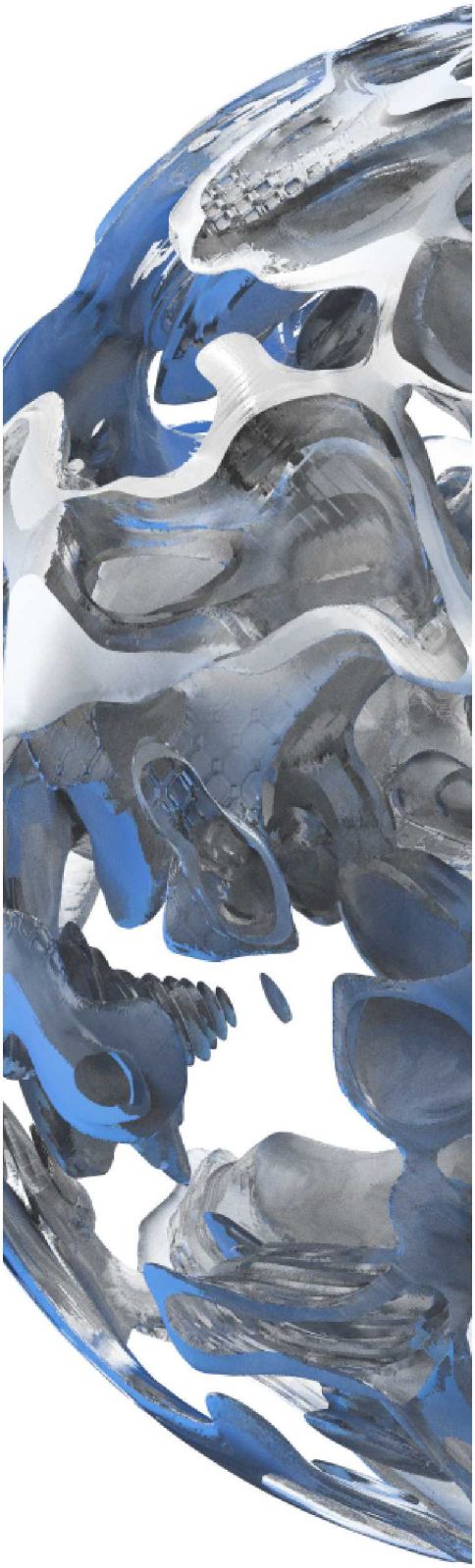
SAND2018-13465B



The VTK-m User's Guide

VTK-m version 1.3

Kenneth Moreland

With contributions from:
Matthew Letter, Robert Maynard, Sujin Philip,
David Pugmire, Allison Vacanti, Abhishek Yenpure,
La-ti Lo, James Kress, and the VTK-m community

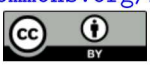
November 26, 2018

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

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY



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

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.
SAND 2018-0475 B

Printed and produced in the United States of America.

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:

Matthew Letter for his help keeping the user's guide up to date with the VTK-m source code.

Sujin Philip, **Robert Maynard**, and **James Kress** for their descriptions of numerous filters.

Allison Vacanti for her documentation of several VTK-m features in Sections 10.2.9 and 10.2.10.

David Pugmire for his documentation of multi-block data sets (Section 12.5) and select filters.

Abhishek Yenpure and **La-ti Lo** for their documentation of locator structures (Chapter 16).

ABOUT THE COVER

The cover image is a visualization of the temperature field computed by the Nek5000 thermal hydraulics simulator. In the simulation twin inlets pump air into a box with a temperature difference between the 2 inlets. The visualization is provided by Matthew Larsen at Lawrence Livermore National Laboratory.

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

The cover design was done by Steve Jordan.

Join the VTK-m Community at 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.

Did you know?

VTK-m is written in C++ and makes extensive use of templates. The toolkit is implemented as a header library, meaning that all the code is implemented in header files (with extension .h) and completely included in any code that uses it. This allows the compiler to inline and specialize code for better performance.

1.1 How to Use This Guide

This user's guide is organized into four parts to help guide novice to advanced users and to provide a convenient reference. Part I, Getting Started, provides everything needed to get up and running with VTK-m. In this part we learn the basics of reading and writing data files, using filters to process data, and performing basic rendering to view the results.

Part II, Using VTK-m, dives deeper into the VTK-m library and provides all the information needed to customize VTK-m's data structures and support multiple devices.

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 III, Developing with VTK-m, documents how to use VTK-m's framework to develop new or custom visualization algorithms. This part describes the concept of a *worklet*, how they are used to implement and execute algorithms, and how to use worklets to implement new filters.

Part IV, Advanced 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 13, 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:

http://m.vtk.org/index.php/VTK-m_Releases

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

The source code in the VTK-m repository is access through the git version control tool. If you have not used git before, there are several resources available to help you get familiar with it. Github has a nice setup guide (https://help.github.com/articles/set-up-git) to help you get up and running quickly. For more complete documentation, we recommend the *Pro Git* book (https://git-scm.com/book).

To get a copy of the VTK-m repository, issue a git clone command.

Example 2.1: Cloning the main VTK-m git repository.

```
1 git clone https://gitlab.kitware.com/vtk/vtk-m.git
```

The git clone command will create a copy of all the source code to your local machine. As time passes and you want to get an update of changes in the repository, you can do that with the git pull command.

Example 2.2: Updating a git repository with the pull command.

```
1 git pull
```

Did you know?

The proceeding examples for using git are based on the git *command line tool, which is particularly prevalent on Unix-based and Mac systems. There also exist several GUI tools for accessing git repositories. These tools each have their own interface and they can be quite different. However, they all should have roughly equivalent commands named "clone" to download a repository given a url and "pull" to update an existing repository.*

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.

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 2.3: Running CMake on a cloned VTK-m repository.

```
1 git clone https://gitlab.kitware.com/vtk/vtk-m.git
2 mkdir vtkm-build
3 cd vtkm-build
4 cmake-gui ../vtk-m
```

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

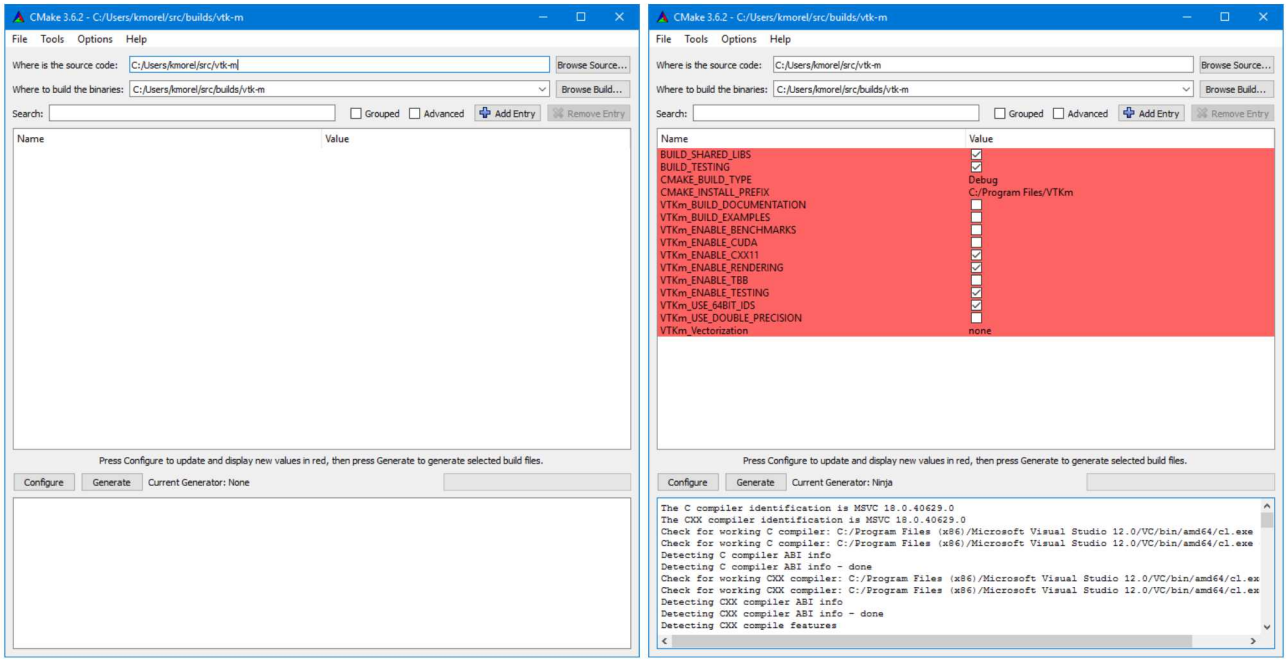


Figure 2.1: The CMake GUI configuring the VTK-m project. At left is the initial blank configuration. At right is the state after a configure pass.

the 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, and volta.

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_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 2.3

Example 2.4: Using make to build VTK-m.

```
1 git clone https://gitlab.kitware.com/vtk/vtk-m.git
2 mkdir vtkm-build
3 cd vtkm-build
4 cmake-gui ../vtk-m
5 make -j
6 make test
7 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 2.4.

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?

A good portion of VTK-m is a header-only library, which does not need to be built in a traditional sense. However, VTK-m contains a significant amount of tests to ensure that the header code does compile and run correctly on a given system. 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-*X.X* directory of the install or build of VTK-m. The configurable CMake variable VTKm_DIR can be set to the directory that contains this file.

Example 2.5: 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 2.6: 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_cont)
```

The following libraries are made available.

vtkm_cont Contains the base objects used to control VTK-m. This library should always be linked in.

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

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_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 2.7: 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

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 Part II we explore efficient ways to define VTK-m data structures. In particular, Section 12.1 describes how to build VTK-m data set objects and Section 10.4 documents how to adapt data structures defined in other libraries to be used directly in VTK-m.

3.1 Readers

All reader classes provided by VTK-m are located in the `vtkm::io::reader` 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. Chapter 12 provides much more details about the contents of a data set object, but for now we treat `DataSet` as an opaque object that can be passed around readers, writers, filters, and rendering units.

3.1.1 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::reader::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.

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

Example 3.1: Reading a legacy VTK file.

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

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

  return reader.ReadDataSet();
}
```

3.2 Writers

All writer classes provided by VTK-m are located in the `vtkm::io::writer` 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.

3.2.1 Legacy VTK File Writer

Legacy VTK files can be written using the `vtkm::io::writer::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 3.2: Writing a legacy VTK file.

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

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

  writer.WriteDataSet(data);
}
```

CHAPTER

FOUR

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 introduced with the file I/O operations in Chapter 3 and are described in more detail in Chapter 12. For now we treat `DataSet` mostly as an opaque object that can be passed around readers, writers, filters, and rendering units.

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.

4.1 Field Filters

Every `vtkm::cont::DataSet` object contains a list of *fields*. A field describes some numerical value associated with different parts of the data set in space. Fields often represent physical properties such as temperature, pressure, or velocity. *Field filters* are a class of filters that generate a new field. These new fields are typically 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.

Before a filter is run, it is important to set up the state of the filter object to the parameters of the algorithm. The state parameters will vary from one filter to the next, but one state parameter that all field filters share is the "active" field for the operation. The active field is set with a call to the `SetActiveField` method. The argument to `SetActiveField` is a string that names this input field. Alternatively, you can call the `SetUseCoordinateSystemAsField` with an argument of `true` to use the point coordinates as the input field rather than a specified field. See Sections 12.3 and 12.4 for more information on fields and coordinate systems, respectively. Finally, `SetOutputFieldName`, specifies the name assigned to the generated field. If not specified, then the filter will use a default name.

All field filters contain an `Execute` method. When calling `Execute` a `vtkm::cont::DataSet` or `vtkm::cont::MultiBlock` object with the input data is provided as an argument. The `Execute` method returns a `DataSet` or

MultiBlock object (matching the type of the input to Execute), which contains the data generated.

The following example provides a simple demonstration of using a field filter. It specifically uses the point elevation filter, which is one of the field filters.

Example 4.1: Using PointElevation, which is a field filter.

```
VTKM_CONT
vtkm::cont::DataSet ComputeAirPressure(vtkm::cont::DataSet dataSet)
{
  vtkm::filter::PointElevation elevationFilter;

  // Use the elevation filter to estimate atmospheric pressure based on the
  // height of the point coordinates. Atmospheric pressure is 101325 Pa at
  // sea level and drops about 12 Pa per meter.
  elevationFilter.SetOutputFieldName("pressure");
  elevationFilter.SetLowPoint(0.0, 0.0, 0.0);
  elevationFilter.SetHighPoint(0.0, 0.0, 2000.0);
  elevationFilter.SetRange(101325.0, 77325.0);

  elevationFilter.SetUseCoordinateSystemAsField(true);

  vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);

  return result;
}
```

4.1.1 Cell Average

vtkm::filter::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.

SetActiveCellSetIndex/GetActiveCellSetIndex Specifies the index for the cell set to use from the data set provided to the Execute method. The default index is 0, which is the first cell set.

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 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 4.4.2 for more details.

SetRuntimeDeviceTracker/GetRuntimeDeviceTracker Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.2 Coordinate System Transforms

VTK-m provides multiple filters to translate between different coordiante systems.

Cylindrical Coordinate System Transform

`vtkm::filter::CylindricalCoordinateSystemTransform` 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 default name for the output field is "cylindricalCoordinateSystemTransform", but that can be overridden as always using the `SetOutputFieldName` method.

In addition the standard `SetOutputFieldName` and `Execute` methods, `CylindricalCoordinateSystemTransform` 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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

Spherical Coordinate System Transform

`vtkm::filter::SphericalCoordinateSystemTransform` 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 default name for the output field is "sphericalCoordinateSystemTransform", but that can be overridden as always using the `SetOutputFieldName` method.

In addition the standard `SetOutputFieldName` and `Execute` methods, `CylindricalCoordinateSystemTransform` 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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.3 Cross Product

`vtkm::filter::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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.4 Dot Product

`vtkm::filter::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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.5 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 5.8.

FieldToColors has three modes it can use to select how it should treat the input field.

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

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

FieldToColors::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.

FieldToColors::RGB Output colors are represented as RGB values with each component represented by an unsigned byte. Specifically, these are vtkm::Vec <vtkm::UInt8,3> values.

FieldToColors::RGBA Output colors are represented as RGBA values with each component represented by an unsigned byte. Specifically, these are vtkm::Vec <vtkm::UInt8,4> 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 SCALAR, MAGNITUDE, or COMPONENT selectors described previously.

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

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

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

IsMappingScalar Returns true if the input mapping mode is scalar (FieldToColors::SCALAR).

IsMappingMagnitude Returns true if the input mapping mode is magnitude (FieldToColors::MAGNITUDE).

IsMappingComponent Returns true if the input mapping mode is component (FieldToColors::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 COMPONENT.

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

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

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

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

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

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.6 Gradients

`vtkm::filter::Gradients` 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 overriden as always using the `SetOutputFieldName` method.

`Gradients` 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".

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.7 Point Average

`vtkm::filter::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.

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

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.8 Point Elevation

`vtkm::filter::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 4.1 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.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.9 Point Transform

`vtkm::filter::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.

In addition the standard `SetOutputFieldName` and `Execute` methods, `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 Z 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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.10 Surface Normals

`vtkm::filter::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 cell normals are computed based on each cell's winding order using vector cross-product. 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.

`SetGeneratePointNormals/GetGeneratePointNormals` Specifies whether the point normals should be generated.

`SetNormalizeCellNormals/GetNormalizeCellNormals` Specifies whether cell normals should be normalized (made unit length). Default value is true. 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.

`SetCellNormalsName/GetCellNormalsName` Specifies the output cell normals field name.

`SetPointNormalsName/GetPointNormalsName` Specifies the output point normals field name.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.1.11 Vector Magnitude

`vtkm::filter::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.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.2 Data Set Filters

Data set filters are a class of filters that generate a new data set with a new topology. This new topology is typically derived from an existing data set. For example, a data set can be significantly altered by adding, removing, or replacing cells.

Before a filter is run, it is important to set up the state of the filter object to the parameters of the algorithm. The state parameters will vary from one filter to the next, but one state parameter that all data set filters share is the "active" cell set for the operation. The active cell set is set with a call to the `SetActiveCellSetIndex`. Likewise, `SetActiveCoordinateSystem` selects which coordinate system to operate on. By default, the filter will operate on the first cell set and coordinate system. (See Sections 12.2 and 12.4 for more information about cell sets and coordinate systems, respectively.)

All data set filters contain an `Execute` method. When calling `Execute` a `vtkm::cont::DataSet` or `vtkm::cont::MultiBlock` object with the input data is provided as an argument. The `Execute` method returns a `DataSet` or `MultiBlock` object (matching the type of the input to `Execute`), which contains the data generated.

The following example provides a simple demonstration of using a data set filter. It specifically uses the vertex clustering filter, which is one of the data set filters.

Example 4.2: Using `VertexClustering`, which is a data set filter.

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

4.2.1 Clean Grid

`vtkm::filter::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 12.2.2.) One benefit of using `CleanGrid` is that it can optionally remove unused points. Another benefit is that the resulting cell set will be of a known specific type.

Common Errors

The result of `vtkm::filter::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. If false, then the output coordinate systems will be a shallow copy of the input coordinate systems.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.2.2 External Faces

vtkm::filter::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.")

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.2.3 Vertex Clustering

vtkm::filter::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. Example 4.2 provides a demonstration of the vertex clustering filter.

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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.3 Data Set with Field Filters

Data set with field filters are a class of filters that generate a new data set with a new topology. This new topology is derived from an existing data set and at least one of the fields in the data set. For example, a field might determine how each cell is culled, clipped, or sliced.

Before a filter is run, it is important to set up the state of the filter object to the parameters of the algorithm. The state parameters will vary from one filter to the next, but one state parameter that all data set with field filters share is the "active" field for the operation. The active field is set with a call to the `SetActiveField` method. The argument to `SetActiveField` is a string that names this input field. Another state parameters all data set with field filters share is the "active" cell set for the operation. The active cell set is set with a call to the `SetActiveCellSetIndex`. Likewise, `SetActiveCoordinateSystem` selects which coordinate system to operate on. By default, the filter will operate on the first cell set and coordinate system. (See Sections 12.2 and 12.4 for more information about cell sets and coordinate systems, respectively.) Finally, `SetOutputFieldName`, specifies the name assigned to the generated field. If not specified, then the filter will use a default name.

All data set with field filters contain an `Execute` method. When calling `Execute` a `vtkm::cont::DataSet` or `vtkm::cont::MultiBlock` object with the input data is provided as an argument. The `Execute` method returns a `DataSet` or `MultiBlock` object (matching the type of the input to `Execute`), which contains the data generated.

The following example provides a simple demonstration of using a data set with field filter. It specifically uses the Marching Cubes filter, which is one of the data set with field filters.

Example 4.3: Using `MarchingCubes`, which is a data set with field filter.

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

4.3.1 Marching Cubes

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*. Marching Cubes is a well know algorithm for computing contours and is implemented by `vtkm::filter::MarchingCubes`. Example 4.3 provides a demonstration of the Marching Cubes filter.

`MarchingCubes` provides the following methods.

`SetIsoValue/GetIsoValue` Specifies the value on which to extract the contour. The contour will be the surface where the field (provided to `Execute`) is equal to this value.

`SetMergeDuplicatePoints/GetMergeDuplicatePoints` Specifies whether coincident points in the data set should be merged. Because the Marching Cubes 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.

`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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.3.2 Threshold

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

`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.

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

`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.

`SetActiveCellSetIndex/GetActiveCellSetIndex` Specifies the index for the cell set to use from the data set provided to the `Execute` method. The default index is 0, which is the first cell set.

`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 4.4.2 for more details.

`SetRuntimeDeviceTracker/GetRuntimeDeviceTracker` Specifies which devices to use during execution. Runtime device trackers are described in Section 8.3.

4.3.3 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.

`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.

Example 4.4: Using `Streamline`, which is a data set with field filter.

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

4.4 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.

4.4.1 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::ASSOC_ANY` Any field regardless of the association. (This is the default if no association is given.)

`vtkm::cont::Field::ASSOC_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::ASSOC_CELL_SET` 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::ASSOC_WHOLE_MESH` 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 4.5: 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 differentiatable 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 12.4 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.)

4.4.2 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 4.6: 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 4.7: 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 4.8: 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 4.9: 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 4.10: Using vtkm::filter::FieldSelection.

```
1 |     vtkm::filter::FieldSelection fieldSelection;
2 |     fieldSelection.AddField("scalars");
3 |     fieldSelection.AddField("cellvar", vtkm::cont::Field::Association::CELL_SET);
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 4.11: Selecting one field and its association for a filter to pass.

```
filter.SetFieldsToPass("pointvar", vtkm::cont::Field::Association::POINTS);
```

Example 4.12: Selecting a list of fields and their associations for a filter to pass.

```
filter.SetFieldsToPass(
  { vtkm::make_Pair("pointvar", vtkm::cont::Field::Association::POINTS),
    vtkm::make_Pair("cellvar", vtkm::cont::Field::Association::CELL_SET),
    vtkm::make_Pair("scalars", vtkm::cont::Field::Association::ANY) });
```

CHAPTER

FIVE

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.

5.1 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. `DataSet` is presented in Chapters 3 and 4. For now we treat `DataSet` mostly as an opaque object that can be passed around readers, writers, filters, and rendering units. Detailed documentation for `DataSet` is provided in Chapter 12.

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 5.1: 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);
```

5.2 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 5.2: Creating a canvas for rendering.

```
vtkm::rendering::CanvasRayTracer canvas(1920, 1080);
```

5.3 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::MapperPoint Uses VTK-m's built in ray tracing system to render the visible points/vertices of a mesh. MapperPoint 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.

5.4 Views

A *view* is a unit that collects all the structures needed to perform rendering. It contains everything needed to take a `Scene` (Section 5.1) and use a `Mapper` (Section 5.3) to render it onto a `Canvas` (Section 5.2). 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 5.3: 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);
view.Initialize();
```

Once the `View` is created but before it is used to render, the `Initialize` method should be called. This is demonstrated in Example 5.3.

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 5.4: 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 5.5: Using `Canvas::Paint` in a display callback.

```
view.Paint();
```

Putting together Examples 5.3, 5.4, and 5.5, the final render of a view looks like that in Figure 5.1.

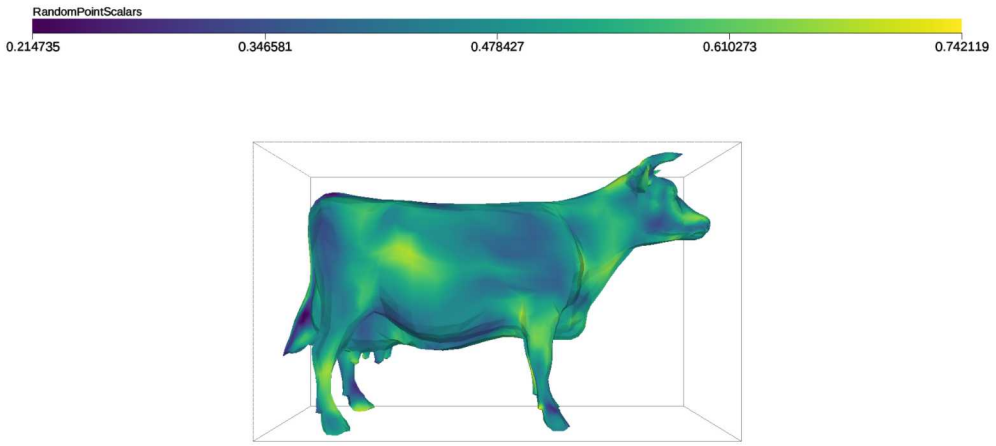


Figure 5.1: Example output of VTK-m's rendering system.

Of course, the `vtkm::rendering::CanvasRayTracer` created in 5.3 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 saves the file in the portable pixelmap (PPM) format.

Example 5.6: Saving the result of a render as an image file.

```
1 | view.SaveAs("BasicRendering.ppm");
```

We visit doing interactive rendering in a GUI later in Section 5.7.

5.5 Changing Rendering Modes

Example 5.3 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 5.3. For example, say you just wanted to see a wireframe representation of your data. You can achieve this by using `vtkm::rendering::MapperWireframer`.

Example 5.7: 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::MapperPoint` renders the points as spheres and also optionally can scale the spheres based on field values.

Example 5.8: Creating a mapper for point representation.

```
1 | vtkm::rendering::MapperPoint mapper;
2 | mapper.UseVariableRadius(true);
3 | mapper.SetRadiusDelta(10.0f);
4 |
5 | vtkm::rendering::View3D view(scene, mapper, canvas);
```

These mappers respectively render the images shown in Figure 5.2. Other mappers, such as those that can render translucent volumes, are also available.

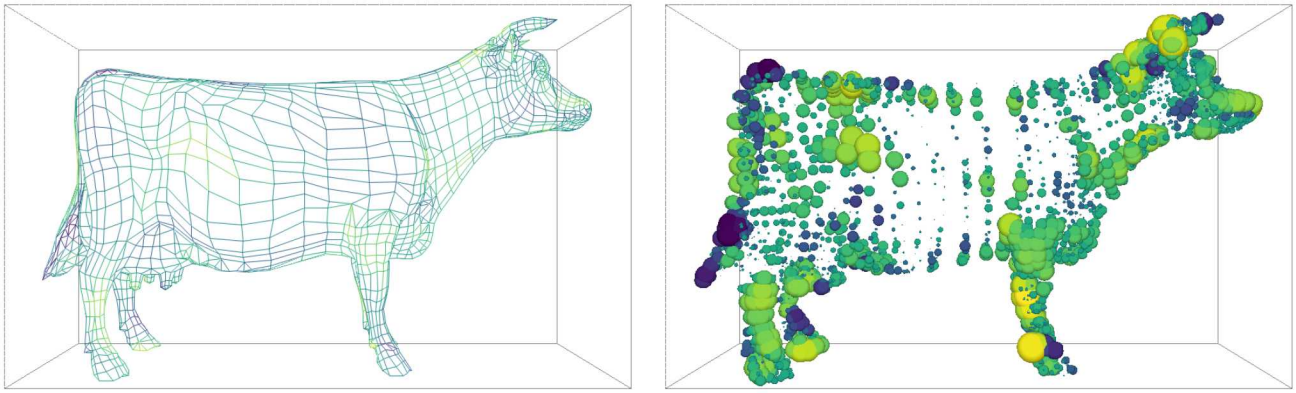


Figure 5.2: Examples of alternate rendering modes using different mappers. The left image is rendered with `MapperWireframer`. The right image is rendered with `MapperPoint`.

5.6 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 5.7.2.

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.

5.6.1 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 `SetViewRange2D` on `Camera`. 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 5.3.

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 6.4.4 and 6.4.5, respectively.

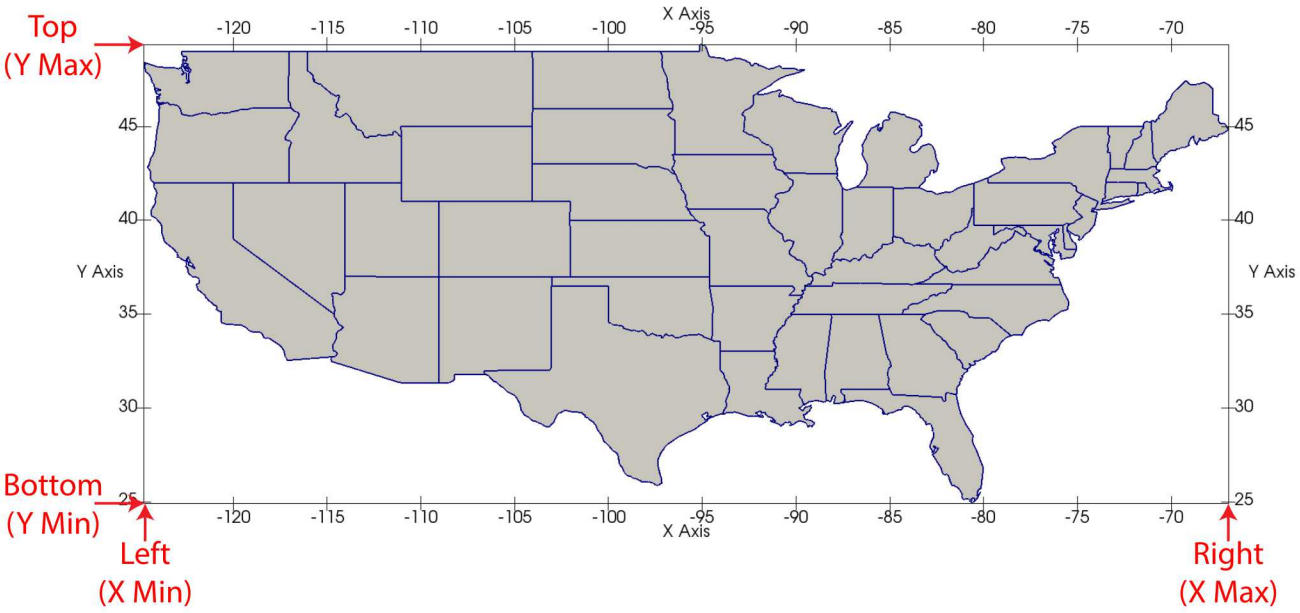


Figure 5.3: The view range bounds to give a `Camera`.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the `Pan` method on `Camera`. `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 5.9: 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 `Zoom` method on `Camera`. `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 5.10: Zooming the camera.

```
1 | view.GetCamera().Zoom(zoomFactor);
```

5.6.2 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 `SetPosition` method of `Camera`. 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 `SetLookAt` method of `Camera`. Figure 5.4 shows the relationship between the position and look at points.

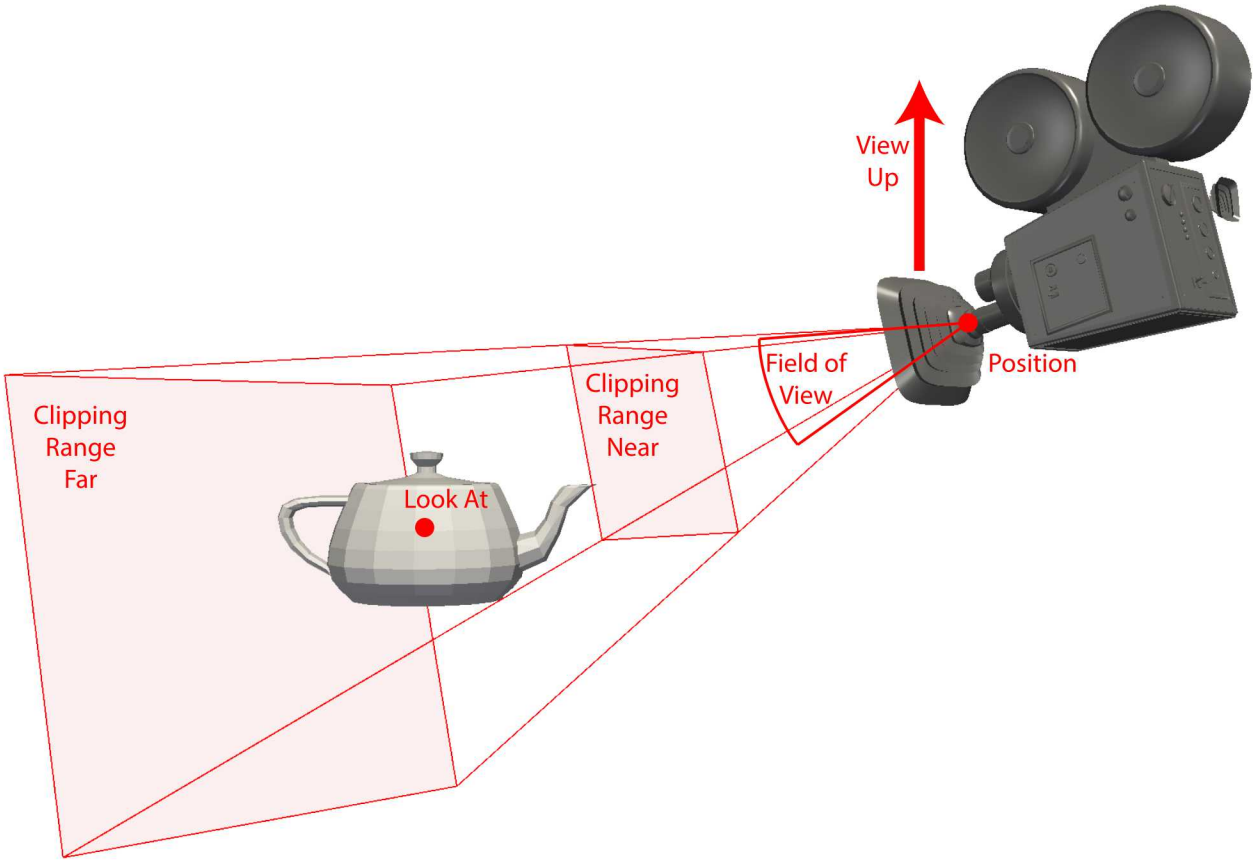


Figure 5.4: The position and orientation parameters for a `Camera`.

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 `SetViewUp` method in `Camera`. 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 5.4.

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 `SetFieldOfView` method in the `Camera`. The field of view angle in relation to the camera orientation is shown in Figure 5.4. 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 `SetClippingRange` method in `Camera`. Figure 5.4 shows the clipping planes in relationship to the camera position and orientation.

Example 5.11: Directly setting `vtkm::rendering::Camera` position and orientation.

```
camera.SetPosition(vtkm::make_Vec(10.0, 6.0, 6.0));
camera.SetLookAt(vtkm::make_Vec(0.0, 0.0, 0.0));
camera.SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
camera.SetFieldOfView(60.0);
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.

Two such methods are elevation and azimuth, which move the camera around the sphere centered at the look at point. `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. `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 5.5 shows the relative movements of `Elevation` and `Azimuth`.

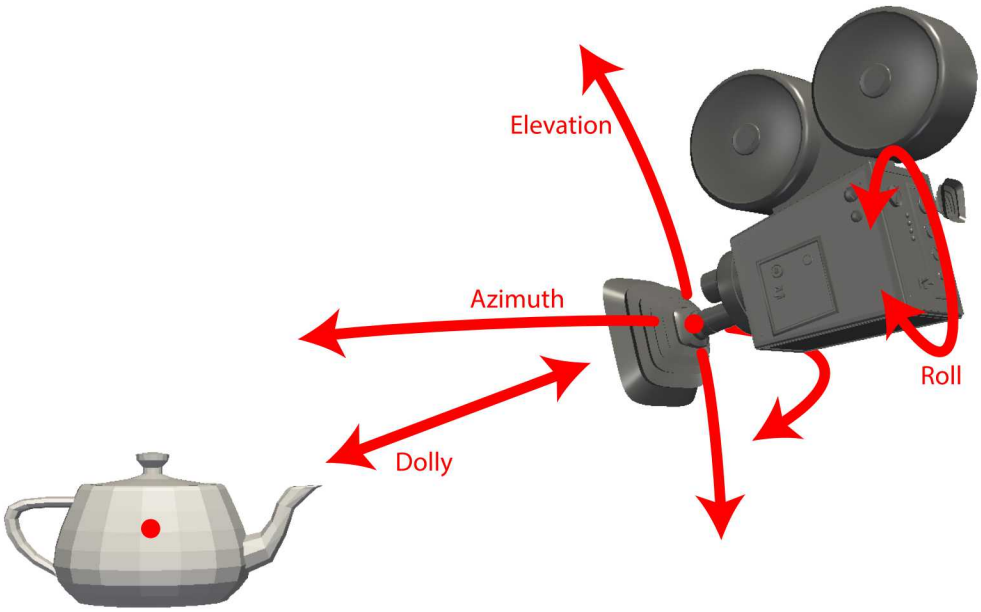


Figure 5.5: `Camera` movement functions relative to position and orientation.

Example 5.12: Moving the camera around the look at point.

```
view.GetCamera().Azimuth(45.0);
view.GetCamera().Elevation(45.0);
```

Common Errors

The `Elevation` *and* `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.*

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 `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 5.5.

Finally, the `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 5.5.

Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the `Pan` method on `Camera`. `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 5.13: 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 `Zoom` method on `Camera`. `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 5.14: 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 `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 5.4, is the central object for managing rendering.

Example 5.15: Resetting a `Camera` to view geometry.

```
void ResetCamera(vtkm::rendering::View& view)
{
  vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
  view.GetCamera().ResetToBounds(bounds);
}
```

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 5.16: Resetting a `Camera` to be axis aligned.

```
  view.GetCamera().SetPosition(vtkm::make_Vec(0.0, 0.0, 0.0));
  view.GetCamera().SetLookAt(vtkm::make_Vec(0.0, 0.0, -1.0));
  view.GetCamera().SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
  vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
  view.GetCamera().ResetToBounds(bounds);
```

5.7 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.

