdapterMemoryManagerShared
16 {
17 public:
18   VTKM_CONT vtkm::cont::DeviceAdapterId GetDevice() const override
19   {
20     return vtkm::cont::DeviceAdapterTagCxx11Thread{};
```

```
21    }
22  };
23
24  }
25  }
26  } // namespace vtkm::cont::internal
```

Did you know?

You may notice that although `vtkm::cont::internal::DeviceAdapterMemoryManager` *requires methods to allocate memory, it has no methods to delete memory. This is because all memory created by a* `vtkm::cont::internal::DeviceAdapterMemoryManager` *is wrapped in a* `vtkm::cont::internal::BufferInfo` *object. Responsibility for the memory management is taken over by* `BufferInfo` *and the memory will be automatically deleted once it is no longer used.*

40.4 Runtime Device Configuration

VTK-m defines a template named `vtkm::cont::internal::RuntimeDeviceConfiguration` that makes it possible to initialize various runtime configuration parameters of the underlying devices. `RuntimeDeviceConfiguration` has a single template argument that is the device adapter tag.

Example 40.8: Prototype for `RuntimeDeviceConfiguration`.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename DeviceAdapterTag>
9  class RuntimeDeviceConfiguration;
10
11 }
12 }
13 } // namespace vtkm::cont::internal
```

All device adapter implementations must create a specialization of `RuntimeDeviceConfiguration`. This specialization of `RuntimeDeviceConfiguration` must inherit from `vtkm::cont::internal::RuntimeDeviceConfigurationBase`. The `RuntimeDeviceConfiguration` provides various `RuntimeDeviceConfigurationBase::Set*` and `RuntimeDeviceConfigurationBase::Get*` methods for setting and accessing device specific runtime parameters. The superclass provides the `RuntimeDeviceConfigurationBase::Initialize` method that takes in a `RuntimeDeviceConfigurationOptions` argument used to set various device parameters when VTK-m is initialized.

Specializations of `RuntimeDeviceConfiguration` must override the `GetDevice` virtual method, which returns a `vtkm::cont::DeviceAdapterId` for the device that this runtime device configuration is overseeing. Specializations of `RuntimeDeviceConfiguration` are not required to override the following methods defined in `RuntimeDeviceConfigurationBase`. These methods should be overridden only if suitable device specific runtime parameters can be set or queried.

`SetThreads` Takes the provided `vtkm::Id` and attempts to set the number of threads to use for this specific

device. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/-failure of the device operation.

SetNumaRegions Takes the provided vtkm::Id and attempts to set the number of numa regions to use for this specific device Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

SetDeviceInstance Takes the provided vtkm::Id and attempts to set the specific device instance to use for this device Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

GetThreads Takes the provided vtkm::Id and attempts to set it to the number of threads this device is currently specified to use. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

GetNumaRegions Takes the provided vtkm::Id and attempts to set it to the number of numa regions this device is currently specified to use. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

GetDeviceInstance Takes the provided vtkm::Id and attempts to set it to the specific device instance this device is currently set to use. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

GetMaxThreads Takes the provided vtkm::Id and attempts to set it to the maximum number of threads allowed by this device. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

GetMaxDevices Takes the provided vtkm::Id and attempts to set it to the mximum number of devices currently allowed by this device. Return a vtkm::cont::internal::RuntimeDeviceConfigReturnCode representing the success/failure of the device operation.

ParseExtraArguments Called before RuntimeDeviceConfigurationBase::Initialize, used to perform extra command line argument parsing specific for a given device. Currently only overriden by the vtkm::cont::-internal::RuntimeDeviceConfigurationKokkos device.

InitializeSubsystem Called at the very end of RuntimeDeviceConfigurationBase::Initialize, and is used to perform additional subystem initialize for a given device. Currently over overriden by the vtkm::cont::-internal::RuntimeDeviceConfigurationKokkos device.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of RuntimeDeviceConfiguration, which by convention would be placed in the vtkm/cont/cxx11/internal/RuntimeDeviceConfigurationCxx11Thread.h header file.

Example 40.9: Specialization of RuntimeDeviceConfiguration.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/internal/RuntimeDeviceConfiguration.h>
4
5  #include <thread>
6
7  namespace vtkm
8  {
9  namespace cont
10 {
11 namespace internal
12 {
13
```

```
14 template<>
15 class RuntimeDeviceConfiguration<vtkm::cont::DeviceAdapterTagCxx11Thread>
16   : public vtkm::cont::internal::RuntimeDeviceConfigurationBase
17 {
18 public:
19   VTKM_CONT RuntimeDeviceConfiguration<vtkm::cont::DeviceAdapterTagCxx11Thread>()
20     : NumThreads(std::thread::hardware_concurrency())
21   {
22   }
23
24   VTKM_CONT vtkm::cont::DeviceAdapterId GetDevice() const override
25   {
26     return vtkm::cont::DeviceAdapterTagCxx11Thread{};
27   }
28
29   VTKM_CONT vtkm::cont::internal::RuntimeDeviceConfigReturnCode GetThreads(
30     vtkm::Id& value) const override
31   {
32     value = this->NumThreads;
33     return vtkm::cont::internal::RuntimeDeviceConfigReturnCode::SUCCESS;
34   }
35
36   VTKM_CONT vtkm::cont::internal::RuntimeDeviceConfigReturnCode SetThreads(
37     const vtkm::Id& value) override
38   {
39     if ((value <= 0) ||
40         (value > static_cast<vtkm::Id>(std::thread::hardware_concurrency())))
41     {
42       this->NumThreads = std::thread::hardware_concurrency();
43     }
44     else
45     {
46       this->NumThreads = value;
47     }
48     return vtkm::cont::internal::RuntimeDeviceConfigReturnCode::SUCCESS;
49   }
50
51   VTKM_CONT vtkm::cont::internal::RuntimeDeviceConfigReturnCode GetMaxThreads(
52     vtkm::Id& value) const override
53   {
54     value = std::thread::hardware_concurrency();
55     return vtkm::cont::internal::RuntimeDeviceConfigReturnCode::SUCCESS;
56   }
57
58 private:
59   vtkm::Id NumThreads;
60 };
61
62 }
63 }
64 } // namespace vtkm::cont::internal
```

Common Errors

`vtkm::cont::Initialize` *automatically initializes the* `vtkm::cont::internal::RuntimeDeviceConfiguration` *for all available devices using parse VTK-m command line arguments. These device runtime configurations are statically managed through the* `vtkm::cont::RuntimeDeviceInformation` *class, which ensures that there is exactly one initialized instance of each* `vtkm::cont::internal::RuntimeDeviceConfiguration` *available for each device. This guarantees that* `vtkm::cont::internal::RuntimeDeviceCon-`

figuration *device classes cannot be initialized more than once, but may lead to device initialization inconsistencies when attempting to access a* vtkm::cont::internal::RuntimeDeviceConfiguration *before calling* vtkm::cont::Initialize. *When creating a new* vtkm::cont::internal::RuntimeDeviceConfiguration *it is important to add an include for the new* DeviceAdapterRuntimeDetector::h*eader to* vtkm::cont::RuntimeDeviceInformation *so that the new device is compiled correctly. Additionally, it is important to note that accessing a* vtkm::cont::internal::RuntimeDeviceConfiguration *via* RuntimeDeviceInformation::GetRuntimeConfiguration *inside the* DeviceAdapterRuntimeDetector::Exists *method will initialize the underlying device incorrectly since VTK-m performs device existence checks while parsing command line arguments.*

40.5 Algorithms

A device adapter implementation must also provide a specialization of vtkm::cont::DeviceAdapterAlgorithm, which provides the underlying implementation of the algorithms described in Chapter 35. The implementation for the device adapter algorithms is typically placed in a header file with a prefix of DeviceAdapterAlgorithm.

Although there are many methods in DeviceAdapterAlgorithm, it is seldom necessary to implement them all. Instead, VTK-m comes with vtkm::cont::internal::DeviceAdapterAlgorithmGeneral that provides generic implementation for most of the required algorithms. By deriving the specialization of DeviceAdapterAlgorithm from DeviceAdapterAlgorithmGeneral, only the implementations for Schedule and Synchronize need to be implemented. All other algorithms can be derived from those.

That said, not all of the algorithms implemented in DeviceAdapterAlgorithmGeneral are optimized for all types of devices. Thus, it is worthwhile to provide algorithms optimized for the specific device when possible. In particular, it is best to provide specializations for the sort, scan, and reduce algorithms.

It is standard practice to implement a specialization of DeviceAdapterAlgorithm by having it inherit from vtkm::cont::internal::DeviceAdapterAlgorithmGeneral and specializing those methods that are optimized for a particular system. DeviceAdapterAlgorithmGeneral is a templated class that takes as its single template parameter the type of the subclass. For example, a device adapter algorithm structure named DeviceAdapterAlgorithm<DeviceAdapterTagFoo> will subclass DeviceAdapterAlgorithmGeneral<DeviceAdapterAlgorithm<DeviceAdapterTagFoo> >.

Did you know?

The convention of having a subclass be templated on the derived class' type is known as the Curiously Recurring Template Pattern (CRTP). In the case of DeviceAdapterAlgorithmGeneral, *VTK-m uses this CRTP behavior to allow the general implementation of these algorithms to run* Schedule *and other specialized algorithms in the subclass.*

One point to note when implementing the Schedule methods is to make sure that errors handled in the execution environment are handled correctly. As described in Chapter 23, errors are signaled in the execution environment by calling RaiseError on a functor or worklet object. This is handled internally by the vtkm::exec::internal::ErrorMessageBuffer class. ErrorMessageBuffer really just holds a small string buffer, which must be provided by the device adapter's Schedule method.

So, before Schedule executes the functor it is given, it should allocate a small string array in the execution environment, initialize it to the empty string, encapsulate the array in an ErrorMessageBuffer object, and set

this buffer object in the functor. When the execution completes, `Schedule` should check to see if an error exists in this buffer and throw a `vtkm::cont::ErrorExecution` if an error has been reported.

Common Errors

Exceptions are generally not supposed to be thrown in the execution environment, but it could happen on devices that support them. Nevertheless, few thread schedulers work well when an exception is thrown in them. Thus, when implementing adapters for devices that do support exceptions, it is good practice to catch them within the thread and report them through the `ErrorMessageBuffer`.

The following example is a minimal implementation of device adapter algorithms using C++11's `std::thread` class. Note that no attempt at providing optimizations has been attempted (and many are possible). By convention this code would be placed in the vtkm/cont/cxx11/internal/DeviceAdapterAlgorithmCxx11Thread.h header file.

Example 40.10: Minimal specialization of `DeviceAdapterAlgorithm`.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/DeviceAdapterAlgorithm.h>
4  #include <vtkm/cont/ErrorExecution.h>
5  #include <vtkm/cont/internal/DeviceAdapterAlgorithmGeneral.h>
6
7  #include <thread>
8
9  namespace vtkm
10 {
11 namespace cont
12 {
13
14 template<>
15 struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
16   : vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
17       DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>,
18       vtkm::cont::DeviceAdapterTagCxx11Thread>
19 {
20 private:
21   template<typename FunctorType>
22   struct ScheduleKernel1D
23   {
24     VTKM_CONT
25     ScheduleKernel1D(const FunctorType& functor)
26       : Functor(functor)
27     {
28     }
29
30     VTKM_EXEC
31     void operator()() const
32     {
33       try
34       {
35         for (vtkm::Id threadId = this->BeginId; threadId < this->EndId; threadId++)
36         {
37           this->Functor(threadId);
38           // If an error is raised, abort execution.
39           if (this->ErrorMessage.IsErrorRaised())
40           {
41             return;
```

```
42            }
43          }
44        }
45        catch (const vtkm::cont::Error& error)
46        {
47          this->ErrorMessage.RaiseError(error.GetMessage().c_str());
48        }
49        catch (const std::exception& error)
50        {
51          this->ErrorMessage.RaiseError(error.what());
52        }
53        catch (...)
54        {
55          this->ErrorMessage.RaiseError("Unknown exception raised.");
56        }
57      }
58
59      FunctorType Functor;
60      vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
61      vtkm::Id BeginId;
62      vtkm::Id EndId;
63    };
64
65    template<typename FunctorType>
66    struct ScheduleKernel3D
67    {
68      VTKM_CONT
69      ScheduleKernel3D(const FunctorType& functor, vtkm::Id3 maxRange)
70        : Functor(functor)
71        , MaxRange(maxRange)
72      {
73      }
74
75      VTKM_EXEC
76      void operator()() const
77      {
78        vtkm::Id3 threadId3D(this->BeginId % this->MaxRange[0],
79                             (this->BeginId / this->MaxRange[0]) % this->MaxRange[1],
80                             this->BeginId / (this->MaxRange[0] * this->MaxRange[1]));
81
82        try
83        {
84          for (vtkm::Id threadId = this->BeginId; threadId < this->EndId; threadId++)
85          {
86            this->Functor(threadId3D);
87            // If an error is raised, abort execution.
88            if (this->ErrorMessage.IsErrorRaised())
89            {
90              return;
91            }
92
93            threadId3D[0]++;
94            if (threadId3D[0] >= MaxRange[0])
95            {
96              threadId3D[0] = 0;
97              threadId3D[1]++;
98              if (threadId3D[1] >= MaxRange[1])
99              {
100               threadId3D[1] = 0;
101               threadId3D[2]++;
102             }
103           }
104         }
105       }
```

```
106       catch (const vtkm::cont::Error& error)
107       {
108         this->ErrorMessage.RaiseError(error.GetMessage().c_str());
109       }
110       catch (const std::exception& error)
111       {
112         this->ErrorMessage.RaiseError(error.what());
113       }
114       catch (...)
115       {
116         this->ErrorMessage.RaiseError("Unknown exception raised.");
117       }
118     }
119
120     FunctorType Functor;
121     vtkm::exec::internal::ErrorMessageBuffer ErrorMessage;
122     vtkm::Id BeginId;
123     vtkm::Id EndId;
124     vtkm::Id3 MaxRange;
125   };
126
127   template<typename KernelType>
128   VTKM_CONT static void DoSchedule(KernelType kernel, vtkm::Id numInstances)
129   {
130     if (numInstances < 1)
131     {
132       return;
133     }
134
135     const vtkm::Id MESSAGE_SIZE = 1024;
136     char errorString[MESSAGE_SIZE];
137     errorString[0] = '\0';
138     vtkm::exec::internal::ErrorMessageBuffer errorMessage(errorString, MESSAGE_SIZE);
139     kernel.Functor.SetErrorMessageBuffer(errorMessage);
140     kernel.ErrorMessage = errorMessage;
141
142     vtkm::Id numThreads;
143
144     auto config = internal::RuntimeDeviceConfiguration<
145       vtkm::cont::DeviceAdapterTagCxx11Thread>();
146     config.SetThreads(numInstances);
147     config.GetThreads(numThreads);
148     vtkm::Id numInstancesPerThread = (numInstances + numThreads - 1) / numThreads;
149
150     std::thread* threadPool = new std::thread[numThreads];
151     vtkm::Id beginId = 0;
152     for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
153     {
154       vtkm::Id endId = std::min(beginId + numInstancesPerThread, numInstances);
155       KernelType threadKernel = kernel;
156       threadKernel.BeginId = beginId;
157       threadKernel.EndId = endId;
158       std::thread newThread(threadKernel);
159       threadPool[threadIndex].swap(newThread);
160       beginId = endId;
161     }
162
163     for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
164     {
165       threadPool[threadIndex].join();
166     }
167
168     delete[] threadPool;
169
```

```
170     if (errorMessage.IsErrorRaised())
171     {
172       throw vtkm::cont::ErrorExecution(errorString);
173     }
174   }
175
176 public:
177   template<typename FunctorType>
178   VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id numInstances)
179   {
180     DoSchedule(ScheduleKernel1D<FunctorType>(functor), numInstances);
181   }
182
183   template<typename FunctorType>
184   VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id3 maxRange)
185   {
186     vtkm::Id numInstances = maxRange[0] * maxRange[1] * maxRange[2];
187     DoSchedule(ScheduleKernel3D<FunctorType>(functor, maxRange), numInstances);
188   }
189
190   VTKM_CONT
191   static void Synchronize()
192   {
193     // Nothing to do. This device schedules all of its operations using a
194     // split/join paradigm. This means that the if the control threaad is
195     // calling this method, then nothing should be running in the execution
196     // environment.
197   }
198 };
199
200 } // namespace cont
201 } // namespace vtkm
```

40.6 Timer Implementation

The VTK-m timer, described in Chapter 13, delegates to an internal class named `vtkm::cont::DeviceAdapterTimerImplementation`. The interface for this class is the same as that for `vtkm::cont::Timer`. A default implementation of this templated class uses the system timer and the `Synchronize` method in the device adapter algorithms.

However, some devices might provide alternate or better methods for implementing timers. For example, the TBB and CUDA libraries come with high resolution timers that have better accuracy than the standard system timers. Thus, the device adapter can optionally provide a specialization of `DeviceAdapterTimerImplementation`, which is typically placed in the same header file as the device adapter algorithms.

Continuing our example of a custom device adapter using C++11's `std::thread` class, we could use the default timer and it would work fine. But C++11 also comes with a `std::chrono` package that contains some portable time functions. The following code demonstrates creating a custom timer for our device adapter using this package. By convention, `DeviceAdapterTimerImplementation` is placed in the same header file as `DeviceAdapterAlgorithm`.

Example 40.11: Specialization of `DeviceAdapterTimerImplementation`.

```
1 #include <chrono>
2
3 namespace vtkm
4 {
5 namespace cont
6 {
```

```
7
8  template<>
9  class DeviceAdapterTimerImplementation<vtkm::cont::DeviceAdapterTagCxx11Thread>
10 {
11 public:
12   VTKM_CONT
13   DeviceAdapterTimerImplementation() { this->Reset(); }
14
15   VTKM_CONT
16   void Reset()
17   {
18     vtkm::cont::DeviceAdapterAlgorithm<
19       vtkm::cont::DeviceAdapterTagCxx11Thread>::Synchronize();
20     this->StartTime = std::chrono::high_resolution_clock::now();
21   }
22
23   VTKM_CONT
24   vtkm::Float64 GetElapsedTime()
25   {
26     vtkm::cont::DeviceAdapterAlgorithm<
27       vtkm::cont::DeviceAdapterTagCxx11Thread>::Synchronize();
28     std::chrono::high_resolution_clock::time_point endTime =
29       std::chrono::high_resolution_clock::now();
30
31     std::chrono::high_resolution_clock::duration elapsedTicks =
32       endTime - this->StartTime;
33
34     std::chrono::duration<vtkm::Float64> elapsedSeconds(elapsedTicks);
35
36     return elapsedSeconds.count();
37   }
38
39 private:
40   std::chrono::high_resolution_clock::time_point StartTime;
41 };
42
43 } // namespace cont
44 } // namespace vtkm
```

CHAPTER

FORTYONE

FUNCTION INTERFACE OBJECTS

For flexibility's sake a worklet is free to declare a `ControlSignature` with whatever number of arguments are sensible for its operation. The `Invoker` is expected to support arguments that match these arguments, and part of the invocation operation may require these arguments to be augmented before the worklet is scheduled. This leaves the invoker with the tricky task of managing some collection of arguments of unknown size and unknown types.

To simplify this management, VTK-m has the `vtkm::internal::FunctionInterface` class. `FunctionInterface` is a templated class that manages a generic set of arguments and return value from a function. An instance of `FunctionInterface` holds an instance of each argument. You can apply the arguments in a `FunctionInterface` object to a functor of a compatible prototype, and the resulting value of the function call is saved in the `FunctionInterface`.

41.1 Declaring and Creating

`vtkm::internal::FunctionInterface` is a templated class with a single parameter. The parameter is the *signature* of the function. A signature is a function type. The syntax in C++ is the return type followed by the argument types encased in parentheses.

Example 41.1: Declaring `vtkm::internal::FunctionInterface`.

```
1  // FunctionInterfaces matching some common POSIX functions.
2  vtkm::internal::FunctionInterface<size_t(const char*)> strlenInterface;
3
4  vtkm::internal::FunctionInterface<char*(char*, const char* s2, size_t)>
5    strncpyInterface;
```

The `vtkm::internal::make_FunctionInterface` function provides an easy way to create a `FunctionInterface` and initialize the state of all the parameters. `make_FunctionInterface` takes a variable number of arguments, one for each parameter. Since the return type is not specified as an argument, you must always specify it as a template parameter.

Example 41.2: Using `vtkm::internal::make_FunctionInterface`.

```
1  const char* s = "Hello World";
2  static const size_t BUFFER_SIZE = 100;
3  char* buffer = (char*)malloc(BUFFER_SIZE);
4
5  strlenInterface = vtkm::internal::make_FunctionInterface<size_t>(s);
6
7  strncpyInterface =
8    vtkm::internal::make_FunctionInterface<char*>(buffer, s, BUFFER_SIZE);
```

41.2 Parameters

Once created, FunctionInterface contains methods to query and manage the parameters and objects associated with them. The number of parameters can be retrieved either with the constant field ARITY or with the GetArity method.

Example 41.3: Getting the arity of a FunctionInterface.

```
1  VTKM_STATIC_ASSERT(vtkm::internal::FunctionInterface<size_t(const char*)>::ARITY ==
2                     1);
3
4  vtkm::IdComponent arity = strncpyInterface.GetArity(); // arity = 3
```

You can use the vtkm::internal::ParameterGet function to retrieve a parameter from a FunctionInterface. When using ParameterGet, you have to specify the index of the parameter using a template argument. Note that the parameters in FunctionInterface start at index 1. Although this is uncommon in C++, it is customary to number function arguments starting at 1.

Example 41.4: Using ParameterGet.

```
1 template<typename FunctionSignature>
2 void GetFirstParameter(
3   const vtkm::internal::FunctionInterface<FunctionSignature>& interface)
4 {
5   // The following two uses of GetParameter are equivalent
6   std::cout << vtkm::internal::ParameterGet<1>(interface) << std::endl;
7 }
```

41.3 Transformations

Rather than replace a single item in a FunctionInterface, it is desirable to change them all in a similar way. FunctionInterface supports a "static transform" that will replace all of the arguments with new types defined at compile time.

The static transform method (named StaticTransformCont) operates by accepting a functor that defines a function with two arguments. The first argument is the FunctionInterface parameter to transform. The second argument is an instance of the vtkm::internal::IndexTag templated class that statically identifies the parameter index being transformed. An IndexTag object has no state, but the class contains a static integer named INDEX. The function returns the transformed argument.

The functor must also contain a templated class named ReturnType with an internal type named type that defines the return type of the transform for a given parameter type. ReturnType must have two template parameters. The first template parameter is the type of the FunctionInterface parameter to transform. It is the same type as passed to the operator. The second template parameter is a vtkm::IdComponent specifying the index.

The transformation is only applied to the parameters of the function. The return argument is unaffected.

The return type can be determined with the StaticTransformType template in the FunctionInterface class. StaticTransformType has a single parameter that is the transform functor and contains a type named type that is the transformed FunctionInterface.

In the following example, a static transform is used to convert a FunctionInterface to a new object that has the pointers to the parameters rather than the values themselves. The parameter index is always ignored as all parameters are uniformly transformed.

Example 41.5: Using a static transform of function interface class.

```
struct ParametersToPointersFunctor
{
  template<typename T, vtkm::IdComponent Index>
  struct ReturnType
  {
    using type = const T*;
  };

  template<typename T, vtkm::IdComponent Index>
  VTKM_CONT const T* operator()(const T& x, vtkm::internal::IndexTag<Index>) const
  {
    return &x;
  }
};

template<typename FunctionInterfaceType>
VTKM_CONT typename FunctionInterfaceType::template StaticTransformType<
  ParametersToPointersFunctor>::type
ParametersToPointers(FunctionInterfaceType& functionInterface)
{
  return functionInterface.StaticTransformCont(ParametersToPointersFunctor());
}
```

CHAPTER

FORTYTWO

WORKLET ARGUMENTS

From the `ControlSignature` and `ExecutionSignature` defined in worklets, VTK-m uses template metaprogramming to build the code required to manage data from the control to the execution environment. These signatures contain tags that define the meaning of each argument and control how the argument data are transferred from the control to execution environments and broken up for each worklet instance.

Chapter 21 documents the many `ControlSignature` and `ExecutionSignature` tags that come with the worklet types. This chapter discusses the internals of these tags and how they control data management. Defining new worklet argument types can allow you to define new data structures in VTK-m. New worklet arguments are also usually a critical components for making new worklet types, as described in Chapter 43.

The management of data in worklet arguments is handled by three classes that provide type checking, transportation, and fetching, respectively. This chapter will first describe these type checking, transportation, and fetching classes and then describe how `ControlSignature` and `ExecutionSignature` tags specify these classes.

Throughout this chapter we demonstrate the definition of worklet arguments using an example of a worklet argument that represents line segments in 2D. The input for such an argument expects an `ArrayHandle` containing floating point `vtkm::Vec` s of size 2 to represent coordinates in the plane. The values in the array are paired up to define the two endpoints of each segment, and the worklet instance will receive a `Vec`-2 of `Vec`-2's representing the two endpoints. In practice, it is generally easier to use a `vtkm::cont::ArrayHandleGroupVec` (see Section 26.13), but this is a simple example for demonstration purposes. Plus, we will use this special worklet argument for our example of a custom worklet type in Chapter 43.

42.1 Type Checks

Before attempting to move data from the control to the execution environment, the VTK-m invokers check the input types to ensure that they are compatible with the associated `ControlSignature` concept. This is done with the `vtkm::cont::arg::TypeCheck` `struct`.