5.7.1 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 `GetColorBuffer` on the `Canvas` object. This will return a `vtkm::cont::ArrayHandle` object containing the image's pixel color data. (`ArrayHandle`s are discussed in detail in Chapter 7.) A raw pointer can be pulled out of this `ArrayHandle` by calling `GetStorage().GetBasePointer()`. 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 5.17: Rendering a `View` and pasting the result to an active OpenGL context.

```
view.Paint();

// Get the color buffer containing the rendered image.
vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::Float32, 4>> colorBuffer =
  view.GetCanvas().GetColorBuffer();

// Pull the C array out of the arrayhandle.
void* colorArray = colorBuffer.GetStorage().GetBasePointer();

// Write the C array to an OpenGL buffer.
glDrawPixels((GLint)view.GetCanvas().GetWidth(),
             (GLint)view.GetCanvas().GetHeight(),
             GL_RGBA,
             GL_FLOAT,
             colorArray);

// Swap the OpenGL buffers (system dependent).
```

5.7.2 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 5.18: 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 5.19: 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 5.20: 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);
}
```

5.8 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 5.1.)

Example 5.21: 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.

Part II

Using VTK-m

CHAPTER

SIX

BASIC PROVISIONS

This section describes the core facilities provided by VTK-m. These include macros, types, and classes that define the environment in which code is run, the core types of data stored, and template introspection. We also start with a description of package structure used by VTK-m.

6.1 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 small functor or kernel designed to operate on a small element of data. (The name "worklet" means a small amount of 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.

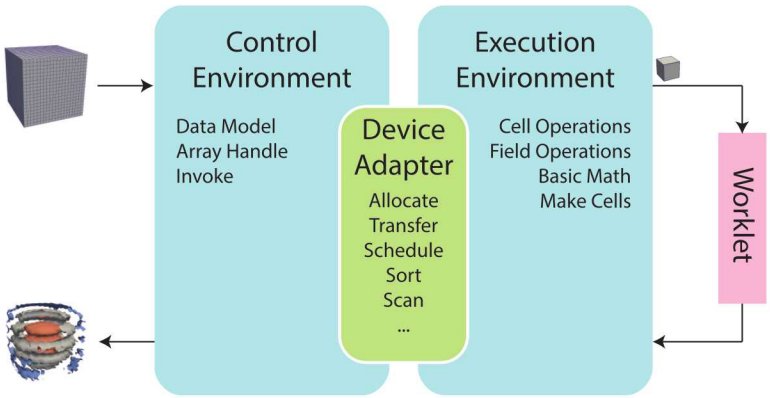


Figure 6.1: Diagram of the VTK-m framework.

Figure 6.1 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.

6.2 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 in Section 6.1, 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 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, described in Chapter 13, 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 I.

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 6.2 provides a diagram of the VTK-m package hierarchy.

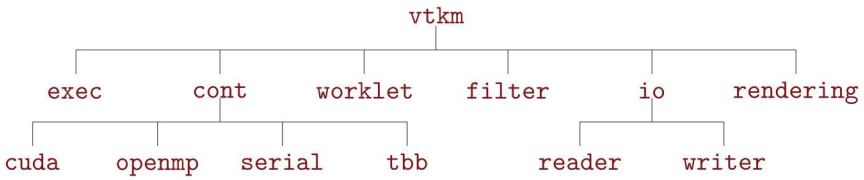


Figure 6.2: VTK-m package hierarchy.

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::ArrayHandle` class is found in the vtkm/cont/ArrayHandle.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.

6.3 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,

[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

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 6.1: 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 6.2: 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();
  }
}
```

6.4 Core Data Types

Except in rare circumstances where precision is not a concern, VTK-m does not directly use the core C types like `int`, `float`, and `double`. Instead, VTK-m provides its own core types, which are declared in vtkm/Types.h.

6.4.1 Single Number Types

To ensure portability across different compilers and architectures, VTK-m provides type aliases for the following basic types with explicit precision: vtkm::Float32, vtkm::Float64, vtkm::Int8, vtkm::Int16, vtkm::Int32, vtkm::Int64, vtkm::UInt8, vtkm::UInt16, vtkm::UInt32, and vtkm::UInt64. Under most circumstances when using VTK-m (and performing visualization in general) the type of data is determined by the source of the data or resolved through templates. In the case where a specific type of data is required, these VTK-m–defined types should be preferred over basic C types like `int` or `float`.

Many of the structures in VTK-m require indices to identify elements like points and cells. All indices for arrays and other lists use the type vtkm::Id. By default this type is a 32-bit wide integer but can be easily changed by compile options. The CMake configuration option VTKM_USE_64BIT_IDS can be used to change vtkm::Id to be 64 bits wide. This configuration can be overridden by defining the C macro VTKM_USE_64BIT_IDS or VTKM_NO_64BIT_IDS to force vtkm::Id to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

There is also a secondary index type named vtkm::IdComponent that is used to index components of short vectors (discussed in Section 6.4.2). This type is an integer that might be a shorter width than vtkm::Id.

There is also the rare circumstance in which an algorithm in VTK-m computes data values for which there is no indication what the precision should be. For these circumstances, the type vtkm::FloatDefault is provided. By default this type is a 32-bit wide floating point number but can be easily changed by compile options. The CMake configuration option VTKM_USE_DOUBLE_PRECISION can be used to change vtkm::FloatDefault to be 64 bits wide. This configuration can be overridden by defining the C macro VTKM_USE_DOUBLE_PRECISION or VTKM_NO_DOUBLE_PRECISION to force vtkm::FloatDefault to be either 64 or 32 bits. These macros must be defined before any VTK-m header files are included to take effect.

For convenience, you can include either vtkm/internal/ConfigureFor32.h or vtkm/internal/ConfigureFor64.h to force both vtkm::Id and vtkm::FloatDefault to be 32 or 64 bits.

6.4.2 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 6.3: Creating vector types.

```
1 vtkm::Vec<vtkm::Float32, 3> A{ 1 };                      // A is (1, 1, 1)
2 A[1] = 2;                                                // A is now (1, 2, 1)
3 vtkm::Vec<vtkm::Float32, 3> B{ 1, 2, 3 };                // B is (1, 2, 3)
4 vtkm::Vec<vtkm::Float32, 3> C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
5 // creation with initializer lists
6 vtkm::Vec<vtkm::Float32, 5> D{ 1 };                  // D is (1, 1, 1, 1, 1)
7 vtkm::Vec<vtkm::Float32, 5> E{ 1, 2, 3, 4, 5 };     // E is (1, 2, 3, 4, 5)
8 vtkm::Vec<vtkm::Float32, 5> F = { 6, 7, 8, 9, 10 }; // F is (6, 7, 8, 9, 10)
9 auto G = vtkm::make_Vec(1, 3, 5, 7, 9);             // G is (1, 3, 5, 7, 9)
```

The types vtkm::Id2 and vtkm::Id3 are type aliases of vtkm::Vec <vtkm::Id,2> and vtkm::Vec <vtkm::Id,2>. These are used to index arrays of 2 and 3 dimensions, which is common.

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 6.4: Vector operations.

```
vtkm::Vec<vtkm::Float32, 3> A{ 1, 2, 3 };
vtkm::Vec<vtkm::Float32, 3> B{ 4, 5, 6.5 };
vtkm::Vec<vtkm::Float32, 3> C = A + B;     // C is (5, 7, 9.5)
vtkm::Vec<vtkm::Float32, 3> 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 6.5: Repurposing a `vtkm::Vec`.

```
vtkm::Vec<vtkm::Vec<vtkm::Float32, 2>, 3> equilateralTriangle(
  vtkm::make_Vec(0.0, 0.0),
  vtkm::make_Vec(1.0, 0.0),
  vtkm::make_Vec(0.5, 0.8660254));
```

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 6.6: Using `vtkm::VecCConst` with a constant array.

```
VTKM_EXEC
vtkm::VecCConst<vtkm::IdComponent> HexagonIndexToIJK(vtkm::IdComponent index)
{
  static const vtkm::IdComponent HexagonIndexToIJKTable[8][3] = {
    { 0, 0, 0 }, { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 0 },
    { 0, 0, 1 }, { 1, 0, 1 }, { 1, 1, 1 }, { 0, 1, 1 }
  };

  return vtkm::make_VecC(HexagonIndexToIJKTable[index], 3);
}
```

```
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 6.6.

Example 6.7: 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::Vec<vtkm::IdComponent, 3> 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 7.2. 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 12.

6.4.3 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 environment, 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.

6.4.4 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.

`Range::IsNonEmpty` Returns true if the range covers at least one value.

`Range::Contains` Takes a single number and returns true if that number is contained within the range.

`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.

`Range::Center` Returns the number equidistant to `Min` and `Max`. If the range is empty, NaN is returned.

`Range::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.

`Range::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 6.8: Using `vtkm::Range`.

```
1  vtkm::Range range;            // default constructor is empty range
2  bool b1 = range.IsNonEmpty(); // b1 is false
3
4  range.Include(0.5);           // range now is [0.5 .. 0.5]
5  bool b2 = range.IsNonEmpty(); // b2 is true
6  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
```

6.4.5 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 6.4.4. `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.

`Bounds::IsNonEmpty` Returns true if the bounds cover at least one value.

`Bounds::Contains` Takes a `vtkm::Vec` of size 3 and returns true if those point coordinates are contained within the range.

`Bounds::Center` Returns the point at the center of the range as a `vtkm::Vec <vtkm::Float64,3>`.

`Bounds::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.

`Bounds::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 6.9: Using `vtkm::Bounds`.

```
vtkm::Bounds bounds;           // default constructor makes empty
bool b1 = bounds.IsNonEmpty(); // b1 is false

bounds.Include(vtkm::make_Vec(0.5, 2.0, 0.0));           // bounds contains only
                                                         // the point [0.5, 2, 0]
bool b2 = bounds.IsNonEmpty();                           // b2 is true
bool b3 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b3 is true
bool b4 = bounds.Contains(vtkm::make_Vec(1, 1, 1));      // b4 is false
bool b5 = bounds.Contains(vtkm::make_Vec(0, 0, 0));      // b5 is false

bounds.Include(vtkm::make_Vec(4, -1, 2)); // bounds is region [0.5 .. 4] in X,
                                          //                 [-1 .. 2] in Y,
                                          //             and [0 .. 2] in Z
bool b6 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b6 is true
bool b7 = bounds.Contains(vtkm::make_Vec(1, 1, 1));       // b7 is true
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::Vec<vtkm::Float64, 3> lower(bounds.X.Min, bounds.Y.Min, bounds.Z.Min);
25   // lower is [0, -1, 0]
26   vtkm::Vec<vtkm::Float64, 3> upper(bounds.X.Max, bounds.Y.Max, bounds.Z.Max);
27   // upper is [4, 3, 3]
28
29   vtkm::Vec<vtkm::Float64, 3> center = bounds.Center(); // center is [2, 1, 1.5]
```

6.5 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 Mayers for a description of traits classes and their uses.

6.5.1 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 Section 6.4. `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 6.10: 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 6.11: Using `TypeTraits` for a generic remainder.

```
#include <vtkm/TypeTraits.h>

#include <vtkm/Math.h>

template<typename T>
T AnyRemainder(const T& numerator, const T& denominator);

namespace detail
{

template<typename T>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   vtkm::TypeTraitsIntegerTag,
                   vtkm::TypeTraitsScalarTag)
{
  return numerator % denominator;
}

template<typename T>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   vtkm::TypeTraitsRealTag,
                   vtkm::TypeTraitsScalarTag)
{
  // The VTK-m math library contains a Remainder function that operates on
  // floating point numbers.
  return vtkm::Remainder(numerator, denominator);
}

template<typename T, typename NumericTag>
T AnyRemainderImpl(const T& numerator,
                   const T& denominator,
                   NumericTag,
                   vtkm::TypeTraitsVectorTag)
{
  T result;
  for (int componentIndex = 0; componentIndex < T::NUM_COMPONENTS; componentIndex++)
  {
    result[componentIndex] =
      AnyRemainder(numerator[componentIndex], denominator[componentIndex]);
  }
  return result;
}

} // namespace detail

template<typename T>
T AnyRemainder(const T& numerator, const T& denominator)
{
  return detail::AnyRemainderImpl(numerator,
                                  denominator,
                                  typename vtkm::TypeTraits<T>::NumericTag(),
                                  typename vtkm::TypeTraits<T>::DimensionalityTag());
}
```

6.5.2 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 6.12: Definition of vtkm::VecTraits <vtkm::Id3 >.

```
1  namespace vtkm {
2
3  template<>
4  struct VecTraits<vtkm::Id3>
5  {
6    using ComponentType = vtkm::Id;
7    static const int NUM_COMPONENTS = 3;
8    using IsSizeStatic = vtkm::VecTraitsTagSizeStatic;
9    using HasMultipleComponents = vtkm::VecTraitsTagMultipleComponents;
10
11   VTKM_EXEC_CONT
12   static vtkm::IdComponent GetNumberOfComponents(const vtkm::Id3&)
13   {
```

```
14      return NUM_COMPONENTS;
15    }
16
17    VTKM_EXEC_CONT
18    static const vtkm::Id& GetComponent(const vtkm::Id3& vector, int component)
19    {
20      return vector[component];
21    }
22    VTKM_EXEC_CONT
23    static vtkm::Id& GetComponent(vtkm::Id3& vector, int component)
24    {
25      return vector[component];
26    }
27
28    VTKM_EXEC_CONT
29    static void SetComponent(vtkm::Id3& vector, int component, vtkm::Id value)
30    {
31      vector[component] = value;
32    }
33
34    template<vtkm::IdComponent DestSize>
35    VTKM_EXEC_CONT static void CopyInto(const vtkm::Id3& src,
36                                        vtkm::Vec<vtkm::Id, DestSize>& dest)
37    {
38      for (vtkm::IdComponent index = 0; (index < NUM_COMPONENTS) && (index < DestSize);
39           index++)
40      {
41        dest[index] = src[index];
42      }
43    }
44  };
45
46  } // 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 6.13: Using `VecTraits` for less functors.

```
1  #include <vtkm/VecTraits.h>
2
3  // This functor provides a total ordering of vectors. Every compared vector
4  // will be either less, greater, or equal (assuming all the vector components
5  // also have a total ordering).
6  template<typename T>
7  struct LessTotalOrder
8  {
9    VTKM_EXEC_CONT
10   bool operator()(const T& left, const T& right)
11   {
12     for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
13     {
14       using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
15       const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
16       const ComponentType& rightValue =
17         vtkm::VecTraits<T>::GetComponent(right, index);
18       if (leftValue < rightValue)
19       {
20         return true;
21       }
22       if (rightValue < leftValue)
23       {
```

```
24          return false;
25        }
26      }
27      // If we are here, the vectors are equal (or at least equivalent).
28      return false;
29    }
30  };
31
32  // This functor provides a partial ordering of vectors. It returns true if and
33  // only if all components satisfy the less operation. It is possible for
34  // vectors to be neither less, greater, nor equal, but the transitive closure
35  // is still valid.
36  template<typename T>
37  struct LessPartialOrder
38  {
39    VTKM_EXEC_CONT
40    bool operator()(const T& left, const T& right)
41    {
42      for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
43      {
44        using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
45        const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
46        const ComponentType& rightValue =
47          vtkm::VecTraits<T>::GetComponent(right, index);
48        if (!(leftValue < rightValue))
49        {
50          return false;
51        }
52      }
53      // If we are here, all components satisfy less than relation.
54      return true;
55    }
56  };
```

6.6 List Tags

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 6.6.2. 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.

VTK-m uses a tag-based mechanism for defining lists, which differs significantly from lists in many other template metaprogramming libraries such as with `boost::mpl::vector` or `boost::vector`. Rather than enumerating all list entries as template arguments, the list is referenced by a single tag class with a descriptive name. The intention is to make fully resolved types shorter and more readable. (Anyone experienced with template programming knows how insanely long and unreadable types can get in compiler errors and warnings.)

6.6.1 Building List Tags

List tags are constructed in VTK-m by defining a `struct` that publicly inherits from another list tags. The base list tags are defined in the vtkm/ListTag.h header.

The most basic list is defined with `vtkm::ListTagEmpty`. This tag represents an empty list.

vtkm::ListTagBase <T, ...> represents a list of the types given as template parameters. vtkm::ListTagBase supports a variable number of parameters with the maximum specified by VTKM_MAX_BASE_LIST.

Finally, lists can be combined together with vtkm::ListTagJoin <ListTag1,ListTag2>, which concatinates two lists together.

The following example demonstrates how to build list tags using these base lists classes. Note first that all the list tags are defined as `struct` rather than `class`. Although these are roughly synonymous in C++, `struct` inheritance is by default public, and public inheritance is important for the list tags to work. Note second that these tags are created by inheritance rather than using a type alias. Although a type alias defined with `using` will work, it will lead to much uglier type names defined by the compiler.

Example 6.14: Creating list tags.

```
#include <vtkm/ListTag.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.

struct FooList : vtkm::ListTagBase<Foo>
{
};

struct FooBarList : vtkm::ListTagBase<Foo, Bar>
{
};

struct BazQuxXyzzyList : vtkm::ListTagBase<Baz, Qux, Xyzzy>
{
};

struct QuxBazBarFooList : vtkm::ListTagBase<Qux, Baz, Bar, Foo>
{
};

struct FooBarBazQuxXyzzyList : vtkm::ListTagJoin<FooBarList, BazQuxXyzzyList>
{
};
```

6.6.2 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/TypeListTag.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::TypeListTagId Contains the single item vtkm::Id.

vtkm::TypeListTagId2 Contains the single item vtkm::Id2.

vtkm::TypeListTagId3 Contains the single item vtkm::Id3.

`vtkm::TypeListTagIndex` A list of all types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`.

`vtkm::TypeListTagFieldScalar` 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::TypeListTagFieldVec2` A list containing types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec3` A list containing types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagFieldVec4` A list containing types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers.

`vtkm::TypeListTagField` A list containing all the types generally used for fields. It is the combination of `vtkm::TypeListTagFieldScalar`, `vtkm::TypeListTagFieldVec2`, `vtkm::TypeListTagFieldVec3`, and `vtkm::TypeListTagFieldVec4`.

`vtkm::TypeListTagScalarAll` 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::TypeListTagVecCommon` 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::TypeListTagVecAll` A list of all `vtkm::Vec` classes with standard integers or floating points as components and lengths between 2 and 4.

`vtkm::TypeListTagAll` A list of all types included in vtkm/Types.h with `vtkm::Vec` s with up to 4 components.

`vtkm::TypeListTagCommon` 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 dynamic arrays (described in Chapter 11).

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 6.6.1 as demonstrated in the following example.

Example 6.15: Defining new type lists.

```
#define VTKM_DEFAULT_TYPE_LIST_TAG MyCommonTypes

#include <vtkm/ListTag.h>
#include <vtkm/TypeListTag.h>

// A list of 2D vector types.
struct Vec2List : vtkm::ListTagBase<vtkm::Id2,
                                    vtkm::Vec<vtkm::Float32, 2>,
                                    vtkm::Vec<vtkm::Float64, 2>>
{
};

// An application that uses 2D geometry might commonly encounter this list of
// types.
struct MyCommonTypes : vtkm::ListTagJoin<Vec2List, vtkm::TypeListTagCommon>
{
};
```

The vtkm/TypeListTag.h header also defines a macro named VTKM_DEFAULT_TYPE_LIST_TAG that defines a default list of types to use in classes like vtkm::cont::DynamicArrayHandle (Chapter 11). This list can be overridden by defining the VTKM_DEFAULT_TYPE_LIST_TAG 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 6.15 also contains an example of overriding the VTKM_DEFAULT_TYPE_LIST_TAG macro.

6.6.3 Operating on Lists

VTK-m template metaprogramming lists are typically just passed to VTK-m methods that internally operate on the lists. Although not typically used outside of the VTK-m library, these operations are also available.

The vtkm/ListTag.h header comes with a vtkm::ListForEach function that takes a functor object and a list tag. 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 coverting a dynamically-typed array to a statically-typed array similar to what is done in VTK-m classes like vtkm::cont::DynamicArrayHandle (which is documented in Chapter 11).

Example 6.16: Converting dynamic types to static types with ListForEach.

```
1  struct MyArrayBase
2  {
3    // A virtual destructor makes sure C++ RTTI will be generated. It also helps
4    // ensure subclass destructors are called.
5    virtual ~MyArrayBase() {}
6  };
7
8  template<typename T>
9  struct MyArrayImpl : public MyArrayBase
10 {
11   std::vector<T> Array;
12 };
13
14 template<typename T>
15 void PrefixSum(std::vector<T>& array)
16 {
17   T sum(typename vtkm::VecTraits<T>::ComponentType(0));
18   for (typename std::vector<T>::iterator iter = array.begin(); iter != array.end();
19        iter++)
20   {
21     sum = sum + *iter;
22     *iter = sum;
23   }
24 }
25
26 struct PrefixSumFunctor
27 {
28   MyArrayBase* ArrayPointer;
29
30   PrefixSumFunctor(MyArrayBase* arrayPointer)
31     : ArrayPointer(arrayPointer)
32   {
33   }
34
35   template<typename T>
36   void operator()(T)
37   {
```

```
38      using ConcreteArrayType = MyArrayImpl<T>;
39      ConcreteArrayType* concreteArray =
40        dynamic_cast<ConcreteArrayType*>(this->ArrayPointer);
41      if (concreteArray != NULL)
42      {
43        PrefixSum(concreteArray->Array);
44      }
45    }
46  };
47
48  void DoPrefixSum(MyArrayBase* array)
49  {
50    PrefixSumFunctor functor = PrefixSumFunctor(array);
51    vtkm::ListForEach(functor, vtkm::TypeListTagCommon());
52  }
```

6.7 Error Handling

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 6.17: 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 (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 occured 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.

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 6.18: Using `VTKM_ASSERT`.

```
template<typename T>
VTKM_CONT T GetArrayValue(vtkm::cont::ArrayHandle<T> arrayHandle, vtkm::Id index)
{
  VTKM_ASSERT(index >= 0);
  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.

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 6.19: 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 6.19 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 6.19 demonstrates getting the Boolean literal from the result of* `std::is_same`.

6.8 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

SEVEN

ARRAY HANDLES

An *array handle*, 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 allocate and 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 up to two copies of the array, one for the control environment and one for the execution environment. 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 is 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.

`ArrayHandle::GetNumberOfValues` Returns the number of entries in the array.

`ArrayHandle::Allocate` Resizes the array to include the number of entries given. Any previously stored data might be discarded.

`ArrayHandle::Shrink` Resizes the array to the number of entries given. Any data stored in the array is preserved. The number of entries must be less than those given in the last call to `Allocate`.

`ArrayHandle::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.

`ArrayHandle::ReleaseResources` Releases all memory managed by this `ArrayHandle`. Any data in this memory is lost.

`ArrayHandle::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.

`ArrayHandle::GetPortalControl` Returns an array portal that can be used to access the data in the array handle in the control environment. Array portals are described in Section 7.2.

`ArrayHandle::GetPortalConstControl` Like `GetPortalControl` but returns a read-only array portal rather than a read/write array portal.

`ArrayHandle::PrepareForInput` Readies the data as input to a parallel algorithm. See Section 7.6 for more details.

`ArrayHandle::PrepareForOutput` Readies the data as output to a parallel algorithm. See Section 7.6 for more details.

`ArrayHandle::PrepareForInPlace` Readies the data as input and output to a parallel algorithm. See Section 7.6 for more details.

`ArrayHandle::GetDeviceAdapterId` Returns a `vtkm::cont::DeviceAdapterId` describing on which device adapter, if any, the array h andle's data is available. Device adapter ids are described in Section 8.2.

`ArrayHandle::GetStorage` Returns the `vtkm::cont::Storage` object that manages the data. The type of the storage object is defined by the storage tag template parameter of the `ArrayHandle`. Storage objects are described in detail in Chapter 10.

7.1 Creating Array Handles

`vtkm::cont::ArrayHandle` is a templated class with two template parameters. The first template parameter is the only one required and specifies the base type of the entries in the array. The second template parameter specifies the storage used when storing data in the control environment. Storage objects are discussed later in Chapter 10, and for now we will use the default value.

Example 7.1: Declaration of the `vtkm::cont::ArrayHandle` templated class.

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
class ArrayHandle;
```

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 7.2: Creating an `ArrayHandle` for output data.

```
  vtkm::cont::ArrayHandle<vtkm::Float32> outputArray;
```

Constructing an `ArrayHandle` that points to a provided C array or `std::vector` is straightforward with the `vtkm::cont::make_ArrayHandle` functions. These functions will make an array handle that points to the array data that you provide.

Example 7.3: 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);
```

Example 7.4: Creating an `ArrayHandle` that points to a provided `std::vector`.

```
1  std::vector<vtkm::Float32> dataBuffer;
2  // Populate dataBuffer with meaningful data. Perhaps read data from a file.
3
4  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
5    vtkm::cont::make_ArrayHandle(dataBuffer);
```

Be aware that `vtkm::cont::make_ArrayHandle` makes a shallow pointer copy. This means that if you change or delete the data provided, the internal state of `ArrayHandle` becomes invalid and undefined behavior can ensue. The most 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. For example, if the code in Example 7.4 where to be placed within a callable function or method, it could cause the `vtkm::cont::ArrayHandle` to become invalid.

Common Errors

Because `ArrayHandle` does not manage data provided by `make_ArrayHandle`, you should only use these as temporary objects. Example 7.5 demonstrates a method of copying one of these temporary arrays into safe managed memory, and Section 7.3 describes how to put data directly into an `ArrayHandle` object.

Example 7.5: Invalidating an `ArrayHandle` by letting the source `std::vector` leave scope.

```
1  VTKM_CONT
2  vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
3  {
4    std::vector<vtkm::Float32> dataBuffer;
5    // Populate dataBuffer with meaningful data. Perhaps read data from a file.
6
7    vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
8      vtkm::cont::make_ArrayHandle(dataBuffer);
9
10   return inputArray;
11   // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
12   // memory. However, inputArray has a pointer to that memory, which becomes an
13   // invalid pointer in the returned object. Bad things will happen when the
14   // ArrayHandle is used.
15 }
16
17 VTKM_CONT
18 vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad()
19 {
20   std::vector<vtkm::Float32> dataBuffer;
21   // Populate dataBuffer with meaningful data. Perhaps read data from a file.
22
23   vtkm::cont::ArrayHandle<vtkm::Float32> tmpArray =
24     vtkm::cont::make_ArrayHandle(dataBuffer);
25
26   // This copies the data from one ArrayHandle to another (in the execution
27   // environment). Although it is an extraneous copy, it is usually pretty fast
28   // on a parallel device. Another option is to make sure that the buffer in
29   // the std::vector never goes out of scope before all the ArrayHandle
30   // references, but this extra step allows the ArrayHandle to manage its own
31   // memory and ensure everything is valid.
32   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray;
33   vtkm::cont::ArrayCopy(tmpArray, inputArray);
34
35   return inputArray;
```

```
36   // This is safe.
37 }
```

7.2 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 (const) 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 (const) 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 7.6: A simple array portal implementation.

```
1  template<typename T>
2  class SimpleScalarArrayPortal
3  {
4  public:
5    using ValueType = T;
6
7    // There is no specification for creating array portals, but they generally
8    // need a constructor like this to be practical.
9    VTKM_EXEC_CONT
10   SimpleScalarArrayPortal(ValueType* array, vtkm::Id numberOfValues)
11     : Array(array)
12     , NumberOfValues(numberOfValues)
13   {
14   }
15
16   VTKM_EXEC_CONT
17   SimpleScalarArrayPortal()
18     : Array(NULL)
19     , NumberOfValues(0)
20   {
21   }
22
23   VTKM_EXEC_CONT
24   vtkm::Id GetNumberOfValues() const { return this->NumberOfValues; }
25
26   VTKM_EXEC_CONT
27   ValueType Get(vtkm::Id index) const { return this->Array[index]; }
28
29   VTKM_EXEC_CONT
30   void Set(vtkm::Id index, ValueType value) const { this->Array[index] = value; }
```

```
31
32 private:
33   ValueType* Array;
34   vtkm::Id NumberOfValues;
35 };
```

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 convienient 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.

`ArrayPortalToIterators::IteratorType` The type of an STL-compatible random-access iterator that can provide the same access as the array portal.

`ArrayPortalToIterators::GetBegin` A method that returns an STL-compatible iterator of type `IteratorType` that points to the beginning of the array.

`ArrayPortalToIterators::GetEnd` A method that returns an STL-compatible iterator of type `IteratorType` that points to the end of the array.

Example 7.7: Using `ArrayPortalToIterators`.

```
1  template<typename PortalType>
2  VTKM_CONT std::vector<typename PortalType::ValueType> CopyArrayPortalToVector(
3    const PortalType& portal)
4  {
5    using ValueType = typename PortalType::ValueType;
6    std::vector<ValueType> result(
7      static_cast<std::size_t>(portal.GetNumberOfValues()));
8
9    vtkm::cont::ArrayPortalToIterators<PortalType> iterators(portal);
10
11   std::copy(iterators.GetBegin(), iterators.GetEnd(), result.begin());
12
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 7.8: 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 `PortalControl` and `PortalConstControl`, which define read-write and read-only (const) array portals, respectively.

`ArrayHandle` also contains similar type definitions for array portals in the execution environment. Because these types are dependent on the device adapter used for execution, these type definitions are embedded in a templated class named `ExecutionTypes`. Within `ExecutionTypes` are the type definitions `Portal` and `PortalConst` defining the read-write and read-only (const) array portals, respectively, for the execution environment for the given device adapter tag.

Because `vtkm::cont::ArrayHandle` is control environment object, it provides the methods `GetPortalControl` and `GetPortalConstControl` to get the associated array portal objects. 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 `GetPortalControl` will invalidate any copy in the execution environment, meaning that any subsequent use will cause the data to be copied back again.

Example 7.9: Using portals from an `ArrayHandle`.

```
template<typename T>
void SortCheckArrayHandle(vtkm::cont::ArrayHandle<T> arrayHandle)
{
  using PortalType = typename vtkm::cont::ArrayHandle<T>::PortalControl;
  using PortalConstType = typename vtkm::cont::ArrayHandle<T>::PortalConstControl;

  PortalType readwritePortal = arrayHandle.GetPortalControl();
  // This is actually pretty dumb. Sorting would be generally faster in
  // parallel in the execution environment using the device adapter algorithms.
  std::sort(vtkm::cont::ArrayPortalToIteratorBegin(readwritePortal),
            vtkm::cont::ArrayPortalToIteratorEnd(readwritePortal));

  PortalConstType readPortal = arrayHandle.GetPortalConstControl();
  for (vtkm::Id index = 1; index < readPortal.GetNumberOfValues(); index++)
  {
    if (readPortal.Get(index - 1) > readPortal.Get(index))
    {
      std::cout << "Sorting is wrong!" << std::endl;
      break;
    }
  }
}
```

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.

7.3 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 7.10: Allocating an `ArrayHandle`.

```
  vtkm::cont::ArrayHandle<vtkm::Float32> arrayHandle;

  const vtkm::Id ARRAY_SIZE = 50;
  arrayHandle.Allocate(ARRAY_SIZE);
```

Common Errors

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 can be particularly surprising when the originally managed memory is empty. For example, older versions of* `std::vector` *initialized all its values by setting them to the same object. When a* `vector` *of* `ArrayHandle`*s was created and one entry was allocated, all entries changed to the same allocation.*

Once an `ArrayHandle` is allocated, it can be populated by using the portal returned from `ArrayHandle::GetPortalControl`, as described in Section 7.2. This is roughly the method used by the readers in the I/O package (Chapter 3).

Example 7.11: 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  using PortalType = vtkm::cont::ArrayHandle<vtkm::Float32>::PortalControl;
7  PortalType portal = arrayHandle.GetPortalControl();
8
9  for (vtkm::Id index = 0; index < ARRAY_SIZE; index++)
10 {
11   portal.Set(index, GetValueForArray(index));
12 }
```

7.4 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.

To simplify copying the data, 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 7.12: Using `ArrayCopy`.

```
1  vtkm::cont::ArrayCopy(tmpArray, inputArray);
```

Did you know?

`ArrayCopy` *will copy data in parallel. If desired, you can specify the device as the third argument to* `ArrayCopy` *using either a device adapter tag or a runtime device tracker. Both the tags and tracker are described in Chapter 8.*

7.5 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 `Ranges` 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 6.4.4.

Example 7.13: Using `ArrayRangeCompute`.

```
1  vtkm::cont::ArrayHandle<vtkm::Range> rangeArray =
2    vtkm::cont::ArrayRangeCompute(arrayHandle);
3  auto rangePortal = rangeArray.GetPortalConstControl();
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 8.3.*

7.6 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 or worklets, this transition is handled automatically. However, it is also possible to invoke the transfer for use in your own scheduled algorithms.

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 a single argument that is the device adapter tag where execution will take place (see Section 8.1 for more information on device adapter tags). `PrepareForOutput` takes two arguments: the size of the space to allocate and the device adapter tag. Each of these methods returns an array portal that can be used in the execution environment. `PrepareForInput` returns an object of type `ArrayHandle::ExecutionTypes<`*DeviceAdapterTag*`>::PortalConst` whereas `PrepareForInPlace` and `PrepareForOutput` each return an object of type `ArrayHandle::ExecutionTypes<`*DeviceAdapterTag*`>::Portal`.

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. Typically this is done with a call to `vtkm::cont::[` DeviceAdapterAlgorithm]Schedule. This and other device adapter algorithms are described in detail in Section 8.4, but here is a quick example of using these execution array portals in a simple functor.

Example 7.14: Using an execution array portal from an `ArrayHandle`.