The `TypeCheck` `struct` is templated with two parameters. The first parameter is a tag that identifies which check to perform. The second parameter is the type of the control argument (after any dynamic casts). The `TypeCheck` class contains a static constant Boolean named `value` that is `true` if the type in the second parameter is compatible with the tag in the first or `false` otherwise.

Type checks are implemented with a defined type check tag (which, by convention, is defined in the `vtkm::cont::arg` namespace and starts with `TypeCheckTag`) and a partial specialization of the `vtkm::cont::arg::TypeCheck` structure. The following type checks (identified by their tags) are provided in VTK-m.

`vtkm::cont::arg::TypeCheckTagExecObject` True if the type is an execution object. All execution objects

must derive from vtkm::cont::ExecutionObjectBase and follow the conventions of that class.

vtkm::cont::arg::TypeCheckTagArrayIn True if the type is a vtkm::cont::ArrayHandle that is capable of reading.

vtkm::cont::arg::TypeCheckTagArrayOut True if the type is a vtkm::cont::ArrayHandle that is capable of writing.

vtkm::cont::arg::TypeCheckTagArrayInOut True if the type is a vtkm::cont::ArrayHandle that is capable of reading and writing.

vtkm::cont::arg::TypeCheckTagAtomicArray Similar to TypeCheckTagArrayInOut except it only returns true for array types with values that are supported for atomic arrays.

vtkm::cont::arg::TypeCheckTagBitField True if the type is a vtkm::cont::BitField.

vtkm::cont::arg::TypeCheckTagCellSet True if and only if the object is a vtkm::cont::CellSet or one of its subclasses.

vtkm::cont::arg::TypeCheckTagCellSetStructured True if the object is a vtkm::cont::CellSetStructured.

vtkm::cont::arg::TypeCheckTagExecObject True if the type is a subclass of vtkm::cont::ExecutionObjectBase. See Chapter 29 for more information on execution objects for worklets.

vtkm::cont::arg::TypeCheckTagKeys True if and only if the object is a vtkm::worklet::Keys class.

Here are some trivial examples of using TypeCheck. Typically these checks are done internally in the base VTK-m invoker code, so these examples are for demonstration only.

Example 42.1: Behavior of vtkm::cont::arg::TypeCheck.

```
1  struct MyExecObject : vtkm::cont::ExecutionObjectBase
2  {
3    vtkm::Id Value;
4  };
5
6  void DoTypeChecks()
7  {
8    using vtkm::cont::arg::TypeCheck;
9    using vtkm::cont::arg::TypeCheckTagArrayIn;
10   using vtkm::cont::arg::TypeCheckTagExecObject;
11
12   bool check1 = TypeCheck<TypeCheckTagExecObject, MyExecObject>::value; // true
13   bool check2 = TypeCheck<TypeCheckTagExecObject, vtkm::Id>::value;     // false
14
15   using ArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
16
17   bool check3 = TypeCheck<TypeCheckTagArrayIn, ArrayType>::value;    // true
18   bool check4 = TypeCheck<TypeCheckTagExecObject, ArrayType>::value; // false
19 }
```

A type check is created by first defining a type check tag object, which by convention is placed in the vtkm::cont::arg namespace and whose name starts with TypeCheckTag. Then, create a specialization of the vtkm::cont::arg::TypeCheck template class with the first template argument matching the aforementioned tag. As stated previously, the TypeCheck class must contain a `value` static constant Boolean representing whether the type is acceptable for the corresponding Invoker argument.

This example of a TypeCheck returns true for control objects that are ArrayHandles with a value type that is a floating point vtkm::Vec of size 2.

Example 42.2: Defining a custom TypeCheck.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

struct TypeCheckTag2DCoordinates
{
};

template<typename ArrayType>
struct TypeCheck<TypeCheckTag2DCoordinates, ArrayType>
{
  static constexpr bool value = false;
};

template<typename T, typename Storage>
struct TypeCheck<TypeCheckTag2DCoordinates, vtkm::cont::ArrayHandle<T, Storage>>
{
  static constexpr bool value = vtkm::ListHas<vtkm::TypeListFieldVec2, T>::value;
};

} // namespace arg
} // namespace cont
} // namespace vtkm
```

42.2 Transport

After all the argument types are checked, the VTK-m dispatch mechanism must load the data into the execution environment before scheduling a job to run there. This is done with the vtkm::cont::arg::Transport struct.

The Transport struct is templated with three parameters. The first parameter is a tag that identifies which transport to perform. The second parameter is the type of the control parameter (after any dynamic casts). The third parameter is a device adapter tag for the device on which the data will be loaded.

A Transport contains a type named ExecObjectType that is the type used after data is moved to the execution environment. A Transport also has a const parenthesis operator that takes 5 arguments: the control-side object that is to be transported to the execution environment, the control-side object that represents the input domain, the size of the input domain, the size of the output domain, and a reference to a vtkm::cont::Token object that is used to define the scope of any generated execution environment objects. The parenthesis operator returns an execution-side object. This operator is called in the control environment, and the operator returns an object that is ready to be used in the execution environment.

Transports are implemented with a defined transport tag (which, by convention, is defined in the vtkm::cont::arg namespace and starts with TransportTag) and a partial specialization of the vtkm::cont::arg::Transport structure. The following transports (identified by their tags) are provided in VTK-m.

vtkm::cont::arg::TransportTagExecObject Calls PrepareForInput on the provided object. The returned execution object is what is provided by PrepareForInput. See Chapter 29 for more information on execution objects for worklets.

vtkm::cont::arg::TransportTagArrayIn Loads data from a vtkm::cont::ArrayHandle onto the specified device using the array handle's PrepareForInput method. The size of the array must be the same as the input domain. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagArrayOut` Allocates data onto the specified device for a `vtkm::cont::ArrayHandle` using the array handle's `PrepareForOutput` method. The array is allocated to the size of the output domain. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagArrayInOut` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInPlace` method. The size of the array must be the same size as the output domain (which is not necessarily the same size as the input domain). The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagWholeArrayIn` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInput` method. This transport is designed to be used with random access whole arrays, so unlike `TransportTagArrayIn` the array size can be unassociated with the input domain. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagWholeArrayOut` Readies data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForOutput` method. This transport is designed to be used with random access whole arrays, so unlike `TransportTagArrayOut` the array size can be unassociated with the input domain. Thus, the array must be pre-allocated and its size is not changed. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagWholeArrayInOut` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInPlace` method. This transport is designed to be used with random access whole arrays, so unlike `TransportTagArrayInOut` the array size can be unassociated with the input domain. The returned execution object is an array portal.

`vtkm::cont::arg::TransportTagAtomicArray` Loads data from a `vtkm::cont::ArrayHandle` and creates a `vtkm::exec::AtomicArray`.

`vtkm::cont::arg::TransportTagBitFieldIn` Loads data from a `vtkm::cont::BitField` onto the specified device using the `PrepareForInput` method.

`vtkm::cont::arg::TransportTagBitFieldInOut` Loads data from a `vtkm::cont::BitField` onto the specified device using the `PrepareForInPlace` method.

`vtkm::cont::arg::TransportTagBitFieldInOut` This is essentially the same as `TransportTagBitFieldInOut`. Because a bit field does not follow the size of the domain, the array must be sized before passed to an invoke.

`vtkm::cont::arg::TransportTagCellSetIn` Loads data from a `vtkm::cont::CellSet` object. The `TransportTagCellSetIn` it a templated class with two parameters: the "from" topology and the "to" topology. (See Section 21.2.3 for a description of "from" and "to" topologies.) The returned execution object is a connectivity object (as described in Section 28.3).

`vtkm::cont::arg::TransportTagTopologyFieldIn` Similar to `TransportTagArrayIn` except that the size is checked against the "from" topology of a cell set for the input domain. The input domain object is assumed to be a `vtkm::cont::CellSet`.

`vtkm::cont::arg::TransportTagKeysIn` Loads data from a `vtkm::worklet::Keys` object. This transport is intended to be used for the input domain of a `vtkm::worklet::WorkletReduceByKey`. The returned execution object is of type `vtkm::exec::internal::ReduceByKeyLookup`.

`vtkm::cont::arg::TransportTagKeyedValuesIn` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInput` method. This transport uses the input domain object, which is expected to be a `vtkm::worklet::Keys` object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

`vtkm::cont::arg::TransportTagKeyedValuesOut` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForOutput` method. This transport uses the input domain object, which is expected to be a `vtkm::worklet::Keys` object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

`vtkm::cont::arg::TransportTagKeyedValuesInOut` Loads data from a `vtkm::cont::ArrayHandle` onto the specified device using the array handle's `PrepareForInPlace` method. This transport uses the input domain object, which is expected to be a `vtkm::worklet::Keys` object, and groups the entries in the array by unique keys. The returned execution object is an array portal of grouped values.

Here are some trivial examples of using `Transport`. Typically this movement is done internally in the VTK-m dispatching code, so these examples are for demonstration only.

Example 42.3: Behavior of `vtkm::cont::arg::Transport`.

```
template<typename ArrayType, typename Device>
void DoTransport(ArrayType inArray, ArrayType outArray, Device)
{
  VTKM_IS_ARRAY_HANDLE(ArrayType);
  VTKM_IS_DEVICE_ADAPTER_TAG(Device);

  using vtkm::cont::arg::Transport;
  using vtkm::cont::arg::TransportTagArrayIn;
  using vtkm::cont::arg::TransportTagArrayOut;
  using vtkm::cont::arg::TransportTagWholeArrayInOut;

  vtkm::cont::Token token;

  // The array in transport returns a read-only array portal.
  using ArrayInTransport = Transport<TransportTagArrayIn, ArrayType, Device>;
  typename ArrayInTransport::ExecObjectType inPortal =
    ArrayInTransport()(inArray, inArray, 10, 10, token);

  // The array out transport returns an allocated array portal.
  using ArrayOutTransport = Transport<TransportTagArrayOut, ArrayType, Device>;
  typename ArrayOutTransport::ExecObjectType outPortal =
    ArrayOutTransport()(outArray, inArray, 10, 10, token);

  // The whole array in transport returns a read-only array portal wrapped in
  // a vtkm::exec::ExecutionWholeArrayConst.
  using WholeArrayTransport =
    Transport<TransportTagWholeArrayInOut, ArrayType, Device>;
  typename WholeArrayTransport::ExecObjectType wholeArray =
    WholeArrayTransport()(inArray, inArray, 10, 10, token);
}
```

A transport is created by first defining a transport tag object, which by convention is placed in the `vtkm::cont::arg` namespace and whose name starts with `TransportTag`. Then, create a specialization of the `vtkm::cont::arg::Transport` template class with the first template argument matching the aforementioned tag. As stated previously, the `Transport` class must contain an `ExecObjectType` type and a parenthesis operator turning the associated control argument into an execution environment object.

This example internally uses a `vtkm::cont::ArrayHandleGroupVec` to take values from an input `ArrayHandle` and pair them up to represent line segments. The resulting execution object is an array portal containing `Vec`-2 values of `Vec`-2's.

Example 42.4: Defining a custom `Transport`.