```
template<typename T, typename Device>
struct DoubleFunctor : public vtkm::exec::FunctorBase
{
  using InputPortalType = typename vtkm::cont::ArrayHandle<
    T>::template ExecutionTypes<Device>::PortalConst;
  using OutputPortalType =
    typename vtkm::cont::ArrayHandle<T>::template ExecutionTypes<Device>::Portal;

  VTKM_CONT
  DoubleFunctor(InputPortalType inputPortal, OutputPortalType outputPortal)
    : InputPortal(inputPortal)
    , OutputPortal(outputPortal)
  {
  }

  VTKM_EXEC
  void operator()(vtkm::Id index) const
  {
    this->OutputPortal.Set(index, 2 * this->InputPortal.Get(index));
  }

  InputPortalType InputPortal;
  OutputPortalType OutputPortal;
};

template<typename T, typename Device>
void DoubleArray(vtkm::cont::ArrayHandle<T> inputArray,
                 vtkm::cont::ArrayHandle<T> outputArray,
                 Device)
{
  vtkm::Id numValues = inputArray.GetNumberOfValues();

  DoubleFunctor<T, Device> functor(
    inputArray.PrepareForInput(Device()),
    outputArray.PrepareForOutput(numValues, Device()));

  vtkm::cont::DeviceAdapterAlgorithm<Device>::Schedule(functor, numValues);
}
```

It should be noted that the array handle will expect any use of the execution array portal to finish before the next call to any `ArrayHandle` method. Since these `Prepare` methods are typically used in the process of scheduling an algorithm in the execution environment, this is seldom an issue.

Common Errors

There are many operations on `ArrayHandle` *that can invalidate the array portals, so do not keep them around indefinitely. It is generally better to keep a reference to the* `ArrayHandle` *and use one of the* `Prepare` *each time the data are accessed in the execution environment.*

CHAPTER

EIGHT

DEVICE ADAPTERS

As multiple vendors vie to provide accelerator-type processors, a great variance in the computer architecture exists. Likewise, there exist multiple compiler environments and libraries for these devices such as CUDA, OpenMP, and various threading libraries. These compiler technologies also vary from system to system.

To make porting among these systems at all feasible, we require a base language support, and the language we use is C++. The majority of the code in VTK-m is constrained to the standard C++ language constructs to minimize the specialization from one system to the next.

Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a *device adapter*. Thus, porting VTK-m to a new architecture can be done by adding only a device adapter.

The device adapter is shown diagrammatically as the connection between the control and execution environments in Figure 6.1 on page 54. The functionality of the device adapter comprises two main parts: a collection of parallel algorithms run in the execution environment and a module to transfer data between the control and execution environments.

This chapter describes how tags are used to specify which devices to use for operations within VTK-m. The chapter also outlines the features provided by a device adapter that allow you to directly control a device. Finally, we document how to create a new device adapter to port or specialize VTK-m for a different device or system.

8.1 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.

There are two ways to select a device adapter. The first is to make a global selection of a default device adapter. The second is to specify a specific device adapter as a template parameter.

8.1.1 Default Device Adapter

A default device adapter tag is specified in vtkm/cont/DeviceAdapter.h. If no other information is given, VTK-m attempts to choose a default device adapter that is a best fit for the system it is compiled on. VTK-m currently select the default device adapter with the following sequence of conditions.

- If the source code is being compiled by CUDA, the CUDA device is used.

- If the compiler does not support CUDA and VTK-m was configured to use Intel Threading Building Blocks, then that device is used.
- If neither CUDA nor TBB is being used and VTK-m was configured to use OpenMP compiler directives, then the OpenMP device is used.
- If no parallel device adapters are found, then VTK-m falls back to a serial device.

You can also set the default device adapter specifically by setting the VTKM_DEVICE_ADAPTER macro. This macro must be set *before* including any VTK-m files. You can set VTKM_DEVICE_ADAPTER to any one of the following.

VTKM_DEVICE_ADAPTER_SERIAL Performs all computation on the same single thread as the control environment. This device is useful for debugging. This device is always available.

VTKM_DEVICE_ADAPTER_CUDA 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.

VTKM_DEVICE_ADAPTER_OPENMP 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.

VTKM_DEVICE_ADAPTER_TBB 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.

These macros provide a useful mechanism for quickly reconfiguring code to run on different devices. The following example shows a typical block of code at the top of a source file that could be used for quick reconfigurations.

Example 8.1: Macros to port VTK-m code among different devices

```
1 // Uncomment one (and only one) of the following to reconfigure the VTK-m
2 // code to use a particular device. Comment them all to automatically pick a
3 // device.
4 #define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_SERIAL
5 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_CUDA
6 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_OPENMP
7 //#define VTKM_DEVICE_ADAPTER VTKM_DEVICE_ADAPTER_TBB
8
9 #include <vtkm/cont/DeviceAdapter.h>
```

Did you know?

Filters do not actually use the default device adapter tag. They have a more sophisticated device selection mechanism that determines the devices available at runtime and will attempt running on multiple devices.

The default device adapter can always be overridden by specifying a device adapter tag, as described in the next section. There is actually one more internal default device adapter named VTKM_DEVICE_ADAPTER_ERROR that will cause a compile error if any component attempts to use the default device adapter. This feature is most often used in testing code to check when device adapters should be specified.

8.1.2 Specifying Device Adapter Tags

In addition to setting a global default device adapter, it is possible to explicitly set which device adapter to use in any feature provided by VTK-m. This is done by providing a device adapter tag as a template argument to VTK-m templated objects. 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.

The following example uses the tag for the Intel Threading Building blocks device adapter to prepare an output array for that device. In this case, the device adapter tag is passed as a parameter for the `ArrayHandle::PrepareForOutput` method.

Example 8.2: Specifying a device using a device adapter tag.

```
1 |   arrayHandle.PrepareForOutput(50, vtkm::cont::DeviceAdapterTagTBB());
```

Common Errors

A device adapter tag is a class just like every other type in C++. Thus it is possible to accidently use a type that is not a device adapter tag when one is expected as a template argument. This leads to unexpected errors in strange parts of the code. To help identify these errors, it is good practice to use the `VTKM_IS_DEVICE_ADAPTER_TAG` *macro to verify the type is a valid device adapter tag. Example 8.3 uses this macro on line 4.*

When structuring your code to always specify a particular device adapter, consider setting the default device adapter (with the `VTKM_DEVICE_ADAPTER` macro) to `VTKM_DEVICE_ADAPTER_ERROR`. This will cause the compiler to produce an error if any object is instantiated with the default device adapter, which checks to make sure the code properly specifies every device adapter parameter.

VTK-m also defines a macro named `VTKM_DEFAULT_DEVICE_ADAPTER_TAG`, which can be used in place of an explicit device adapter tag to use the default tag. This macro is used to create new templates that have template parameters for device adapters that can use the default. The following example defines a functor to be used with the `DeviceAdapterAlgorithm::Schedule` operation (to be described later) that is templated on the device it uses.

Example 8.3: Specifying a default device for template parameters.

```
template<typename Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG>
struct SetPortalFunctor : vtkm::exec::FunctorBase
{
  VTKM_IS_DEVICE_ADAPTER_TAG(Device);

  using ExecPortalType =
    typename vtkm::cont::ArrayHandle<vtkm::Id>::ExecutionTypes<Device>::Portal;
  ExecPortalType Portal;

  VTKM_CONT
  SetPortalFunctor(vtkm::cont::ArrayHandle<vtkm::Id> array, vtkm::Id size)
    : Portal(array.PrepareForOutput(size, Device()))
  {
  }

  VTKM_EXEC
  void operator()(vtkm::Id index) const
  {
    using ValueType = typename ExecPortalType::ValueType;
    this->Portal.Set(index, TestValue(index, ValueType()));
  }
};
```

Common Errors

There was a time when VTK-m required all user code to specify a device and have a single default device adapter. Since then, VTK-m has become more flexible in the device that it uses and instead encourages code to support multiple device adapters specified by a runtime device tracker. Thus, the use of `VTKM_DEFAULT_DEVICE_ADAPTER_TAG` *is now discouraged and may be removed in future version of VTK-m. Instead, use the runtime device tracker described in Section 8.3.*

8.2 Device Adapter Traits

In Section 6.5 we see that VTK-m defines multiple *traits* classes to publish and retrieve information about types. In addition to traits about basic data types, VTK-m also has instances of defining traits for other features. One such traits class is `vtkm::cont::DeviceAdapterTraits`, which provides some basic information about a device adapter tag. The `DeviceAdapterTraits` class provides the following features.

`DeviceAdapterTraits::GetId` A `static` method taking no arguments that returns a unique integer identifier for the device adapter. The integer identifier is stored in a type named `vtkm::cont::DeviceAdapterId`, which is currently aliased to `vtkm::Int8`. The device adapter id is useful for storing run time information about a device without directly compiling for the class.

`DeviceAdapterTraits::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.

`DeviceAdapterTraits::Valid` A `static const bool` that is true if the implementation of the device is available. The valid flag is useful for conditionally compiling code depending on whether a device is available.

The following example demonstrates using the vtkm::cont::DeviceAdapterId to check whether an array already has its data available on a particular device. Code like this can be used to attempt find a device on which data is already available to avoid moving data across devices. For simplicity, this example just outputs a message.

Example 8.4: Using DeviceAdapterTraits.

```
template<typename ArrayHandleType, typename DeviceAdapterTag>
void CheckArrayHandleDevice(const ArrayHandleType& array, DeviceAdapterTag)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);
  VTKM_IS_DEVICE_ADAPTER_TAG(DeviceAdapterTag);

  vtkm::cont::DeviceAdapterId currentDevice = array.GetDeviceAdapterId();
  if (currentDevice == DeviceAdapterTag())
  {
    std::cout << "Array is already on device " << DeviceAdapterTag().GetName()
              << std::endl;
  }
  else
  {
    std::cout << "Copying array to device " << DeviceAdapterTag().GetName()
              << std::endl;
    array.PrepareForInput(DeviceAdapterTag());
  }
}
```

VTK-m contains multiple devices that might not be available for a variety of reasons. For example, the CUDA device is only available when code is being compiled with the special nvcc compiler. To make it easier to manage devices that may be available in some configurations but not others, VTK-m always defines the device adapter tag structure, but signals whether the device features are available through the traits `Valid` flag. For example, VTK-m always provides the vtkm/cont/cuda/DeviceAdapterCuda.h header file and the vtkm::cont::DeviceAdapterTagCuda tag defined in it. However, vtkm::cont::DeviceAdapterTraits <vtkm::cont::DeviceAdapterTagCuda >::Valid is true if and only if VTK-m was configured to use CUDA and the code is being compiled with nvcc. The following example uses DeviceAdapterTraits to wrap the function defined in Exampleex:DeviceAdapterTraits in a safer version of the function that checks whether the device is valid and will compile correctly in either case.

Example 8.5: Managing invalid devices without compile time errors.

```
namespace detail
{

template<typename ArrayHandleType, typename DeviceAdapterTag>
void SafeCheckArrayHandleDeviceImpl(const ArrayHandleType& array,
                                    DeviceAdapterTag,
                                    std::true_type)
{
  CheckArrayHandleDevice(array, DeviceAdapterTag());
}

template<typename ArrayHandleType, typename DeviceAdapterTag>
void SafeCheckArrayHandleDeviceImpl(const ArrayHandleType&,
                                    DeviceAdapterTag,
                                    std::false_type)
{
  std::cout << "Device " << DeviceAdapterTag().GetName() << " is not available"
            << std::endl;
}

} // namespace detail

```

```
23 template<typename ArrayHandleType, typename DeviceAdapterTag>
24 void SafeCheckArrayHandleDevice(const ArrayHandleType& array, DeviceAdapterTag)
25 {
26   static const bool deviceValid = DeviceAdapterTag::IsEnabled;
27   detail::SafeCheckArrayHandleDeviceImpl(
28     array, DeviceAdapterTag(), std::integral_constant<bool, deviceValid>());
29 }
```

Note that Example 8.5 makes use of `std::integral_constant` to make it easier to overload a function based on a `bool` value. The example also makes use of `std::true_type` and `std::false_type`, which are aliases of true and false Boolean integral constants. They save on typing and make the code more clear.

Did you know?

It is rare that you have to directly query whether a particular device is valid. If you wish to write functions that support multiple devices, it is common to wrap them in a `vtkm::cont::TryExecute`*, which takes care of invalid devices for you.* `TryExecute` *is described in Chapter19.*

Common Errors

Be aware that even though supported VTK-m devices always have a tag and associated traits defined, the rest of the implementation will likely be missing for devices that are not valid. Thus, you are likely to get errors in code that uses an invalid tag in any class that is not `DeviceAdapterTraits`*. For example, you might be tempted to implement the behavior of Example 8.5 by simply adding an* `if` *condition to the function in Example 8.4. However, if you did that, then you would get compile errors in other* `if` *branches that use the invalid device tag (even though they can never be reached). This is why Example 8.5 instead uses function overloading to avoid compiling any code that attempts to use an invalid device adapter.*

8.3 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 4, 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. `RuntimeDeviceTracker` has the following methods.

`RuntimeDeviceTracker::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.

`RuntimeDeviceTracker::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.

`RuntimeDeviceTracker::ResetDevice` Takes a device adapter tag 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.

`RuntimeDeviceTracker::Reset` Resets all devices. This equivocally calls `ResetDevice` for all devices supported by VTK-m.

`RuntimeDeviceTracker::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.

`RuntimeDeviceTracker::DeepCopy` `RuntimeDeviceTracker` behaves like a smart pointer for its state. That is, if you copy a `RuntimeDeviceTracker` and then change the state of one (by, for example, calling `DisableDevice` on one), then the state changes for both. If you want to copy the state of a `RuntimeDeviceTracker` but do not want future changes to effect each other, then use `DeepCopy`. There are two versions of `DeepCopy`. The first version takes no arguments and returns a new `RuntimeDeviceTracker`. The second version takes another instance of a `RuntimeDeviceTracker` and copies its state into this class.

`RuntimeDeviceTracker::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.

A `RuntimeDeviceTracker` can be used to specify which devices to consider for a particular operation. For example, let us say that we want to perform a deep copy of an array using the `vtkm::cont::ArrayCopy` method (described in Section 7.4). 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. `ArrayCopy` takes an optional `RuntimeDeviceTracker` argument, so we can pass in a tracker with the CUDA device disabled.

Example 8.6: Disabling a device with `RuntimeDeviceTracker`.

```
1  vtkm::cont::RuntimeDeviceTracker tracker;
2  tracker.DisableDevice(vtkm::cont::DeviceAdapterTagCuda());
3
4  vtkm::cont::ArrayCopy(srcArray, destArray, tracker);
```

Did you know?

Section 8.2 warned that using device adapter tags for devices that are not available can cause compile time errors when used with most features of VTK-m. This is not the case for `RuntimeDeviceTracker`*. You may pass* `RuntimeDeviceTracker` *any device adapter tag regardless of whether VTK-m is configured for that device or whether the current compiler supports that device. This allows you to set up a* `RuntimeDeviceTracker` *in a translation unit that does not support a particular device and pass it to function compiled in a unit that does.*

It can be tedious to maintain your own `RuntimeDeviceTracker` and pass it to every function that chooses a device. To make this easier, VTK-m maintains a global runtime device tracker, which can be retrieved with the `vtkm::cont::GetGlobalRuntimeDeviceTracker` function. Specifying a `RuntimeDeviceTracker` is almost always optional, and the global runtime device tracker is used if none is specified.

One of the nice features about having a global runtime device tracker is that when an algorithm encounters a problem with a device, it can be marked as disabled and future algorithms can skip over that non-functioning device. That said, it may be the case where you want to re-enable a device previously marked as disabled. For example, an algorithm may disable a device in the global tracker because that device ran out of memory.

However, your code might want to re-enable such devices if moving on to a different data set. This can be done by simply calling a reset method on the global runtime device tracker.

Example 8.7: Resetting the global RuntimeDeviceTracker.

```
vtkm::cont::GetGlobalRuntimeDeviceTracker().Reset();
```

It is also possible to restrict devices that are used through the global runtime device adapter. For example, if you are debugging some code, you might find it useful to restrict VTK-m to use the serial device.

Example 8.8: Globally restricting which devices VTK-m uses.

```
vtkm::cont::GetGlobalRuntimeDeviceTracker().ForceDevice(
  vtkm::cont::DeviceAdapterTagSerial());
```

8.4 Device Adapter Algorithms

VTK-m comes with the templated class vtkm::cont::DeviceAdapterAlgorithm that provides a set of algorithms that can be invoked in the control environment and are run on the execution environment. The template has a single argument that specifies the device adapter tag.

Example 8.9: Prototype for vtkm::cont::DeviceAdapterAlgorithm.

```
namespace vtkm
{
namespace cont
{

template<typename DeviceAdapterTag>
struct DeviceAdapterAlgorithm;
}
} // namespace vtkm
```

DeviceAdapterAlgorithm 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* DeviceAdapterAlgorithm::Copy *method.*

8.4.1 Copy

The DeviceAdapterAlgorithm::Copy method copies data from an input array to an output array. The copy takes place in the execution environment.

Example 8.10: Using the device adapter Copy algorithm.

```
  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
  vtkm::cont::ArrayHandle<vtkm::Int32> input =
    vtkm::cont::make_ArrayHandle(inputBuffer);

```

```
5     vtkm::cont::ArrayHandle<vtkm::Int32> output;
6
7     vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Copy(input, output);
8
9     // output has { 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 }
```

8.4.2 CopyIf

The `DeviceAdapterAlgorithm::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.

`DeviceAdapterAlgorithm::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. A unary predicate is a simple functor with a parentheses argument that has a single argument (in this case, a value of the stencil), and returns true or false. 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 8.11: Using the device adapter `CopyIf` algorithm.

```
1     std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
2     std::vector<vtkm::UInt8> stencilBuffer{ 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1 };
3     vtkm::cont::ArrayHandle<vtkm::Int32> input =
4       vtkm::cont::make_ArrayHandle(inputBuffer);
5     vtkm::cont::ArrayHandle<vtkm::UInt8> stencil =
6       vtkm::cont::make_ArrayHandle(stencilBuffer);
7
8     vtkm::cont::ArrayHandle<vtkm::Int32> output;
9
10    vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::CopyIf(
11      input, stencil, output);
12
13    // output has { 0, 5, 3, 8, 3 }
14
15    struct LessThan5
16    {
17      VTKM_EXEC_CONT bool operator()(vtkm::Int32 x) const { return x < 5; }
18    };
19
20    vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::CopyIf(
21      input, input, output, LessThan5());
22
23    // output has { 0, 1, 1, 4, 3, 3 }
```

8.4.3 CopySubRange

The `DeviceAdapterAlgorithm::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 8.12: Using the device adapter `CopySubRange` algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> output;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::CopySubRange(
  input, 1, 7, output);

// output has { 0, 1, 1, 5, 5, 4, 3 }
```

8.4.4 LowerBounds

The `DeviceAdapterAlgorithm::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 `DeviceAdapterAlgorithm::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` 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 8.13: Using the device adapter `LowerBounds` algorithm.

```
std::vector<vtkm::Int32> sortedBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 };
std::vector<vtkm::Int32> valuesBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
  vtkm::cont::make_ArrayHandle(sortedBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> values =
  vtkm::cont::make_ArrayHandle(valuesBuffer);

vtkm::cont::ArrayHandle<vtkm::Id> output;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::LowerBounds(
  sorted, values, output);

// output has { 8, 0, 1, 1, 6, 6, 5, 3, 8, 10, 11, 3 }

std::vector<vtkm::Int32> revSortedBuffer{ 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 };
vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
  vtkm::cont::make_ArrayHandle(revSortedBuffer);

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::LowerBounds(
  reverseSorted, values, output, vtkm::SortGreater());

// output has { 2, 11, 9, 9, 4, 4, 6, 7, 2, 1, 0, 7 }
```

8.4.5 Reduce

The `DeviceAdapterAlgorithm::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 8.14: Using the device adapter `Reduce` algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 1, 1, 5, 5 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::Int32 sum =
  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Reduce(input, 0);

// sum is 12

vtkm::Int32 product =
  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Reduce(
    input, 1, vtkm::Multiply());

// product is 25
```

8.4.6 ReduceByKey

The `DeviceAdapterAlgorithm::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 8.15: Using the device adapter `ReduceByKey` algorithm.

```
std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Id> uniqueKeys;
vtkm::cont::ArrayHandle<vtkm::Int32> sums;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ReduceByKey(
  keys, input, uniqueKeys, sums, vtkm::Add());

// uniqueKeys is { 0, 3, 5, 6 }
// sums is { 7, 12, 4, 30 }

vtkm::cont::ArrayHandle<vtkm::Int32> products;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ReduceByKey(
  keys, input, uniqueKeys, products, vtkm::Multiply());

// products is { 0, 25, 4, 4536 }
```

8.4.7 ScanExclusive

The `DeviceAdapterAlgorithm::ScanExclusive` method takes an input and an output `ArrayHandle` and performs a running sum on the input array. 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. There are two forms of `ScanExclusive`. The first performs the sum using addition. The second form other accepts a custom binary function to use as the "sum" operator and a custom initial value (instead of 0).

Example 8.16: Using the device adapter `ScanExclusive` algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanExclusive(input,
                                                                    runningSum);

// runningSum is { 0, 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50 }

vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanExclusive(
  input, runningMax, vtkm::Maximum(), -1);

// runningMax is { -1, 7, 7, 7, 7, 7, 7, 7, 7, 8, 9 }
```

8.4.8 ScanExclusiveByKey

The `DeviceAdapterAlgorithm::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 8.17: Using the device adapter `ScanExclusiveByKey` algorithm.

```
std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanExclusiveByKey(
  keys, input, runningSums);

// runningSums is { 0, 7, 0, 1, 2, 7, 0, 0, 3, 10, 18, 27 }

vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanExclusiveByKey(
  keys, input, runningMaxes, -1, vtkm::Maximum());

// runningMax is { -1, 7, -1, 1, 1, 5, -1, -1, 3, 7, 8, 9 }
```

8.4.9 ScanInclusive

The `DeviceAdapterAlgorithm::ScanInclusive` method takes an input and an output `ArrayHandle` and performs a running sum on the input array. 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. `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 8.18: Using the device adapter `ScanInclusive` algorithm.

```
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> runningSum;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanInclusive(input,
                                                                    runningSum);

// runningSum is { 7, 7, 8, 9, 14, 19, 23, 26, 33, 41, 50, 53 }

vtkm::cont::ArrayHandle<vtkm::Int32> runningMax;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanInclusive(
  input, runningMax, vtkm::Maximum());

// runningMax is { 7, 7, 7, 7, 7, 7, 7, 7, 8, 9, 9 }
```

8.4.10 ScanInclusiveByKey

The `DeviceAdapterAlgorithm::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 8.19: Using the device adapter `ScanInclusiveByKey` algorithm.

```
std::vector<vtkm::Id> keyBuffer{ 0, 0, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6 };
std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Id> keys = vtkm::cont::make_ArrayHandle(keyBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> input =
  vtkm::cont::make_ArrayHandle(inputBuffer);

vtkm::cont::ArrayHandle<vtkm::Int32> runningSums;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanInclusiveByKey(
  keys, input, runningSums);

// runningSums is { 7, 7, 1, 2, 7, 12, 4, 3, 10, 18, 27, 30 }

vtkm::cont::ArrayHandle<vtkm::Int32> runningMaxes;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::ScanInclusiveByKey(
  keys, input, runningMaxes, vtkm::Maximum());

// runningMax is { 7, 7, 1, 1, 5, 5, 4, 3, 7, 8, 9, 9 }
```

8.4.11 Schedule

The `DeviceAdapterAlgorithm::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.

8.4.12 Sort

The `DeviceAdapterAlgorithm::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 8.20: Using the device adapter `Sort` algorithm.

```
1  std::vector<vtkm::Int32> inputBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };
2  vtkm::cont::ArrayHandle<vtkm::Int32> array =
3    vtkm::cont::make_ArrayHandle(inputBuffer);
4
5  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Sort(array);
6
7  // array has { 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 }
8
9  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Sort(array,
10                                                            vtkm::SortGreater());
11
12 // array has { 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 }
```

8.4.13 SortByKey

The `DeviceAdapterAlgorithm::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 8.21: Using the device adapter `SortByKey` algorithm.

```
1  std::vector<vtkm::Int32> keyBuffer{ 7, 0, 1, 5, 4, 8, 9, 3 };
2  std::vector<vtkm::Id> valueBuffer{ 0, 1, 2, 3, 4, 5, 6, 7 };
3
4  vtkm::cont::ArrayHandle<vtkm::Int32> keys =
5    vtkm::cont::make_ArrayHandle(keyBuffer);
6  vtkm::cont::ArrayHandle<vtkm::Id> values =
7    vtkm::cont::make_ArrayHandle(valueBuffer);
8
9  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::SortByKey(keys, values);
10
11 // keys has   { 0, 1, 3, 4, 5, 7, 8, 9 }
12 // values has { 1, 2, 7, 4, 3, 0, 5, 6 }
13
14 vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::SortByKey(
15   keys, values, vtkm::SortGreater());
16
17 // keys has   { 9, 8, 7, 5, 4, 3, 1, 0 }
```

```
18 |     // values has { 6, 5, 0, 3, 4, 7, 2, 1 }
```

8.4.14 Synchronize

The `Synchronize` method waits for any asynchronous operations running on the device to complete and then returns.

8.4.15 Unique

The `DeviceAdapterAlgorithm::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 8.22: Using the device adapter `Unique` algorithm.

```
1  |  std::vector<vtkm::Int32> valuesBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 7, 9 };
2  |  vtkm::cont::ArrayHandle<vtkm::Int32> values =
3  |    vtkm::cont::make_ArrayHandle(valuesBuffer);
4  |
5  |  vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::Unique(values);
6  |
7  |  // values has {0, 1, 3, 4, 5, 7, 9}
8  |
9  |  std::vector<vtkm::Float64> fvaluesBuffer{ 0.0, 0.001, 0.0, 1.5, 1.499, 2.0 };
10 |  vtkm::cont::ArrayHandle<vtkm::Float64> fvalues =
11 |    vtkm::cont::make_ArrayHandle(fvaluesBuffer);
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::DeviceAdapterAlgorithm<DeviceAdapterTag>::Unique(
22 |    fvalues, AlmostEqualFunctor());
23 |
24 |  // values has {0.0, 1.5, 2.0}
```

8.4.16 UpperBounds

The `DeviceAdapterAlgorithm::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 8.23: Using the device adapter `UpperBounds` algorithm.

```
std::vector<vtkm::Int32> sortedBuffer{ 0, 1, 1, 3, 3, 4, 5, 5, 7, 7, 8, 9 };
std::vector<vtkm::Int32> valuesBuffer{ 7, 0, 1, 1, 5, 5, 4, 3, 7, 8, 9, 3 };

vtkm::cont::ArrayHandle<vtkm::Int32> sorted =
  vtkm::cont::make_ArrayHandle(sortedBuffer);
vtkm::cont::ArrayHandle<vtkm::Int32> values =
  vtkm::cont::make_ArrayHandle(valuesBuffer);

vtkm::cont::ArrayHandle<vtkm::Id> output;

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::UpperBounds(
  sorted, values, output);

// output has { 10, 1, 3, 3, 8, 8, 6, 5, 10, 11, 12, 5 }

std::vector<vtkm::Int32> revSortedBuffer{ 9, 8, 7, 7, 5, 5, 4, 3, 3, 1, 1, 0 };
vtkm::cont::ArrayHandle<vtkm::Int32> reverseSorted =
  vtkm::cont::make_ArrayHandle(revSortedBuffer);

vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapterTag>::UpperBounds(
  reverseSorted, values, output, vtkm::SortGreater());

// output has { 4, 12, 11, 11, 6, 6, 7, 9, 4, 2, 1, 9 }
```

CHAPTER

NINE

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. In the VTK-m control environment timing is simplified because the control environment operates on a single thread. However, operations invoked in the execution environment may run asynchronously to operations in the control environment.

To ensure that accurate timings can be made, VTK-m provides a `vtkm::cont::Timer` class that is templated on the device adapter to provide an accurate measurement of operations that happen on the device. If not template parameter is provided, the default device adapter is used.

The timing starts when the `Timer` is constructed. The time elapsed can be retrieved with a call to the `Timer::GetElapsedTime` method. This method will block until all operations in the execution environment complete so as to return an accurate time. The timer can be restarted with a call to the `Timer::Reset` method.

Example 9.1: Using `vtkm::cont::Timer`.

```
1  vtkm::filter::PointElevation elevationFilter;
2  elevationFilter.SetUseCoordinateSystemAsField(true);
3  elevationFilter.SetOutputFieldName("elevation");
4
5  vtkm::cont::Timer<> timer;
6
7  vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
8
9  // This code makes sure data is pulled back to the host in a host/device
10 // architecture.
11 vtkm::cont::ArrayHandle<vtkm::Float64> outArray;
12 result.GetField("elevation").GetData().CopyTo(outArray);
13 outArray.GetPortalConstControl();
14
15 vtkm::Float64 elapsedTime = timer.GetElapsedTime();
16
17 std::cout << "Time to run: " << elapsedTime << std::endl;
```

Common Errors

Make sure the `Timer` being used is match to the device adapter used for the computation. This will ensure that the parallel computation is synchronized.

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.

CHAPTER

TEN

FANCY ARRAY STORAGE

Chapter 7 introduces the `vtkm::cont::ArrayHandle` class. In it, 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 10.1.

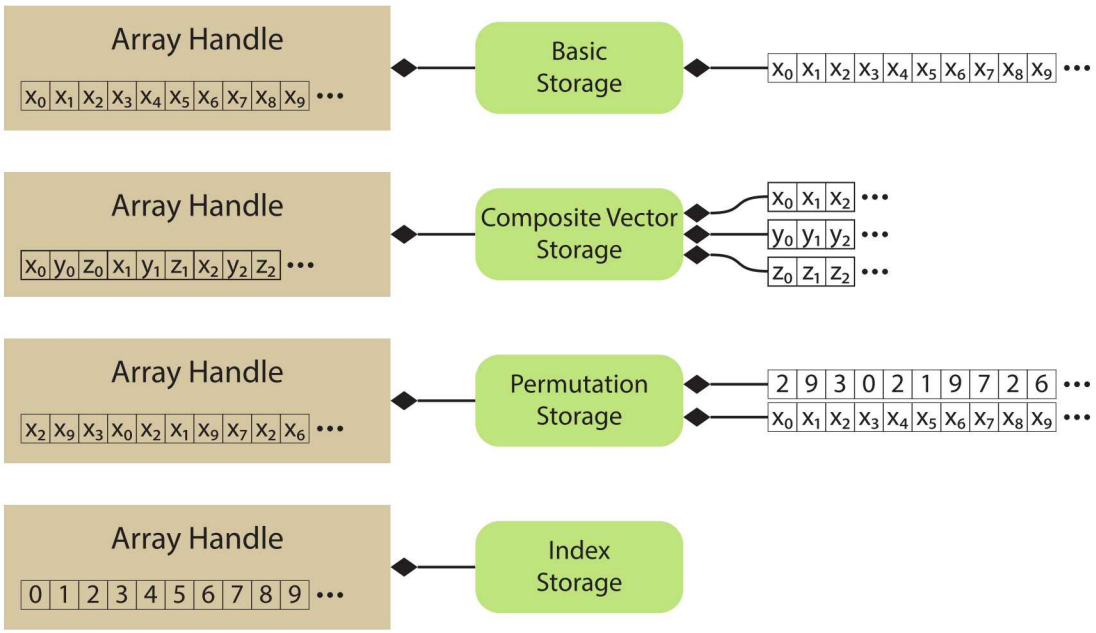


Figure 10.1: Array handles, storage objects, and the underlying data source.

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). In this chapter we explore the many ways you can manipulate

the ArrayHandle storage.

Did you know?

VTK-m comes with many "fancy" array handles that can change the data in other arrays without modifying the memory or can generate data on the fly to behave like an array without actually using any memory. These fancy array handles are documented later in this chapter, and they can be very handy when developing with VTK-m.

10.1 Basic Storage

As previously discussed in Chapter 7, vtkm::cont::ArrayHandle takes two template arguments.

Example 10.1: Declaration of the vtkm::cont::ArrayHandle templated class (again).

```
template<
    typename T,
    typename StorageTag = VTKM_DEFAULT_STORAGE_TAG>
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.

In this and the following sections we describe this storage mechanism. A default storage is specified in much the same way as a default device adapter is defined (as described in Section 8.1.1. It is done by setting the VTKM_STORAGE macro. This macro must be set before including any VTK-m header files. Currently the only practical storage provided by VTK-m is the basic storage, which simply allocates a continuous section of memory of the given base type. This storage can be explicitly specified by setting VTKM_STORAGE to VTKM_STORAGE_BASIC although the basic storage will also be used as the default if no other storage is specified (which is typical).

The default storage can always be overridden by specifying an array storage tag. The tag for the basic storage is located in the vtkm/cont/StorageBasic.h header file and is named vtkm::cont::StorageTagBasic. Here is an example of specifying the storage type when declaring an array handle.

Example 10.2: Specifying the storage type for an ArrayHandle.

```
vtkm::cont::ArrayHandle<vtkm::Float32, vtkm::cont::StorageTagBasic> arrayHandle;
```

VTK-m also defines a macro named VTKM_DEFAULT_STORAGE_TAG that can be used in place of an explicit storage tag to use the default tag. This macro is used to create new templates that have template parameters for storage that can use the default.

10.2 Provided Fancy Arrays

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 do many other operations. 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. This modified behavior for arrays is called "fancy" arrays.

VTK-m provides many of the fancy arrays, which we explore in this section. Later Section 10.3 describes many different ways in which new fancy arrays can be implemented.

10.2.1 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 10.3: Using `ArrayHandleConstant`.

```
// Create an array of 50 entries, all containing the number 3. This could be
// used, for example, to represent the sizes of all the polygons in a set
// where we know all the polygons are triangles.
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 10.4: Using `make_ArrayHandleConstant`.

```
  // Create an array of 50 entries, all containing the number 3.
  vtkm::cont::make_ArrayHandleConstant(3, 50)
```

10.2.2 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 10.5: 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 10.6: 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 10.7: Using `make_ArrayHandleCounting`.

```
// Create an array of 50 entries, all containing the number 3.
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 10.8: 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 10.9: 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)
```

10.2.3 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 10.10: 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::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 10.11: Using make_ArrayHandleCast.

```
vtkm::cont::make_ArrayHandleCast<vtkm::Float64>(originalArray)
```

10.2.4 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 10.12: Using ArrayHandleDiscard.

```
template<typename InputArrayType,
         typename OutputArrayType1,
         typename OutputArrayType2>
VTKM_CONT void DoFoo(InputArrayType input,
                     OutputArrayType1 output1,
                     OutputArrayType2 output2);

template<typename InputArrayType>
VTKM_CONT inline vtkm::cont::ArrayHandle<vtkm::FloatDefault> DoBar(
  InputArrayType input)
{
  VTKM_IS_ARRAY_HANDLE(InputArrayType);

  vtkm::cont::ArrayHandle<vtkm::FloatDefault> keepOutput;

  vtkm::cont::ArrayHandleDiscard<vtkm::FloatDefault> discardOutput;

  DoFoo(input, keepOutput, discardOutput);

  return keepOutput;
}
```

10.2.5 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 10.13: Using ArrayHandlePermutation.

```
  using IdArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
  using IdPortalType = IdArrayType::PortalControl;

  using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Float64>;
  using ValuePortalType = ValueArrayType::PortalControl;

  // Create array with values [0.0, 0.1, 0.2, 0.3]
```

```
8   ValueArrayType valueArray;
9   valueArray.Allocate(4);
10  ValuePortalType valuePortal = valueArray.GetPortalControl();
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.GetPortalControl();
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.GetPortalControl();
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 10.14: 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.

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.

10.2.6 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 10.15: Using `ArrayHandleZip`.

```
using ArrayType1 = vtkm::cont::ArrayHandle<vtkm::Id>;
using PortalType1 = ArrayType1::PortalControl;

using ArrayType2 = vtkm::cont::ArrayHandle<vtkm::Float64>;
using PortalType2 = ArrayType2::PortalControl;

// Create an array of vtkm::Id with values [3, 0, 1]
ArrayType1 array1;
array1.Allocate(3);
PortalType1 portal1 = array1.GetPortalControl();
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.GetPortalControl();
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 10.16: Using `make_ArrayHandleZip`.

```
vtkm::cont::make_ArrayHandleZip(array1, array2)
```

10.2.7 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 12, 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 10.17: 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::Vec<vtkm::FloatDefault, 3>{ -5.0f, -3.0f, -3.0f },
  vtkm::Vec<vtkm::FloatDefault, 3>{ 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 10.18: Using a `ArrayHandleCartesianProduct`.

```
using AxisArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
using AxisPortalType = AxisArrayType::PortalControl;

// Create array for x axis coordinates with values [0.0, 1.1, 5.0]
AxisArrayType xAxisArray;
xAxisArray.Allocate(3);
AxisPortalType xAxisPortal = xAxisArray.GetPortalControl();
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.GetPortalControl();
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.GetPortalControl();
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::
  ArrayHandleCartesianProduct<AxisArrayType, AxisArrayType, AxisArrayType>
    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 10.19: 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.

10.2.8 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 10.2.6) 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 `Vec`s are generated as the array is used. Writing `Vec`s 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 10.20: 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::PortalControl arrayPortal2 = array2.GetPortalControl();
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::PortalControl arrayPortal3 = array3.GetPortalControl();
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 10.21: Using make_ArrayHandleCompositeVector.

```
1  vtkm::cont::make_ArrayHandleCompositeVector(array1, array2, array3, array4)
```

10.2.9 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 10.2.8) 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 10.22: Extracting components of Vecs in an array with ArrayHandleExtractComponent.

```
1  using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::Float64, 3>>;
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.GetPortalControl();
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 10.23: Using make_ArrayHandleExtractComponent.

```
1  vtkm::cont::make_ArrayHandleExtractComponent(valueArray, 2)
```

10.2.10 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 10.24: Swizzling components of Vecs in an array with ArrayHandleSwizzle.

```
using ValueArrayType = vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::Float64, 4>>;

// 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.GetPortalControl();
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::Vec<vtkm::IdComponent, 3>(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 10.25: Using make_ArrayHandleSwizzle.

```
  vtkm::cont::make_ArrayHandleSwizzle(valueArray, 2, 0, 3)
```

10.2.11 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 10.26: 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)]
```

```
6   vtkm::cont::ArrayHandleGroupVec<ArrayType, 2> vec2Array(sourceArray);
7
8   // Create an array containing [(0,1,2), (3,4,5), (6,7,8), (9,10,11)]
9   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 `Vec`s. You must specify the size of the `Vec`s as a template parameter when using `vtkm::cont::make_ArrayHandleGroupVec`.

Example 10.27: Using `make_ArrayHandleGroupVec`.