```
namespace vtkm
{
namespace cont
```

```
4  {
5  namespace arg
6  {
7
8  struct TransportTag2DLineSegmentsIn
9  {
10 };
11
12 template<typename ContObjectType, typename Device>
13 struct Transport<vtkm::cont::arg::TransportTag2DLineSegmentsIn,
14                  ContObjectType,
15                  Device>
16 {
17   VTKM_IS_ARRAY_HANDLE(ContObjectType);
18
19   using GroupedArrayType = vtkm::cont::ArrayHandleGroupVec<ContObjectType, 2>;
20
21   using ExecObjectType = typename GroupedArrayType::ReadPortalType;
22
23   template<typename InputDomainType>
24   VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
25                                       const InputDomainType&,
26                                       vtkm::Id inputRange,
27                                       vtkm::Id,
28                                       vtkm::cont::Token& token) const
29   {
30     if (object.GetNumberOfValues() != inputRange * 2)
31     {
32       throw vtkm::cont::ErrorBadValue(
33         "2D line segment array size does not agree with input size.");
34     }
35
36     GroupedArrayType groupedArray(object);
37     return groupedArray.PrepareForInput(Device{}, token);
38   }
39 };
40
41 } // namespace arg
42 } // namespace cont
43 } // namespace vtkm
```

Common Errors

It is fair to assume that the `Transport`*'s control object type matches whatever the associated* `TypeCheck` *allows. However, it is good practice to provide a secondary compile-time check in the* `Transport` *class, like the one on line 17 in Example 42.4, for debugging purposes in case there is a problem with the* `TypeCheck` *or this* `Transport` *is used with an unexpected* `TypeCheck`*.*

42.3 Fetch

Before the function of a worklet is invoked, the VTK-m internals pull the appropriate data out of the execution object and pass it to the worklet function. A class named `vtkm::exec::arg::Fetch` is responsible for pulling this data out and putting computed data in to the execution objects.

The `Fetch` `struct` is templated with three parameters. The first parameter is a tag that identifies which type of fetch to perform. The second parameter is a different tag that identifies the aspect of the data to fetch.

The third template parameter to a `Fetch` `struct` is the type of the execution object that is created by the `Transport` (as described in Section 42.2). This is generally where the data are fetched from.

A `Fetch` also has a pair of methods named `Load` and `Store` that get data from and add data to the execution object at a given domain or thread index.

In both `Load` and `Store` methods, its first parameter is a thread indices object which manages the multiple indices that are relevant to the worklet invocation including the input index, output index, and visit index, all of which can be different.

The specific type of the thread indices object depends on the type of worklet begin invoked, but all thread indices classes implement methods named `GetInputIndex`, `GetOutputIndex`, and `GetVisitIndex` to get those respective indices. The thread indices object may also contain other methods to get information pertinent to the associated worklet's execution. For example a thread indices object associated with a topology map has methods to get the shape identifier and incident from indices of the current input object. Thread indices objects are discussed in more detail in Section 43.2.

Fetches are specified with a pair of fetch and aspect tags. Fetch tags are by convention defined in the `vtkm::exec::arg` namespace and start with `FetchTag`. Likewise, aspect tags are also defined in the `vtkm::exec::arg` namespace and start with `AspectTag`. The `Fetch` class is partially specialized on these two tags.

The most common aspect tag is `vtkm::exec::arg::AspectTagDefault`, and all fetch tags should have a specialization of `vtkm::exec::arg::Fetch` with this tag. The following list of fetch tags describes the execution objects they work with and the data they pull for each aspect tag they support.

`vtkm::exec::arg::FetchTagExecObject` Simply returns an execution object. This fetch only supports the `AspectTagDefault` aspect. The `Load` returns the executive object in the associated parameter. The `Store` does nothing.

`vtkm::exec::arg::FetchTagWholeCellSetIn` Loads data from a cell set. The `Load` simply returns the execution object created with a `TransportTagCellSetIn` and the `Store` does nothing.

`vtkm::exec::arg::FetchTagArrayDirectIn` Loads data from an array portal. This fetch only supports the `AspectTagDefault` aspect. The `Load` gets data directly from the domain (thread) index. The `Store` does nothing.

`vtkm::exec::arg::FetchTagArrayDirectOut` Stores data to an array portal. This fetch only supports the `AspectTagDefault` aspect. The `Store` sets data directly to the domain (thread) index. The `Load` does nothing.

`vtkm::exec::arg::FetchTagCellSetIn` Load data from a cell set. This fetch is used with the worklet topology maps to pull topology information from a cell set. The `Load` simply returns the cell shape of the given input cells and the `Store` method does nothing. This tag is typically used with the input domain object, and aspects like `vtkm::exec::arg::AspectTagIncidentElementCount` and `vtkm::exec::arg::AspectTagIncidentElementIndices` are used to get more detailed information.

`vtkm::exec::arg::FetchTagArrayTopologyMapIn` Loads data from the "from" topology in a topology map. For example, in a point to cell topology map, this fetch will get the field values for all points attached to the cell being visited. The `Load` returns a `Vec`-like object containing all the incident field values whereas the `Store` method does nothing. This fetch is designed for use in topology maps and expects the input domain to be a cell set.

A fetch is created by first defining a fetch tag object, which by convention is placed in the `vtkm::exec::arg` namespace and whose name starts with `FetchTag`. Then, create a specialization of the `vtkm::exec::arg::Fetch` template class with the first template argument matching the aforementioned tag. As stated previously, the `Fetch` class must contain a pair of `Load` and `Store` methods that get a value out of the data and store a value in the data, respectively.

Example 42.5: Defining a custom `Fetch`.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct FetchTag2DLineSegmentsIn
{
};

template<typename ExecObjectType>
struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
             vtkm::exec::arg::AspectTagDefault,
             ExecObjectType>
{
  using ValueType = typename ExecObjectType::ValueType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename ThreadIndicesType>
  VTKM_EXEC ValueType Load(const ThreadIndicesType& indices,
                           const ExecObjectType& arrayPortal) const
  {
    return arrayPortal.Get(indices.GetInputIndex());
  }

  template<typename ThreadIndicesType>
  VTKM_EXEC void Store(const ThreadIndicesType&,
                       const ExecObjectType&,
                       const ValueType&) const
  {
    // Store is a no-op for this fetch.
  }
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

Did you know?

The fetch defined in Example 42.5 could actually be replaced by the more general `FetchTagArrayDirectIn` that already comes with VTK-m. This example is mostly provided for demonstrative purposes.

In addition to the aforementioned aspect tags that are explicitly paired with fetch tags, VTK-m also provides some aspect tags that either modify the behavior of a general fetch or simply ignore the type of fetch.

`vtkm::exec::arg::AspectTagDefault` Performs the "default" fetch. Every fetch tag should have an implementation of `vtkm::exec::arg::Fetch` with that tag and `AspectTagDefault`.

`vtkm::exec::arg::AspectTagWorkIndex` Simply returns the domain (or thread) index ignoring any associated data. This aspect is used to implement the `WorkIndex` execution signature tag.

`vtkm::exec::arg::AspectTagInputIndex` Returns the index of the element being used from the input domain. This is often the same as the work index but can be different if a scatter is being used. (See Section 31.1 for information on scatters in worklets.)

`vtkm::exec::arg::AspectTagOutputIndex` Returns the index of the element being written to the output. This is generally the same as the work index.

`vtkm::exec::arg::AspectTagVisitIndex` Returns the visit index corresponding to the current input. Together the pair of input index and visit index are unique.

`vtkm::exec::arg::AspectTagCellShape` Returns the cell shape from the input domain. This aspect is designed to be used with topology maps.

`vtkm::exec::arg::AspectTagIncidentElementCount` Returns the number of elements associated with the "from" topology that are incident to the input element of the "to" topology. This aspect is designed to be used with topology maps.

`vtkm::exec::arg::AspectTagIncidentElementIndices` Returns a `Vec`-like object containing the indices to the elements associated with the "from" topology that are incident to the input element of the "to" topology. This aspect is designed to be used with topology maps.

`vtkm::exec::arg::AspectTagValueCount` Returns the number of times the key associated with the current input. This aspect is designed to be used with reduce by key maps.

An aspect is created by first defining an aspect tag object, which by convention is placed in the `vtkm::exec::arg` namespace and whose name starts with `AspectTag`. Then, create specializations of the `vtkm::exec::arg::Fetch` template class where appropriate with the second template argument matching the aforementioned tag.

This example creates a specialization of a `Fetch` to retrieve the first point of a line segment.

Example 42.6: Defining a custom `Aspect`.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct AspectTagFirstPoint
{
};

template<typename ExecObjectType>
struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
             vtkm::exec::arg::AspectTagFirstPoint,
             ExecObjectType>
{
  using ValueType = typename ExecObjectType::ValueType::ComponentType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename ThreadIndicesType>
  VTKM_EXEC ValueType Load(const ThreadIndicesType& indices,
                           const ExecObjectType& arrayPortal) const
  {
    return arrayPortal.Get(indices.GetInputIndex())[0];
  }
```

```
26
27   template<typename ThreadIndicesType>
28   VTKM_EXEC void Store(const ThreadIndicesType&,
29                        const ExecObjectType&,
30                        const ValueType&) const
31   {
32     // Store is a no-op for this fetch.
33   }
34 };
35
36 } // namespace arg
37 } // namespace exec
38 } // namespace vtkm
```

42.4 Creating New ControlSignature Tags

The type checks, transports, and fetches defined in the previous sections of this chapter conspire to interpret the arguments given to a Invoker and provide data to an instance of a worklet. What remains to be defined are the tags used in the ControlSignature and ExecutionSignature that bring these three items together. These two types of tags are defined differently. In this section we discuss the ControlSignature tags.

A ControlSignature tag is defined by a `struct` (or equivocally a `class`). This `struct` is typically defined inside a worklet (or, more typically, a worklet superclass) so that it can be used without qualifying its namespace. VTK-m has requirements for every defined ControlSignature tag.

The first requirement of a ControlSignature tag is that it must inherit from vtkm::cont::arg::ControlSignatureTagBase. You will get a compile error if you attempt to use a type that is not a subclass of ControlSignatureTagBase in a ControlSignature.

The second requirement of a ControlSignature tag is that it must contain the following three types: `TypeCheckTag`, `TransportTag`, and `FetchTag`. As the names would imply, these specify tags for TypeCheck, Transport, and Fetch classes, respectively, which were discussed earlier in this chapter.

The following example defines a ControlSignature tag for an array that represents 2D line segments using the classes defined in previous examples.

Example 42.7: Defining a new ControlSignature tag.

```
1 struct LineSegment2DCoordinatesIn : vtkm::cont::arg::ControlSignatureTagBase
2 {
3   using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
4   using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
5   using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
6 };
```

Once defined, this tag can be used like any other ControlSignature tag.

Example 42.8: Using a custom ControlSignature tag.

```
1 using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
2                               FieldOut vecOut,
3                               FieldIn index);
```

42.5 Creating New ExecutionSignature Tags

An ExecutionSignature tag is defined by a `struct` (or equivocally a `class`). This `struct` is typically defined inside a worklet (or, more typically, a worklet superclass) so that it can be used without qualifying its namespace.

VTK-m has requirements for every defined ExecutionSignature tag.

The first requirement of an ExecutionSignature tag is that it must inherit from vtkm::exec::arg::ExecutionSignatureTagBase. You will get a compile error if you attempt to use a type that is not a subclass of ExecutionSignatureTagBase in an ExecutionSignature.

The second requirement of an ExecutionSignature tag is that it must contain a type named AspectTag, which is set to an aspect tag. As discussed in Section 42.3, the aspect tag is passed as a template argument to the vtkm::exec::arg::Fetch class to modify the data it loads and stores. The numerical ExecutionSignature tags (i.e. _1, _2, etc.) operate by setting the AspectTag to vtkm::exec::arg::AspectTagDefault, effectively engaging the default fetch.

The third requirement of an ExecutionSignature tag is that it contains an INDEX member that is a static const vtkm::IdComponent. The number that INDEX is set to refers to the ControlSignature argument from which that data come from (indexed starting at 1). The numerical ExecutionSignature tags (i.e. _1, _2, etc.) operate by setting their INDEX values to the corresponding number (i.e. 1, 2, etc.). An ExecutionSignature tag might take another tag as a template argument and copy the INDEX from one to another. This allows you to use a tag to modify the aspect of another tag. Most often this is used to apply a particular aspect to a numerical ExecutionSignature tag (i.e. _1, _2, etc.). Still other ExecutionSignature tags might not need direct access to any ControlSignature arguments (such as those that pull information from thread indices). If the INDEX does not matter (because the execution object parameter to the Fetch Load and Store is ignored). In this case, the ExecutionSignature tag can set the INDEX to 1, because there is guaranteed to be at least one control argument.

The following example defines an ExecutionSignature tag to get the coordinates for only the first point in a 2D line segment. The defined tag takes as an argument another tag (generally one of the numeric tags), which is expected to point to a ControlSignature argument with a LineSegment2DCoordinatesIn (as defined in Example 42.7).

Example 42.9: Defining a new ExecutionSignature tag.

```
1  template<typename ArgTag>
2  struct FirstPoint : vtkm::exec::arg::ExecutionSignatureTagBase
3  {
4    static const vtkm::IdComponent INDEX = ArgTag::INDEX;
5    using AspectTag = vtkm::exec::arg::AspectTagFirstPoint;
6  };
```

Once defined, this tag can be used like any other ExecutionSignature tag.

Example 42.10: Using a custom ExecutionSignature tag.

```
1  using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
2                                FieldOut vecOut,
3                                FieldIn index);
4  using ExecutionSignature = void(FirstPoint<_1>, SecondPoint<_1>, _2);
```

CHAPTER

FORTYTHREE

NEW WORKLET TYPES

The basic building block for an algorithm in VTK-m is the worklet. Chapter 21 describes the different types of worklet types provided by VTK-m and how to use them to create algorithms. However, it is entirely possible that this set of worklet types does not directly cover what is needed to implement a particular algorithm. One way around this problem is to use some of the numerous back doors provided by VTK-m to provide less restricted access in the execution environment such as using whole arrays for random access.

However, it make come to pass that you encounter a particular pattern of execution that you find useful for implementing several algorithms. If such is the case, it can be worthwhile to create a new worklet type that directly supports such a pattern. Creating a new worklet type can provide two key advantages. First, it makes implementing algorithms of this nature easier, which saves developer time. Second, it can make the implementation of such algorithms safer. By encapsulating the management of structures and regulating the data access, users of the worklet type can be more assured of correct behavior.