```
1     // Create an array containing [(0,1,2,3), (4,5,6,7), (8,9,10,11)]
2     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 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 10.28: Using `ArrayHandleGroupVecVariable`.

```
1   // Create an array of counts containing [4, 2, 3, 3]
2   vtkm::IdComponent countBuffer[4] = { 4, 2, 3, 3 };
3   vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray =
4     vtkm::cont::make_ArrayHandle(countBuffer, 4);
5
6   // Convert the count array to an offset array [0, 4, 6, 9]
7   // Returns the number of total components: 12
8   vtkm::Id sourceArraySize;
9   using OffsetArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;
10  OffsetArrayType offsetArray =
11    vtkm::cont::ConvertNumComponentsToOffsets(countArray, sourceArraySize);
12
13  // Create an array containing [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
14  using SourceArrayType = vtkm::cont::ArrayHandleIndex;
15  SourceArrayType sourceArray(sourceArraySize);
16
17  // Create an array containing [(0,1,2,3), (4,5), (6,7,8), (9,10,11)]
18  vtkm::cont::ArrayHandleGroupVecVariable<SourceArrayType, OffsetArrayType>
19    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 10.29: 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 6.4.2 and 6.5.2 for more information on the difference between* `vtkm::Vec` *and* `Vec`*-like objects.*

10.3 Implementing Fancy Arrays

Although the behavior of fancy arrays might seem complicated, they are actually straightforward to implement. VTK-m provides several mechanisms to implement fancy arrays.

10.3.1 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. 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 10.30: Functor that doubles an index.

```
struct DoubleIndexFunctor
{
  VTKM_EXEC_CONT
  vtkm::Id operator()(vtkm::Id index) const { return 2 * index; }
};
```

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 10.31: Declaring a `ArrayHandleImplicit`.

```
  vtkm::cont::ArrayHandleImplicit<DoubleIndexFunctor> implicitArray(
    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 10.32: Using `make_ArrayHandleImplicit`.

```
  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 10.33: 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 10.33 is not templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS_NT macro. (The other macro is described in Section 10.3.2 on page 118). 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.

10.3.2 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 first array applied to the functor.

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 10.34: Functor to scale and bias a value.

```
1  template<typename T>
2  struct ScaleBiasFunctor
3  {
4    VTKM_EXEC_CONT
5    ScaleBiasFunctor(T scale = T(1), T bias = T(0))
6      : Scale(scale)
7      , Bias(bias)
8    {
9    }
10
11   VTKM_EXEC_CONT
12   T operator()(T x) const { return this->Scale * x + this->Bias; }
13
14   T Scale;
15   T Bias;
16 };
```

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 10.35: 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 10.36: 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>>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleScaleBias,
    (ArrayHandleScaleBias<ArrayHandleType>),
    (vtkm::cont::ArrayHandleTransform<
      ArrayHandleType,
      ScaleBiasFunctor<typename ArrayHandleType::ValueType>>));

  VTKM_CONT
  ArrayHandleScaleBias(const ArrayHandleType& array, ValueType scale, ValueType bias)
    : Superclass(array, ScaleBiasFunctor<ValueType>(scale, bias))
  {
  }
};

template<typename ArrayHandleType>
VTKM_CONT ArrayHandleScaleBias<ArrayHandleType> make_ArrayHandleScaleBias(
  const ArrayHandleType& array,
  typename ArrayHandleType::ValueType scale,
  typename ArrayHandleType::ValueType bias)
{
  return ArrayHandleScaleBias<ArrayHandleType>(array, scale, bias);
}
```

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 10.36 is templated, so it uses the VTKM_ARRAY_HANDLE_SUBCLASS macro. (The other macro is described in Section 10.4 on page 131). 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.

10.3.3 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 10.3.2) 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 concatenated together. We could of course actually copy the data, but we can also do it in place.

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 (or have a separate version for control and execution).

Example 10.37: Derived array portal for concatenated arrays.

```
#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/ArrayPortal.h>

template<typename P1, typename P2>
class ArrayPortalConcatenate
{
public:
  using PortalType1 = P1;
  using PortalType2 = P2;
  using ValueType = typename PortalType1::ValueType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  ArrayPortalConcatenate()
    : Portal1()
    , Portal2()
  {
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  ArrayPortalConcatenate(const PortalType1& portal1, const PortalType2 portal2)
    : Portal1(portal1)
    , Portal2(portal2)
  {
  }

  /// Copy constructor for any other ArrayPortalConcatenate 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).
  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OtherP1, typename OtherP2>
  VTKM_EXEC_CONT ArrayPortalConcatenate(
    const ArrayPortalConcatenate<OtherP1, OtherP2>& src)
    : Portal1(src.GetPortal1())
    , Portal2(src.GetPortal2())
  {
  }

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return this->Portal1.GetNumberOfValues() + this->Portal2.GetNumberOfValues();
  }
```

```
46
47    VTKM_SUPPRESS_EXEC_WARNINGS
48    VTKM_EXEC_CONT
49    ValueType Get(vtkm::Id index) const
50    {
51      if (index < this->Portal1.GetNumberOfValues())
52      {
53        return this->Portal1.Get(index);
54      }
55      else
56      {
57        return this->Portal2.Get(index - this->Portal1.GetNumberOfValues());
58      }
59    }
60
61    VTKM_SUPPRESS_EXEC_WARNINGS
62    VTKM_EXEC_CONT
63    void Set(vtkm::Id index, const ValueType& value) const
64    {
65      if (index < this->Portal1.GetNumberOfValues())
66      {
67        this->Portal1.Set(index, value);
68      }
69      else
70      {
71        this->Portal2.Set(index - this->Portal1.GetNumberOfValues(), value);
72      }
73    }
74
75    VTKM_EXEC_CONT
76    const PortalType1& GetPortal1() const { return this->Portal1; }
77    VTKM_EXEC_CONT
78    const PortalType2& GetPortal2() const { return this->Portal2; }
79
80  private:
81    PortalType1 Portal1;
82    PortalType2 Portal2;
83  };
```

Like in an adapter storage, the next step in creating a derived storage is to define a tag for the adapter. We shall call ours `StorageTagConcatenate` 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 implementation for a `Storage` for a derived storage is usually trivial compared to an adapter storage because the majority of the work is deferred to the derived arrays.

Example 10.38: `Storage` for derived container of concatenated arrays.

```
1  template<typename ArrayHandleType1, typename ArrayHandleType2>
2  struct StorageTagConcatenate
3  {
4  };
5
6  namespace vtkm
7  {
8  namespace cont
9  {
10 namespace internal
11 {
12
13 template<typename ArrayHandleType1, typename ArrayHandleType2>
14 class Storage<typename ArrayHandleType1::ValueType,
15               StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>
16 {
17 public:
```

```
18  using ValueType = typename ArrayHandleType1::ValueType;
19
20  using PortalType =
21    ArrayPortalConcatenate<typename ArrayHandleType1::PortalControl,
22                           typename ArrayHandleType2::PortalControl>;
23  using PortalConstType =
24    ArrayPortalConcatenate<typename ArrayHandleType1::PortalConstControl,
25                           typename ArrayHandleType2::PortalConstControl>;
26
27  VTKM_CONT
28  Storage()
29    : Valid(false)
30  {
31  }
32
33  VTKM_CONT
34  Storage(const ArrayHandleType1 array1, const ArrayHandleType2 array2)
35    : Array1(array1)
36    , Array2(array2)
37    , Valid(true)
38  {
39  }
40
41  VTKM_CONT
42  PortalType GetPortal()
43  {
44    VTKM_ASSERT(this->Valid);
45    return PortalType(this->Array1.GetPortalControl(),
46                      this->Array2.GetPortalControl());
47  }
48
49  VTKM_CONT
50  PortalConstType GetPortalConst() const
51  {
52    VTKM_ASSERT(this->Valid);
53    return PortalConstType(this->Array1.GetPortalConstControl(),
54                           this->Array2.GetPortalConstControl());
55  }
56
57  VTKM_CONT
58  vtkm::Id GetNumberOfValues() const
59  {
60    VTKM_ASSERT(this->Valid);
61    return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
62  }
63
64  VTKM_CONT
65  void Allocate(vtkm::Id numberOfValues)
66  {
67    VTKM_ASSERT(this->Valid);
68    // This implementation of allocate, which allocates the same amount in both
69    // arrays, is arbitrary. It could, for example, leave the size of Array1
70    // alone and change the size of Array2. Or, probably most likely, it could
71    // simply throw an error and state that this operation is invalid.
72    vtkm::Id half = numberOfValues / 2;
73    this->Array1.Allocate(numberOfValues - half);
74    this->Array2.Allocate(half);
75  }
76
77  VTKM_CONT
78  void Shrink(vtkm::Id numberOfValues)
79  {
80    VTKM_ASSERT(this->Valid);
81    if (numberOfValues < this->Array1.GetNumberOfValues())
```

```
82      {
83        this->Array1.Shrink(numberOfValues);
84        this->Array2.Shrink(0);
85      }
86      else
87      {
88        this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
89      }
90    }
91
92    VTKM_CONT
93    void ReleaseResources()
94    {
95      VTKM_ASSERT(this->Valid);
96      this->Array1.ReleaseResources();
97      this->Array2.ReleaseResources();
98    }
99
100   // Requried for later use in ArrayTransfer class.
101   VTKM_CONT
102   const ArrayHandleType1& GetArray1() const
103   {
104     VTKM_ASSERT(this->Valid);
105     return this->Array1;
106   }
107   VTKM_CONT
108   const ArrayHandleType2& GetArray2() const
109   {
110     VTKM_ASSERT(this->Valid);
111     return this->Array2;
112   }
113
114 private:
115   ArrayHandleType1 Array1;
116   ArrayHandleType2 Array2;
117   bool Valid;
118 };
119
120 } // namespace internal
121 } // namespace cont
122 } // namespace vtkm
```

One of the responsibilities of an array handle is to copy data between the control and execution environments. The default behavior is to request the device adapter to copy data items from one environment to another. This might involve transferring data between a host and device. For an array of data resting in memory, this is necessary. However, implicit storage (described in the previous section) overrides this behavior to pass nothing but the functional array portal. Likewise, it is undesirable to do a raw transfer of data with derived storage. The underlying arrays being derived may be used in other contexts, and it would be good to share the data wherever possible. It is also sometimes more efficient to copy data independently from the arrays being derived than from the derived storage itself.

The mechanism that controls how a particular storage gets transferred to and from the execution environment is encapsulated in the templated `vtkm::cont::internal::ArrayTransfer` class. By creating a specialization of `vtkm::cont::internal::ArrayTransfer`, we can modify the transfer behavior to instead transfer the arrays being derived and use the respective copies in the control and execution environments.

`vtkm::cont::internal::ArrayTransfer` has three template arguments: the base type of the array, the storage tag, and the device adapter tag.

Example 10.39: Prototype for `vtkm::cont::internal::ArrayTransfer`.

```
1 namespace vtkm
```

```
{
namespace cont
{
namespace internal
{

template<typename T, typename StorageTag, typename DeviceAdapterTag>
class ArrayTransfer;
}
} // namespace cont
} // namespace vtkm
```

All `vtkm::cont::internal::ArrayTransfer` implementations must have a constructor method that accepts a pointer to a `vtkm::cont::internal::Storage` object templated to the same base type and storage tag as the `ArrayTransfer` object. Assuming that an `ArrayHandle` is templated using the parameters in Example 10.39, the prototype for the constructor must be equivalent to the following.

Example 10.40: Prototype for `ArrayTransfer` constructor.

```
ArrayTransfer(vtkm::cont::internal::Storage<T, StorageTag> *storage);
```

Typically the constructor either saves the `Storage` pointer or other relevant objects from the `Storage` for later use in the methods.

In addition to this non-default constructor, the `vtkm::cont::internal::ArrayTransfer` specialization must define the following items.

`ArrayTransfer::ValueType` The type for each item in the array. This is the same type as the first template argument.

`ArrayTransfer::PortalControl` The type of an array portal that is used to access the underlying data in the control environment.

`ArrayTransfer::PortalConstControl` A read-only (const) version of `PortalControl`.

`ArrayTransfer::PortalExecution` The type of an array portal that is used to access the underlying data in the execution environment.

`ArrayTransfer::PortalConstExecution` A read-only (const) version of `PortalExecution`.

`ArrayTransfer::GetNumberOfValues` A method that returns the number of values currently allocated in the execution environment. The results may be undefined if none of the load or allocate methods have yet been called.

`ArrayTransfer::PrepareForInput` A method responsible for transferring data from the control to the execution for input. `PrepareForInput` has one Boolean argument that controls whether this transfer should actually take place. When true, data from the `Storage` object given in the constructor should be transferred to the execution environment; otherwise the data should not be copied. An `ArrayTransfer` for a derived array typically ignores this parameter since the arrays being derived manages this transfer already. Regardless of the Boolean flag, a `PortalConstExecution` is returned.

`ArrayTransfer::PrepareForInPlace` A method that behaves just like `PrepareForInput` except that the data in the execution environment is used for both reading and writing so the method returns a `PortalExecution`. If the array is considered read-only, which is common for derived arrays, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`ArrayTransfer::PrepareForOutput` A method that takes a size (in a `vtkm::Id`) and allocates an array in the execution environment of the specified size. The initial memory can be uninitialized. The method returns a

`PortalExecution` for the allocated data. If the array is considered read-only, which is common for derived arrays, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`ArrayTransfer::RetrieveOutputData` This method takes an array storage pointer (which is the same as that passed to the constructor, but provided for convenience), allocates memory in the control environment, and copies data from the execution environment into it. If the derived array is considered read-only and both `PrepareForInPlace` and `PrepareForOutput` throw exceptions, then this method should never be called. If it is, then that is probably a bug in `ArrayHandle`, and it is OK to throw `vtkm::cont::ErrorControlInternal`.

`ArrayTransfer::Shrink` A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked. If the derived array is considered read-only, then this method should throw a `vtkm::cont::ErrorControlBadValue`.

`ArrayTransfer::ReleaseResources` A method that frees any resources (typically memory) in the execution environment.

Continuing our example derived storage that concatenates two arrays started in Examples 10.37 and 10.38, the following provides an `ArrayTransfer` appropriate for the derived storage.

Example 10.41: `ArrayTransfer` for derived storage of concatenated arrays.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename ArrayHandleType1, typename ArrayHandleType2, typename Device>
9  class ArrayTransfer<typename ArrayHandleType1::ValueType,
10                     StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>,
11                     Device>
12 {
13 public:
14   using ValueType = typename ArrayHandleType1::ValueType;
15
16 private:
17   using StorageTag = StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>;
18   using StorageType = vtkm::cont::internal::Storage<ValueType, StorageTag>;
19
20 public:
21   using PortalControl = typename StorageType::PortalType;
22   using PortalConstControl = typename StorageType::PortalConstType;
23
24   using PortalExecution = ArrayPortalConcatenate<
25     typename ArrayHandleType1::template ExecutionTypes<Device>::Portal,
26     typename ArrayHandleType2::template ExecutionTypes<Device>::Portal>;
27   using PortalConstExecution = ArrayPortalConcatenate<
28     typename ArrayHandleType1::template ExecutionTypes<Device>::PortalConst,
29     typename ArrayHandleType2::template ExecutionTypes<Device>::PortalConst>;
30
31   VTKM_CONT
32   ArrayTransfer(StorageType* storage)
33     : Array1(storage->GetArray1())
34     , Array2(storage->GetArray2())
35   {
36   }
37
38   VTKM_CONT
```

```
39    vtkm::Id GetNumberOfValues() const
40    {
41      return this->Array1.GetNumberOfValues() + this->Array2.GetNumberOfValues();
42    }
43
44    VTKM_CONT
45    PortalConstExecution PrepareForInput(bool vtkmNotUsed(updateData))
46    {
47      return PortalConstExecution(this->Array1.PrepareForInput(Device()),
48                                  this->Array2.PrepareForInput(Device()));
49    }
50
51    VTKM_CONT
52    PortalExecution PrepareForInPlace(bool vtkmNotUsed(updateData))
53    {
54      return PortalExecution(this->Array1.PrepareForInPlace(Device()),
55                             this->Array2.PrepareForInPlace(Device()));
56    }
57
58    VTKM_CONT
59    PortalExecution PrepareForOutput(vtkm::Id numberOfValues)
60    {
61      // This implementation of allocate, which allocates the same amount in both
62      // arrays, is arbitrary. It could, for example, leave the size of Array1
63      // alone and change the size of Array2. Or, probably most likely, it could
64      // simply throw an error and state that this operation is invalid.
65      vtkm::Id half = numberOfValues / 2;
66      return PortalExecution(
67        this->Array1.PrepareForOutput(numberOfValues - half, Device()),
68        this->Array2.PrepareForOutput(half, Device()));
69    }
70
71    VTKM_CONT
72    void RetrieveOutputData(StorageType* vtkmNotUsed(storage)) const
73    {
74      // Implementation of this method should be unnecessary. The internal
75      // array handles should automatically retrieve the output data as
76      // necessary.
77    }
78
79    VTKM_CONT
80    void Shrink(vtkm::Id numberOfValues)
81    {
82      if (numberOfValues < this->Array1.GetNumberOfValues())
83      {
84        this->Array1.Shrink(numberOfValues);
85        this->Array2.Shrink(0);
86      }
87      else
88      {
89        this->Array2.Shrink(numberOfValues - this->Array1.GetNumberOfValues());
90      }
91    }
92
93    VTKM_CONT
94    void ReleaseResources()
95    {
96      this->Array1.ReleaseResourcesExecution();
97      this->Array2.ReleaseResourcesExecution();
98    }
99
100 private:
101   ArrayHandleType1 Array1;
102   ArrayHandleType2 Array2;
```

```
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

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 10.42: `ArrayHandle` for derived storage of concatenated arrays.

```
template<typename ArrayHandleType1, typename ArrayHandleType2>
class ArrayHandleConcatenate
  : public vtkm::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>
{
public:
  VTKM_ARRAY_HANDLE_SUBCLASS(
    ArrayHandleConcatenate,
    (ArrayHandleConcatenate<ArrayHandleType1, ArrayHandleType2>),
    (vtkm::cont::ArrayHandle<
      typename ArrayHandleType1::ValueType,
      StorageTagConcatenate<ArrayHandleType1, ArrayHandleType2>>));

private:
  using StorageType = vtkm::cont::internal::Storage<ValueType, StorageTag>;

public:
  VTKM_CONT
  ArrayHandleConcatenate(const ArrayHandleType1& array1,
                         const ArrayHandleType2& array2)
    : Superclass(StorageType(array1, array2))
  {
  }
};
```

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 10.42 is templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS` macro. (The other macro is described in Section 10.4 on page 131). 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.

`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.

10.4 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.

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 `FooFields` 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 `std::deque` object.

Example 10.43: Fictitious field storage used in custom array storage examples.

```
#include <deque>

struct FooFields
{
  float Pressure;
  float Temperature;
  float Velocity[3];
  // And so on...
};

using FooFieldsDeque = std::deque<FooFields>;
```

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 `FooFieldsDeque` object.

The first step in creating an adapter storage is to create a control environment array portal to the data. This is described in more detail in Section 7.2 and is generally straightforward for simple containers like this. Here is an example implementation for our `FooFieldsDeque` container.

Example 10.44: Array portal to adapt a third-party container to VTK-m.

```
#include <vtkm/Assert.h>
#include <vtkm/cont/internal/IteratorFromArrayPortal.h>

// DequeType expected to be either FooFieldsDeque or const FooFieldsDeque
template<typename DequeType>
class ArrayPortalFooPressure
{
public:
  using ValueType = float;

  VTKM_CONT
  ArrayPortalFooPressure()
    : Container(NULL)
```

```
14    {
15    }
16
17    VTKM_CONT
18    ArrayPortalFooPressure(DequeType* container)
19      : Container(container)
20    {
21    }
22
23    // Required to copy compatible types of ArrayPortalFooPressure. Really needed
24    // to copy from non-const to const versions of array portals.
25    template<typename OtherDequeType>
26    VTKM_CONT ArrayPortalFooPressure(
27      const ArrayPortalFooPressure<OtherDequeType>& other)
28      : Container(other.GetContainer())
29    {
30    }
31
32    VTKM_CONT
33    vtkm::Id GetNumberOfValues() const
34    {
35      return static_cast<vtkm::Id>(this->Container->size());
36    }
37
38    VTKM_CONT
39    ValueType Get(vtkm::Id index) const
40    {
41      VTKM_ASSERT(index >= 0);
42      VTKM_ASSERT(index < this->GetNumberOfValues());
43      return (*this->Container)[index].Pressure;
44    }
45
46    VTKM_CONT
47    void Set(vtkm::Id index, ValueType value) const
48    {
49      VTKM_ASSERT(index >= 0);
50      VTKM_ASSERT(index < this->GetNumberOfValues());
51      (*this->Container)[static_cast<std::size_t>(index)].Pressure = value;
52    }
53
54    // Here for the copy constructor.
55    VTKM_CONT
56    DequeType* GetContainer() const { return this->Container; }
57
58  private:
59    DequeType* Container;
60  };
```

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 10.45: Prototype for `vtkm::cont::internal::Storage`.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
```

```
template<typename T, class StorageTag>
class Storage;
}
} // namespace cont
} // namespace vtkm
```

The `vtkm::cont::internal::Storage` must define the following items.

`Storage::ValueType` The type of each item in the array. This is the same type as the first template argument.

`Storage::PortalType` The type of an array portal that can be used to access the underlying data. This array portal needs to work only in the control environment.

`Storage::PortalConstType` A read-only (const) version of `PortalType`.

`Storage::GetPortal` A method that returns an array portal of type `PortalType` that can be used to access the data manged in this storage.

`Storage::GetPortalConst` Same as `GetPortal` except it returns a read-only (const) array portal.

`Storage::GetNumberOfValues` A method that returns the number of values the storage is currently allocated for.

`Storage::Allocate` A method that allocates the array to a given size. All values stored in the previous allocation may be destroyed.

`Storage::Shrink` A method like `Allocate` with two differences. First, the size of the allocation must be smaller than the existing allocation when the method is called. Second, any values currently stored in the array will be valid after the array is resized. This constrained form of allocation allows the array to be resized and values valid without ever having to copy data.

`Storage::ReleaseResources` A method that instructs the storage to free all of its memory.

The following provides an example implementation of our adapter to a `FooFieldsDeque`. It relies on the `ArrayPortalFooPressure` provided in Example 10.44.

Example 10.46: Storage to adapt a third-party container to VTK-m.

```
// Includes or definition for ArrayPortalFooPressure

struct StorageTagFooPressure
{
};

namespace vtkm
{
namespace cont
{
namespace internal
{

template<>
class Storage<float, StorageTagFooPressure>
{
public:
  using ValueType = float;

  using PortalType = ArrayPortalFooPressure<FooFieldsDeque>;
  using PortalConstType = ArrayPortalFooPressure<const FooFieldsDeque>;

```

```
  VTKM_CONT
  Storage()
    : Container(NULL)
  {
  }

  VTKM_CONT
  Storage(FooFieldsDeque* container)
    : Container(container)
  {
  }

  VTKM_CONT
  PortalType GetPortal() { return PortalType(this->Container); }

  VTKM_CONT
  PortalConstType GetPortalConst() const { return PortalConstType(this->Container); }

  VTKM_CONT
  vtkm::Id GetNumberOfValues() const
  {
    return static_cast<vtkm::Id>(this->Container->size());
  }

  VTKM_CONT
  void Allocate(vtkm::Id numberOfValues)
  {
    this->Container->resize(static_cast<std::size_t>(numberOfValues));
  }

  VTKM_CONT
  void Shrink(vtkm::Id numberOfValues)
  {
    this->Container->resize(static_cast<std::size_t>(numberOfValues));
  }

  VTKM_CONT
  void ReleaseResources() { this->Container->clear(); }

private:
  FooFieldsDeque* Container;
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

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 10.47: Array handle to adapt a third-party container to VTK-m.

```
class ArrayHandleFooPressure
  : public vtkm::cont::ArrayHandle<float, StorageTagFooPressure>
{
private:
  using StorageType = vtkm::cont::internal::Storage<float, StorageTagFooPressure>;

public:
  VTKM_ARRAY_HANDLE_SUBCLASS_NT(
    ArrayHandleFooPressure,
    (vtkm::cont::ArrayHandle<float, StorageTagFooPressure>));

```

```
12    VTKM_CONT
13    ArrayHandleFooPressure(FooFieldsDeque* container)
14      : Superclass(StorageType(container))
15    {
16    }
17  };
```

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 10.47 is not templated, so it uses the `VTKM_ARRAY_HANDLE_SUBCLASS_NT` macro. (The other macro is described in Section 10.3.2 on page 118). 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.

With this new version of `ArrayHandle`, VTK-m can now read to and write from the `FooFieldsDeque` structure directly.

Example 10.48: Using an `ArrayHandle` with custom container.

```
1  VTKM_CONT
2  void GetElevationAirPressure(vtkm::cont::DataSet grid, FooFieldsDeque* fields)
3  {
4    // Make an array handle that points to the pressure values in the fields.
5    ArrayHandleFooPressure pressureHandle(fields);
6
7    // Use the elevation worklet to estimate atmospheric pressure based on the
8    // height of the point coordinates. Atmospheric pressure is 101325 Pa at
9    // sea level and drops about 12 Pa per meter.
10   vtkm::worklet::PointElevation elevation;
11   elevation.SetLowPoint(vtkm::make_Vec(0.0, 0.0, 0.0));
12   elevation.SetHighPoint(vtkm::make_Vec(0.0, 0.0, 2000.0));
13   elevation.SetRange(101325.0, 77325.0);
14
15   vtkm::worklet::DispatcherMapField<vtkm::worklet::PointElevation> dispatcher(
16     elevation);
17   dispatcher.Invoke(grid.GetCoordinateSystem().GetData(), pressureHandle);
18
19   // Make sure the values are flushed back to the control environment.
20   pressureHandle.SyncControlArray();
21
22   // Now the pressure field is in the fields container.
23 }
```

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 10.48.*

Most of the code in VTK-m will create ArrayHandles using the default storage, which is set to the basic storage if not otherwise specified. If you wish to replace the default storage used, then set the VTKM_STORAGE macro to VTKM_STORAGE_UNDEFINED and set the VTKM_DEFAULT_STORAGE_TAG to your tag class. These definitions have to happen *before* including any VTK-m header files. You will also have to declare the tag class (or at least a prototype of it) before including VTK-m header files.

Example 10.49: Redefining the default array handle storage.

```
1 #define VTKM_STORAGE VTKM_STORAGE_UNDEFINED
2 #define VTKM_DEFAULT_STORAGE_TAG StorageTagFooPressure
3
4 struct StorageTagFooPressure;
```

Common Errors

ArrayHandles are often stored in dynamic objects like dynamic arrays (Chapter 11) or data sets (Chapter 12). When this happens, the array's type information, including the storage used, is lost. VTK-m will have to guess the storage, and if you do not tell VTK-m to try your custom storage, you will get a runtime error when the array is used. The most common ways of doing this are to change the default storage tag (described here), adding the storage tag to the default storage list (Section 11.3) or specifying the storage tag in the policy when executing filters.

CHAPTER

ELEVEN

DYNAMIC 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 `DynamicArrayHandle` class provides a mechanism to manage arrays of data with unspecified types.

`vtkm::cont::DynamicArrayHandle` holds a reference to an array. Unlike `ArrayHandle`, `DynamicArrayHandle` is *not* templated. Instead, it uses C++ run-type type information to store the array without type and cast it when appropriate.

A `DynamicArrayHandle` can be established by constructing it with or assigning it to an `ArrayHandle`. The following example demonstrates how a `DynamicArrayHandle` might be used to load an array whose type is not known until run-time.

Example 11.1: Creating a `DynamicArrayHandle`.

```
1  VTKM_CONT
2  vtkm::cont::DynamicArrayHandle LoadDynamicArray(const void* buffer,
3                                                  vtkm::Id length,
4                                                  std::string type)
5  {
6    vtkm::cont::DynamicArrayHandle handle;
7    if (type == "float")
8    {
9      vtkm::cont::ArrayHandle<vtkm::Float32> concreteArray =
10       vtkm::cont::make_ArrayHandle(reinterpret_cast<const vtkm::Float32*>(buffer),
11                                    length);
12     handle = concreteArray;
13   }
14   else if (type == "int")
15   {
16     vtkm::cont::ArrayHandle<vtkm::Int32> concreteArray =
17       vtkm::cont::make_ArrayHandle(reinterpret_cast<const vtkm::Int32*>(buffer),
18                                    length);
19     handle = concreteArray;
20   }
21   return handle;
22 }
```

11.1 Querying and Casting

Data pointed to by a `DynamicArrayHandle` is not directly accessible. However, there are a few generic queries you can make without directly knowing the data type. The `GetNumberOfValues` method returns the length of the array with respect to its base data type. It is also common in VTK-m to use data types, such as `vtkm::Vec`,

with multiple components per value. The `GetNumberOfComponents` method returns the number of components in a vector-like type (or 1 for scalars).

Example 11.2: Non type-specific queries on `DynamicArrayHandle`.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle scalarDynamicHandle(
  vtkm::cont::make_ArrayHandle(scalarBuffer));

// This returns 10.
vtkm::Id scalarArraySize = scalarDynamicHandle.GetNumberOfValues();

// This returns 1.
vtkm::IdComponent scalarComponents = scalarDynamicHandle.GetNumberOfComponents();

std::vector<vtkm::Vec<vtkm::Float32, 3>> vectorBuffer(20);
vtkm::cont::DynamicArrayHandle vectorDynamicHandle(
  vtkm::cont::make_ArrayHandle(vectorBuffer));

// This returns 20.
vtkm::Id vectorArraySize = vectorDynamicHandle.GetNumberOfValues();

// This returns 3.
vtkm::IdComponent vectorComponents = vectorDynamicHandle.GetNumberOfComponents();
```

It is also often desirable to create a new array based on the underlying type of a `DynamicArrayHandle`. For example, when a filter creates a field, it is common to make this output field the same type as the input. To satisfy this use case, `DynamicArrayHandle` has a method named `NewInstance` that creates a new empty array with the same underlying type as the original array.

Example 11.3: Using `NewInstance`.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::DynamicArrayHandle dynamicHandle(
  vtkm::cont::make_ArrayHandle(scalarBuffer));

// This creates a new empty array of type Float32.
vtkm::cont::DynamicArrayHandle newDynamicArray = dynamicHandle.NewInstance();
```

Before the data with a `DynamicArrayHandle` can be accessed, the type and storage of the array must be established. This is usually done internally within VTK-m when a worklet or filter? is invoked. However, it is also possible to query the types and cast to a concrete `ArrayHandle`.

You can query the component type and storage type using the `IsType`, `IsSameType`, and `IsTypeAndStorage` methods. `IsType` takes an example array handle type and returns whether the underlying array matches the given static array type. `IsSameType` behaves the same as `IsType` but accepts an instance of an `ArrayHandle` object to automatically resolve the template parameters. `IsTypeAndStorage` takes an example component type and an example storage type as arguments and returns whether the underlying array matches both types.

Example 11.4: Querying the component and storage types of a `DynamicArrayHandle`.

```
std::vector<vtkm::Float32> scalarBuffer(10);
vtkm::cont::ArrayHandle<vtkm::Float32> concreteHandle =
  vtkm::cont::make_ArrayHandle(scalarBuffer);
vtkm::cont::DynamicArrayHandle dynamicHandle(concreteHandle);

// This returns true
bool isFloat32Array = dynamicHandle.IsSameType(concreteHandle);

// This returns false
bool isIdArray = dynamicHandle.IsType<vtkm::cont::ArrayHandle<vtkm::Id>>();

// This returns true
```

```
13  bool isFloat32 =
14    dynamicHandle.IsTypeAndStorage<vtkm::Float32, VTKM_DEFAULT_STORAGE_TAG>();
15
16  // This returns false
17  bool isId = dynamicHandle.IsTypeAndStorage<vtkm::Id, VTKM_DEFAULT_STORAGE_TAG>();
18
19  // This returns false
20  bool isErrorStorage = dynamicHandle.IsTypeAndStorage<
21    vtkm::Float32,
22    vtkm::cont::ArrayHandleCounting<vtkm::Float32>::StorageTag>();
```

Once the type of the DynamicArrayHandle is known, it can be cast to a concrete ArrayHandle, which has access to the data as described in Chapter 7. The easiest way to do this is to use the CopyTo method. This templated method takes a reference to an ArrayHandle as an argument and sets that array handle to point to the array in DynamicArrayHandle. If the given types are incorrect, then CopyTo throws a vtkm::cont::ErrorControlBadValue exception.

Example 11.5: Casting a DynamicArrayHandle to a concrete ArrayHandle.

```
1  dynamicHandle.CopyTo(concreteHandle);
```

Common Errors

Remember that ArrayHandle and DynamicArrayHandle represent pointers to the data, so this "copy" is a shallow copy. There is still only one copy of the data, and if you change the data in one array handle that change is reflected in the other.

11.2 Casting to Unknown Types

Using CopyTo is fine as long as the correct types are known, but often times they are not. For this use case DynamicArrayHandle has a method named CastAndCall that attempts to cast the array to some set of types.

The CastAndCall 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.

Example 11.6: Operating on DynamicArrayHandle with CastAndCall.

```
1  struct PrintArrayContentsFunctor
2  {
3    template<typename T, typename Storage>
4    VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<T, Storage>& array) const
5    {
6      this->PrintArrayPortal(array.GetPortalConstControl());
7    }
8
9  private:
10   template<typename PortalType>
11   VTKM_CONT void PrintArrayPortal(const PortalType& portal) const
12   {
13     for (vtkm::Id index = 0; index < portal.GetNumberOfValues(); index++)
14     {
15       // All ArrayPortal objects have ValueType for the type of each value.
16       using ValueType = typename PortalType::ValueType;
17
```

```
18        ValueType value = portal.Get(index);
19
20        vtkm::IdComponent numComponents =
21          vtkm::VecTraits<ValueType>::GetNumberOfComponents(value);
22        for (vtkm::IdComponent componentIndex = 0; componentIndex < numComponents;
23             componentIndex++)
24        {
25          std::cout << " "
26                    << vtkm::VecTraits<ValueType>::GetComponent(value, componentIndex);
27        }
28        std::cout << std::endl;
29      }
30    }
31  };
32
33  template<typename DynamicArrayType>
34  void PrintArrayContents(const DynamicArrayType& array)
35  {
36    array.CastAndCall(PrintArrayContentsFunctor());
37  }
```

Common Errors

It is possible to store any form of `ArrayHandle` *in a* `DynamicArrayHandle`*, but it is not possible for* `CastAndCall` *to check every possible form of* `ArrayHandle`*. If* `CastAndCall` *cannot determine the* `ArrayHandle` *type, then an* `ErrorControlBadValue` *is thrown. The following section describes how to specify the forms of* `ArrayHandle` *to try.*

11.3 Specifying Cast Lists

The `CastAndCall` method can only check a finite number of types. The default form of `CastAndCall` uses a default set of common types. These default lists can be overridden using the VTK-m list tags facility, which is discussed at length in Section 6.6. There are separate lists for value types and for storage types.

Common type lists for value are defined in vtkm/TypeListTag.h and are documented in Section 6.6.2. This header also defines `VTKM_DEFAULT_TYPE_LIST_TAG`, which defines the default list of value types tried in `CastAndCall`.

Common storage lists are defined in vtkm/cont/StorageListTag.h. There is only one common storage distributed with VTK-m: `StorageBasic`. A list tag containing this type is defined as `vtkm::cont::StorageListTagBasic`.

As with other lists, it is possible to create new storage type lists using the existing type lists and the list bases from Section 6.6.1.

The vtkm/cont/StorageListTag.h header also defines a macro named `VTKM_DEFAULT_STORAGE_LIST_TAG` that defines a default list of types to use in classes like `DynamicArrayHandle`. This list can be overridden by defining the `VTKM_DEFAULT_STORAGE_LIST_TAG` 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.

There is a form of `CastAndCall` that accepts tags for the list of component types and storage types. This can be used when the specific lists are known at the time of the call. However, when creating generic operations like the `PrintArrayContents` function in Example 11.6, passing these tags is inconvenient at best.

To address this use case, DynamicArrayHandle has a pair of methods named ResetTypeList and ResetStorageList. These methods return a new object with that behaves just like a DynamicArrayHandle with identical state except that the cast and call functionality uses the specified component type or storage type instead of the default. (Note that PrintArrayContents in Example 11.6 is templated on the type of DynamicArrayHandle. This is to accommodate using the objects from the Reset*List methods, which have the same behavior but different type names.)