This chapter documents the process for creating new worklet types. The operation of a worklet requires the coordination of several different object types such as dispatchers, argument handlers, and thread indices. This chapter will provide examples of all these required components. To tie all these features together, we start this chapter with a motivating example for an implementation of a custom worklet type. The chapter then discusses the individual components of the worklet, which in the end come together for the worklet type that is then demonstrated.

43.1 Motivating Example

For our motivation to create a new worklet type, let us consider the use case of building fractals. Fractals are generally not a primary concern of visualization libraries like VTK-m, but building a fractal (or approximations of fractals) has similarities the the computational geometry problems in scientific visualization. In particular, we consider the class of fractals that is generated by replacing each line in a shape with some collection of lines. These types of fractals are interesting because, in addition to other reasons, the right parameters result in a shape that has infinite length confined to a finite area.

A simple but well known example of a line fractal is the Koch Snowflake. The Koch Snowflake starts as a line or triangle that gets replaced with the curve shown in Figure 43.1.

The fractal is formed by iteratively replacing the curve's lines with this basic shape. Figure 43.2 shows the second iteration and then several subsequent iterations that create a "fuzzy" curve. The curve is confined to a limited area regardless of how many iterations are performed, but the length of the curve approaches infinity as the number of iterations approaches infinity.

In our finite world we want to estimate the curve of the Koch Snowflake by performing a finite amount of

Figure 43.1: Basic shape for the Koch Snowflake.

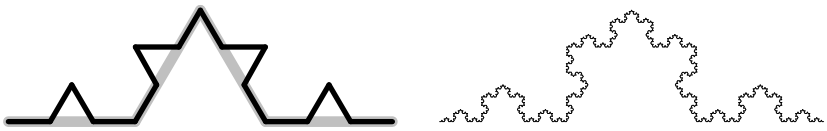


Figure 43.2: The Koch Snowflake after the second iteration (left image) and after several more iterations (right image).

iterations. This is similar to a Lindenmayer system but with less formality. The size of the curve grows quickly and in practice it takes few iterations to make close approximations.

Did you know?

The Koch Snowflake is just one example of many line fractals we can make with this recursive line substitution, which is why it is fruitful to create a worklet type to implement such fractals. We use the Koch Snowflake to set up the example here. Section 43.6 provides several more examples.

To implement line fractals of this nature, we want to be able to define the lines of the base shape in terms of parametric coordinates and then transform the coordinates to align with a line segment. For example, the Koch Snowflake base shape could be defined with parametric coordinates shown in Figure 43.3.

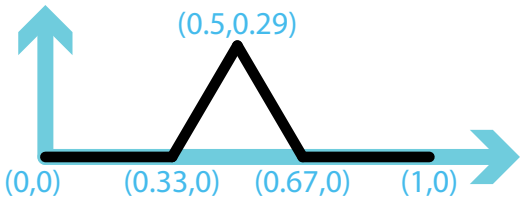


Figure 43.3: Parametric coordinates for the Koch Snowflake shape.

Given these parametric coordinates, for each line we define an axis with the main axis along the line segment and the secondary axis perpendicular to that. Given this definition, we can perform each fractal iteration by applying this transform for each line segment as shown in Figure 43.4.

To implement the application of the line fractal demonstrated in Figure 43.4, let us define a class named `LineFractalTransform` that takes as its constructor the coordinates of two ends of the original line. As its operator, `LineFractalTransform` takes a point in parametric space and returns the coordinates in world space in respect to the original line segment. We define this class in the `vtkm::exec` namespace because the intended use case is by worklets of the type we are making. A definition of `LineFractalTransform` is given in Example 43.1

Example 43.1: A support class for a line fractal worklet.

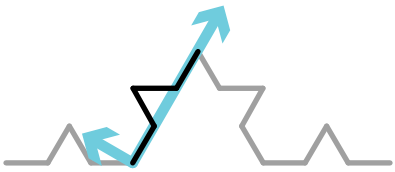


Figure 43.4: Applying the line fractal transform for the Koch Snowflake.

```
namespace vtkm
{
namespace exec
{

class LineFractalTransform
{
public:
  template<typename T>
  VTKM_EXEC LineFractalTransform(const vtkm::Vec<T, 2>& point0,
                                 const vtkm::Vec<T, 2>& point1)
  {
    this->Offset = point0;
    this->UAxis = point1 - point0;
    this->VAxis = vtkm::make_Vec(-this->UAxis[1], this->UAxis[0]);
  }

  template<typename T>
  VTKM_EXEC vtkm::Vec<T, 2> operator()(const vtkm::Vec<T, 2>& ppoint) const
  {
    vtkm::Vec2f ppointCast(ppoint);
    vtkm::Vec2f transform =
      ppointCast[0] * this->UAxis + ppointCast[1] * this->VAxis + this->Offset;
    return vtkm::Vec<T, 2>(transform);
  }

  template<typename T>
  VTKM_EXEC vtkm::Vec<T, 2> operator()(T x, T y) const
  {
    return (*this)(vtkm::Vec<T, 2>(x, y));
  }

private:
  vtkm::Vec2f Offset;
  vtkm::Vec2f UAxis;
  vtkm::Vec2f VAxis;
};

} // namespace exec
} // namespace vtkm
```

Did you know?

The definition of `LineFractalTransform` *(or something like it) is not strictly necessary for implementing a worklet type. However, it is common to implement such supporting classes that operate in the execution environment in support of the operations typically applied by the worklet type.*

The remainder of this chapter is dedicated to defining a `WorkletLineFractal` class and supporting objects that

allow you to easily make line fractals. Example 43.2 demonstrates how we intend to use this worklet type.

Example 43.2: Demonstration of how we want to use the line fractal worklet.

```
struct KochSnowflake
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<4>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.00f, 0.00f);
      segmentsOutVec[0][1] = transform(0.33f, 0.00f);

      segmentsOutVec[1][0] = transform(0.33f, 0.00f);
      segmentsOutVec[1][1] = transform(0.50f, 0.29f);

      segmentsOutVec[2][0] = transform(0.50f, 0.29f);
      segmentsOutVec[2][1] = transform(0.67f, 0.00f);

      segmentsOutVec[3][0] = transform(0.67f, 0.00f);
      segmentsOutVec[3][1] = transform(1.00f, 0.00f);
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.WritePortal().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.WritePortal().Set(1, vtkm::Vec2f(1.0f, 0.0f));

    vtkm::cont::Invoker invoke;
    KochSnowflake::FractalWorklet worklet;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
      invoke(worklet, points, outPoints);
      points = outPoints;
    }

    return points;
  }
};
```

43.2 Thread Indices

The first internal support class for implementing a worklet type is a class that manages indices for a thread. As the name would imply, the thread indices class holds a reference to an index identifying work to be done by the current thread. This includes indices to the current input element and the current output element. The thread indices object can also hold other information (that may not strictly be index data) about the

input and output data. For example, the thread indices object for topology maps (named `vtkm::exec::arg::ThreadIndicesTopologyMap`) maintains cell shape and connection indices for the current input object.

As is discussed briefly in Section 42.3, a thread indices object is passed to the methods of the class `vtkm::exec::arg::Fetch` to retrieve data from the execution object. The thread indices object serves two important functions for the `Fetch`. The first function is to cache information about the current thread that is likely to be used by multiple objects retrieving information. For example, in a point to cell topology map data from point fields must be retrieved by looking up indices in the topology connections. It is more efficient to retrieve the topology connections once and store them in the thread indices than it is to look them up independently for each field.

The second function of thread indices is to make it easier to find information about the input domain when fetching data. Once again, getting point data in a point to cell topology map requires looking up connectivity information in the input domain. However, the `Fetch` object for the point field does not have direct access to the data for the input domain. Instead, it gets this information from the thread indices.

All worklet classes have a method named `GetThreadIndices` that constructs a thread indices object for a given thread. `GetThreadIndices` is called with 5 parameters: a unique index for the thread (i.e. worklet instance), an array portal that maps output indices to input indices (which might not be one-to-one if a scatter is being used), an array portal that gives the visit index for each output index, an array portal that maps each unique thread index to the output index for that thread, and the execution object for the input domain.

The base worklet implementation provides an implementation of `GetThreadIndices` that creates a `vtkm::exec::arg::ThreadIndicesBasic` object. This provides the minimum information required in a thread indices object, but non-trivial worklet types are likely to need to provide their own thread indices type. This following example shows the implementation of `GetThreadIndices` we will use in our worklet type superclass (discussed in more detail in Section 43.4).

Example 43.3: Implementation of `GetThreadIndices` in a worklet superclass.

```
1  VTKM_SUPPRESS_EXEC_WARNINGS
2  template<typename OutToInPortalType,
3           typename VisitPortalType,
4           typename ThreadToOutType,
5           typename InputDomainType>
6  VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
7    vtkm::Id threadIndex,
8    const OutToInPortalType& outToIn,
9    const VisitPortalType& visit,
10   const ThreadToOutType& threadToOut,
11   const InputDomainType& inputPoints) const
12 {
13   vtkm::Id outputIndex = threadToOut.Get(threadIndex);
14   vtkm::Id inputIndex = outToIn.Get(outputIndex);
15   vtkm::IdComponent visitIndex = visit.Get(outputIndex);
16   return vtkm::exec::arg::ThreadIndicesLineFractal(
17     threadIndex, inputIndex, visitIndex, outputIndex, inputPoints);
18 }
```

As we can see in Example 43.3, our new worklet type needs a custom thread indices class. Specifically, we want the thread indices class to manage the coordinate information of the input line segment.

Did you know?

The implementation of a thread indices object we demonstrate here stores point coordinate information in addition to actual indices. It is acceptable for a thread indices object to store data that are not strictly indices. That said, the thread indices object should only load data (index or not) that is almost certain to

be used by any worklet implementation. The thread indices object is created before any time that the worklet operator is called. If the thread indices object loads data that is never used by a worklet, that is a waste.

An implementation of a thread indices object usually derives from `vtkm::exec::arg::ThreadIndicesBasic` (or some other existing thread indices class) and adds to it information specific to a particular worklet type.

Example 43.4: Implementation of a thread indices class.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

class ThreadIndicesLineFractal : public vtkm::exec::arg::ThreadIndicesBasic
{
  using Superclass = vtkm::exec::arg::ThreadIndicesBasic;

public:
  using CoordinateType = vtkm::Vec2f;

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename InputPointPortal>
  VTKM_EXEC ThreadIndicesLineFractal(vtkm::Id threadIndex,
                                     vtkm::Id inputIndex,
                                     vtkm::IdComponent visitIndex,
                                     vtkm::Id outputIndex,
                                     const InputPointPortal& inputPoints)
    : Superclass(threadIndex, inputIndex, visitIndex, outputIndex)
  {
    this->Point0 = inputPoints.Get(this->GetInputIndex())[0];
    this->Point1 = inputPoints.Get(this->GetInputIndex())[1];
  }

  VTKM_EXEC
  const CoordinateType& GetPoint0() const { return this->Point0; }

  VTKM_EXEC
  const CoordinateType& GetPoint1() const { return this->Point1; }

private:
  CoordinateType Point0;
  CoordinateType Point1;
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

43.3 Signature Tags

It is common that when defining a new worklet type, the new worklet type is associated with new types of data. Thus, it is common that implementing new worklet types involves defining custom tags for `ControlSignature`s and `ExecutionSignature`s. This in turn typically requires creating custom `TypeCheck`, `Transport`, and `Fetch` classes.

Chapter 42 describes in detail the process of defining new worklet types and the associated code to manage data from a `vtkm::cont::Invoker` argument to the data that are passed to the worklet operator. Rather than repeat

the discussion, readers should review Chapter 42 for details on how custom arguments are defined for a new worklet type. In particular, we use the code from Examples 42.2 (page 371), 42.4 (page 373), and 42.5 (page 376) to implement an argument representing 2D line segments (which is our input domain). All these examples culminate in the definition of a `ControlSignature` tag in our worklet superclass.

Example 43.5: Custom `ControlSignature` tag for the input domain of our example worklet type.

```
struct SegmentsIn : vtkm::cont::arg::ControlSignatureTagBase
{
  using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
  using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
  using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
};
```

As you have worked with different existing worklet types, you have likely noticed that different worklet types have special `ExecutionSignature` tags to point to information in the input domain. For example, a point to cell topology map has special `ExecutionSignature` tags for getting the input cell shape and the indices to all points incident on the current input cell. We described in the beginning of the chapter that we wanted our worklet type to provide worklet implementations an object named `LineFractalTransform` (Example 43.1), so it makes sense to define our own custom `ExecutionSignature` tag to provide this object.

Chapter 42 gives an example of a custom `ExecutionSignature` tag that modifies what information is fetched from an argument (Examples 42.6 and 42.9). However, `ExecutionSignature` tags that only pull data from input domain behave a little differently because they only get information from the thread indices object and ignore the associated data object. This is done by providing a partial specialization of `vtkm::exec::arg::Fetch` that specializes on the aspect tag but not on the fetch tag.

Example 43.6: A `Fetch` for an aspect that does not depend on any control argument.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct AspectTagLineFractalTransform
{
};

template<typename FetchTag, typename ExecObjectType>
struct Fetch<FetchTag,
             vtkm::exec::arg::AspectTagLineFractalTransform,
             ExecObjectType>
{
  using ValueType = LineFractalTransform;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC
  ValueType Load(const vtkm::exec::arg::ThreadIndicesLineFractal& indices,
                 const ExecObjectType&) const
  {
    return ValueType(indices.GetPoint0(), indices.GetPoint1());
  }

  VTKM_EXEC
  void Store(const vtkm::exec::arg::ThreadIndicesLineFractal&,
             const ExecObjectType&,
             const ValueType&) const
  {
    // Store is a no-op for this fetch.
  }
```

```
};

} // namespace arg
} // namespace exec
} // namespace vtkm
```

The definition of an associated `ExecutionSignature` tag simply has to use the define aspect as its `AspectTag`. The tag also has to define a `INDEX` member (which is required of all `ExecutionSignature` tags). This is problematic as this execution argument does not depend on any particular control argument. Thus, it is customary to simply set the `INDEX` to 1. There is guaranteed to be at least one `ControlSignature` argument for any worklet implementation. Thus, the first argument is sure to exist and can then be ignored.

Example 43.7: Custom `ExecutionSignature` tag that only relies on input domain information in the thread indices.

```
  struct Transform : vtkm::exec::arg::ExecutionSignatureTagBase
  {
    static const vtkm::IdComponent INDEX = 1;
    using AspectTag = vtkm::exec::arg::AspectTagLineFractalTransform;
  };
```

So far we have discussed how to get input line segments into our worklet. We also need a `ControlSignature` tag to represent the output line segments created by instances of our worklet. The motivating example has each worklet outputting a fixed number (greater than 1) of line segments for each input line segment. To manage this, we will define another `ControlSignature` tag that outputs these line segments (as two `Vec`-2 coordinates). This is defined as a `Vec` of `Vec`-2's. The tag takes the number of line segments as a template argument.

Example 43.8: Output `ControlSignature` tag for our motivating example.

```
  template<vtkm::IdComponent NumSegments>
  struct SegmentsOut : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumSegments>;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
  };
```

You can see that the tag in Example 43.8 relies on a custom transport named `TransportTag2DLineSegmentsOut`. There is nothing particularly special about this transport, but we provide the implementation here for completeness.

Example 43.9: Implementation of `Transport` for the output in our motivating example.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

template<vtkm::IdComponent NumOutputPerInput>
struct TransportTag2DLineSegmentsOut
{
};

template<vtkm::IdComponent NumOutputPerInput,
         typename ContObjectType,
         typename Device>
struct Transport<vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumOutputPerInput>,
                 ContObjectType,
                 Device>
{
```

```
  VTKM_IS_ARRAY_HANDLE(ContObjectType);

  using GroupedArrayType = vtkm::cont::ArrayHandleGroupVec<
    vtkm::cont::ArrayHandleGroupVec<ContObjectType, 2>,
    NumOutputPerInput>;

  using ExecObjectType = typename GroupedArrayType::WritePortalType;

  template<typename InputDomainType>
  VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
                                      const InputDomainType&,
                                      vtkm::Id,
                                      vtkm::Id outputRange,
                                      vtkm::cont::Token& token) const
  {
    GroupedArrayType groupedArray(vtkm::cont::make_ArrayHandleGroupVec<2>(object));
    return groupedArray.PrepareForOutput(outputRange, Device{}, token);
  }
};

} // namespace arg
} // namespace cont
} // namespace vtkm
```

In addition to these special `ControlSignature` tags that are specific to the nature of our worklet type, it is common to need to replicate some more common or general `ControlSignature` tags. One such tag, which is appropriate for our worklet type, is a "field" type that takes an array with exactly one value associated with each input or output element. We can build these field tags using existing type checks, transports, and fetches. The following example defines a `FieldIn` tag for our fractal worklet type. A `FieldOut` tag can be made in a similar manner.

Example 43.10: Implementing a `FieldIn` tag.

```
  struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArrayIn;
    using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
  };
```

43.4 Worklet Superclass

The penultimate step in defining a new worklet type is to define a class that will serve as the superclass of all implementations of worklets of this type. This class itself must inherit from `vtkm::worklet::internal::WorkletBase`. By convention the worklet superclass is placed in the `vtkm::worklet` namespace and its name starts with `Worklet`.

Within the worklet superclass we define the signature tags (as discussed in Section 43.3) and the `GetThreadIndices` method (as discussed in Section 43.2. The worklet superclass can also override other default behavior of the `WorkletBase` (such as special scatter). And the worklet superclass can provide other items that might be particularly useful to its subclasses (such as commonly used tags). Also, the worklet superclass must declare a `Dispatcher` template that points to a *dispatcher* object used to invoke the worklet. The dispatcher is created in Section 43.5.

Example 43.11: Superclass for a new type of worklet.

```
namespace vtkm
{
```

```
3  namespace worklet
4  {
5
6  template<typename WorkletType>
7  class DispatcherLineFractal;
8
9  class WorkletLineFractal : public vtkm::worklet::internal::WorkletBase
10 {
11 public:
12   /// The dispatcher used to invoke worklets of this type.
13   ///
14   template<typename Worklet>
15   using Dispatcher = vtkm::worklet::DispatcherLineFractal<Worklet>;
16
17   /// Control signature tag for line segments in the plane. Used as the input
18   /// domain.
19   ///
20   struct SegmentsIn : vtkm::cont::arg::ControlSignatureTagBase
21   {
22     using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
23     using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
24     using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
25   };
26
27   /// Control signature tag for a group of output line segments. The template
28   /// argument specifies how many line segments are outputted for each input.
29   /// The type is a Vec-like (of size NumSegments) of Vec-2's.
30   ///
31   template<vtkm::IdComponent NumSegments>
32   struct SegmentsOut : vtkm::cont::arg::ControlSignatureTagBase
33   {
34     using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
35     using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsOut<NumSegments>;
36     using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
37   };
38
39   /// Control signature tag for input fields. There is one entry per input line
40   /// segment. This tag takes a template argument that is a type list tag that
41   /// limits the possible value types in the array.
42   ///
43   struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
44   {
45     using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArrayIn;
46     using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
47     using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
48   };
49
50   /// Control signature tag for input fields. There is one entry per input line
51   /// segment. This tag takes a template argument that is a type list tag that
52   /// limits the possible value types in the array.
53   ///
54   struct FieldOut : vtkm::cont::arg::ControlSignatureTagBase
55   {
56     using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArrayOut;
57     using TransportTag = vtkm::cont::arg::TransportTagArrayOut;
58     using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
59   };
60
61   /// Execution signature tag for a LineFractalTransform from the input.
62   ///
63   struct Transform : vtkm::exec::arg::ExecutionSignatureTagBase
64   {
65     static const vtkm::IdComponent INDEX = 1;
66     using AspectTag = vtkm::exec::arg::AspectTagLineFractalTransform;
```

```
67    };
68
69    VTKM_SUPPRESS_EXEC_WARNINGS
70    template<typename OutToInPortalType,
71             typename VisitPortalType,
72             typename ThreadToOutType,
73             typename InputDomainType>
74    VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
75      vtkm::Id threadIndex,
76      const OutToInPortalType& outToIn,
77      const VisitPortalType& visit,
78      const ThreadToOutType& threadToOut,
79      const InputDomainType& inputPoints) const
80    {
81      vtkm::Id outputIndex = threadToOut.Get(threadIndex);
82      vtkm::Id inputIndex = outToIn.Get(outputIndex);
83      vtkm::IdComponent visitIndex = visit.Get(outputIndex);
84      return vtkm::exec::arg::ThreadIndicesLineFractal(
85        threadIndex, inputIndex, visitIndex, outputIndex, inputPoints);
86    }
87  };
88
89  } // namespace worklet
90  } // namespace vtkm
```

Common Errors

Be wary of creating worklet superclasses that are templated. The C++ compiler rules for superclass templates that are only partially specialized are non-intuitive. If a subclass does not fully resolve the template, features of the superclass such as signature tags will have to be qualified with `typename` *keywords, which reduces the usability of the class.*

43.5 Dispatcher

Worklets are instantiated in the control environment and run in the execution environment. This means that the control environment must have a means to *invoke* worklets that start running in the execution environment. This is ostensibly done by the `vtkm::cont::Invoker` object, but the `Invoker` does this through a *dispatcher* object.

A dispatcher object is an object in the control environment that has an instance of a worklet and can invoke that worklet with a set of arguments. There are multiple types of dispatcher objects, each corresponding to a type of worklet object. All dispatcher objects have at least one template parameter: the worklet class being invoked, which is always the first argument. All dispatcher objects must be constructed with an instance of the worklet they are to invoke.

All dispatcher classes have a method named `Invoke` that launches the worklet in the execution environment. The arguments to `Invoke` must match those expected by the worklet, which is specified by something called a *control signature*.

The following is a list of the dispatchers defined in VTK-m. The dispatcher classes correspond to the list of worklet types specified in Chapter 21.

`vtkm::worklet::DispatcherMapField` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapField`. The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherMapTopology` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletMapTopology` or one of its sibling classes (such as `vtkm::worklet::WorkletVisitCellsWithPoints`). The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherPointNeighborhood` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletPointNeighborhood`. The dispatcher class has one template argument: the worklet type.

`vtkm::worklet::DispatcherReduceByKey` The dispatcher used in conjunction with a worklet that subclasses `vtkm::worklet::WorkletReduceByKey`. The dispatcher class has one template argument: the worklet type.

The final element required for a new worklet type is an associated dispatcher class for invoking the worklet.

Example 43.12: Standard template arguments for a dispatcher class.

```
1 template<typename WorkletType>
2 class DispatcherLineFractal
```

A dispatcher implementation inherits from `vtkm::worklet::internal::DispatcherBase`. `DispatcherBase` is itself a templated class with the following three templated arguments.

1. The dispatcher class that is subclassing `DispatcherBase`. All template arguments must be given.
2. The type of the worklet being dispatched (which by convention is the first argument of the dispatcher's template).
3. The expected superclass of the worklet, which is associated with the dispatcher implementation. `DispatcherBase` will check that the worklet has the appropriate superclass and provide a compile error if there is a mismatch.

Did you know?

The convention of having a subclass be templated on the derived class' type is known as the Curiously Recurring Template Pattern (CRTP). In the case of `DispatcherBase`*, VTK-m uses this CRTP behavior to allow the general implementation of* `Invoke` *to run* `DoInvoke` *in the subclass, which as we see in a moment is itself templated.*

Example 43.13: Subclassing `DispatcherBase`.

```
1 template<typename WorkletType>
2 class DispatcherLineFractal
3   : public vtkm::worklet::internal::DispatcherBase<
4       DispatcherLineFractal<WorkletType>,
5       WorkletType,
6       vtkm::worklet::WorkletLineFractal>
```

The dispatcher should have two constructors. The first constructor takes a worklet and a dispatcher. Both arguments should have a default value that is a new object created with its default constructor. It is good practice to put a warning on this constructor letting users know if they get a compile error there it is probably because the worklet or dispatcher does not have a default constructor and they need to provide one. The second constructor just takes a dispatcher.

Example 43.14: Typical constructor for a dispatcher.

```
// If you get a compile error here about there being no appropriate constructor
// for ScatterType, then that probably means that the worklet you are trying to
// execute has defined a custom ScatterType and that you need to create one
// (because there is no default way to construct the scatter). By convention,
// worklets that define a custom scatter type usually provide a static method
// named MakeScatter that constructs a scatter object.
VTKM_CONT
DispatcherLineFractal(const WorkletType& worklet = WorkletType(),
                      const ScatterType& scatter = ScatterType())
  : Superclass(worklet, scatter)
{
}

VTKM_CONT
DispatcherLineFractal(const ScatterType& scatter)
  : Superclass(WorkletType(), scatter)
{
}
```

Finally, the dispatcher must implement a const method named `DoInvoke`. The `DoInvoke` method should take a single argument. The argument will be an object of type `vtkm::internal::Invocation` although it is usually more convenient to just express the argument type as a single template parameter. The `Invocation` could contain several data items, so it is best to pass this argument as a constant reference.

Example 43.15: Declaration of `DoInvoke` of a dispatcher.

```
template<typename Invocation>
VTKM_CONT void DoInvoke(Invocation& invocation) const
```

`Invocation` is an object that encapsulates the state and data relevant to the invoke. `Invocation` contains multiple types and data items. For brevity only the ones most likely to be used in a `DoInvoke` method are documented here. We discuss these briefly before getting back to the implementation of `DoInvoke`.