So the default component type list contains a subset of the basic VTK-m types. If you wanted to accommodate more types, you could use ResetTypeList.

Example 11.7: Trying all component types in a DynamicArrayHandle.

```
1  PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagAll()));
```

Likewise, if you happen to know a particular type of the dynamic array, that can be specified to reduce the amount of object code created by templates in the compiler.

Example 11.8: Specifying a single component type in a DynamicArrayHandle.

```
1  PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagId()));
```

Storage type lists can be changed similarly.

Example 11.9: Specifying different storage types in a DynamicArrayHandle.

```
1  struct MyIdStorageList : vtkm::ListTagBase<vtkm::cont::StorageTagBasic,
2                                             vtkm::cont::ArrayHandleIndex::StorageTag>
3  {
4  };
5
6  void PrintIds(vtkm::cont::DynamicArrayHandle array)
7  {
8    PrintArrayContents(array.ResetStorageList(MyIdStorageList()));
9  }
```

Common Errors

The ResetTypeList *and* ResetStorageList *do not change the object they are called on. Rather, they return a new object with different type information. Calling these methods has no effect unless you do something with the returned value.*

Both the component type list and the storage type list can be modified by chaining these reset calls.

Example 11.10: Specifying both component and storage types in a DynamicArrayHandle.

```
1  PrintArrayContents(dynamicArray.ResetTypeList(vtkm::TypeListTagId())
2                                 .ResetStorageList(MyIdStorageList()));
```

The ResetTypeList and ResetStorageList work by returning a vtkm::cont::DynamicArrayHandleBase object. DynamicArrayHandleBase specifies the value and storage tag lists as template arguments and otherwise behaves just like DynamicArrayHandle.

Did you know?

I lied earlier when I said at the beginning of this chapter that `DynamicArrayHandle` *is a class that is not templated. This symbol is really just a type alias of* `DynamicArrayHandleBase`*. Because the* `DynamicArrayHandle` *fully specifies the template arguments, it behaves like a class, but if you get a compiler error it will show up as* `DynamicArrayHandleBase`*.*

Most code does not need to worry about working directly with `DynamicArrayHandleBase`. However, it is sometimes useful to declare it in templated functions that accept dynamic array handles so that works with every type list. The function in Example 11.6 did this by making the dynamic array handle class itself the template argument. This will work, but it is prone to error because the template will resolve to any type of argument. When passing objects that are not dynamic array handles will result in strange and hard to diagnose errors. Instead, we can define the same function using `DyamicArrayHandleBase` so that the template will only match dynamic array handle types.

Example 11.11: Using `DynamicArrayHandleBase` to accept generic dynamic array handles.

```
1 template<typename TypeList, typename StorageList>
2 void PrintArrayContents(
3   const vtkm::cont::DynamicArrayHandleBase<TypeList, StorageList>& array)
4 {
5   array.CastAndCall(PrintArrayContentsFunctor());
6 }
```

CHAPTER

TWELVE

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 must have at least one cell set, but can have more than one cell set defined. This makes it possible to define groups of cells with different properties. For example, a simulation might model some subset of elements as boundary that contain properties the other elements do not. Another example is the representation of a molecule that requires atoms and bonds, each having very different properties associated with them.

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::MultiBlock` to represent multi-block data sets. A `MultiBlock` is implemented as a collection of `DataSet` objects. Multi-block data sets are described later in Section 12.5.

12.1 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 3.

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.

12.1.1 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 12.1: Creating a uniform grid.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

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 12.2: Creating a uniform grid with custom origin and spacing.

```
vtkm::cont::DataSetBuilderUniform dataSetBuilder;

vtkm::cont::DataSet dataSet =
  dataSetBuilder.Create(vtkm::Id3(101, 101, 26),
                        vtkm::Vec<vtkm::FloatDefault, 3>(-50.0, -50.0, -50.0),
                        vtkm::Vec<vtkm::FloatDefault, 3>(1.0, 1.0, 4.0));
```

12.1.2 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, 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 12.3: 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));
}

```

```
8   // Make y coordinates range from 0 to 2 with tighter spacing near 2.
9   std::vector<vtkm::Float32> yCoordinates;
10  for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
11  {
12    yCoordinates.push_back(vtkm::Sqrt(y));
13  }
14
15  // Make z coordinates rangefrom -1 to 1 with even spacing.
16  std::vector<vtkm::Float32> zCoordinates;
17  for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
18  {
19    zCoordinates.push_back(z);
20  }
21
22  vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;
23
24  vtkm::cont::DataSet dataSet =
25    dataSetBuilder.Create(xCoordinates, yCoordinates, zCoordinates);
```

12.1.3 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.

The cells of an explicit mesh are defined by providing the shape, number of indices, and the points that comprise it for each cell. These three things are stored in separate arrays. Figure 12.1 shows an example of an explicit mesh and the arrays that can be used to define it.

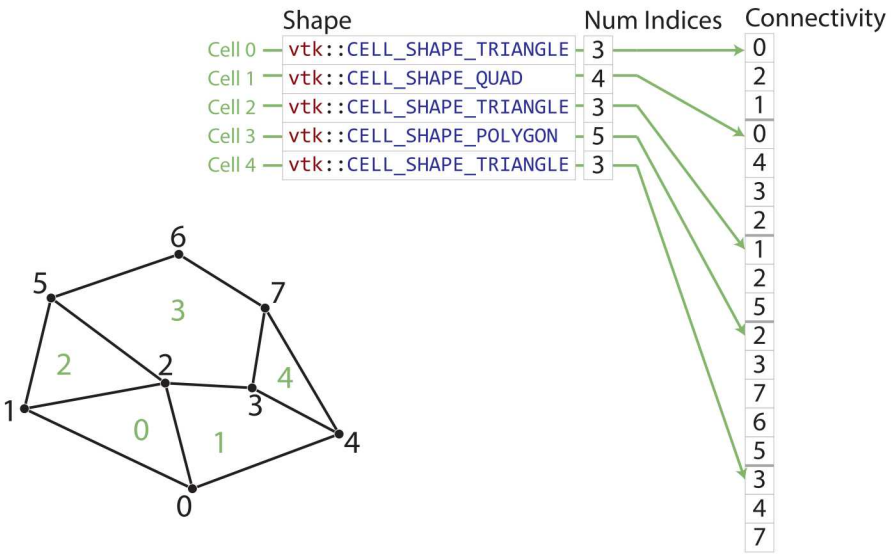


Figure 12.1: An example explicit mesh.

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 12.1.

Example 12.4: Creating an explicit mesh with DataSetBuilderExplicit.

```
// Array of point coordinates.
std::vector<vtkm::Vec<vtkm::Float32, 3>> pointCoordinates;
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(1.1f, 0.0f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(0.2f, 0.4f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(0.9f, 0.6f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(1.4f, 0.5f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(1.8f, 0.3f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(0.4f, 1.0f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(1.0f, 1.2f, 0.0f));
pointCoordinates.push_back(vtkm::Vec<vtkm::Float32, 3>(1.5f, 0.9f, 0.0f));

// Array of shapes.
std::vector<vtkm::UInt8> shapes;
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
shapes.push_back(vtkm::CELL_SHAPE_QUAD);
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);

// Array of number of indices per cell.
std::vector<vtkm::IdComponent> numIndices;
numIndices.push_back(3);
numIndices.push_back(4);
numIndices.push_back(3);
numIndices.push_back(5);
numIndices.push_back(3);

// Connectivity array.
std::vector<vtkm::Id> connectivity;
connectivity.push_back(0); // Cell 0
connectivity.push_back(2);
connectivity.push_back(1);
connectivity.push_back(0); // Cell 1
connectivity.push_back(4);
connectivity.push_back(3);
connectivity.push_back(2);
connectivity.push_back(1); // Cell 2
connectivity.push_back(2);
connectivity.push_back(5);
connectivity.push_back(2); // Cell 3
connectivity.push_back(3);
connectivity.push_back(7);
connectivity.push_back(6);
connectivity.push_back(5);
connectivity.push_back(3); // Cell 4
connectivity.push_back(4);
connectivity.push_back(7);

// Copy these arrays into a DataSet.
vtkm::cont::DataSetBuilderExplicit dataSetBuilder;

vtkm::cont::DataSet dataSet =
  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 12.1 except this time using DataSetBuilderExplicitIterative.

Example 12.5: Creating an explicit mesh with DataSetBuilderExplicitIterative.

```
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();
```

12.1.4 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 `vtkm::cont::DataSetFieldAdd` class. This class works on `DataSet`s of any type. It has methods named `AddPointField` and `AddCellField` that define a field for either points or cells. Every field must have an associated field name.

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::vector`s, in which case the data are copied. Field arrays can also be passed in a `ArrayHandle`, 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 12.6: Adding fields to a `DataSet`.

```
1   // 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);

  // This is the helper object to add fields to a data set.
  vtkm::cont::DataSetFieldAdd dataSetFieldAdd;

  // 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);
        }
      }
    }
  }

  dataSetFieldAdd.AddPointField(dataSet, "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);
        }
        else
        {
          boundaryCells.push_back(0);
        }
      }
    }
  }

  dataSetFieldAdd.AddCellField(dataSet, "boundary_cells", boundaryCells);
```

12.2 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 12.2 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 15.1 starting on page 201.

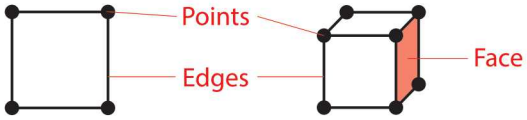


Figure 12.2: The relationship between a cell shape and its topological elements (points, edges, and faces).

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.

12.2.1 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 12.3 demonstrates this arrangement.

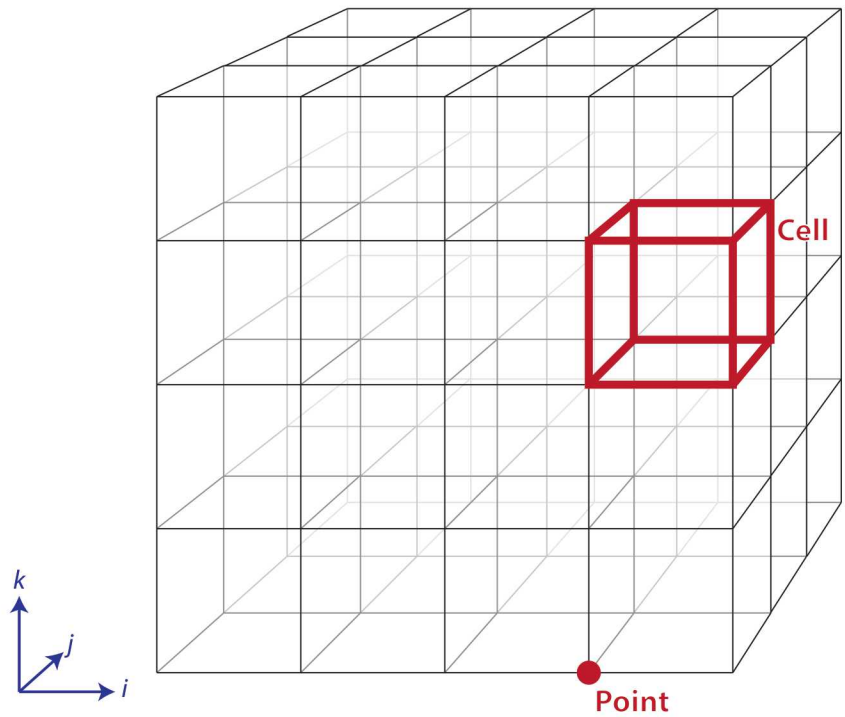


Figure 12.3: The arrangement of points and cells in a 3D structured grid.

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 12.4 starting on page 149.

12.2.2 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. 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. (Cell shapes are discussed in detail in Section 15.1 starting on page 201.) The second array identifies how many points are in each cell. The third array has a sequence of point indices that make up each cell. Figure 12.4 shows a simple example of an explicit cell set.

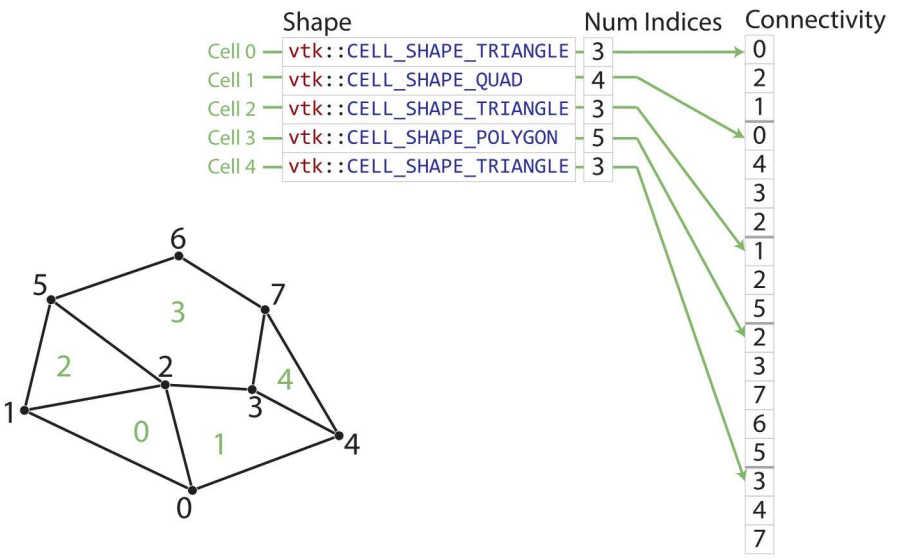


Figure 12.4: Example of cells in a `CellSetExplict` and the arrays that define them.

An explicit cell set may also have other topological arrays such as an array of offsets of each cell into the connectivity array or an array of cells incident on each point. Although these arrays can be provided, they are optional and can be internally derived from the shape, num indices, and connectivity arrays.

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

12.2.3 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.

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 12.7: Subsampling a data set with CellSetPermutation.

```
1  // Create a simple data set.
2  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
3  vtkm::cont::DataSet originalDataSet = dataSetBuilder.Create(vtkm::Id3(33, 33, 26));
4  vtkm::cont::CellSetStructured<3> originalCellSet;
5  originalDataSet.GetCellSet().CopyTo(originalCellSet);
6
7  // Create a permutation array for the cells. Each value in the array refers
8  // to a cell in the original cell set. This particular array selects every
9  // 10th cell.
10 vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);
11
12 // Create a permutation of that cell set containing only every 10th cell.
13 vtkm::cont::CellSetPermutation<vtkm::cont::CellSetStructured<3>,
14                                vtkm::cont::ArrayHandleCounting<vtkm::Id>>
15   permutedCellSet(permutationArray, originalCellSet);
```

12.2.4 Dynamic Cell Sets

vtkm::cont::DataSet must hold an arbitrary collection of vtkm::cont::CellSet objects, which it cannot do while knowing their types at compile time. To manage storing CellSets without knowing their types, DataSet actually holds references using vtkm::cont::DynamicCellSet.

DynamicCellSet is similar in nature to DynamicArrayHandle except that it, of course, holds CellSets instead of ArrayHandles. The interface for the two classes is similar, and you should review the documentation for DynamicArrayHandle (in Chapter 11 starting on page 133) to understand DynamicCellSet.

vtkm::cont::DynamicCellSet has a method named CastToBase that returns a const reference to the held cell set as the abstract CellSet class. This can be used to easily access the virtual methods in the CellSet interface. You can also create a new instance of a cell set with the same type using the NewInstance method.

The DynamicCellSet::IsType method can be used to determine whether the cell set held in the dynamic cell set is of a given type. If the cell set type is known, DynamicCellSet::Cast can be used to safely downcast the cell set object, or DynamicCellSet::CopyTo can be used to safely copy to a reference of the appropriate type.

When a typed version of the cell set stored in the DynamicCellSet is needed but the type is not known, which happens regularly in the internal workings of VTK-m, the CastAndCall method can be used to make this transition. CastAndCall works by taking a functor and calls it with the appropriately cast cell set object.

The CastAndCall method works by attempting to cast to a known set of types. This set of types used is defined by the macro VTKM_DEFAULT_CELL_SET_LIST_TAG, which is declared in vtkm/cont/CellSetListTag.h. This list can be overridden globally by defining the VTKM_DEFAULT_CELL_SET_LIST_TAG macro *before* any VTK-m headers are included.

The set of types used in a CastAndCall can also be changed only for a particular instance of a dynamic cell set by calling its ResetCellSetList. This method takes a list of cell types and returns a new dynamic array handle of a slightly different type that will use this new list of cells for dynamic casting.

12.2.5 Blocks and Assemblies

Rather than just one cell set, a vtkm::cont::DataSet can hold multiple cell sets. This can be used to construct multiblock data structures or assemblies of parts. Multiple cell sets can also be used to represent subsets of the data with particular properties such as all cells filled with a material of a certain type. Or these multiple cells might represent particular features in the data, such as the set of faces representing a boundary in the simulation.

12.2.6 Zero Cell Sets

It is also possible to construct a vtkm::cont::DataSet that contains no cell set objects whatsoever. This can be used to manage data that does not contain any topological structure. For example, a collection of series that come from columns in a table could be stored as multiple fields in a data set with no cell set.

12.3 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. Field holds its data with a DynamicArrayHandle, which itself is a container for an ArrayHandle. Field also maintains the association and, optionally, the name of a cell set for which the field is valid.

The data array can be retrieved as a DynamicArrayHandle using the GetData method of Field. Field also 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.

12.4 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.

12.5 Multi-Block Data

A multi-block data set, implemented with `vtkm::cont::MultiBlock`, comprises a set of `vtkm::cont::DataSet` objects. The `MultiBlock` interface allows for adding, inserting, and replacing `DataSet`s in its list with the `AddBlock` (or `AddBlocks`), `InsertBlock`, and `ReplaceBlock` methods, respectively. The `GetBlocks` method returns a list of `DataSet` objects in a `std::vector`.

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

Example 12.8: Creating a `MultiBlock`.

```
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::MultiBlock multiBlock;
9  multiBlock.AddBlock(dataSet1);
10 multiBlock.AddBlock(dataSet2);
```

It is always possible to retrieve the independent blocks in a `MultiBlock`, 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. However, `MultiBlock` also offers several helper methods to collect metadata information about the collection as a whole. Each function is in its own respective header file.

`vtkm::cont::BoundsCompute` Queries the bounds of all the `DataSet`s contained in the given `MultiBlock` 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 `MultiBlock`, 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 12.3).

`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::MultiBlock`.

Example 12.9: Queries on a `MultiBlock`.

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

Did you know?

The aforementioned functions for querying a `MultiBlock` 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 `MultiBlock` 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 12.10: Applying a filter to multi block data.

```
1  vtkm::filter::CellAverage cellAverage;
2  cellAverage.SetActiveField("pointvar", vtkm::cont::Field::Association::POINTS);
3
4  vtkm::cont::MultiBlock results = cellAverage.Execute(multiBlock);
```

Part III

Developing with VTK-m

CHAPTER

THIRTEEN

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. This chapter explains the basic mechanics of defining and using worklets.

13.1 Worklet Types

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. Details on how to create worklets of each type are given in Section 13.5. It is also possible to create new worklet types in VTK-m. This is an advanced topic covered in Chapter 23.

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 sibling 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::WorkletMapPointToCell` 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::WorkletMapCellToPoint` calls the worklet operation for each point and makes every incident cell available.

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.

13.2 Dispatchers

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 invocation is done through a set of *dispatcher* objects. 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 two template parameters: the worklet class being invoked, which is always the first argument, and the device adapter tag, which is always the last argument and will be set to the default device adapter if not specified.

All dispatcher objects must be constructed with an instance of the worklet they are to invoke. If no worklet is provided to the constructor, a new worklet object is created with the default constructor. Many worklets do not require any state, so allowing the dispatcher to construct its own is fine. However, if the worklet holds some parameters (e.g. a threshold value), then you will have to construct a worklet and pass it to a dispatcher as it is created.

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 expected arguments for worklets provided by VTK-m are documented in Section 13.3. Also, for any worklet, the `Invoke` arguments can be gleaned from the control signature, which is described in Section 13.4.1.

The following is a list of the dispatchers defined in VTK-m. The dispatcher classes correspond to the list of worklet types specified in Section 13.1. Many examples of using these dispatchers are provided in Section 13.3.

`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::WorkletMapPointToCell`). 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.

13.3 Provided Worklets

VTK-m comes with several worklet implementations. These worklet implementations for the most part provide the underlying implementations of the filters described in Chapter 4. The easiest way to execute a filter is to run it from the associated filter class. However, if your data is not in a `vtkm::cont::DataSet` structure or you have knowledge of the specific data types used in the `DataSet`, it might be more efficient to run the worklet directly. Note that many of the filters use multiple worklets under the covers to implement the full functionality.

The following example demonstrates using the simple `vtkm::worklet::PointElevation` worklet directly.

Example 13.1: Using the provided `PointElevation` worklet.

```
VTKM_CONT
vtkm::cont::ArrayHandle<vtkm::FloatDefault> ComputeAirPressure(
  vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 3>> pointCoordinates)
{
  vtkm::worklet::PointElevation elevationWorklet;

  // Use the elevation worklet to estimate atmospheric pressure based on the
  // height of the point coordinates. Atmospheric pressure is 101325 Pa at
  // sea level and drops about 12 Pa per meter.
  elevationWorklet.SetLowPoint(vtkm::Vec<vtkm::Float64, 3>(0.0, 0.0, 0.0));
  elevationWorklet.SetHighPoint(vtkm::Vec<vtkm::Float64, 3>(0.0, 0.0, 2000.0));
  elevationWorklet.SetRange(101325.0, 77325.0);

  vtkm::worklet::DispatcherMapField<vtkm::worklet::PointElevation>
    elevationDispatcher(elevationWorklet);

  vtkm::cont::ArrayHandle<vtkm::FloatDefault> pressure;

  elevationDispatcher.Invoke(pointCoordinates, pressure);

  return pressure;
}
```

13.4 Creating Worklets

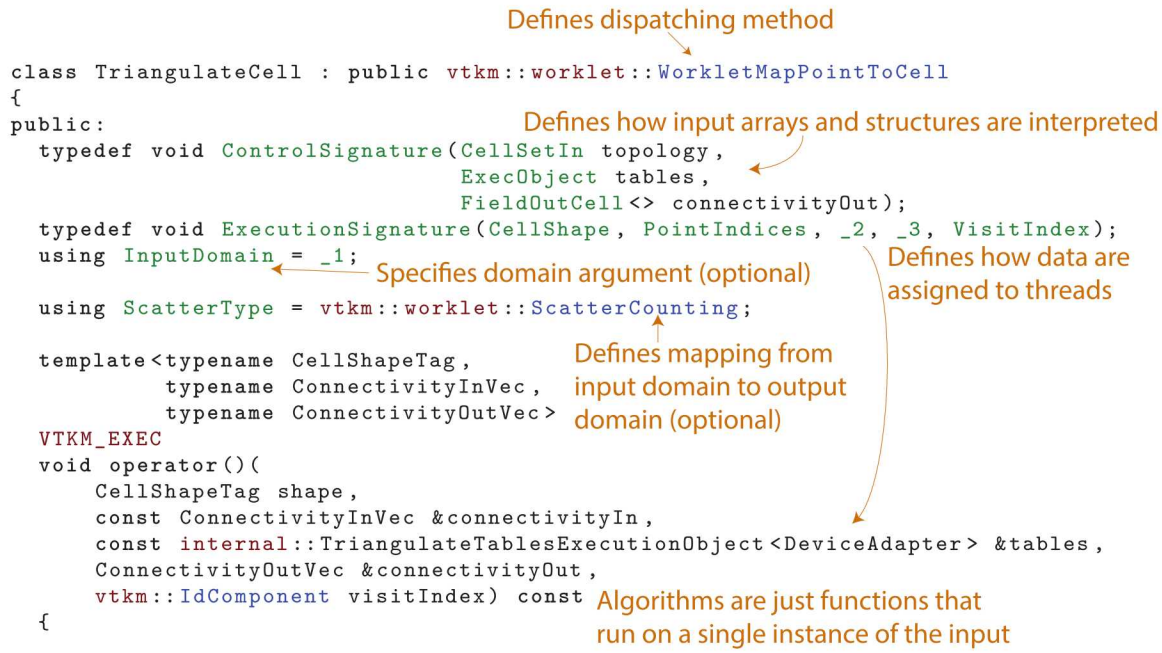
A worklet is created by implementing a `class` or `struct` with the following features.

1. The class must contain a functional type named `ControlSignature`, which specifies what arguments are expected when invoking the class with a dispatcher in the control environment.
2. The class must contain a functional type named `ExecutionSignature`, which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.
3. The class must contain a type named `InputDomain`, which identifies which input parameter defines the input domain of the data.
4. The class may define a scatter operation to override a 1:1 mapping from input to output.
5. The class must contain an implementation of the parenthesis operator, which is the method that is executed in the execution environment. The parenthesis operator must be declared `const`.
6. The class must publicly inherit from a base worklet class that specifies the type of operation being performed.

Figure 13.1 demonstrates all of the required components of a worklet.

13.4.1 Control Signature

The control signature of a worklet is a functional type named `ControlSignature`. The function prototype matches the calling specification used with the dispatcher `Invoke` function.




```
class TriangulateCell : public vtkm::worklet::WorkletMapPointToCell
{
public:
  typedef void ControlSignature(CellSetIn topology,
                                ExecObject tables,
                                FieldOutCell<> connectivityOut);
  typedef void ExecutionSignature(CellShape, PointIndices, _2, _3, VisitIndex);
  using InputDomain = _1;

  using ScatterType = vtkm::worklet::ScatterCounting;

  template<typename CellShapeTag,
           typename ConnectivityInVec,
           typename ConnectivityOutVec>
  VTKM_EXEC
  void operator()(
      CellShapeTag shape,
      const ConnectivityInVec &connectivityIn,
      const internal::TriangulateTablesExecutionObject<DeviceAdapter> &tables,
      ConnectivityOutVec &connectivityOut,
      vtkm::IdComponent visitIndex) const
  {
```

Figure 13.1: Annotated example of a worklet declaration.

Example 13.2: A `ControlSignature`.

```
1     using ControlSignature = void(FieldIn<VecAll> inputVectors,
2                                   FieldOut<Scalar> outputMagnitudes);
```

The return type of the function prototype is always `void` because the dispatcher `Invoke` functions do not return values. 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 Section 13.5.

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.

Type List Tags

Some tags are templated to have modifiers. For example, `Field` tags have a template argument that is set to a type list tag defining what types of field data are supported. (See Section 6.6.2 for a description of type lists.) In fact, this type list modifier is so common that the following convenience subtags used with `Field` tags are defined for all worklet types.

Did you know?

Any type list will work as modifiers for `ControlSignature` tags. However, these common type lists are provided for convenience and to make the `ControlSignature` shorter and more readable.

`AllTypes` All possible types.

`CommonTypes` 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. The same as `vtkm::TypeListTagCommon`.

`IdType` Contains the single item `vtkm::Id`. The same as `vtkm::TypeListTagId`.

`Id2Type` Contains the single item `vtkm::Id2`. The same as `vtkm::TypeListTagId2`.

`Id3Type` Contains the single item `vtkm::Id3`. The same as `vtkm::TypeListTagId3`.

`Index` All types used to index arrays. Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`. The same as `vtkm::TypeListTagIndex`.

`FieldCommon` A list containing all the types generally used for fields. It is the combination of `Scalar`, `Vec2`, `Vec3`, and `Vec4`. The same as `vtkm::TypeListTagField`.

`Scalar` Types used for scalar fields. Specifically, it contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`). The same as `vtkm::TypeListTagFieldScalar`.

`ScalarAll` 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. The same as `vtkm::TypeListTagScalarAll`.

`Vec2` Types for values of fields with 2 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec2`.

`Vec3` Types for values of fields with 3 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec3`.

`Vec4` Types for values of fields with 4 dimensional vectors. All these vectors use floating point numbers. The same as `vtkm::TypeListTagFieldVec4`.

`VecAll` All `vtkm::Vec` classes with standard integers or floating points as components and lengths between 2 and 4. The same as `vtkm::TypeListTagVecAll`.

`VecCommon` 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. The same as `vtkm::TypeListTagVecCommon`.

13.4.2 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 13.4.4) in terms of arity and argument semantics.

Example 13.3: An `ExecutionSignature`.

```
1 |     using ExecutionSignature = _2(_1);
```

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.

13.4.3 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. One of the arguments given to the dispatcher's `Invoke` in the control environment must specify the domain.

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 13.4: 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.

13.4.4 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 13.5: An overloaded parenthesis operator of a worklet.

```
1 |     template<typename T, vtkm::IdComponent Size>
2 |     VTKM_EXEC T operator()(const vtkm::Vec<T, Size>& inVector) 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.

13.5 Worklet Type Reference

There are multiple worklet types provided by VTK-m, each designed to support a particular type of operation. Section 13.1 gave a brief overview of each type of worklet. This section gives a much more detailed reference for each of the worklet types including identifying the generic superclass that a worklet instance should derive, listing the signature tags and their meanings, and giving an example of the worklet in use.

13.5.1 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 `WorkletMapField` subclass is invoked with a `vtkm::worklet::DispatcherMapField`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array.

`FieldIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldInOut` This tag represents field that is both an input and an output. A `FieldInOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.7 starting on page 181.

`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 13.9 starting on page 186.

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 13.10).

`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 13.10).

`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 23.2, 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 13.6: Implementation and use of a field map worklet.

```
1  #include <vtkm/worklet/DispatcherMapField.h>
2  #include <vtkm/worklet/WorkletMapField.h>
3
4  #include <vtkm/cont/ArrayHandle.h>
5  #include <vtkm/cont/DynamicArrayHandle.h>
6
7  #include <vtkm/VecTraits.h>
8  #include <vtkm/VectorAnalysis.h>
9
10 namespace vtkm
```

```
{
namespace worklet
{

struct Magnitude
{
  class ComputeMagnitude : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn<VecAll> inputVectors,
                                  FieldOut<Scalar> outputMagnitudes);
    using ExecutionSignature = _2(_1);

    using InputDomain = _1;

    template<typename T, vtkm::IdComponent Size>
    VTKM_EXEC T operator()(const vtkm::Vec<T, Size>& inVector) const
    {
      return vtkm::Magnitude(inVector);
    }
  };

  template<typename ValueType, typename Storage>
  VTKM_CONT static vtkm::cont::ArrayHandle<
    typename vtkm::VecTraits<ValueType>::ComponentType>
  Run(const vtkm::cont::ArrayHandle<ValueType, Storage>& input)
  {
    using ComponentType = typename vtkm::VecTraits<ValueType>::ComponentType;
    vtkm::cont::ArrayHandle<ComponentType> output;

    vtkm::worklet::DispatcherMapField<ComputeMagnitude> dispatcher;
    dispatcher.Invoke(input, output);

    return output;
  }
};

} // namespace worklet
} // namespace vtkm
```

Did you know?

Example 13.6 is using an informal but helpful convention where worklets are defined inside another class that also contains a static method named `Run` *that runs the worklet using a common set of operations. The* `Run` *method helps make clear the intention and use of the worklet. This is especially important for operations that require a series of worklet invocations that might be non-obvious.*

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 Sections 13.6 and 13.9, 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 13.7: Leveraging field maps and field maps for general processing.

```
namespace vtkm
{
```

```
3  namespace worklet
4  {
5
6  struct ReverseArrayCopy
7  {
8    struct PermuteArrayValues : vtkm::worklet::WorkletMapField
9    {
10     using ControlSignature = void(FieldIn<> inputArray, WholeArrayOut<> outputArray);
11     using ExecutionSignature = void(_1, _2, WorkIndex);
12     using InputDomain = _1;
13
14     template<typename InputType, typename OutputArrayPortalType>
15     VTKM_EXEC void operator()(const InputType& inputValue,
16                               const OutputArrayPortalType& outputArrayPortal,
17                               vtkm::Id workIndex) const
18     {
19       vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
20       if (outIndex >= 0)
21       {
22         outputArrayPortal.Set(outIndex, inputValue);
23       }
24       else
25       {
26         this->RaiseError("Output array not sized correctly.");
27       }
28     }
29   };
30
31   template<typename T, typename Storage>
32   VTKM_CONT static vtkm::cont::ArrayHandle<T> Run(
33     const vtkm::cont::ArrayHandle<T, Storage>& inArray)
34   {
35     vtkm::cont::ArrayHandle<T> outArray;
36     outArray.Allocate(inArray.GetNumberOfValues());
37
38     vtkm::worklet::DispatcherMapField<PermuteArrayValues> dispatcher;
39     dispatcher.Invoke(inArray, outArray);
40
41     return outArray;
42   }
43 };
44
45 } // namespace worklet
46 } // namespace vtkm
```

13.5.2 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. Regardless of their names, they are all defined in vtkm/worklet/WorkletMapTopology.h and are all invoked with `vtkm::worklet::DispatcherMapTopology`.

Point to Cell Map

A worklet deriving `vtkm::worklet::WorkletMapPointToCell` 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 `WorkletMapPointToCell` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A point to cell map 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 a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 15.)

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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldInCell` This tag represents an input field that is associated with the cells. A `FieldInCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldOutCell` This tag represents an output field, which is necessarily associated with cells. A `FieldOutCell` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOutCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`FieldInOutCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.7 starting on page 181.

`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 13.9 starting on page 186.

A field 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 cell. (Cell shapes and the operations you can do with cells are discussed in Chapter 15.) 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 13.10).

`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 13.10).

`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 23.2, 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 15 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 13.8: Implementation and use of a map point to cell worklet.

```
#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/cont/DataSet.h>
#include <vtkm/cont/DataSetFieldAdd.h>

#include <vtkm/exec/CellInterpolate.h>
#include <vtkm/exec/ParametricCoordinates.h>

namespace vtkm
{
namespace worklet
{

struct CellCenter
{
  class InterpolateCenter : public vtkm::worklet::WorkletMapPointToCell
  {
  public:
    using ControlSignature = void(CellSetIn cellSet,
                                  FieldInPoint<> inputPointField,
                                  FieldOut<> outputCellField);
    using ExecutionSignature = _3(_1, PointCount, _2);

    using InputDomain = _1;

    template<typename CellShape, typename InputPointFieldType>
    VTKM_EXEC typename InputPointFieldType::ComponentType operator()(
      CellShape shape,
      vtkm::IdComponent numPoints,
      const InputPointFieldType& inputPointField) const
    {
      vtkm::Vec<vtkm::FloatDefault, 3> parametricCenter =
        vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, *this);
      return vtkm::exec::CellInterpolate(
        inputPointField, parametricCenter, shape, *this);
    }
  };

  template<typename CellSetType, typename ValueType, typename StorageType>
  VTKM_CONT static vtkm::cont::ArrayHandle<ValueType> Run(
```

```
42      const CellSetType& cellSet,
43      const vtkm::cont::ArrayHandle<ValueType, StorageType>& inPointField)
44    {
45      vtkm::cont::ArrayHandle<ValueType> outCellField;
46
47      vtkm::worklet::DispatcherMapTopology<
48        vtkm::worklet::CellCenter::InterpolateCenter>
49        dispatcher;
50      dispatcher.Invoke(cellSet, inPointField, outCellField);
51
52      return outCellField;
53    }
54  };
55
56  } // namespace worklet
57  } // namespace vtkm
```

Cell To Point Map

A worklet deriving `vtkm::worklet::WorkletMapCellToPoint` 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 `WorkletMapCellToPoint` subclass is invoked with a `vtkm::worklet::DispatcherMapTopology`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

A cell to point map 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 a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`.

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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInCell` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldInPoint` This tag represents an input field that is associated with the points. A `FieldInPoint` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldOutPoint` This tag represents an output field, which is necessarily associated with points. A `FieldOutPoint` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the

dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOutPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

`FieldInOutPoint` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.7 starting on page 181.

`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 13.9 starting on page 186.

A field map 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 13.10).

`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 13.10).

`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 23.2, 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 13.9: Implementation and use of a map cell to point worklet.