`vtkm::internal::Invocation` contains a data member named `Parameters` that contains the data passed to the `Invoke` method of the dispatcher (with some possible transformations applied). `Parameters` is stored in a `vtkm::internal::FunctionInterface` template object. (`FunctionInterface` is described in Chapter 41.) The specific type of `Parameters` is defined as type `ParameterInterface` in the `Invoke` object.

The `Invoke` object also contains the types `ControlInterface` and `ExecutionInterface` that are `FunctionInterface` classes built from the `ControlSignature` and `ExecutionSignature` of the worklet. These `FunctionInterface` classes provide a simple mechanism for introspecting the arguments of the worklet's signatures.

All worklets must also define an input domain index, which points to one of the `ControlSignature`/`Invoke` arguments. This number is also captured in the `vtkm::internal::Invocation` object in a field named `InputDomainIndex`. For convenience, `Invocation` also has the type `InputDomainTag` set to be the same as the `ControlSignature` argument corresponding to the input domain. Likewise, `Invocation` has the type `InputDomainType` set to be the same type as the (transformed) input domain argument to `Invoke`. `Invocation` also has a method name `GetInputDomain` that returns the invocation object passed to `Invoke`.

Getting back to the implementation of a dispatcher, the `DoInvoke` should first verify that the `ControlSignature` argument associated with the input domain is of the expected type. This can be done by comparing the `Invocation::InputDomainTag` with the expected signature tag using a tool like `std::is_same`. This step is not strictly necessary, but is invaluable to users diagnosing issues with using the dispatcher. It does not hurt to also check that the `Invoke` argument for the input domain is also the same as expected (by checking `Invocation::InputDomainType`). It is additionally helpful to have a descriptive comment near these checks.

Example 43.16: Checking the input domain tag and type.

```
    // Get the control signature tag for the input domain.
```

```
    using InputDomainTag = typename Invocation::InputDomainTag;

    // If you get a compile error on this line, then you have set the input
    // domain to something that is not a SegmentsIn parameter, which is not
    // valid.
    VTKM_STATIC_ASSERT(
      (std::is_same<InputDomainTag,
                    vtkm::worklet::WorkletLineFractal::SegmentsIn>::value));

    // This is the type for the input domain
    using InputDomainType = typename Invocation::InputDomainType;

    // If you get a compile error on this line, then you have tried to use
    // something that is not a vtkm::cont::ArrayHandle as the input domain to a
    // topology operation (that operates on a cell set connection domain).
    VTKM_IS_ARRAY_HANDLE(InputDomainType);
```

Next, `DoInvoke` must determine the size in number of elements of the input domain. When the default identity scatter is used, the input domain size corresponds to the number of instances the worklet is executed. (Other scatters will transform the input domain size to an output domain size, and that output domain size will determine the number of instances.) The input domain size is generally determined by using `Invocation::::GetInputDomain` and querying the input domain argument. In our motivating example, the input domain is an `ArrayHandle` and the input domain size is half the size of the array (since array entries are paired up into line segments).

The final thing `DoInvoke` does is call `BasicInvoke` on its `DispatcherBase` superclass. `BasicInvoke` does the complicated work of transferring arguments, scheduling the parallel job, and calling the worklet's operator. `BasicInvoke` takes three arguments: the `Invocation` object, the size of the input domain, and the device adapter tag to run on.

Example 43.17: Calling `BasicInvoke` from a dispatcher's `DoInvoke`.

```
    // We can pull the input domain parameter (the data specifying the input
    // domain) from the invocation object.
    const InputDomainType& inputDomain = invocation.GetInputDomain();

    // Now that we have the input domain, we can extract the range of the
    // scheduling and call BasicInvoke.
    this->BasicInvoke(invocation, inputDomain.GetNumberOfValues() / 2);
```

Putting this all together, the following example demonstrates the full implementation of the dispatcher for our motivating example.

Example 43.18: Implementation of a dispatcher for a new type of worklet.

```
namespace vtkm
{
namespace worklet
{

template<typename WorkletType>
class DispatcherLineFractal
  : public vtkm::worklet::internal::DispatcherBase<
      DispatcherLineFractal<WorkletType>,
      WorkletType,
      vtkm::worklet::WorkletLineFractal>
{
  using Superclass =
    vtkm::worklet::internal::DispatcherBase<DispatcherLineFractal<WorkletType>,
                                            WorkletType,
                                            vtkm::worklet::WorkletLineFractal>;
  using ScatterType = typename Superclass::ScatterType;

public:
```

```
20    // If you get a compile error here about there being no appropriate constructor
21    // for ScatterType, then that probably means that the worklet you are trying to
22    // execute has defined a custom ScatterType and that you need to create one
23    // (because there is no default way to construct the scatter). By convention,
24    // worklets that define a custom scatter type usually provide a static method
25    // named MakeScatter that constructs a scatter object.
26    VTKM_CONT
27    DispatcherLineFractal(const WorkletType& worklet = WorkletType(),
28                          const ScatterType& scatter = ScatterType())
29      : Superclass(worklet, scatter)
30    {
31    }
32
33    VTKM_CONT
34    DispatcherLineFractal(const ScatterType& scatter)
35      : Superclass(WorkletType(), scatter)
36    {
37    }
38
39    template<typename Invocation>
40    VTKM_CONT void DoInvoke(Invocation& invocation) const
41    {
42      // Get the control signature tag for the input domain.
43      using InputDomainTag = typename Invocation::InputDomainTag;
44
45      // If you get a compile error on this line, then you have set the input
46      // domain to something that is not a SegmentsIn parameter, which is not
47      // valid.
48      VTKM_STATIC_ASSERT(
49        (std::is_same<InputDomainTag,
50                      vtkm::worklet::WorkletLineFractal::SegmentsIn>::value));
51
52      // This is the type for the input domain
53      using InputDomainType = typename Invocation::InputDomainType;
54
55      // If you get a compile error on this line, then you have tried to use
56      // something that is not a vtkm::cont::ArrayHandle as the input domain to a
57      // topology operation (that operates on a cell set connection domain).
58      VTKM_IS_ARRAY_HANDLE(InputDomainType);
59
60      // We can pull the input domain parameter (the data specifying the input
61      // domain) from the invocation object.
62      const InputDomainType& inputDomain = invocation.GetInputDomain();
63
64      // Now that we have the input domain, we can extract the range of the
65      // scheduling and call BasicInvoke.
66      this->BasicInvoke(invocation, inputDomain.GetNumberOfValues() / 2);
67    }
68  };
69
70  } // namespace worklet
71  } // namespace vtkm
```

43.6 Using the Worklet

Now that we have our full implementation of a worklet type that generates line fractals, let us have some fun with it. The beginning of this chapter shows an implementation of the Koch Snowflake. The remainder of this chapter demonstrates other fractals that are easily implemented with our worklet type.

43.6.1 Quadratic Type 2 Curve

There are multiple variants of the Koch Snowflake. One simple but interesting version is the quadratic type 1 curve. This fractal has a shape similar to what we used for Koch but has right angles and goes both up and down as shown in Figure 43.5.

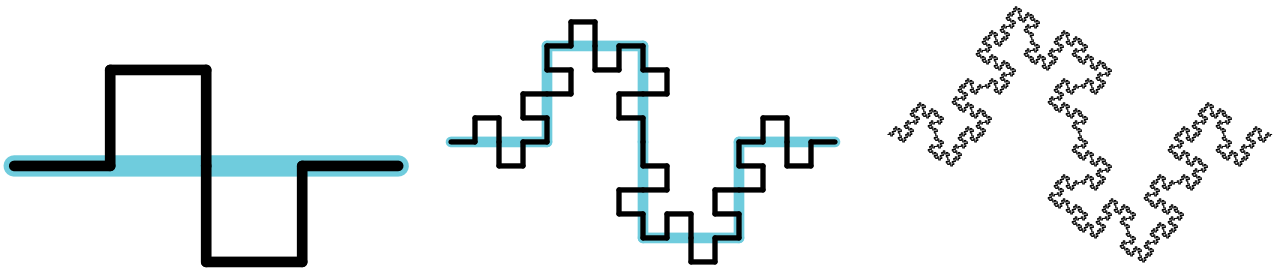


Figure 43.5: The quadratic type 2 curve fractal. The left image gives the first iteration. The middle image gives the second iteration. The right image gives the result after a few iterations.

The quadratic type 2 curve is implemented exactly like the Koch Snowflake except we output 8 lines to every input instead of 4, and, of course, the positions of the lines we generate are different.

Example 43.19: A worklet to generate a quadratic type 2 curve fractal.

```
struct QuadraticType2
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<8>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.00f, 0.00f);
      segmentsOutVec[0][1] = transform(0.25f, 0.00f);

      segmentsOutVec[1][0] = transform(0.25f, 0.00f);
      segmentsOutVec[1][1] = transform(0.25f, 0.25f);

      segmentsOutVec[2][0] = transform(0.25f, 0.25f);
      segmentsOutVec[2][1] = transform(0.50f, 0.25f);

      segmentsOutVec[3][0] = transform(0.50f, 0.25f);
      segmentsOutVec[3][1] = transform(0.50f, 0.00f);

      segmentsOutVec[4][0] = transform(0.50f, 0.00f);
      segmentsOutVec[4][1] = transform(0.50f, -0.25f);

      segmentsOutVec[5][0] = transform(0.50f, -0.25f);
      segmentsOutVec[5][1] = transform(0.75f, -0.25f);

      segmentsOutVec[6][0] = transform(0.75f, -0.25f);
      segmentsOutVec[6][1] = transform(0.75f, 0.00f);

      segmentsOutVec[7][0] = transform(0.75f, 0.00f);
      segmentsOutVec[7][1] = transform(1.00f, 0.00f);
    }
```

```
37    };
38
39    VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
40      vtkm::IdComponent numIterations)
41    {
42      vtkm::cont::ArrayHandle<vtkm::Vec2f> points;
43
44      // Initialize points array with a single line
45      points.Allocate(2);
46      points.WritePortal().Set(0, vtkm::Vec2f(0.0f, 0.0f));
47      points.WritePortal().Set(1, vtkm::Vec2f(1.0f, 0.0f));
48
49      vtkm::cont::Invoker invoke;
50      QuadraticType2::FractalWorklet worklet;
51
52      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
53      {
54        vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
55        invoke(worklet, points, outPoints);
56        points = outPoints;
57      }
58
59      return points;
60    }
61  };
```

43.6.2 Tree Fractal

Another type of fractal we can make is a tree fractal. We will make a fractal similar to a Pythagoras tree except using lines instead of squares. Our fractal will start with a vertical line that will be replaced with the off-center "Y" shape shown in Figure 43.6. Iterative replacing using this "Y" shape produces a bushy tree shape.

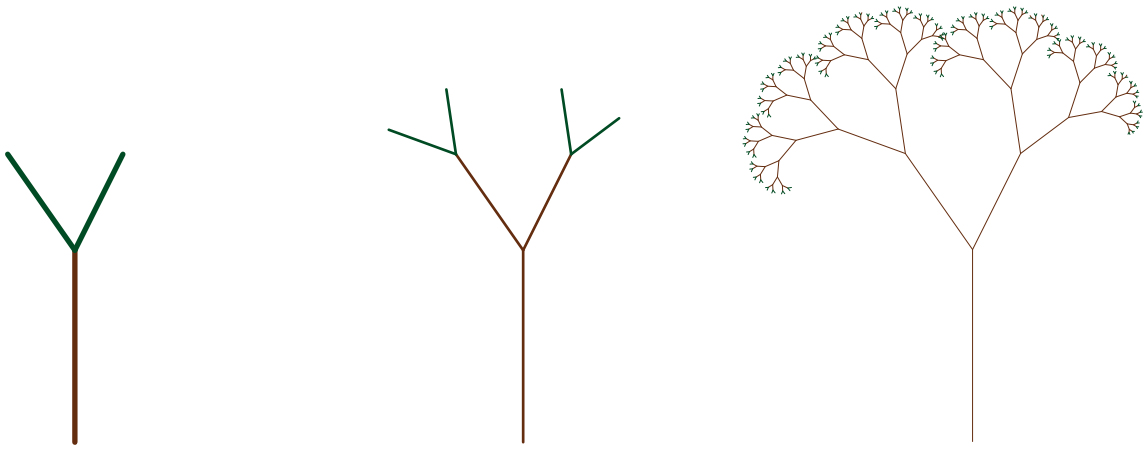


Figure 43.6: The tree fractal replaces each line with the "Y" shape shown at left. An iteration grows branches at the end (middle). After several iterations the tree branches out to the bushy shape at right.

One complication of implementing this tree fractal is that we really only want to apply the "Y" shape to the "leaves" of the tree. For example, once we apply the "Y" to the trunk, we do not want to apply it to the trunk again. If we were to apply it to the trunk again, we would create duplicates of the first layer of branches.

We can implement this feature in our worklet by using a count scatter. (Worklet scatters are described in Section 31.1.) Instead of directing the fractal worklet to generate 3 output line segments for every input line

segment, we tell the fractal worklet to generate just 1 output line segment. We then use a scatter counting to generate 3 line segments for the leaves and 1 line segment for all other line segments. The count array for the initial iteration is initialized to a single 3. Each iteration then creates the count array for the next iteration by writing a 1 for the base line segment and a 3 from the other two line segments.

Example 43.20: A worklet to generate a tree fractal.

```
struct TreeFractal
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn,
                                  SegmentsOut<1>,
                                  FieldOut countNextIteration);
    using ExecutionSignature = void(Transform, VisitIndex, _2, _3);
    using InputDomain = _1;

    using ScatterType = vtkm::worklet::ScatterCounting;

    template<typename Storage>
    VTKM_CONT static ScatterType MakeScatter(
      const vtkm::cont::ArrayHandle<vtkm::IdComponent, Storage>& count)
    {
      return ScatterType(count);
    }

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    vtkm::IdComponent visitIndex,
                    SegmentsOutVecType& segmentsOutVec,
                    vtkm::IdComponent& countNextIteration) const
    {
      switch (visitIndex)
      {
        case 0:
          segmentsOutVec[0][0] = transform(0.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.0f, 0.0f);
          countNextIteration = 1;
          break;
        case 1:
          segmentsOutVec[0][0] = transform(1.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.5f, -0.25f);
          countNextIteration = 3;
          break;
        case 2:
          segmentsOutVec[0][0] = transform(1.0f, 0.0f);
          segmentsOutVec[0][1] = transform(1.5f, 0.35f);
          countNextIteration = 3;
          break;
        default:
          this->RaiseError("Unexpected visit index.");
      }
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
    vtkm::cont::ArrayHandle<vtkm::Vec2f> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.WritePortal().Set(0, vtkm::Vec2f(0.0f, 0.0f));
    points.WritePortal().Set(1, vtkm::Vec2f(0.0f, 1.0f));
```

```
58
59      vtkm::cont::ArrayHandle<vtkm::IdComponent> count;
60
61      // Initialize count array with 3 (meaning iterate)
62      count.Allocate(1);
63      count.WritePortal().Set(0, 3);
64
65      vtkm::cont::Invoker invoke;
66      TreeFractal::FractalWorklet worklet;
67
68      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
69      {
70        auto scatter = TreeFractal::FractalWorklet::MakeScatter(count);
71        vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
72        invoke(worklet, scatter, points, outPoints, count);
73        points = outPoints;
74      }
75
76      return points;
77    }
78  };
```

43.6.3 Dragon Fractal

The next fractal we will implement is known as the dragon fractal. The dragon fractal is also sometimes known as the Heighway dragon or the Harter-Heighway dragon after creators John Heighway, Bruce Banks, and William Harter. It is also sometimes colloquially referred to as the Jurassic Park dragon as the fractal was prominently featured in the *Jurassic Park* novel by Michael Crichton.

The basic building block is simple. Each line segment is replaced by two line segments bent at 90 degrees and attached to the original segments endpoints as shown in Figure 43.7. As you can see by the fourth iteration a more complicated pattern starts to emerge. Figure 43.8 shows the twelfth iteration a demonstrates a repeating spiral.

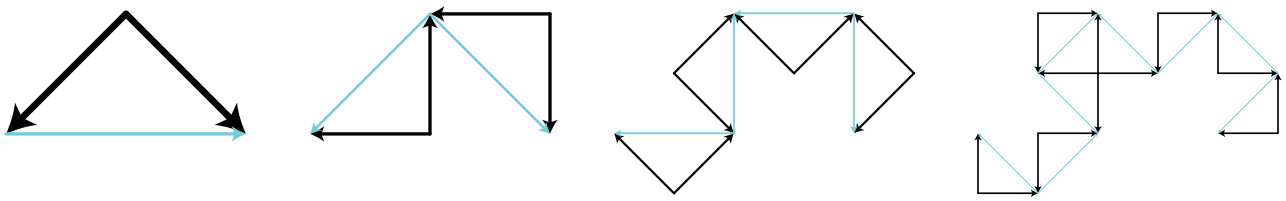


Figure 43.7: The first four iterations of the dragon fractal. The cyan lines give the previous iteration for reference.

What makes the dragon fractal different than the Koch Snowflake and similar fractals like the the quadratic curves implementation-wise is that the direction shape flips from one side to another. Note in the second image of Figure 43.7 the first bend is under the its associated line segment whereas the second is above its line segment. The easiest way for us to control the bend is to alternate the direction of the line segments. In Figure 43.7 each line segment has an arrowhead indicating the orientation of the first and second point with the arrowhead at the second point. Note that the shape is defined such that the first point of both line segments meet at the right angle. With the shape defined this way, each iteration is applied to put the bend to the left of the segment with respect to an observer at the first point looking at the second point.

Other than reversing the direction of half the line segments, the implementation of the dragon fractal is nearly identical to the Koch Snowflake.

Figure 43.8: The dragon fractal after 12 iterations.

Example 43.21: A worklet to generate the dragon fractal.

```
struct DragonFractal
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn, SegmentsOut<2>);
    using ExecutionSignature = void(Transform, _2);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    SegmentsOutVecType& segmentsOutVec) const
    {
      segmentsOutVec[0][0] = transform(0.5f, 0.5f);
      segmentsOutVec[0][1] = transform(0.0f, 0.0f);

      segmentsOutVec[1][0] = transform(0.5f, 0.5f);
      segmentsOutVec[1][1] = transform(1.0f, 0.0f);
    }
  };

  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
    vtkm::IdComponent numIterations)
  {
```

```
24      vtkm::cont::ArrayHandle<vtkm::Vec2f> points;
25
26      // Initialize points array with a single line
27      points.Allocate(2);
28      points.WritePortal().Set(0, vtkm::Vec2f(0.0f, 0.0f));
29      points.WritePortal().Set(1, vtkm::Vec2f(1.0f, 0.0f));
30
31      vtkm::cont::Invoker invoke;
32      DragonFractal::FractalWorklet worklet;
33
34      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
35      {
36        vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
37        invoke(worklet, points, outPoints);
38        points = outPoints;
39      }
40
41      return points;
42    }
43  };
```

43.6.4 Hilbert Curve

For our final example we will look into using our fractal worklet to construct a space-filling curve. A space-filling curve is a type of fractal that defines a curve that, when iterated to its infinite length, completely fills a space. Space-filling curves have several practical uses by allowing you to order points in a 2 dimensional or higher space in a 1 dimensional array in such a way that points close in the higher dimensional space are usually close in the 1 dimensional ordering. For this fractal we will be generating the well-known Hilbert curve. (Specifically, we will be generating the 2D Hilbert curve.)

The 2D Hilbert curve fills in a rectangular region in space. (Our implementation will fill a unit square in the [0,1] range, but a simple scaling can generalize it to any rectangle.) Without loss of generality, we will say that the curve starts in the lower left corner of the region and ends in the lower right corner. The Hilbert curve starts by snaking around the lower-left corner then into the upper-left followed by the upper-right and then lower-right. The curve is typically generated by recursively dividing and orienting these quadrants.

To generate the Hilbert curve in our worklet system, we will define our line segments as the connection from the lower left of (entrance to) the region to the lower right of (exit from) the region. The fractal generation breaks this line to a 4 segment curve that moves up, then right, then back down. Figure 43.9 demonstrates the Hilbert curve. (Readers familiar with the Hilbert curve might notice the shape is a bit different than other representations. Where many derivations derive the Hilbert curve by connecting the center of oriented boxes, our derivation uses a line segment along one edge of these boxes. The result is a more asymmetrical shape in early iterations, but the two approaches are equivalent as the iterations approach infinity.)

Like the dragon fractal, the Hilbert curve needs to flip the shape in different directions. For example, the first iteration, shown at left in Figure 43.9, is drawn to the “left” of the initial line along the horizontal axis. The next iteration, the second image in Figure 43.9, is created by drawing the shape to the “right” of the vertical line segments but to the left of the horizontal segments.

Section 43.6.3 solved this problem for the dragon fractal by flipping the direction of some of the line segments. Such an approach would work for the Hilbert curve, but it results in line segments being listed out of order and with inconsistent directions with respect to the curve. For the dragon fractal, the order and orientation of line segments is of little consequence. But for many applications of a space-filling curve the distance along the curve is the whole point, so we want the order of the line segments to be consistent with the curve.

To support this flipped shape while preserving the line segment order, we will use a data field attached to the

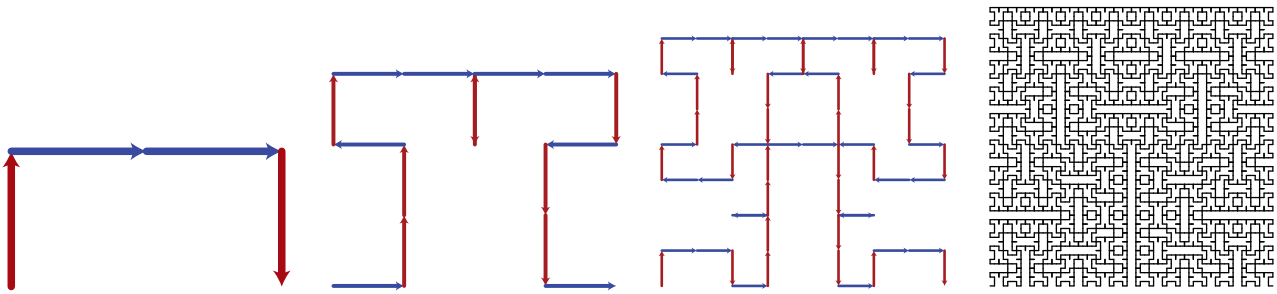


Figure 43.9: The first, second, third, and sixth iterations, respectively, of the Hilbert curve fractal.

line segments. That is, each line segment will have a value to represent which way to draw the shape. If the field value is set to 1 (represented by the blue line segments in Figure 43.9), then the shape is drawn to the "left." If the field value is set to -1 (represented by the red line segments in Figure 43.9), then the shape is inverted and drawn to the "right." This field is passed in and out of the worklet using the `FieldIn` and `FieldOut` tags.

Example 43.22: A worklet to generate the Hilbert curve.

```
1  struct HilbertCurve
2  {
3    struct FractalWorklet : vtkm::worklet::WorkletLineFractal
4    {
5      using ControlSignature = void(SegmentsIn,
6                                    FieldIn directionIn,
7                                    SegmentsOut<4>,
8                                    FieldOut directionOut);
9      using ExecutionSignature = void(Transform, _2, _3, _4);
10     using InputDomain = _1;
11
12     template<typename SegmentsOutVecType>
13     void operator()(const vtkm::exec::LineFractalTransform& transform,
14                     vtkm::Int8 directionIn,
15                     SegmentsOutVecType& segmentsOutVec,
16                     vtkm::Vec4i_8& directionOut) const
17     {
18       segmentsOutVec[0][0] = transform(0.0f, directionIn * 0.0f);
19       segmentsOutVec[0][1] = transform(0.0f, directionIn * 0.5f);
20       directionOut[0] = -directionIn;
21
22       segmentsOutVec[1][0] = transform(0.0f, directionIn * 0.5f);
23       segmentsOutVec[1][1] = transform(0.5f, directionIn * 0.5f);
24       directionOut[1] = directionIn;
25
26       segmentsOutVec[2][0] = transform(0.5f, directionIn * 0.5f);
27       segmentsOutVec[2][1] = transform(1.0f, directionIn * 0.5f);
28       directionOut[2] = directionIn;
29
30       segmentsOutVec[3][0] = transform(1.0f, directionIn * 0.5f);
31       segmentsOutVec[3][1] = transform(1.0f, directionIn * 0.0f);
32       directionOut[3] = -directionIn;
33     }
34   };
35
36   VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec2f> Run(
37     vtkm::IdComponent numIterations)
38   {
39     vtkm::cont::ArrayHandle<vtkm::Vec2f> points;
40
```

```
41      // Initialize points array with a single line
42      points.Allocate(2);
43      points.WritePortal().Set(0, vtkm::Vec2f(0.0f, 0.0f));
44      points.WritePortal().Set(1, vtkm::Vec2f(1.0f, 0.0f));
45
46      vtkm::cont::ArrayHandle<vtkm::Int8> directions;
47
48      // Initialize direction with positive.
49      directions.Allocate(1);
50      directions.WritePortal().Set(0, 1);
51
52      vtkm::cont::Invoker invoke;
53      HilbertCurve::FractalWorklet worklet;
54
55      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
56      {
57        vtkm::cont::ArrayHandle<vtkm::Vec2f> outPoints;
58        vtkm::cont::ArrayHandle<vtkm::Int8> outDirections;
59        invoke(worklet,
60               points,
61               directions,
62               outPoints,
63               vtkm::cont::make_ArrayHandleGroupVec<4>(outDirections));
64        points = outPoints;
65        directions = outDirections;
66      }
67
68      return points;
69    }
70  };
```

Part VI

Appendix

INDEX