```
#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/cont/DataSet.h>
#include <vtkm/cont/DataSetFieldAdd.h>
#include <vtkm/cont/DynamicArrayHandle.h>
#include <vtkm/cont/DynamicCellSet.h>
#include <vtkm/cont/Field.h>

namespace vtkm
{
namespace worklet
{

struct ConvertCellFieldsToPointFields
{
  class AverageCellField : public vtkm::worklet::WorkletMapCellToPoint
  {
  public:
    using ControlSignature = void(CellSetIn cellSet,
                                  FieldInCell<> inputCellField,
                                  FieldOut<> outputPointField);
    using ExecutionSignature = void(CellCount, _2, _3);

    using InputDomain = _1;

    template<typename InputCellFieldType, typename OutputFieldType>
```

```
28      VTKM_EXEC void operator()(vtkm::IdComponent numCells,
29                                const InputCellFieldType& inputCellField,
30                                OutputFieldType& fieldAverage) const
31      {
32        // TODO: This trickery with calling DoAverage with an extra fabricated type
33        // is to handle when the dynamic type resolution provides combinations that
34        // are incompatible. On the todo list for VTK-m is to allow you to express
35        // types that are the same for different parameters of the control
36        // signature. When that happens, we can get rid of this hack.
37        using InputComponentType = typename InputCellFieldType::ComponentType;
38        this->DoAverage(numCells,
39                        inputCellField,
40                        fieldAverage,
41                        vtkm::ListTagBase<InputComponentType, OutputFieldType>());
42      }
43
44    private:
45      template<typename InputCellFieldType, typename OutputFieldType>
46      VTKM_EXEC void DoAverage(
47        vtkm::IdComponent numCells,
48        const InputCellFieldType& inputCellField,
49        OutputFieldType& fieldAverage,
50        vtkm::ListTagBase<OutputFieldType, OutputFieldType>) const
51      {
52        fieldAverage = OutputFieldType(0);
53
54        for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
55        {
56          fieldAverage = fieldAverage + inputCellField[cellIndex];
57        }
58
59        fieldAverage = fieldAverage / OutputFieldType(numCells);
60      }
61
62      template<typename T1, typename T2, typename T3>
63      VTKM_EXEC void DoAverage(vtkm::IdComponent, T1, T2, T3) const
64      {
65        this->RaiseError("Incompatible types for input and output.");
66      }
67    };
68
69    VTKM_CONT
70    static vtkm::cont::DataSet Run(const vtkm::cont::DataSet& inData)
71    {
72      vtkm::cont::DataSet outData;
73
74      // Copy parts of structure that should be passed through.
75      for (vtkm::Id cellSetIndex = 0; cellSetIndex < inData.GetNumberOfCellSets();
76           cellSetIndex++)
77      {
78        outData.AddCellSet(inData.GetCellSet(cellSetIndex));
79      }
80      for (vtkm::Id coordSysIndex = 0;
81           coordSysIndex < inData.GetNumberOfCoordinateSystems();
82           coordSysIndex++)
83      {
84        outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSysIndex));
85      }
86
87      // Copy all fields, converting cell fields to point fields.
88      for (vtkm::Id fieldIndex = 0; fieldIndex < inData.GetNumberOfFields();
89           fieldIndex++)
90      {
91        vtkm::cont::Field inField = inData.GetField(fieldIndex);
```

```
92        if (inField.GetAssociation() == vtkm::cont::Field::Association::CELL_SET)
93        {
94          vtkm::cont::DynamicArrayHandle inFieldData = inField.GetData();
95          vtkm::cont::DynamicCellSet inCellSet =
96            inData.GetCellSet(inField.GetAssocCellSet());
97
98          vtkm::cont::DynamicArrayHandle outFieldData = inFieldData.NewInstance();
99          vtkm::worklet::DispatcherMapTopology<AverageCellField> dispatcher;
100         dispatcher.Invoke(inCellSet, inFieldData, outFieldData);
101
102         vtkm::cont::DataSetFieldAdd::AddCellField(
103           outData, inField.GetName(), outFieldData, inField.GetAssocCellSet());
104       }
105       else
106       {
107         outData.AddField(inField);
108       }
109     }
110
111     return outData;
112   }
113 };
114
115 } // namespace worklet
116 } // namespace vtkm
```

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 "from" topology element, and the second template parameter specifies the "to" topology element. The worklet is scheduled such that each instance is associated with a particular "to" topology element and has access to incident "from" topology elements.

These from and to 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::TopologyElementTagPoint, vtkm::Topolo-
gyElementTagCell >
```

is equivalent to the `vtkm::worklet::WorkletMapPointToCell` class except the signature tags have different names. The names used in the specific topology map superclasses (such as `WorkletMapPointToCell`) 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 a `DynamicCellSet` in the associated parameter of the dispatcher's `Invoke`. Each invocation of the worklet gets a cell shape tag. (Cell shapes and the operations you can do with cells are discussed in Chapter 15.)

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

`FieldInFrom` This tag represents an input field that is associated with the "from" elements. A `FieldInFrom` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The size of the array must be exactly the number of "from" elements.

Each invocation of the worklet gets a `Vec`-like object containing the field values for all the "from" elements incident with the "to" element being visited. If the field is a vector field, then the provided object is a Vec of Vecs.

`FieldInFrom` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldInTo` This tag represents an input field that is associated with the "to" element. A `FieldInTo` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`FieldInTo` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldOut` This tag represents an output field, which is necessarily associated with "to" elements. A `FieldOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`FieldOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`FieldInOut` This tag represents field that is both an input and an output, which is necessarily associated with "to" elements. A `FieldInOut` argument expects an `ArrayHandle` or a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.6 starting on page 178.

`WholeArrayInOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 dispatcher's `Invoke`. 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 13.7 starting on page 181.

`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 13.9 starting on page 186.

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 "to" element. (Cell shapes and the operations you can do with cells are discussed in Chapter 15.) This is the same value that gets provided if you reference the `CellSetIn` parameter.

If the "to" 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.

`FromCount` This tag produces a `vtkm::IdComponent` equal to the number of "from" elements incident on the "to" element being visited. The Vecs provided from a `FieldInFrom` parameter will be the same size as `FromCount`.

`FromIndices` This tag produces a `Vec`-like object of `vtkm::Id` s giving the indices for all incident "from" elements. The order of the entries is consistent with the values of all other `FieldInFrom` 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 13.10).

`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 13.10).

`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 23.2, but most users can get the information they need through other signature tags.

13.5.3 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.

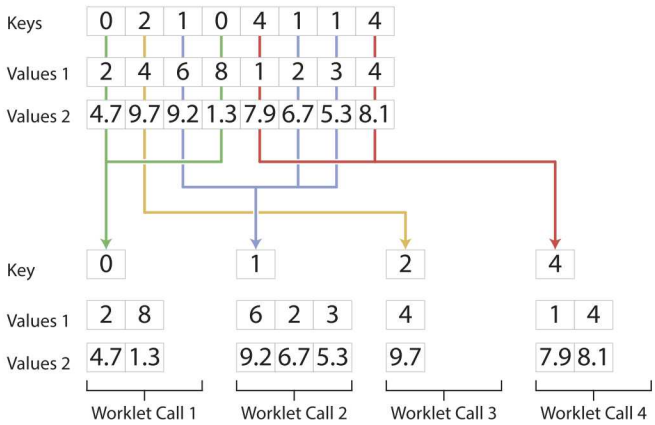


Figure 13.2: The collection of values for a reduce by key worklet.

Figure 13.2 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.)

Within the dispatch of the worklet, 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 `WorkletReduceByKey` subclass is invoked with a `vtkm::worklet::DispatcherReduceByKey`. This dispatcher has two template arguments. The first argument is the type of the worklet subclass. The second argument, which is optional, is a device adapter tag.

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 dispatcher's `Invoke`. 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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. 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.

`ValuesIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

`ReducedValuesOut` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

`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 a `DynamicArrayHandle` in the associated parameter of the dispatcher's `Invoke`. The number of values in the array must equal the number of *unique* keys.

`ReducedValuesIn` has a single template parameter that specifies what data types are acceptable for the array. The type tags are described in Section 13.4.1 starting on page 157.

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.

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 13.10).

`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 13.10).

`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 23.2, 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 13.10: 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 13.11: A simple map worklet to identify histogram bins, which will be used as keys.

```
  struct IdentifyBins : vtkm::worklet::WorkletMapField
  {
    using ControlSignature = void(FieldIn<Scalar> data, FieldOut<IdType> 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 invoke of the reduce by keys dispatcher. This of course happens in the control environment after calling the dispatcher for our worklet for generating the keys.

Example 13.12: Creating a `vtkm::worklet::Keys` object.

```
vtkm::cont::ArrayHandle<vtkm::Id> binIds;
vtkm::worklet::DispatcherMapField<IdentifyBins> identifyDispatcher(bins);
identifyDispatcher.Invoke(valuesArray, binIds);

vtkm::worklet::Keys<vtkm::Id> keys(binIds, DeviceAdapterTag());
```

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 reduce by key dispatcher 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 13.13: 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 13.14: 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>
```

```
10      VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
11        const OriginalValuesVecType& originalValues) const
12      {
13        typename OriginalValuesVecType::ComponentType sum = 0;
14        for (vtkm::IdComponent index = 0;
15             index < originalValues.GetNumberOfComponents();
16             index++)
17        {
18          sum = sum + originalValues[index];
19        }
20        return sum / originalValues.GetNumberOfComponents();
21      }
22    };
```

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 13.15: Using a reduce by key worklet to average values falling into the same bin.

```
1   struct CombineSimilarValues
2   {
3     struct IdentifyBins : vtkm::worklet::WorkletMapField
4     {
5       using ControlSignature = void(FieldIn<Scalar> data, FieldOut<IdType> bins);
6       using ExecutionSignature = _2(_1);
7       using InputDomain = _1;
8
9       BinScalars Bins;
10
11      VTKM_CONT
12      IdentifyBins(const BinScalars& bins)
13        : Bins(bins)
14      {
15      }
16
17      VTKM_EXEC
18      vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
19    };
20
21    struct BinAverage : vtkm::worklet::WorkletReduceByKey
22    {
23      using ControlSignature = void(KeysIn keys,
24                                    ValuesIn<> originalValues,
25                                    ReducedValuesOut<> averages);
26      using ExecutionSignature = _3(_2);
27      using InputDomain = _1;
28
29      template<typename OriginalValuesVecType>
30      VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
31        const OriginalValuesVecType& originalValues) const
32      {
33        typename OriginalValuesVecType::ComponentType sum = 0;
34        for (vtkm::IdComponent index = 0;
35             index < originalValues.GetNumberOfComponents();
36             index++)
37        {
38          sum = sum + originalValues[index];
39        }
40        return sum / originalValues.GetNumberOfComponents();
41      }
42    };
43
44    template<typename InArrayHandleType, typename DeviceAdapterTag>
```

```
45    VTKM_CONT static vtkm::cont::ArrayHandle<typename InArrayHandleType::ValueType>
46    Run(const InArrayHandleType& valuesArray, vtkm::Id numBins, DeviceAdapterTag)
47    {
48      VTKM_IS_ARRAY_HANDLE(InArrayHandleType);
49
50      using ValueType = typename InArrayHandleType::ValueType;
51
52      vtkm::Range range =
53        vtkm::cont::ArrayRangeCompute(valuesArray).GetPortalConstControl().Get(0);
54      BinScalars bins(range, numBins);
55
56      vtkm::cont::ArrayHandle<vtkm::Id> binIds;
57      vtkm::worklet::DispatcherMapField<IdentifyBins> identifyDispatcher(bins);
58      identifyDispatcher.Invoke(valuesArray, binIds);
59
60      vtkm::worklet::Keys<vtkm::Id> keys(binIds, DeviceAdapterTag());
61
62      vtkm::cont::ArrayHandle<ValueType> combinedValues;
63
64      vtkm::worklet::DispatcherReduceByKey<BinAverage> averageDispatcher;
65      averageDispatcher.Invoke(keys, valuesArray, combinedValues);
66
67      return combinedValues;
68    }
69  };
```

13.6 Whole Arrays

As documented in Section 13.5, each worklet type has a set of parameter types that can be used to pass data to and from the worklet invocation. But what happens if you want to pass data that cannot be expressed in these predefined mechanisms. Chapter 23 describes how to create completely new worklet types and parameter tags. However, designing such a system for a one-time use is overkill.

Instead, all VTK-m worklets provide a trio of mechanisms that allow you to pass arbitrary data to a worklet. In this section, we will explore a *whole array* argument that provides random access to an entire array. In a later section we describe an even more general mechanism to describe any execution object.

Common Errors

The VTK-m worklet dispatching 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`. The corresponding argument to the dispatcher's `Invoke` 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. 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 13.7 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 13.16: Using `WholeArrayIn` to access a lookup table in a worklet.

```
#include <vtkm/cont/ArrayHandle.h>
#include <vtkm/cont/DataSet.h>

#include <vtkm/worklet/DispatcherMapTopology.h>
#include <vtkm/worklet/WorkletMapTopology.h>

#include <vtkm/CellShape.h>
#include <vtkm/Math.h>
#include <vtkm/VectorAnalysis.h>

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);
}

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>
```

```
51 VTKM_EXEC_CONT static vtkm::Float32 LookupTriangleQuality(
52   const PortalType& triangleQualityPortal,
53   const vtkm::Vec<T, 3>& point1,
54   const vtkm::Vec<T, 3>& point2,
55   const vtkm::Vec<T, 3>& point3)
56 {
57   vtkm::Vec<T, 3> edgeLengths = TriangleEdgeLengths(point1, point2, point3);
58
59   // To reduce the size of the table, we just store the quality of triangles
60   // with the longest edge of size 1. The table is 2D indexed by the length
61   // of the other two edges. Thus, to use the table we have to identify the
62   // longest edge and scale appropriately.
63   T smallEdge1 = vtkm::Min(edgeLengths[0], edgeLengths[1]);
64   T tmpEdge = vtkm::Max(edgeLengths[0], edgeLengths[1]);
65   T smallEdge2 = vtkm::Min(edgeLengths[2], tmpEdge);
66   T largeEdge = vtkm::Max(edgeLengths[2], tmpEdge);
67
68   smallEdge1 /= largeEdge;
69   smallEdge2 /= largeEdge;
70
71   // Find index into array.
72   vtkm::Id index1 = static_cast<vtkm::Id>(
73     vtkm::Floor(smallEdge1 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
74   vtkm::Id index2 = static_cast<vtkm::Id>(
75     vtkm::Floor(smallEdge2 * (TRIANGLE_QUALITY_TABLE_DIMENSION - 1) + 0.5));
76   vtkm::Id totalIndex = index1 + index2 * TRIANGLE_QUALITY_TABLE_DIMENSION;
77
78   return triangleQualityPortal.Get(totalIndex);
79 }
80
81 } // namespace detail
82
83 struct TriangleQuality
84 {
85   struct TriangleQualityWorklet : vtkm::worklet::WorkletMapPointToCell
86   {
87     using ControlSignature = void(CellSetIn cells,
88                                   FieldInPoint<Vec3> pointCoordinates,
89                                   WholeArrayIn<Scalar> triangleQualityTable,
90                                   FieldOutCell<Scalar> triangleQuality);
91     using ExecutionSignature = _4(CellShape, _2, _3);
92     using InputDomain = _1;
93
94     template<typename CellShape,
95              typename PointCoordinatesType,
96              typename TriangleQualityTablePortalType>
97     VTKM_EXEC vtkm::Float32 operator()(
98       CellShape shape,
99       const PointCoordinatesType& pointCoordinates,
100      const TriangleQualityTablePortalType& triangleQualityTable) const
101    {
102      if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
103      {
104        this->RaiseError("Only triangles are supported for triangle quality.");
105        return vtkm::Nan32();
106      }
107
108      return detail::LookupTriangleQuality(triangleQualityTable,
109                                           pointCoordinates[0],
110                                           pointCoordinates[1],
111                                           pointCoordinates[2]);
112    }
113  };
114
```

```
115   template<typename DeviceAdapterTag>
116   VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Float32> Run(
117     vtkm::cont::DataSet dataSet,
118     DeviceAdapterTag)
119   {
120     vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualityTable =
121       detail::GetTriangleQualityTable();
122
123     vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
124
125     vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet> dispatcher;
126     dispatcher.Invoke(dataSet.GetCellSet(),
127                       dataSet.GetCoordinateSystem().GetData(),
128                       triangleQualityTable,
129                       triangleQualities);
130
131     return triangleQualities;
132   }
133 };
```

13.7 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 dispatcher's `Invoke` 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::AtomicArray` structure. `AtomicArray` has two important methods:

`AtomicArray::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.

`AtomicArray::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 13.5.3 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 13.17: Using `AtomicArrayInOut` to count histogram bins in a worklet.

```
struct CountBins : vtkm::worklet::WorkletMapField
{
  using ControlSignature = void(FieldIn<Scalar> data,
                                AtomicArrayInOut<> histogramBins);
  using ExecutionSignature = void(_1, _2);
  using InputDomain = _1;

  vtkm::Range HistogramRange;
  vtkm::Id NumberOfBins;

  VTKM_CONT
  CountBins(const vtkm::Range& histogramRange, vtkm::Id& numBins)
    : HistogramRange(histogramRange)
    , NumberOfBins(numBins)
  {
  }

  template<typename T, typename AtomicArrayType>
  VTKM_EXEC void operator()(T value, const AtomicArrayType& histogramBins) const
  {
    vtkm::Float64 interp =
      (value - this->HistogramRange.Min) / this->HistogramRange.Length();
    vtkm::Id bin = static_cast<vtkm::Id>(interp * this->NumberOfBins);
    if (bin < 0)
    {
      bin = 0;
    }
    if (bin >= this->NumberOfBins)
    {
      bin = this->NumberOfBins - 1;
    }

    histogramBins.Add(bin, 1);
  }
};
```

13.8 Whole Cell Sets

Section 13.5.2 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 dispatcher's `Invoke` should be a `CellSet` subclass or a `DynamicCellSet` (both of which are described in Section 12.2).

The `WholeCellSetIn` is templated and takes two arguments: the "from" topology type and the "to" 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 "from" and "to" topology types define which topological elements can be queried and which incident elements are returned. The semantics of the "from" and "to" topology is the same as that for the general topology maps described in Section 13.5.2. You can look up an element of the "to" topology by index and then get all of the "from" elements that are incident from it.

For example, a `WholeCellSetIn<Point, Cell>` allows you to find all the points that are incident on each cell (as well as querying the cell shape). Likewise, a `WholeCellSetIn<Cell, Point>` allows you to find all the cells that are incident on each point. The default parameters of `WholeCellSetIn` are from point to cell. That is, `WholeCellSetIn<>` is equivalent to `WholeCellSetIn<Point, Cell>`.

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 15.1.) 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 15.

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 13.3 one of the angles attached to the middle point is labeled θ_j.

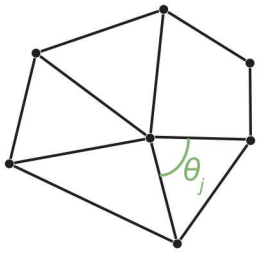


Figure 13.3: The angles incident around a point in a mesh.

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 map cell to point worklet (Section 13.5.2) 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 13.18: Using `WholeCellSetIn` to sum the angles around each point.

```
1  struct SumOfAngles : vtkm::worklet::WorkletMapCellToPoint
2  {
3    using ControlSignature = void(CellSetIn inputCells,
4                                  WholeCellSetIn<>, // Same as inputCells
5                                  WholeArrayIn<> pointCoords,
6                                  FieldOutPoint<Scalar> 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 =
36         cellSet.GetCellShape(cellIndex);
37       typename CellSetType::IndicesType cellConnections =
38         cellSet.GetIndices(cellIndex);
39       vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
```

```
40          vtkm::IdComponent numEdges =
41            vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
42
43          // Iterate over all edges and find the first one with pointIndex.
44          // Use that to find the first vector.
45          vtkm::IdComponent edgeIndex = -1;
46          CoordType vec1;
47          while (true)
48          {
49            ++edgeIndex;
50            if (edgeIndex >= numEdges)
51            {
52              this->RaiseError("Bad cell. Could not find two incident edges.");
53              return;
54            }
55            auto edge =
56              vtkm::make_Vec(vtkm::exec::CellEdgeLocalIndex(
57                               numPointsInCell, 0, edgeIndex, cellShape, *this),
58                             vtkm::exec::CellEdgeLocalIndex(
59                               numPointsInCell, 1, edgeIndex, cellShape, *this));
60            if (cellConnections[edge[0]] == pointIndex)
61            {
62              vec1 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
63              break;
64            }
65            else if (cellConnections[edge[1]] == pointIndex)
66            {
67              vec1 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
68              break;
69            }
70            else
71            {
72              // Continue to next iteration of loop.
73            }
74          }
75
76          // Continue iteration over remaining edges and find the second one with
77          // pointIndex. Use that to find the second vector.
78          CoordType vec2;
79          while (true)
80          {
81            ++edgeIndex;
82            if (edgeIndex >= numEdges)
83            {
84              this->RaiseError("Bad cell. Could not find two incident edges.");
85              return;
86            }
87            auto edge =
88              vtkm::make_Vec(vtkm::exec::CellEdgeLocalIndex(
89                               numPointsInCell, 0, edgeIndex, cellShape, *this),
90                             vtkm::exec::CellEdgeLocalIndex(
91                               numPointsInCell, 1, edgeIndex, cellShape, *this));
92            if (cellConnections[edge[0]] == pointIndex)
93            {
94              vec2 = pointCoordsPortal.Get(cellConnections[edge[1]]) - thisPoint;
95              break;
96            }
97            else if (cellConnections[edge[1]] == pointIndex)
98            {
99              vec2 = pointCoordsPortal.Get(cellConnections[edge[0]]) - thisPoint;
100             break;
101           }
102           else
103           {
```

```
104             // Continue to next iteration of loop.
105           }
106         }
107
108         // The dot product of two unit vectors is equal to the cosine of the
109         // angle between them.
110         vtkm::Normalize(vec1);
111         vtkm::Normalize(vec2);
112         SumType cosine = static_cast<SumType>(vtkm::Dot(vec1, vec2));
113
114         outSum += vtkm::ACos(cosine);
115       }
116     }
117   };
```

13.9 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`, templated on device adapter tag, and passed as an argument. 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 `PrepareForInput` method in `vtkm::cont::ArrayHandle` as described in Section 7.6. Other VTK-m data objects, such as the subclasses of `vtkm::cont::CellSet`, have similar methods.

Returning to the example we have in Section 13.6, we are computing triangle quality quickly by looking up the value in a table. In Example 13.16 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` 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 13.16 to create such an object.

Example 13.19: Using `ExecObject` to access a lookup table in a worklet.

```
1  template<typename DeviceAdapterTag>
2  class TriangleQualityTableExecutionObject
3  {
4  public:
5    VTKM_CONT
6    TriangleQualityTableExecutionObject()
7    {
8      this->TablePortal =
9        detail::GetTriangleQualityTable().PrepareForInput(DeviceAdapterTag());
10   }
11
12   template<typename T>
```

```
13    VTKM_EXEC vtkm::Float32 GetQuality(const vtkm::Vec<T, 3>& point1,
14                                       const vtkm::Vec<T, 3>& point2,
15                                       const vtkm::Vec<T, 3>& point3) const
16    {
17      return detail::LookupTriangleQuality(this->TablePortal, point1, point2, point3);
18    }
19
20  private:
21    using TableArrayType = vtkm::cont::ArrayHandle<vtkm::Float32>;
22    using TableArrayPortalType =
23      typename TableArrayType::ExecutionTypes<DeviceAdapterTag>::PortalConst;
24    TableArrayPortalType TablePortal;
25  };
26
27  class TriangleQualityTable : public vtkm::cont::ExecutionObjectBase
28  {
29  public:
30    template<typename Device>
31    VTKM_CONT TriangleQualityTableExecutionObject<Device> PrepareForExecution(
32      Device) const
33    {
34      return TriangleQualityTableExecutionObject<Device>();
35    }
36  };
37
38  struct TriangleQualityWorklet2 : vtkm::worklet::WorkletMapPointToCell
39  {
40    using ControlSignature = void(CellSetIn cells,
41                                  FieldInPoint<Vec3> pointCoordinates,
42                                  ExecObject triangleQualityTable,
43                                  FieldOutCell<Scalar> triangleQuality);
44    using ExecutionSignature = _4(CellShape, _2, _3);
45    using InputDomain = _1;
46
47    template<typename CellShape,
48             typename PointCoordinatesType,
49             typename TriangleQualityTableType>
50    VTKM_EXEC vtkm::Float32 operator()(
51      CellShape shape,
52      const PointCoordinatesType& pointCoordinates,
53      const TriangleQualityTableType& triangleQualityTable) const
54    {
55      if (shape.Id != vtkm::CELL_SHAPE_TRIANGLE)
56      {
57        this->RaiseError("Only triangles are supported for triangle quality.");
58        return vtkm::Nan32();
59      }
60
61      return triangleQualityTable.GetQuality(
62        pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
63    }
64  };
65
66  // Normally we would encapsulate this call in a filter, but for demonstrative
67  // purposes we are just calling the worklet directly.
68  template<typename DeviceAdapterTag>
69  VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> RunTriangleQuality2(
70    vtkm::cont::DataSet dataSet,
71    DeviceAdapterTag)
72  {
73    TriangleQualityTable triangleQualityTable;
74
75    vtkm::cont::ArrayHandle<vtkm::Float32> triangleQualities;
76
```

```
77   vtkm::worklet::DispatcherMapTopology<TriangleQualityWorklet2> dispatcher;
78   dispatcher.Invoke(dataSet.GetCellSet(),
79                     dataSet.GetCoordinateSystem().GetData(),
80                     triangleQualityTable,
81                     triangleQualities);
82
83   return triangleQualities;
84 }
```

13.10 Scatter

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::WorkletMapPointToCell` 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* for a worklet. 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. The worklet provides a number at runtime that defines 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 worklet provides 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.

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 dispatcher.

Example 13.20: Declaration of a scatter type in a worklet.

```
1    using ScatterType = vtkm::worklet::ScatterCounting;
2
3    template<typename CountArrayType, typename DeviceAdapterTag>
4    VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray,
5                                             DeviceAdapterTag)
6    {
7      VTKM_IS_ARRAY_HANDLE(CountArrayType);
8      return ScatterType(countArray, DeviceAdapterTag());
9    }
```

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.

It is stated in Section 13.2 that when a dispatcher object is created, it must be given a worklet object (or a default object will be created). Additionally, a dispatcher must have a scatter object when it is created. Like with the worklet, if the required scatter object does not hold any state, then the dispatcher can automatically create a scatter object with the scatter's default constructor. This is the case for the default scatter, so none of the previous examples needed to define a scatter for the dispatcher. However, a `ScatterCounting` needs to be constructed, set up, and passed to the dispatcher's constructor. As mentioned previously, it is generally good practice for the worklet to provide a static `MakeScatter` method to construct a scatter of the correct type and state if the dispatcher requires a custom scatter. If both a worklet object and a scatter object need to be given to a dispatcher's constructor, the worklet is given as the first argument.

Example 13.21: Constructing a dispatcher that requires a custom scatter.

```
1  auto generateScatter =
2    ClipPoints::Generate::MakeScatter(countArray, DeviceAdapterTag());
3  vtkm::worklet::DispatcherMapField<ClipPoints::Generate> dispatcherGenerate(
4    generateScatter);
```

Did you know?

A scatter object does not have to be tied to a single worklet/dispatcher 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::cont::ScatterUniform` of size 2 and using the `VisitIndex` to determine from which array to pull a value.

Example 13.22: Using `ScatterUniform`.

```
1  struct InterleaveArrays : vtkm::worklet::WorkletMapField
2  {
3    using ControlSignature = void(FieldIn<>, FieldIn<>, FieldOut<>);
4    using ExecutionSignature = void(_1, _2, _3, VisitIndex);
5    using InputDomain = _1;
6
7    using ScatterType = vtkm::worklet::ScatterUniform<2>;
8
9    template<typename T>
10   VTKM_EXEC void operator()(const T& input0,
11                             const T& input1,
12                             T& output,
13                             vtkm::IdComponent visitIndex) const
14   {
15     if (visitIndex == 0)
16     {
17       output = input0;
18     }
```

```
    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::cont::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 13.23: Using `ScatterCounting`.

```
struct ClipPoints
{
  class Count : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn<Vec3> points,
                                  FieldOut<IdComponentType> count);
    using ExecutionSignature = _2(_1);
    using InputDomain = _1;

    template<typename T>
    VTKM_CONT Count(const vtkm::Vec<T, 3>& boundsMin,
                    const vtkm::Vec<T, 3>& boundsMax)
      : BoundsMin(boundsMin[0], boundsMin[1], boundsMin[2])
      , BoundsMax(boundsMax[0], boundsMax[1], boundsMax[2])
    {
    }

    template<typename T>
    VTKM_EXEC vtkm::IdComponent operator()(const vtkm::Vec<T, 3>& point) const
    {
      return static_cast<vtkm::IdComponent>(
        (this->BoundsMin[0] < point[0]) && (this->BoundsMin[1] < point[1]) &&
        (this->BoundsMin[2] < point[2]) && (this->BoundsMax[0] > point[0]) &&
        (this->BoundsMax[1] > point[1]) && (this->BoundsMax[2] > point[2]));
    }

  private:
    vtkm::Vec<vtkm::FloatDefault, 3> BoundsMin;
    vtkm::Vec<vtkm::FloatDefault, 3> BoundsMax;
  };

  class Generate : public vtkm::worklet::WorkletMapField
  {
  public:
    using ControlSignature = void(FieldIn<Vec3> inPoints, FieldOut<Vec3> outPoints);
    using ExecutionSignature = void(_1, _2);
    using InputDomain = _1;

    using ScatterType = vtkm::worklet::ScatterCounting;

    template<typename CountArrayType, typename DeviceAdapterTag>
    VTKM_CONT static ScatterType MakeScatter(const CountArrayType& countArray,
                                             DeviceAdapterTag)
    {
      VTKM_IS_ARRAY_HANDLE(CountArrayType);
      return ScatterType(countArray, DeviceAdapterTag());
    }

    template<typename InType, typename OutType>
```

```
51      VTKM_EXEC void operator()(const vtkm::Vec<InType, 3>& inPoint,
52                                vtkm::Vec<OutType, 3>& outPoint) const
53      {
54        // The scatter ensures that this method is only called for input points
55        // that are passed to the output (where the count was 1). Thus, in this
56        // case we know that we just need to copy the input to the output.
57        outPoint = vtkm::Vec<OutType, 3>(inPoint[0], inPoint[1], inPoint[2]);
58      }
59    };
60
61    template<typename T, typename Storage, typename DeviceAdapterTag>
62    VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<T, 3>> Run(
63      const vtkm::cont::ArrayHandle<vtkm::Vec<T, 3>, Storage>& pointArray,
64      vtkm::Vec<T, 3> boundsMin,
65      vtkm::Vec<T, 3> boundsMax,
66      DeviceAdapterTag)
67    {
68      vtkm::cont::ArrayHandle<vtkm::IdComponent> countArray;
69
70      ClipPoints::Count workletCount(boundsMin, boundsMax);
71      vtkm::worklet::DispatcherMapField<ClipPoints::Count> dispatcherCount(
72        workletCount);
73      dispatcherCount.Invoke(pointArray, countArray);
74
75      vtkm::cont::ArrayHandle<vtkm::Vec<T, 3>> clippedPointsArray;
76
77      auto generateScatter =
78        ClipPoints::Generate::MakeScatter(countArray, DeviceAdapterTag());
79      vtkm::worklet::DispatcherMapField<ClipPoints::Generate> dispatcherGenerate(
80        generateScatter);
81      dispatcherGenerate.Invoke(pointArray, clippedPointsArray);
82
83      return clippedPointsArray;
84    }
85  };
```

13.11 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 13.24: Raising an error in the execution environment.

```
1  struct SquareRoot : vtkm::worklet::WorkletMapField
2  {
```

```
3  public:
4    using ControlSignature = void(FieldIn<Scalar>, FieldOut<Scalar>);
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 6.7 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

FOURTEEN

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.

14.1 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::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 `NagativeInfinity32` 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::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 byt 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).

14.2 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.

`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.

14.3 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 14.1: 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.

14.4 Newton's Method

VTK-m's matrix methods (documented in Section 14.3) 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 14.2: 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:
  //
  //    x^2 + y^2 = 2
  //    x*y = 1
  //
  // There are two possible solutions, which are (x=1,y=1) and (x=-1,y=-1).
  // The one found depends on the starting value.
  vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer1 =
    vtkm::NewtonsMethod(JacobianFunctor(),
                        FunctionFunctor(),
                        vtkm::make_Vec(2.0f, 1.0f),
                        vtkm::make_Vec(1.0f, 0.0f));
  if (!answer1.Valid || !answer1.Converged)
  {
    // Failed to find solution
  }
  // answer1.Solution is [1,1]

  vtkm::NewtonsMethodResult<vtkm::Float32, 2> answer2 =
    vtkm::NewtonsMethod(JacobianFunctor(),
                        FunctionFunctor(),
                        vtkm::make_Vec(2.0f, 1.0f),
                        vtkm::make_Vec(0.0f, -2.0f));
  if (!answer2.Valid || !answer2.Converged)
  {
    // Failed to find solution
  }
```

```
60 |   // answer2 is [-1,-1]
61 | }
```

CHAPTER

FIFTEEN

WORKING WITH CELLS

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See Section 12.2 starting on page 144 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.

15.1 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 15.1 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.

15.1.1 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

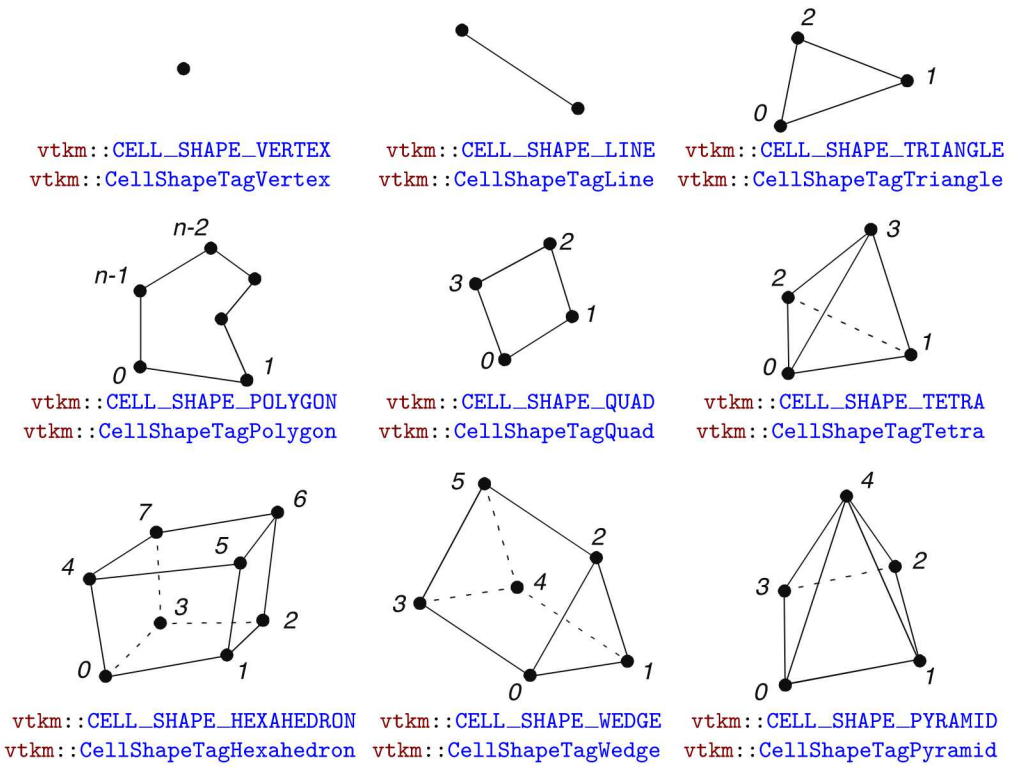


Figure 15.1: Basic Cell Shapes

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 15.1: 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 15.2.

15.1.2 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.

`CellTraits::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.

`CellTraits::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.

`CellTraits::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).

`CellTraits::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 15.2: Using `CellTraits` to implement a polygon normal estimator.

```
1  namespace detail
2  {
3
4  VTKM_SUPPRESS_EXEC_WARNINGS
5  template<typename PointCoordinatesVector, typename WorkletType>
6  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
7    const PointCoordinatesVector& pointCoordinates,
8    vtkm::CellTopologicalDimensionsTag<2>,
9    const WorkletType& worklet)
10 {
11   if (pointCoordinates.GetNumberOfComponents() >= 3)
12   {
13     return vtkm::TriangleNormal(
14       pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
15   }
16   else
17   {
18     worklet.RaiseError("Degenerate polygon.");
19     return typename PointCoordinatesVector::ComponentType();
20   }
21 }
22
23 VTKM_SUPPRESS_EXEC_WARNINGS
24 template<typename PointCoordinatesVector,
25          vtkm::IdComponent Dimensions,
26          typename WorkletType>
27 VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
28   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 }
```

15.2 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 4 arguments: the number of points in the cell, a `vtkm::Vec` of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a `vtkm::Vec <vtkm::FloatDefault,3>` instead of passing it as a parameter.

`vtkm::exec::ParametricCoordinatesPoint` Returns the parametric coordinates for a given point of a given shape. It takes 5 arguments: the number of points in the cell, the index of the point to query, a `vtkm::Vec` of size 3 to store the results, a shape tag, and a worklet object (for raising errors). A second form of this method takes 3 arguments and returns the result as a `vtkm::Vec <<vtkm::FloatDefault,3>` instead of passing it as a parameter.

`vtkm::exec::ParametricCoordinatesToWorldCoordinates` Given 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 worklet object (for raising errors), returns the world coordinates.

`vtkm::exec::WorldCoordinatesToParametricCoordinates` Given 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 worklet object (for raising errors), returns the parametric coordinates. This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

15.3 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, and a worklet object (for raising errors). It returns the field interpolated to the location represented by the given parametric coordinates.

Example 15.3: Interpolating field values to a cell's center.

```
struct CellCenters : vtkm::worklet::WorkletMapPointToCell
{
  using ControlSignature = void(CellSetIn,
                                FieldInPoint<> inputField,
                                FieldOutCell<> outputField);
  using ExecutionSignature = void(CellShape, PointCount, _2, _3);
  using InputDomain = _1;

  template<typename CellShapeTag, typename FieldInVecType, typename FieldOutType>
  VTKM_EXEC void operator()(CellShapeTag shape,
                            vtkm::IdComponent pointCount,
                            const FieldInVecType& inputField,
                            FieldOutType& outputField) const
  {
    vtkm::Vec<vtkm::FloatDefault, 3> center =
      vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
    outputField = vtkm::exec::CellInterpolate(inputField, center, shape, *this);
  }
};
```

15.4 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 15.4: Computing the derivative of the field at cell centers.

```
struct CellDerivatives : vtkm::worklet::WorkletMapPointToCell
{
  using ControlSignature = void(CellSetIn,
                                FieldInPoint<> inputField,
                                FieldInPoint<Vec3> pointCoordinates,
                                FieldOutCell<> outputField);
```

```
7   using ExecutionSignature = void(CellShape, PointCount, _2, _3, _4);
8   using InputDomain = _1;
9
10  template<typename CellShapeTag,
11           typename FieldInVecType,
12           typename PointCoordVecType,
13           typename FieldOutType>
14  VTKM_EXEC void operator()(CellShapeTag shape,
15                            vtkm::IdComponent pointCount,
16                            const FieldInVecType& inputField,
17                            const PointCoordVecType& pointCoordinates,
18                            FieldOutType& outputField) const
19  {
20    vtkm::Vec<vtkm::FloatDefault, 3> center =
21      vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, *this);
22    outputField =
23      vtkm::exec::CellDerivative(inputField, pointCoordinates, center, shape, *this);
24  }
25 };
```

15.5 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 15.1. It contains Figure 15.1 on page 202, 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 15.2 demonstrates the relationship of these constituent elements for some example cell shapes.

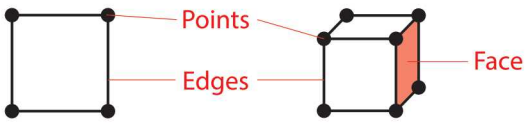


Figure 15.2: The constituent elements (points, edges, and faces) of cells.

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 an instance of the calling worklet (for error reporting). It returns the number of edges the cell has (as a `vtkm::IdComponent`).

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 an instance of the calling worklet. It returns a `vtkm::IdComponent` containing 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 15.2. To get the point indices relative to the data set, the edge indices should be used to reference a `PointIndices` list.

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 an instance of the calling worklet. It returns a `vtkm::Id2` 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 15.5 demonstrates one of many techniques for creating cell sets in a worklet. Chapter 17 describes this and many more such techniques.

Example 15.5: Using cell edge functions.

```
1  struct EdgesCount : vtkm::worklet::WorkletMapPointToCell
2  {
3    using ControlSignature = void(CellSetIn, FieldOutCell<> numEdgesInCell);
4    using ExecutionSignature = _2(CellShape, PointCount);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC vtkm::IdComponent operator()(CellShapeTag cellShape,
9                                           vtkm::IdComponent numPointsInCell) const
10   {
11     return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
12   }
13 };
14
15 struct EdgesExtract : vtkm::worklet::WorkletMapPointToCell
16 {
17   using ControlSignature = void(CellSetIn, FieldOutCell<> edgeIndices);
18   using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
19   using InputDomain = _1;
20
21   using ScatterType = vtkm::worklet::ScatterCounting;
22
23   template<typename CellShapeTag,
24            typename PointIndexVecType,
25            typename EdgeIndexVecType>
26   VTKM_EXEC void operator()(CellShapeTag cellShape,
27                             const PointIndexVecType& globalPointIndicesForCell,
28                             vtkm::IdComponent edgeIndex,
29                             EdgeIndexVecType& edgeIndices) const
30   {
31     vtkm::IdComponent numPointsInCell =
32       globalPointIndicesForCell.GetNumberOfComponents();
33
34     vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
35       numPointsInCell, 0, edgeIndex, cellShape, *this);
36     vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
37       numPointsInCell, 1, edgeIndex, cellShape, *this);
38
39     edgeIndices[0] = globalPointIndicesForCell[pointInCellIndex0];
40     edgeIndices[1] = globalPointIndicesForCell[pointInCellIndex1];
41   }
42 };
```

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 an instance of the calling worklet (for error reporting). It returns the number of faces the cell has (as a `vtkm::IdComponent`).

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 an instance of the calling worklet. It returns the number of points the specified face has (as a `vtkm::IdComponent`).

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 an instance of the calling worklet. It returns a `vtkm::IdComponent` containing 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 15.2. To get the point indices relative to the data set, the face indices should be used to reference a `PointIndices` list.

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 an instance of the calling worklet. It returns a `vtkm::Id3` 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 15.6: Using cell face functions.

```
1  struct FacesCount : vtkm::worklet::WorkletMapPointToCell
2  {
3    using ControlSignature = void(CellSetIn, FieldOutCell<> numFacesInCell);
4    using ExecutionSignature = _2(CellShape);
5    using InputDomain = _1;
6
7    template<typename CellShapeTag>
8    VTKM_EXEC vtkm::IdComponent operator()(CellShapeTag cellShape) const
9    {
10     return vtkm::exec::CellFaceNumberOfFaces(cellShape, *this);
11   }
12 };
13
14 struct FacesCountPoints : vtkm::worklet::WorkletMapPointToCell
15 {
16   using ControlSignature = void(CellSetIn,
17                                 FieldOutCell<> numPointsInFace,
18                                 FieldOutCell<> faceShape);
19   using ExecutionSignature = void(CellShape, VisitIndex, _2, _3);
20   using InputDomain = _1;
21
22   using ScatterType = vtkm::worklet::ScatterCounting;
23
24   template<typename CellShapeTag>
25   VTKM_EXEC void operator()(CellShapeTag cellShape,
26                             vtkm::IdComponent faceIndex,
27                             vtkm::IdComponent& numPointsInFace,
28                             vtkm::UInt8& faceShape) const
29   {
30     numPointsInFace =
31       vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, *this);
32     switch (numPointsInFace)
33     {
34       case 3:
35         faceShape = vtkm::CELL_SHAPE_TRIANGLE;
36         break;
```

```
37          case 4:
38            faceShape = vtkm::CELL_SHAPE_QUAD;
39            break;
40          default:
41            faceShape = vtkm::CELL_SHAPE_POLYGON;
42            break;
43        }
44      }
45    };
46
47    struct FacesExtract : vtkm::worklet::WorkletMapPointToCell
48    {
49      using ControlSignature = void(CellSetIn, FieldOutCell<> faceIndices);
50      using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
51      using InputDomain = _1;
52
53      using ScatterType = vtkm::worklet::ScatterCounting;
54
55      template<typename CellShapeTag,
56               typename PointIndexVecType,
57               typename FaceIndexVecType>
58      VTKM_EXEC void operator()(CellShapeTag cellShape,
59                                const PointIndexVecType& globalPointIndicesForCell,
60                                vtkm::IdComponent faceIndex,
61                                FaceIndexVecType& faceIndices) const
62      {
63        vtkm::IdComponent numPointsInFace = faceIndices.GetNumberOfComponents();
64        VTKM_ASSERT(numPointsInFace ==
65                    vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, *this));
66        for (vtkm::IdComponent pointInFaceIndex = 0;
67             pointInFaceIndex < numPointsInFace;
68             pointInFaceIndex++)
69        {
70          vtkm::IdComponent pointInCellIndex = vtkm::exec::CellFaceLocalIndex(
71            pointInFaceIndex, faceIndex, cellShape, *this);
72          faceIndices[pointInFaceIndex] = globalPointIndicesForCell[pointInCellIndex];
73        }
74      }
75    };
```

CHAPTER

SIXTEEN

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.

16.1 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 subclasses of `vtkm::cont::CellLocator`, 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.

16.1.1 Building a Cell Locator

All Cell Locators in VTK-m inherit from `vtkm::cont::CellLocator`, which provides the 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.

VTK-m currently exposes the implementations of the following Cell Locators.

Bounding Interval Hierarchy

The `vtkm::cont::BoundingIntervalHierarchy` cell locator is based on the bounding interval hierarchy spatial search structure. The implementation in VTK-m 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 be provided as parameters for the constructor or through the `SetNumberOfPlanes` and `SetMaxLeafSize` methods.

Example 16.1: Building a `vtkm::cont::BoundingIntervalHierarchy`.

```
// Create a locator that will use 5 splitting places,
// and will have a maximum of 10 cells per leaf node.
vtkm::cont::BoundingIntervalHierarchy locator(5, 10);
// Provides the CellSet required for the locator
locator.SetCellSet(cellSet)
// Provides the coordinate system required for the locator
locator.SetCoordinates(coords)
// Build the search structure (using a parallel device)
locator.Update();
```

16.1.2 Using Cell Locators in a Worklet

The `vtkm::cont::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 Section 13.9 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 pointer to a `vtkm::exec::CellLocator` object. `vtkm::exec::CellLocator` provides a `FindCell` method that identifies a containing cell given a point location in space.

Common Errors

Note that `vtkm::cont::CellLocator` and `vtkm::exec::CellLocator` are different objects with different interfaces despite the similar names.

The `CellLocator::FindCell()` method takes 4 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). The fourth argument is a reference to the calling worklet (used for error reporting purposes). The following example defines a simple worklet to get the containing cell id and parametric coordinates of a collection of world coordinates.

Example 16.2: Using a `CellLocator` in a worklet.

```
struct QueryCellIds : public vtkm::worklet::WorkletMapField
{
  using ControlSignature = void (FieldIn<>,
                                 ExecObject,
                                 FieldOut<>,
                                 FieldOut<>);
  using ExecutionSignature = void (_1, _2, _3, _4);

  template <typename Point, typename BoundingIntervalHierarchyExecObject>
  VTKM_EXEC vtkm::IdComponent operator()(
      const Point& point,
      BoundingIntervalHierarchyExecObject locator,
      vtkm::Id& cellId,
      vtkm::Vec<vtkm::FloatDefault, 3>& parametric) const
  {
    locator->FindCell(point, cellId, parametric, *this);
  }
}; // struct QueryCellIds

void QueryPointsInDataset(
    vtkm::cont::DataSet& dataSet,
    vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 3>>& queryPoints;
    vtkm::IdComponent numPlanes,
    vtkm::IdComponent maxLeadNodes)
{
  using DeviceAdapter = VTKM_DEFAULT_DEVICE_ADAPTER_TAG;
  using Algorithms = vtkm::cont::DeviceAdapterAlgorithm<DeviceAdapter>;
  using Timer = vtkm::cont::Timer<DeviceAdapter>;

  // Extract the data needed to build the locator from the DataSet
  vtkm::cont::DynamicCellSet cellSet = dataSet.GetCellSet();
  vtkm::cont::ArrayHandleVirtualCoordinates vertices =
      dataSet.GetCoordinateSystem().GetData();

  // Construct the search structure by providing the required data,
  // and then executing the Build method on the locator object.
  vtkm::cont::BoundingIntervalHierarchy locator =
      vtkm::cont::BoundingIntervalHierarchy(numPlanes, maxLeafNodes);
  locator.SetCellSet(cellSet);
  locator.SetCoordinates(dataSet.GetCoordinateSystem());
  locator.Update();

  // Define the arrays for storing the output of queries
  vtkm::cont::ArrayHandle<vtkm::Id> cellIds;
  vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FlaotDefault, 3>> parametric;

  // Invoke the worklet by providing the locator as an ExecObject
  vtkm::worklet::DispatcherMapField<QueryCellIds>().Invoke(queryPoints,
                                                           locator,
                                                           cellIds,
                                                           parametric);
  // Continue to Process
}
```

16.2 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 cell locators is a two step process. The first step is to build the search structure. This is done by instantiating one of the subclasses of vtkm::cont::PointLocator, 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 cell locator is built, it can be used in the execution environment within a filter or worklet.

16.2.1 Building Point Locators

All point Locators in VTK-m inherit from vtkm::cont::PointLocator, which provides the 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

VTK-m currently exposes the implementations of the following Point Locators.

Uniform Grid Point Locator

The vtkm::cont::PointLocatorUniformGrid 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 constructor of vtkm::cont::PointLocatorUniformGrid takes three parameters: the minimum position of the bounding box of the search space, the maximum position of the bounding box, and the number of grid cells to divide the search space.

Example 16.3: Building a vtkm::cont::PointLocatorUniformGrid.

```
// Create a PointLocatorUniformGrid with bounding box of 0, 0, 0 to
// 10, 10, 10 and divide the space into a 5 by 5 by 5 3D grid.
vtkm::cont::PointLocatorUniformGrid locator(
      { 0.0f, 0.0f, 0.0f }, { 10.0f, 10.0f, 10.0f }, { 5, 5, 5 });
// Set the position training points
locator.SetCoords(coord);
// Build the search structure
locator.Build();
```

16.2.2 Using Point Locators in a Worklet

The vtkm::cont::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 Section 13.9 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 pointer to a vtkm::exec::PointLocator object. vtkm::exec::PointLocator 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 `vtkm::exec::CellLocator::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 the point is not 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 16.4: Using a `PointLocator` in a worklet.

```
class PointLocatorUniformGridWorklet : public vtkm::worklet::WorkletMapField
{
public:
  using ControlSignature = void (FieldIn<> qcIn,
                                 ExecObject locator,
                                 FieldOut<> nnIdOut,
                                 FieldOut<> nnDistOut);

  using ExecutionSignature = void (_1, _2, _3, _4);

  VTKM_CONT
  PointLocatorUniformGridWorklet() {}

  template <typename CoordiVecType,
            typename Locator,
            typename IndexType,
            typename CoordiType>
  VTKM_EXEC void operator()(const CoordiVecType& qc,
                            const Locator& locator,
                            IndexType& nnIdOut,
                            CoordiType& nnDis) const
  {
    locator->FindNearestNeighbor(qc, nnIdOut, nnDis);
  }
};
///// randomly generate testing points/////
std::vector<vtkm::Vec<vtkm::Float32, 3>> qcVec;
for (vtkm::Int32 i = 0; i < nTestingPoint; i++)
{
  qcVec.push_back(vtkm::make_Vec(dr(dre), dr(dre), dr(dre)));
}
auto qc_Handle = vtkm::cont::make_ArrayHandle(qcVec);

vtkm::cont::ArrayHandle<vtkm::Id> nnId_Handle;
vtkm::cont::ArrayHandle<vtkm::Float32> nnDis_Handle;

PointLocatorUniformGridWorklet pointLocatorUniformGridWorklet;
vtkm::worklet::DispatcherMapField<PointLocatorUniformGridWorklet, DeviceAdapter>
  locatorDispatcher(pointLocatorUniformGridWorklet);
locatorDispatcher.Invoke(qc_Handle, locator, nnId_Handle, nnDis_Handle);
```

CHAPTER

SEVENTEEN

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 12 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 13. 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.

17.1 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 17.1: A simple worklet to count the number of edges on each cell.

```
1  struct CountEdges : vtkm::worklet::WorkletMapPointToCell
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        return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
13      }
14    };
```

This count array generated in Example 17.1 can be used in a `vtkm::worklet::ScatterCounting` of a subsequent worklet that generates the output cells. (See Section 13.10 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's 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::Vec <vtkm::Id,2>`.

Example 17.2: A worklet to generate indices for line cells.

```
1   class EdgeIndices : public vtkm::worklet::WorkletMapPointToCell
2   {
3   public:
4     using ControlSignature = void(CellSetIn cellSet, FieldOut<> connectivityOut);
5     using ExecutionSignature = void(CellShape, PointIndices, _2, VisitIndex);
6     using InputDomain = _1;
7
8     using ScatterType = vtkm::worklet::ScatterCounting;
9
10    template<typename CellShapeTag, typename PointIndexVecType>
11    VTKM_EXEC void operator()(CellShapeTag cellShape,
12                              const PointIndexVecType& globalPointIndicesForCell,
13                              vtkm::Vec<vtkm::Id, 2>& connectivityOut,
14                              vtkm::IdComponent edgeIndex) const
15    {
16      vtkm::IdComponent numPointsInCell =
17        globalPointIndicesForCell.GetNumberOfComponents();
18
19      vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
20        numPointsInCell, 0, edgeIndex, cellShape, *this);
21      vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
22        numPointsInCell, 1, edgeIndex, cellShape, *this);
23
24      connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
25      connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
26    }
27  };
```

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 17.2.

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 10.2.11) to make a flat array of indices look like an array of `Vec` objects. The following example shows a `Run` method in a worklet helper class. Note the use of `make_ArrayHandleGroupVec` when calling the `Invoke` method to make this conversion.

Example 17.3: Invoking worklets to extract edges from a cell set.

```
template<typename CellSetType>
VTKM_CONT vtkm::cont::CellSetSingleType<> Run(const CellSetType& inCellSet)
{
  VTKM_IS_DYNAMIC_OR_STATIC_CELL_SET(CellSetType);

  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  vtkm::worklet::DispatcherMapTopology<CountEdges> countEdgeDispatcher;
  countEdgeDispatcher.Invoke(inCellSet, edgeCounts);

  vtkm::worklet::ScatterCounting scatter(edgeCounts);
  this->OutputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());

  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  vtkm::worklet::DispatcherMapTopology<EdgeIndices> edgeIndicesDispatcher(scatter);
  edgeIndicesDispatcher.Invoke(
    inCellSet, vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  vtkm::cont::CellSetSingleType<> outCellSet(inCellSet.GetName());
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  return outCellSet;
}
```

Another feature to note in Example 17.3 is that the method calls `GetOutputToInputMap` on the `Scatter` object it creates and squirrels it away for later use. The reason for this behavior is to implement mapping fields that are attached to the input cells to the indices of the output. In practice, this worklet is going to be called on `DataSet` objects to create new `DataSet` objects. The method in Example 17.3 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 following example shows another convenience method that takes this saved `OutputToInputCellMap` array and converts an array from an input cell field to an output cell field array. Note that we can do this by using the `OutputToInputCellMap` as an index array in a `vtkm::cont::ArrayHandlePermutation`.

Example 17.4: Converting cell fields using a simple permutation.

```
template<typename ValueType, typename Storage>
VTKM_CONT vtkm::cont::ArrayHandle<ValueType> ProcessCellField(
  const vtkm::cont::ArrayHandle<ValueType, Storage>& inCellField) const
{
  vtkm::cont::ArrayHandle<ValueType> outCellField;
  vtkm::cont::ArrayCopy(vtkm::cont::make_ArrayHandlePermutation(
                          this->OutputToInputCellMap, inCellField),
                        outCellField);
  return outCellField;
```

```
10 | }
```

17.2 Combining Like Elements

Our motivating example in Section 17.1 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 17.1 will create multiple overlapping lines, one for each cell using the edge, as demonstrated in Figure 17.1. What we really want is to have one line for every edge in the mesh rather than many overlapping lines.

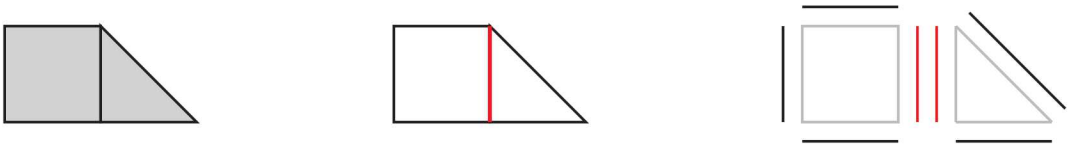


Figure 17.1: Duplicate lines from extracted edges. Consider the small mesh at the left comprising a square and a triangle. If we count the edges in this mesh, we would expect to get 6. However, our naïve implementation in Section 17.1 generates 7 because the shared edge (highlighted in red in the wireframe 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 17.5: A simple worklet to count the number of edges on each cell.

```
1  | struct CountEdges : vtkm::worklet::WorkletMapPointToCell
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 |     return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
13 |   }
14 | };
```

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 13.5.3 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 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 as 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 17.6: Worklet generating canonical edge identifiers.

```
class EdgeIds : public vtkm::worklet::WorkletMapPointToCell
{
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();

    canonicalIdOut = vtkm::exec::CellEdgeCanonicalId(numPointsInCell,
                                                     localEdgeIndex,
                                                     cellShape,
                                                     globalPointIndicesForCell,
                                                     *this);
  }
};
```

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`). The reduce by key dispatcher 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 executed the worklet back in Example 17.6.

As we did in Section 17.1, this worklet writes out the edge information in a `vtkm::Vec <vtkm::Id,2>` (which in some following code will be created with an `ArrayHandleGroupVec`).

Example 17.7: A worklet to generate indices for line cells from combined edges.

```
class EdgeIndices : public vtkm::worklet::WorkletReduceByKey
```

[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).

```
  {
  public:
    using ControlSignature = void(KeysIn keys,
                                  WholeCellSetIn<> inputCells,
                                  ValuesIn<> originCells,
                                  ValuesIn<> originEdges,
                                  ReducedValuesOut<> connectivityOut);
    using ExecutionSignature = void(_2 inputCells,
                                    _3 originCell,
                                    _4 originEdge,
                                    _5 connectivityOut);
    using InputDomain = _1;

    template<typename CellSetType,
             typename OriginCellsType,
             typename OriginEdgesType>
    VTKM_EXEC void operator()(const CellSetType& cellSet,
                              const OriginCellsType& originCells,
                              const OriginEdgesType& originEdges,
                              vtkm::Id2& connectivityOut) const
    {
      // Regardless of how many cells/edges are in our local input, we know they are
      // all the same, so just pick the first one.
      vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(originCells[0]);
      vtkm::IdComponent edgeIndex = originEdges[0];
      auto cellShape = cellSet.GetCellShape(originCells[0]);

      vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
        numPointsInCell, 0, edgeIndex, cellShape, *this);
      vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
        numPointsInCell, 1, edgeIndex, cellShape, *this);

      auto globalPointIndicesForCell = cellSet.GetIndices(originCells[0]);
      connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
      connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
    }
  };
```

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 17.7. 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 10.2.11) to make a flat array of indices look like an array of `Vec` objects. The

following example shows a `Run` method in a worklet helper class. Note the use of `make_ArrayHandleGroupVec` when calling the `Invoke` method to make this conversion.

Example 17.8: Invoking worklets to extract unique edges from a cell set.

```
template<typename CellSetType>
VTKM_CONT vtkm::cont::CellSetSingleType<> Run(const CellSetType& inCellSet)
{
  VTKM_IS_DYNAMIC_OR_STATIC_CELL_SET(CellSetType);

  // First, count the edges in each cell.
  vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
  vtkm::worklet::DispatcherMapTopology<CountEdges> countEdgeDispatcher;
  countEdgeDispatcher.Invoke(inCellSet, edgeCounts);

  vtkm::worklet::ScatterCounting scatter(edgeCounts);
  this->OutputToInputCellMap =
    scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
  vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
    scatter.GetVisitArray(inCellSet.GetNumberOfCells());

  // Second, for each edge, extract a canonical id.
  vtkm::cont::ArrayHandle<vtkm::Id2> canonicalIds;

  vtkm::worklet::DispatcherMapTopology<EdgeIds> edgeIdsDispatcher(scatter);
  edgeIdsDispatcher.Invoke(inCellSet, canonicalIds);

  // Third, use a Keys object to combine all like edge ids.
  this->CellToEdgeKeys = vtkm::worklet::Keys<vtkm::Id2>(canonicalIds);

  // Fourth, use a reduce-by-key to extract indices for each unique edge.
  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
  vtkm::worklet::DispatcherReduceByKey<EdgeIndices> edgeIndicesDispatcher;
  edgeIndicesDispatcher.Invoke(
    this->CellToEdgeKeys,
    inCellSet,
    this->OutputToInputCellMap,
    outputToInputEdgeMap,
    vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));

  // Fifth, use the created connectivity array to build a cell set.
  vtkm::cont::CellSetSingleType<> outCellSet(inCellSet.GetName());
  outCellSet.Fill(
    inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);

  return outCellSet;
}
```

Another feature to note in Example 17.8 is that the method calls `GetOutputToInputMap` on the `Scatter` object it creates and squirrels it away for later use. It also saves the `vtkm::worklet::Keys` object created for later user. The reason for this behavior is to implement mapping fields that are attached to the input cells to the indices of the output. In practice, these worklet are going to be called on `DataSet` objects to create new `DataSet` objects. The method in Example 17.8 creates a new `CellSet`, but we also need a method to transform the `Field`s on the data set. The saved `OutputToInputCellMap` array and `Keys` object allow us to transform input fields to output fields.

The following example shows another convenience method that takes these saved objects and converts an array from an input cell field to an output cell field array. Because in general there are several cells that contribute to each edge/line in the output, we need a method to combine all these cell values to one. The most appropriate combination is likely an average of all the values. Because this is a common operation, VTK-m provides the `vtkm::worklet::AverageByKey` to help perform exactly this operation. `AverageByKey` provides a `Run` method that takes a `Keys` object, an array of in values, and a device adapter tag and produces and array of values

averaged by key.

Example 17.9: Converting cell fields that average collected values.

```
template<typename ValueType, typename Storage>
VTKM_CONT vtkm::cont::ArrayHandle<ValueType> ProcessCellField(
  const vtkm::cont::ArrayHandle<ValueType, Storage>& inCellField) const
{
  return vtkm::worklet::AverageByKey::Run(
    this->CellToEdgeKeys,
    vtkm::cont::make_ArrayHandlePermutation(this->OutputToInputCellMap,
                                            inCellField));
}
```

17.3 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 17.1 we provided a pair of worklets that generate one line per edge per cell. In Section 17.2 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 17.8), 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 when collisions happen. Resolving hash value collisions adds overhead, but 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 17.2 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 17.10: A simple worklet to count the number of edges on each cell.

```
struct CountEdges : vtkm::worklet::WorkletMapPointToCell
{
  using ControlSignature = void(CellSetIn cellSet, FieldOut<> numEdges);
  using ExecutionSignature = _2(CellShape, PointCount);
```

[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).

```
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      return vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, *this);
13    }
14  };
```

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 17.11: Worklet generating hash values.

```
1   class EdgeHashes : public vtkm::worklet::WorkletMapPointToCell
2   {
3   public:
4     using ControlSignature = void(CellSetIn cellSet, FieldOut<> hashValues);
5     using ExecutionSignature = _2(CellShape cellShape,
6                                   PointIndices globalPointIndices,
7                                   VisitIndex localEdgeIndex);
8     using InputDomain = _1;
9
10    using ScatterType = vtkm::worklet::ScatterCounting;
11
12    template<typename CellShapeTag, typename PointIndexVecType>
13    VTKM_EXEC vtkm::HashType operator()(
14      CellShapeTag cellShape,
15      const PointIndexVecType& globalPointIndicesForCell,
16      vtkm::IdComponent localEdgeIndex) const
17    {
18      vtkm::IdComponent numPointsInCell =
19        globalPointIndicesForCell.GetNumberOfComponents();
20      return vtkm::Hash(vtkm::exec::CellEdgeCanonicalId(numPointsInCell,
21                                                        localEdgeIndex,
22                                                        cellShape,
23                                                        globalPointIndicesForCell,
24                                                        *this));
25    }
26  };
```

The hash values generated by the worklet in Example 17.11 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 17.12: Worklet to resolve hash collisions occurring on edge identifiers.

```
1   class EdgeHashCollisions : public vtkm::worklet::WorkletReduceByKey
2   {
3   public:
4     using ControlSignature = void(KeysIn keys,
5                                   WholeCellSetIn<> inputCells,
6                                   ValuesIn<> originCells,
7                                   ValuesIn<> originEdges,
```

```
                                  ValuesOut<> localEdgeIndices,
                                  ReducedValuesOut<> numEdges);
  using ExecutionSignature = _6(_2 inputCells,
                                  _3 originCells,
                                  _4 originEdges,
                                  _5 localEdgeIndices);
  using InputDomain = _1;

  template<typename CellSetType,
           typename OriginCellsType,
           typename OriginEdgesType,
           typename localEdgeIndicesType>
  VTKM_EXEC vtkm::IdComponent operator()(
    const CellSetType& cellSet,
    const OriginCellsType& originCells,
    const OriginEdgesType& originEdges,
    localEdgeIndicesType& localEdgeIndices) const
  {
    vtkm::IdComponent numEdgesInHash = localEdgeIndices.GetNumberOfComponents();

    // Sanity checks.
    VTKM_ASSERT(originCells.GetNumberOfComponents() == numEdgesInHash);
    VTKM_ASSERT(originEdges.GetNumberOfComponents() == numEdgesInHash);

    // Clear out localEdgeIndices
    for (vtkm::IdComponent index = 0; index < numEdgesInHash; ++index)
    {
      localEdgeIndices[index] = -1;
    }

    // Count how many unique edges there are and create an id for each;
    vtkm::IdComponent numUniqueEdges = 0;
    for (vtkm::IdComponent firstEdgeIndex = 0; firstEdgeIndex < numEdgesInHash;
         ++firstEdgeIndex)
    {
      if (localEdgeIndices[firstEdgeIndex] == -1)
      {
        vtkm::IdComponent edgeId = numUniqueEdges;
        localEdgeIndices[firstEdgeIndex] = edgeId;
        // Find all matching edges.
        vtkm::Id firstCellIndex = originCells[firstEdgeIndex];
        vtkm::Id2 canonicalEdgeId = vtkm::exec::CellEdgeCanonicalId(
          cellSet.GetNumberOfIndices(firstCellIndex),
          originEdges[firstEdgeIndex],
          cellSet.GetCellShape(firstCellIndex),
          cellSet.GetIndices(firstCellIndex),
          *this);
        for (vtkm::IdComponent laterEdgeIndex = firstEdgeIndex + 1;
             laterEdgeIndex < numEdgesInHash;
             ++laterEdgeIndex)
        {
          vtkm::Id laterCellIndex = originCells[laterEdgeIndex];
          vtkm::Id2 otherCanonicalEdgeId = vtkm::exec::CellEdgeCanonicalId(
            cellSet.GetNumberOfIndices(laterCellIndex),
            originEdges[laterEdgeIndex],
            cellSet.GetCellShape(laterCellIndex),
            cellSet.GetIndices(laterCellIndex),
            *this);
          if (canonicalEdgeId == otherCanonicalEdgeId)
          {
            localEdgeIndices[laterEdgeIndex] = edgeId;
          }
        }
        ++numUniqueEdges;
```

```
72          }
73        }
74
75        return numUniqueEdges;
76      }
77    };
```

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 dispatch 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 17.11.

As in the previous sections, this worklet writes out the edge information in a `vtkm::Vec <vtkm::Id,2>` (which in some following code will be created with an `ArrayHandleGroupVec`).

Example 17.13: A worklet to generate indices for line cells from combined edges and potential collisions.

```
1   class EdgeIndices : public vtkm::worklet::WorkletReduceByKey
2   {
3   public:
4     using ControlSignature = void(KeysIn keys,
5                                   WholeCellSetIn<> inputCells,
6                                   ValuesIn<> originCells,
7                                   ValuesIn<> originEdges,
8                                   ValuesIn<> localEdgeIndices,
9                                   ReducedValuesOut<> connectivityOut);
10    using ExecutionSignature = void(_2 inputCells,
11                                    _3 originCell,
12                                    _4 originEdge,
13                                    _5 localEdgeIndices,
14                                    VisitIndex localEdgeIndex,
15                                    _6 connectivityOut);
16    using InputDomain = _1;
17
18    using ScatterType = vtkm::worklet::ScatterCounting;
19
20    template<typename CellSetType,
21             typename OriginCellsType,
22             typename OriginEdgesType,
23             typename LocalEdgeIndicesType>
24    VTKM_EXEC void operator()(const CellSetType& cellSet,
25                              const OriginCellsType& originCells,
26                              const OriginEdgesType& originEdges,
27                              const LocalEdgeIndicesType& localEdgeIndices,
28                              vtkm::IdComponent localEdgeIndex,
29                              vtkm::Id2& connectivityOut) const
30    {
31      // Find the first edge that matches the index given and return it.
32      for (vtkm::IdComponent edgeIndex = 0;; ++edgeIndex)
33      {
34        if (localEdgeIndices[edgeIndex] == localEdgeIndex)
35        {
36          vtkm::Id cellIndex = originCells[edgeIndex];
37          vtkm::IdComponent numPointsInCell = cellSet.GetNumberOfIndices(cellIndex);
38          vtkm::IdComponent edgeInCellIndex = originEdges[edgeIndex];
39          auto cellShape = cellSet.GetCellShape(cellIndex);
40
41          vtkm::IdComponent pointInCellIndex0 = vtkm::exec::CellEdgeLocalIndex(
42            numPointsInCell, 0, edgeInCellIndex, cellShape, *this);
43          vtkm::IdComponent pointInCellIndex1 = vtkm::exec::CellEdgeLocalIndex(
44            numPointsInCell, 1, edgeInCellIndex, cellShape, *this);
45
46          auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
```

```
47            connectivityOut[0] = globalPointIndicesForCell[pointInCellIndex0];
48            connectivityOut[1] = globalPointIndicesForCell[pointInCellIndex1];
49
50            break;
51          }
52        }
53      }
54    };
```

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 17.7. 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 10.2.11) to make a flat array of indices look like an array of `Vec` objects. The following example shows a `Run` method in a worklet helper class. Note the use of `make_ArrayHandleGroupVec` when calling the `Invoke` method to make this conversion.

Example 17.14: Invoking worklets to extract unique edges from a cell set using hash values.

```
1   template<typename CellSetType>
2   VTKM_CONT vtkm::cont::CellSetSingleType<> Run(const CellSetType& inCellSet)
3   {
4     VTKM_IS_DYNAMIC_OR_STATIC_CELL_SET(CellSetType);
5
6     // First, count the edges in each cell.
7     vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
8     vtkm::worklet::DispatcherMapTopology<CountEdges> countEdgeDispatcher;
9     countEdgeDispatcher.Invoke(inCellSet, edgeCounts);
10
11    vtkm::worklet::ScatterCounting scatter(edgeCounts);
12    this->OutputToInputCellMap =
13      scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
14    vtkm::worklet::ScatterCounting::VisitArrayType outputToInputEdgeMap =
15      scatter.GetVisitArray(inCellSet.GetNumberOfCells());
16
17    // Second, for each edge, extract a hash.
18    vtkm::cont::ArrayHandle<vtkm::HashType> hashValues;
19
20    vtkm::worklet::DispatcherMapTopology<EdgeHashes> EdgeHashesDispatcher(scatter);
21    EdgeHashesDispatcher.Invoke(inCellSet, hashValues);
22
23    // Third, use a Keys object to combine all like hashes.
24    this->CellToEdgeKeys = vtkm::worklet::Keys<vtkm::HashType>(hashValues);
25
26    // Fourth, use a reduce-by-key to collect like hash values, resolve collisions,
27    // and count the number of unique edges associated with each hash.
28    vtkm::cont::ArrayHandle<vtkm::IdComponent> numUniqueEdgesInEachHash;
29
30    vtkm::worklet::DispatcherReduceByKey<EdgeHashCollisions>
31      edgeHashCollisionDispatcher;
32    edgeHashCollisionDispatcher.Invoke(this->CellToEdgeKeys,
33                                       inCellSet,
34                                       this->OutputToInputCellMap,
35                                       outputToInputEdgeMap,
36                                       this->LocalEdgeIndices,
37                                       numUniqueEdgesInEachHash);
```

```
38
39      // Fifth, use a reduce-by-key to extract indices for each unique edge.
40      this->HashCollisionScatter.reset(
41        new vtkm::worklet::ScatterCounting(numUniqueEdgesInEachHash));
42
43      vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
44      vtkm::worklet::DispatcherReduceByKey<EdgeIndices> edgeIndicesDispatcher(
45        *this->HashCollisionScatter);
46      edgeIndicesDispatcher.Invoke(
47        this->CellToEdgeKeys,
48        inCellSet,
49        this->OutputToInputCellMap,
50        outputToInputEdgeMap,
51        this->LocalEdgeIndices,
52        vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));
53
54      // Sixth, use the created connectivity array to build a cell set.
55      vtkm::cont::CellSetSingleType<> outCellSet(inCellSet.GetName());
56      outCellSet.Fill(
57        inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);
58
59      return outCellSet;
60    }
```

As noted in Section 17.2, in practice these worklets are going to be called on `DataSet` objects to create new `DataSet` objects. Although Example 17.14 creates a new `CellSet`, we also need a method to transform the `Field`s on the data set. To do this, we need to save some information. This includes the map from the edge of each cell to its origin cell (`OutputToInputCellMap`), a `Keys` object on the hash values (`CellToEdgeKeys`), an array of indices resolving collisions (`LocalEdgeIndices`), and a `ScatterCounting` to repeat edge outputs when unique edges collide on a hash (`HashCollisionScatter`).

Common Errors

Note that `HashCollisionScatter` *is an object of type* `vtkm::worklet::ScatterCounting`*, which does not have a default constructor. That can be a problem since you do not have the data to construct* `HashCollisionScatter` *until* `Run` *is called. To get around this chicken-and-egg issue, it is best to store the* `ScatterCounter` *as a pointer. It is best practice to use a smart pointer, like* `std::shared_ptr` *to manage it.*

In Section 17.2 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 17.15: A worklet to average values with the same key, resolving for collisions.

```
1   class AverageCellField : public vtkm::worklet::WorkletReduceByKey
2   {
3   public:
4     using ControlSignature = void(KeysIn keys,
5                                   ValuesIn<> inFieldValues,
6                                   ValuesIn<> localEdgeIndices,
7                                   ReducedValuesOut<> averagedField);
8     using ExecutionSignature = _4(_2 inFieldValues,
9                                   _3 localEdgeIndices,
10                                  VisitIndex localEdgeIndex);
11    using InputDomain = _1;
12
```

```
13      using ScatterType = vtkm::worklet::ScatterCounting;
14
15      template<typename InFieldValuesType, typename LocalEdgeIndicesType>
16      VTKM_EXEC typename InFieldValuesType::ComponentType operator()(
17        const InFieldValuesType& inFieldValues,
18        const LocalEdgeIndicesType& localEdgeIndices,
19        vtkm::IdComponent localEdgeIndex) const
20      {
21        using FieldType = typename InFieldValuesType::ComponentType;
22
23        FieldType averageField = FieldType(0);
24        vtkm::IdComponent numValues = 0;
25        for (vtkm::IdComponent reduceIndex = 0;
26             reduceIndex < inFieldValues.GetNumberOfComponents();
27             ++reduceIndex)
28        {
29          if (localEdgeIndices[reduceIndex] == localEdgeIndex)
30          {
31            FieldType fieldValue = inFieldValues[reduceIndex];
32            averageField = averageField + fieldValue;
33            ++numValues;
34          }
35        }
36        VTKM_ASSERT(numValues > 0);
37        return averageField / numValues;
38      }
39    };
```

With this worklet, it is straightforward to process cell fields.

Example 17.16: Invoking the worklet to process cell fields, resolving for collisions.

```
1   template<typename ValueType, typename Storage>
2   VTKM_CONT vtkm::cont::ArrayHandle<ValueType> ProcessCellField(
3     const vtkm::cont::ArrayHandle<ValueType, Storage>& inCellField) const
4   {
5     vtkm::cont::ArrayHandle<ValueType> averageField;
6     vtkm::worklet::DispatcherReduceByKey<AverageCellField> dispatcher(
7       *this->HashCollisionScatter);
8     dispatcher.Invoke(this->CellToEdgeKeys,
9                       vtkm::cont::make_ArrayHandlePermutation(
10                        this->OutputToInputCellMap, inCellField),
11                      this->LocalEdgeIndices,
12                      averageField);
13    return averageField;
14  }
```

17.4 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 set of worklets 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 17.1, 17.2, and 17.3. 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 17.3 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 outputing multiple cell types.

Our algorithm for extracting unique cell faces follows the same algorithm as that in Section 17.3. 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 17.10, 17.11, and 17.12, 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 17.17: A worklet to count the points in the final cells of extracted faces

```
class CountPointsInFace : public vtkm::worklet::WorkletReduceByKey
{
public:
  using ControlSignature = void(KeysIn keys,
                                WholeCellSetIn<> inputCells,
                                ValuesIn<> originCells,
                                ValuesIn<> originFaces,
                                ValuesIn<> localFaceIndices,
                                ReducedValuesOut<> faceShape,
                                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];
        faceShape = vtkm::exec::CellFaceShape(
          originFaces[faceIndex], cellSet.GetCellShape(cellIndex), *this);
        numPointsInFace = vtkm::exec::CellFaceNumberOfPoints(
```

```
            originFaces[faceIndex], cellSet.GetCellShape(cellIndex), *this);
          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 10.2.11, `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 17.17 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 17.18: Converting counts of connectivity groups to offsets for `ArrayHandleGroupVecVariable`.

```
vtkm::cont::ArrayHandle<vtkm::Id> offsetsArray;
vtkm::Id connectivityArraySize;
vtkm::cont::ConvertNumComponentsToOffsets(
  numPointsInEachFace, offsetsArray, connectivityArraySize);
OutCellSetType::ConnectivityArrayType connectivityArray;
connectivityArray.Allocate(connectivityArraySize);
auto connectivityArrayVecs =
  vtkm::cont::make_ArrayHandleGroupVecVariable(connectivityArray, offsetsArray);
```

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 17.13), but we have to correctly handle the `Vec`-like of unknown type and size.

Example 17.19: A worklet to generate indices for polygon cells of different sizes from combined edges and potential collisions.

```
class FaceIndices : 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();

          VTKM_ASSERT(numPointsInFace == vtkm::exec::CellFaceNumberOfPoints(
                                           faceInCellIndex, cellShape, *this));

          auto globalPointIndicesForCell = cellSet.GetIndices(cellIndex);
          for (vtkm::IdComponent localPointI = 0; localPointI < numPointsInFace;
               ++localPointI)
          {
            vtkm::IdComponent pointInCellIndex = vtkm::exec::CellFaceLocalIndex(
              localPointI, faceInCellIndex, cellShape, *this);
            connectivityOut[localPointI] =
              globalPointIndicesForCell[pointInCellIndex];
          }

          break;
        }
      }
    }
  };
```

With these worklets in place, we can use the following helper method to execute the series of worklets to extract unique faces.

Example 17.20: Invoking worklets to extract unique faces froma cell set.

```
    vtkm::cont::ArrayHandle<vtkm::Id> offsetsArray;
    vtkm::Id connectivityArraySize;
    vtkm::cont::ConvertNumComponentsToOffsets(
      numPointsInEachFace, offsetsArray, connectivityArraySize);
    OutCellSetType::ConnectivityArrayType connectivityArray;
    connectivityArray.Allocate(connectivityArraySize);
    auto connectivityArrayVecs =
      vtkm::cont::make_ArrayHandleGroupVecVariable(connectivityArray, offsetsArray);
```

As noted previously, in practice these worklets are going to be 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 17.3 (Examples 17.15 and 17.16).

CHAPTER

EIGHTEEN

CREATING FILTERS

In Chapter 13 we discuss how to implement an algorithm in the VTK-m framework by creating a worklet. Worklets might be straightforward to implement and invoke for those well familiar with the appropriate VTK-m API. However, novice users have difficulty using worklets directly. For simplicity, worklet algorithms are generally wrapped in what are called filter objects for general usage. Chapter 4 introduces the concept of filters and documents those that come with the VTK-m library. In this chapter we describe how to build new filter objects using the worklet examples introduced in Chapter 13.

Unsurprisingly, the base filter objects are contained in the `vtkm::filter` package. The basic implementation of a filter involves subclassing one of the base filter objects and implementing the `DoExecute` method. The `DoExecute` method performs the operation of the filter and returns a new data set containing the result.

As with worklets, there are several flavors of filter types to address different operating behaviors although their is not a one-to-one relationship between worklet and filter types. This chapter is sectioned by the different filter types with an example of implementations for each.

18.1 Field Filters

As described in Section 4.1 (starting on page 17), field filters are a category of filters that generate a new fields. These new fields are typically 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.

Field filters are implemented in classes that derive the `vtkm::filter::FilterField` base class. `FilterField` is a templated class that has a single template argument, which is the type of the concrete subclass.

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 `FilterField` *and other filter base classes, VTK-m uses this CRTP behavior to allow the general implementation of these algorithms to run* `DoExecute` *in the subclass, which as we see in a moment is itself templated.*

All `FilterField` subclasses must implement a `DoExecute` method. The `FilterField` base class implements an `Execute` method (actually several overloaded versions of `Execute`), processes the arguments, and then calls the `DoExecute` method of its subclass. The `DoExecute` method has the following 4 arguments.

- An input data set contained in a `vtkm::cont::DataSet` object. (See Chapter 12 for details on `DataSet`

objects.)

- The field from the DataSet specified in the Execute method to operate on. The field is always passed as an instance of vtkm::cont::ArrayHandle. (See Chapter 7 for details on ArrayHandle objects.) The type of the ArrayHandle is generally not known until the class is used and requires a template type.
- A vtkm::filter::FieldMetadata object that contains the associated metadata of the field not contained in the ArrayHandle of the second argument. The FieldMetadata contains information like the name of the field and what topological element the field is associated with (such as points or cells).
- A policy class. The type of the policy is generally unknown until the class is used and requires a template type.

In this section we provide an example implementation of a field filter that wraps the "magnitude" worklet provided in Example 13.6 (listed on page 160). By convention, filter 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 18.1: Header declaration for a field filter.

```
namespace vtkm
{
namespace filter
{

class FieldMagnitude : public vtkm::filter::FilterField<FieldMagnitude>
{
public:
  VTKM_CONT
  FieldMagnitude();

  template<typename ArrayHandleType, typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet,
    const ArrayHandleType& inField,
    const vtkm::filter::FieldMetadata& fieldMetadata,
    vtkm::filter::PolicyBase<Policy>);
};

template<>
class FilterTraits<vtkm::filter::FieldMagnitude>
{
public:
  struct InputFieldTypeList : vtkm::ListTagBase<vtkm::Vec<vtkm::Float32, 2>,
                                                vtkm::Vec<vtkm::Float64, 2>,
                                                vtkm::Vec<vtkm::Float32, 3>,
                                                vtkm::Vec<vtkm::Float64, 3>,
                                                vtkm::Vec<vtkm::Float32, 4>,
                                                vtkm::Vec<vtkm::Float64, 4>>
  {
  };
};

} // namespace filter
} // namespace vtkm
```

Notice that in addition to declaring the class for FieldMagnitude, Example 18.1 also specializes the vtkm::filter::FilterTraits templated class for FieldMagnitude. The FilterTraits class, declared in vtkm/filter/FilterTraits.h, provides hints used internally in the base filter classes to modify its behavior based on the subclass. A FilterTraits class is expected to define the following types.

`FilterTraits::InputFieldTypeList` A type list containing all the types that are valid for the input field. For example, a filter operating on positions in space might limit the types to three dimensional vectors. Type lists are discussed in detail in Section 6.6.2.

In the particular case of our `FieldMagnitude` filter, the filter expects to operate on some type of vector field. Thus, the `InputFieldTypeList` is modified to a list of all standard floating point `Vec`s.

Once the filter class and (optionally) the `FilterTraits` are declared in the .h file, the implementation filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named FieldMagnitude.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of `DoExecute` is straightforward. A worklet is invoked to compute a new field array. `DoExecute` then returns a newly constructed `vtkm::cont::DataSet` object containing the result. There are numerous ways to create a `DataSet`, but to simplify making filters, VTK-m provides the `vtkm::filter::internal::CreateResult` function to simplify the process. There are several overloaded versions of `vtkm::filter::internal::CreateResult` (defined in header file vtkm/filter/internal/CreateResult.h, but the simplest one to use for a filter that adds a field takes the following 5 arguments.

- The input data set. This is the same data set passed to the first argument of `DoExecute`.
- The array containing the data for the new field, which was presumably computed by the filter.
- The name of the new field.
- The topological association (e.g. points or cells) of the new field. In the case where the filter is a simple operation on a field array, the association can usually be copied from the `FieldMetadata` passed to `DoExecute`.
- The name of the element set the new field is associated with. This only has meaning if the new field is associated with cells and usually is specified if and only if the new field is associated with cells. This name usually can be copied from the `FieldMetadata` passed to `DoExecute`.

Note that all fields need a unique name, which is the reason for the third argument to `CreateResult`. The `vtkm::filter::FilterField` base class contains a pair of methods named `SetOutputFieldName` and `GetOutputFieldName` 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.

Example 18.2: Implementation of a field filter.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5
6  VTKM_CONT
7  FieldMagnitude::FieldMagnitude()
8  {
9    this->SetOutputFieldName("");
10 }
11
12 template<typename ArrayHandleType, typename Policy>
13 VTKM_CONT cont::DataSet FieldMagnitude::DoExecute(
14   const vtkm::cont::DataSet& inDataSet,
```

```
15    const ArrayHandleType& inField,
16    const vtkm::filter::FieldMetadata& fieldMetadata,
17    vtkm::filter::PolicyBase<Policy>)
18  {
19    VTKM_IS_ARRAY_HANDLE(ArrayHandleType);
20
21    using ComponentType =
22      typename vtkm::VecTraits<typename ArrayHandleType::ValueType>::ComponentType;
23
24    vtkm::cont::ArrayHandle<ComponentType> outField =
25      vtkm::worklet::Magnitude::Run(inField);
26
27    std::string outFieldName = this->GetOutputFieldName();
28    if (outFieldName == "")
29    {
30      outFieldName = fieldMetadata.GetName() + "_magnitude";
31    }
32
33    return vtkm::filter::internal::CreateResult(inDataSet,
34                                                outField,
35                                                outFieldName,
36                                                fieldMetadata.GetAssociation(),
37                                                fieldMetadata.GetCellSetName());
38  }
39
40  } // namespace filter
41  } // namespace vtkm
```

18.2 Field Filters Using Cell Connectivity

A special subset of field filters are those that take into account the connectivity of a cell set to compute derivative fields. These types of field filters should be implemented in classes that derive the `vtkm::filter::FilterCell` base class. `FilterCell` is itself a subclass of `vtkm::filter::FilterField` and behaves essentially the same. `FilterCell` adds the pair of methods `SetActiveCellSetIndex` and `GetActiveCellSetIndex`. The `SetActiveCellSetIndex` method allows users to specify which cell set of the given `DataSet` to use. Likewise, `FilterCell` subclasses should use `GetActiveCellSetIndex` when retrieving a cell set from the given `DataSet`.

Like `FilterField`, `FilterCell` is a templated class that takes as its single template argument the type of the derived class. Also like `FilterField`, a `FilterCell` subclass must implement a method named `DoExecute` with 4 arguments: an input `vtkm::cont::DataSet` object, a `vtkm::cont::ArrayHandle` of the input field, a `vtkm::filter::FieldMetadata` information object, and a policy class.

In this section we provide an example implementation of a field filter on cells that wraps the "cell center" worklet provided in Example 13.8 (listed on page 165). By convention, filter 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 CellCenter.h.

Example 18.3: Header declaration for a field filter using cell topology.

```
1  namespace vtkm
2  {
3  namespace filter
4  {
5
6  class CellCenters : public vtkm::filter::FilterCell<CellCenters>
7  {
8  public:
9    VTKM_CONT
10   CellCenters();
```

```
11
12    template<typename ArrayHandleType, typename Policy>
13    VTKM_CONT vtkm::cont::DataSet DoExecute(
14      const vtkm::cont::DataSet& inDataSet,
15      const ArrayHandleType& inField,
16      const vtkm::filter::FieldMetadata& FieldMetadata,
17      vtkm::filter::PolicyBase<Policy>);
18  };
19
20  } // namespace filter
21  } // namespace vtkm
```

Did you know?

You may have noticed that Example 18.1 provided a specialization for `vtkm::filter::FilterTraits` *but Example 18.3 provides no such specialization. This demonstrates that declaring filter traits is optional. If a filter only works on some limited number of types, then it can use* `FilterTraits` *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 traits.*

Once the filter class and (optionally) the `FilterTraits` are declared in the .h file, the implementation filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named CellCenter.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

Like with a `FilterField`, a subclass of `FilterCell` implements `DoExecute` by invoking a worklet to compute a new field array and then return a newly constructed `vtkm::cont::DataSet` object.

Example 18.4: Implementation of a field filter using cell topology.

```
1   namespace vtkm
2   {
3   namespace filter
4   {
5
6   VTKM_CONT
7   CellCenters::CellCenters()
8   {
9     this->SetOutputFieldName("");
10  }
11
12  template<typename ArrayHandleType, typename Policy>
13  VTKM_CONT cont::DataSet CellCenters::DoExecute(
14    const vtkm::cont::DataSet& inDataSet,
15    const ArrayHandleType& inField,
16    const vtkm::filter::FieldMetadata& fieldMetadata,
17    vtkm::filter::PolicyBase<Policy>)
18  {
19    VTKM_IS_ARRAY_HANDLE(ArrayHandleType);
20
21    if (!fieldMetadata.IsPointField())
22    {
23      throw vtkm::cont::ErrorBadType("Cell Centers filter operates on point data.");
24    }
25
26    vtkm::cont::DynamicCellSet cellSet =
27      inDataSet.GetCellSet(this->GetActiveCellSetIndex());
```

```
28
29    using ValueType = typename ArrayHandleType::ValueType;
30
31    vtkm::cont::ArrayHandle<ValueType> outField = vtkm::worklet::CellCenter::Run(
32      vtkm::filter::ApplyPolicy(cellSet, Policy()), inField);
33
34    std::string outFieldName = this->GetOutputFieldName();
35    if (outFieldName == "")
36    {
37      outFieldName = fieldMetadata.GetName() + "_center";
38    }
39
40    return vtkm::filter::internal::CreateResult(
41      inDataSet,
42      outField,
43      outFieldName,
44      vtkm::cont::Field::Association::CELL_SET,
45      cellSet.GetName());
46  }
47
48  } // namespace filter
49  } // namespace vtkm
```

Common Errors

The policy passed to the `DoExecute` *method contains information on what types of cell sets should be supported by the execution. This list of cell set types could be different than the default types specified the* `DynamicCellSet` *returned from* `DataSet::GetCellSet`*. Thus, it is important to apply the policy to the cell set before passing it to the dispatcher's invoke method. The policy is applied by calling the* `vtkm::filter::ApplyPolicy` *function on the* `DynamicCellSet`*. The use of* `ApplyPolicy` *is demonstrated in Example 18.4.*

18.3 Data Set Filters

As described in Section 4.2 (starting on page 28), data set filters are a category of filters that 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.

Data set filters are implemented in classes that derive the `vtkm::filter::FilterDataSet` base class. `FilterDataSet` is a templated class that has a single template argument, which is the type of the concrete subclass.

All `FilterDataSet` subclasses must implement two methods: `DoExecute` and `DoMapField`. The `FilterDataSet` base class implements `Execute` and `MapFieldOntoOutput` methods that process the arguments and then call the `DoExecute` and `DoMapField` methods, respectively, of its subclass.

The `DoExecute` method has the following 2 arguments.

- An input data set contained in a `vtkm::cont::DataSet` object. (See Chapter 12 for details on `DataSet` objects.)
- A policy class. The type of the policy is generally unknown until the class is used and requires a template type.

The `DoMapField` method has the following 4 arguments.

- A `vtkm::cont::DataSet` object that was returned from the last call to `DoExecute`. The method both needs information in the `DataSet` object and writes its results back to it, so the parameter should be declared as a non-const reference. (See Chapter 12 for details on `DataSet` objects.)
- The field from the `DataSet` specified in the `Execute` method to operate on. The field is always passed as an instance of `vtkm::cont::ArrayHandle`. (See Chapter 7 for details on `ArrayHandle` objects.) The type of the `ArrayHandle` is generally not known until the class is used and requires a template type.
- A `vtkm::filter::FieldMetadata` object that contains the associated metadata of the field not contained in the `ArrayHandle` of the second argument. The `FieldMetadata` contains information like the name of the field and what topological element the field is associated with (such as points or cells).
- A policy class. The type of the policy is generally unknown until the class is used and requires a template type.

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 17. For the sake of argument, we are assuming that all the worklets required to implement edge extraction are wrapped up in structure named `vtkm::worklet ::ExtractEdges`. Furthermore, we assume that `ExtractEdges` has a pair of methods, `Run` and `ProcessCellField`, that create a cell set of lines from the edges in another cell set and average a cell field from input to output, respectively. The `ExtractEdges` may hold state. All of these assumptions are consistent with the examples in Chapter 17.

By convention, filter 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 ExtractEdges.h.

Example 18.5: Header declaration for a data set filter.

```
template<typename Policy>
inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
  const vtkm::cont::DataSet& inData,
  vtkm::filter::PolicyBase<Policy> policy)
{

  const vtkm::cont::DynamicCellSet& inCells =
    inData.GetCellSet(this->GetActiveCellSetIndex());

  vtkm::cont::CellSetSingleType<> outCells =
    this->Worklet.Run(vtkm::filter::ApplyPolicy(inCells, policy));

  vtkm::cont::DataSet outData;

  outData.AddCellSet(outCells);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

Once the filter class and (optionally) the `FilterTraits` are declared in the .h file, the implementation of the filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be

expected in a file named ExtractEdges.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of `DoExecute` first calls the worklet methods to generate a new `CellSet` class. It then constructs a `DataSet` containing this `CellSet`. It also has to pass all the coordinate systems to the new `DataSet`. Finally, it returns the `DataSet`.

Example 18.6: Implementation of the `DoExecute` method of a data set filter.

```
template<typename Policy>
inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
  const vtkm::cont::DataSet& inData,
  vtkm::filter::PolicyBase<Policy> policy)
{

  const vtkm::cont::DynamicCellSet& inCells =
    inData.GetCellSet(this->GetActiveCellSetIndex());

  vtkm::cont::CellSetSingleType<> outCells =
    this->Worklet.Run(vtkm::filter::ApplyPolicy(inCells, policy));

  vtkm::cont::DataSet outData;

  outData.AddCellSet(outCells);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

The implementation of `DoMapField` checks to see what elements the given field is associated with (e.g. points or cells), processes the field data as necessary, and adds the field to the `DataSet`. The `vtkm::filter::FieldMetadata` passed to `DoMapField` provides some features to make this process easier. `FieldMetadata` contains methods to query what the field is associated with such as `IsPointField` and `IsCellField`. `FieldMetadata` also has a method named `AsField` that creates a new `vtkm::cont::Field` object with a new data array and metadata that matches the input.

Example 18.7: Implementation of the `DoMapField` method of a data set filter.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool ExtractEdges::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& input,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field output;

  if (fieldMeta.IsPointField())
  {
    output = fieldMeta.AsField(input); // pass through
  }
  else if (fieldMeta.IsCellField())
  {
    output = fieldMeta.AsField(this->Worklet.ProcessCellField(input));
  }
  else
```

```
19    {
20      return false;
21    }
22
23    result.AddField(output);
24
25    return true;
26  }
```

18.4 Data Set with Field Filters

As described in Section 4.3 (starting on page 31), data set with field filters are a category of filters that 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.

Data set with field filters are implemented in classes that derive the `vtkm::filter::FilterDataSetWithField` base class. `FilterDataSetWithField` is a templated class that has a single template argument, which is the type of the concrete subclass.

All `FilterDataSetWithField` subclasses must implement two methods: `DoExecute` and `DoMapField`. The `FilterDataSetWithField` base class implements `Execute` and `MapFieldOntoOutput` methods that process the arguments and then call the `DoExecute` and `DoMapField` methods, respectively, of its subclass.

The `DoExecute` method has the following 4 arguments. (They are the same arguments for `DoExecute` of field filters as described in Section 18.1.)

- An input data set contained in a `vtkm::cont::DataSet` object. (See Chapter 12 for details on `DataSet` objects.)
- The field from the `DataSet` specified in the `Execute` method to operate on. The field is always passed as an instance of `vtkm::cont::ArrayHandle`. (See Chapter 7 for details on `ArrayHandle` objects.) The type of the `ArrayHandle` is generally not known until the class is used and requires a template type.
- A `vtkm::filter::FieldMetadata` object that contains the associated metadata of the field not contained in the `ArrayHandle` of the second argument. The `FieldMetadata` contains information like the name of the field and what topological element the field is associated with (such as points or cells).
- A policy class. The type of the policy is generally unknown until the class is used and requires a template type.

The `DoMapField` method has the following 4 arguments. (They are the same arguments for `DoExecute` of field filters as described in Section 18.3.)

- A `vtkm::cont::DataSet` object that was returned from the last call to `DoExecute`. The method both needs information in the `DataSet` object and writes its results back to it, so the parameter should be declared as a non-const reference. (See Chapter 12 for details on `DataSet` objects.)
- The field from the `DataSet` specified in the `Execute` method to operate on. The field is always passed as an instance of `vtkm::cont::ArrayHandle`. (See Chapter 7 for details on `ArrayHandle` objects.) The type of the `ArrayHandle` is generally not known until the class is used and requires a template type.
- A `vtkm::filter::FieldMetadata` object that contains the associated metadata of the field not contained in the `ArrayHandle` of the second argument. The `FieldMetadata` contains information like the name of the field and what topological element the field is associated with (such as points or cells).

- A policy class. The type of the policy is generally unknown until the class is used and requires a template type.

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.

We leverage the worklets in the VTK-m source code that implement the `Threshold` functionality. The threshold worklets are templated on a predicate class that, given a field value, returns a flag on whether to pass or cull the given cell. The `vtkm::filter::Threshold` class uses a predicate that is true for field values within a range. Our blank cells filter will instead use the predicate class `vtkm::NotZeroInitialized` that will cull all values where the mask is zero.

By convention, filter 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 BlankCells.h.

Example 18.8: Header declaration for a data set with field filter.

```
namespace vtkm
{
namespace filter
{

class BlankCells : public vtkm::filter::FilterDataSetWithField<BlankCells>
{
public:
  template<typename ArrayHandleType, typename Policy>
  VTKM_CONT vtkm::cont::DataSet DoExecute(
    const vtkm::cont::DataSet& inDataSet,
    const ArrayHandleType& inField,
    const vtkm::filter::FieldMetadata& fieldMetadata,
    vtkm::filter::PolicyBase<Policy>);

  template<typename T, typename StorageType, typename Policy>
  VTKM_CONT bool DoMapField(vtkm::cont::DataSet& result,
                            const vtkm::cont::ArrayHandle<T, StorageType>& input,
                            const vtkm::filter::FieldMetadata& fieldMeta,
                            const vtkm::filter::PolicyBase<Policy>& policy);

private:
  vtkm::worklet::Threshold Worklet;
};


template<>
struct FilterTraits<vtkm::filter::BlankCells>
{
  struct InputFieldTypeList : vtkm::TypeListTagScalarAll
  {
  };
};

} // namespace filter
} // namespace vtkm
```

Note that the filter class declaration is accompanied by a specialization of `FilterTraits` to specify the field is meant to operate specifically on scalar types. Once the filter class and (optionally) the `FilterTraits` are declared in the .h file, the implementation of the filter is by convention given in a separate .hxx file. So the continuation of our example that follows would be expected in a file named BlankCells.hxx. The .h file near its bottom needs an include line to the .hxx file. This convention is set up because a near future version of VTK-m

will allow the building of filter libraries containing default policies that can be used by only including the header declaration.

The implementation of `DoExecute` first calls the worklet methods to generate a new `CellSet` class. It then constructs a `DataSet` containing this `CellSet`. It also has to pass all the coordinate systems to the new `DataSet`. Finally, it returns the `DataSet`.

Example 18.9: Implementation of the `DoExecute` method of a data set with field filter.

```
template<typename ArrayHandleType, typename Policy>
VTKM_CONT vtkm::cont::DataSet BlankCells::DoExecute(
  const vtkm::cont::DataSet& inData,
  const ArrayHandleType& inField,
  const vtkm::filter::FieldMetadata& fieldMetadata,
  vtkm::filter::PolicyBase<Policy>)
{
  VTKM_IS_ARRAY_HANDLE(ArrayHandleType);

  if (!fieldMetadata.IsCellField())
  {
    throw vtkm::cont::ErrorBadValue("Blanking field must be a cell field.");
  }

  const vtkm::cont::DynamicCellSet& inCells =
    inData.GetCellSet(this->GetActiveCellSetIndex());

  vtkm::cont::DynamicCellSet outCells =
    this->Worklet.Run(vtkm::filter::ApplyPolicy(inCells, Policy()),
                      inField,
                      fieldMetadata.GetAssociation(),
                      vtkm::NotZeroInitialized());

  vtkm::cont::DataSet outData;

  outData.AddCellSet(outCells);

  for (vtkm::IdComponent coordSystemIndex = 0;
       coordSystemIndex < inData.GetNumberOfCoordinateSystems();
       ++coordSystemIndex)
  {
    outData.AddCoordinateSystem(inData.GetCoordinateSystem(coordSystemIndex));
  }

  return outData;
}
```

The implementation of `DoMapField` checks to see what elements the given field is associated with (e.g. points or cells), processes the field data as necessary, and adds the field to the `DataSet`. The `vtkm::filter::FieldMetadata` passed to `DoMapField` provides some features to make this process easier. `FieldMetadata` contains methods to query what the field is associated with such as `IsPointField` and `IsCellField`. `FieldMetadata` also has a method named `AsField` that creates a new `vtkm::cont::Field` object with a new data array and metadata that matches the input.

Example 18.10: Implementation of the `DoMapField` method of a data set with field filter.

```
template<typename T, typename StorageType, typename Policy>
inline VTKM_CONT bool BlankCells::DoMapField(
  vtkm::cont::DataSet& result,
  const vtkm::cont::ArrayHandle<T, StorageType>& input,
  const vtkm::filter::FieldMetadata& fieldMeta,
  const vtkm::filter::PolicyBase<Policy>&)
{
  vtkm::cont::Field output;
```

```
9
10    if (fieldMeta.IsPointField())
11    {
12      output = fieldMeta.AsField(input); // pass through
13    }
14    else if (fieldMeta.IsCellField())
15    {
16      output = fieldMeta.AsField(this->Worklet.ProcessCellField(input));
17    }
18    else
19    {
20      return false;
21    }
22
23    result.AddField(output);
24
25    return true;
26  }
```

CHAPTER

NINETEEN

TRY EXECUTE

Throughout this chapter and elsewhere in this book we have seen examples that require specifying the device on which to run using a device adapter tag. This is an important aspect when writing portable parallel algorithms. However, it is often the case that users of these algorithms 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 4, you do not need to specify a device; they choose a device for you.

Internally, the filter superclasses have 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 18. Most of the outward facing interfaces of parallel algorithms in VTK-m are through these filter classes. For everything else, there is the `vtkm::cont::TryExecute` function.

`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 with two arguments. 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. The second argument is a `vtkm::cont::RuntimeDeviceTracker` that specifies what devices to try. `RuntimeDeviceTracker` is documented in Section 8.3.

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 19.1: 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::DeviceAdapterAlgorithm<Device>::Reduce(array, T(0));
5   return sum / T(array.GetNumberOfValues());
6 }
```

The function in Example 19.1 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 19.1, 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 19.2: Using `TryExecute`.

```
1 namespace detail
2 {
3
```

```
4  template<typename T, typename Storage>
5  struct ArrayAverageFunctor
6  {
7    using InArrayType = vtkm::cont::ArrayHandle<T, Storage>;
8
9    InArrayType InArray;
10   T OutValue;
11
12   VTKM_CONT
13   ArrayAverageFunctor(const InArrayType& array)
14     : InArray(array)
15   {
16   }
17
18   template<typename Device>
19   VTKM_CONT bool operator()(Device)
20   {
21     // Call the version of ArrayAverage that takes a DeviceAdapter.
22     this->OutValue = ArrayAverage(this->InArray, Device());
23
24     return true;
25   }
26 };
27
28 } // namespace detail
29
30 template<typename T, typename Storage>
31 VTKM_CONT T ArrayAverage(const vtkm::cont::ArrayHandle<T, Storage>& array,
32                          vtkm::cont::RuntimeDeviceTracker tracker =
33                            vtkm::cont::GetGlobalRuntimeDeviceTracker())
34 {
35   detail::ArrayAverageFunctor<T, Storage> functor(array);
36
37   bool foundAverage = vtkm::cont::TryExecute(functor, tracker);
38
39   if (!foundAverage)
40   {
41     throw vtkm::cont::ErrorExecution("Could not compute array average.");
42   }
43
44   return functor.OutValue;
45 }
```

Did you know?

By convention, functions that use `TryExecute` *to try multiple devices should have an argument that accepts a* `vtkm::cont::RuntimeDeviceTracker`*. This argument should be optional, with the default value set to what the* `vtkm::cont::GetGlobalRuntimeDeviceTracker` *function returns.*

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.*

Part IV

Advanced Development

CHAPTER

TWENTY

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 class to establish virtual objects 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 5 header files with the names DeviceAdapterTag∗.h, DeviceAdapterRuntimeDetector∗.h, ArrayManagerExecution∗.h, VirtualObjectTransfer∗.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 4 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 20.1: 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/ArrayManagerExecutionCxx11Thread.h>
4 #include <vtkm/cont/cxx11/internal/VirtualObjectTransferCxx11Thread.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.

20.1 Tag

The device adapter tag, as described in Section 8.1 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 20.2: Implementation of a device adapter tag.

```
#include <vtkm/cont/internal/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 7

VTKM_VALID_DEVICE_ADAPTER(Cxx11Thread, VTKM_DEVICE_ADAPTER_CXX11_THREAD);
```

20.2 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 20.3: Prototype for DeviceAdapterRuntimeDetector.

```
namespace vtkm
{
namespace cont
{

template<typename DeviceAdapterTag>
class DeviceAdapterRuntimeDetector;
}
} // namespace vtkm
```

All device adapter implementations must create a specialization of DeviceAdapterRuntimeDetector. They must contain a method named DeviceAdapterRuntimeDetectorExists 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 20.4: Implementation of DeviceAdapterRuntimeDetector specialization

```
namespace vtkm
{
namespace cont
{

template<>
class DeviceAdapterRuntimeDetector<vtkm::cont::DeviceAdapterTagCxx11Thread>
{
public:
  VTKM_CONT bool Exists() const
  {
    return vtkm::cont::DeviceAdapterTagCxx11Thread::IsEnabled;
  }
```

```
14 };
15
16 } // namespace cont
17 } // namespace vtkm
```

20.3 Array Manager Execution

VTK-m defines a template named `vtkm::cont::internal::ArrayManagerExecution` that is responsible for allocating memory in the execution environment and copying data between the control and execution environment. `ArrayManagerExecution` is also paired with two helper classes, `vtkm::cont::internal::ExecutionPortalFactoryBasic` and `vtkm::cont::internal::ExecutionArrayInterfaceBasic`, which provide operations for creating and operating on and manipulating data in standard C arrays. All 3 class specializations are typically defined in an internal header file with a prefix of ArrayManagerExecution. The following subsections describe each of these classes.

20.3.1 `ArrayManagerExecution`

Example 20.5: Prototype for `vtkm::cont::internal::ArrayManagerExecution`.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename T, typename StorageTag, typename DeviceAdapterTag>
9  class ArrayManagerExecution;
10 }
11 } // namespace cont
12 } // namespace vtkm
```

A device adapter must provide a partial specialization of `vtkm::cont::internal::ArrayManagerExecution` for its device adapter tag. The implementation for `ArrayManagerExecution` is expected to manage the resources for a single array. All `ArrayManagerExecution` specializations must have a constructor that takes a pointer to a `vtkm::cont::internal::Storage` object. The `ArrayManagerExecution` should store a reference to this `Storage` object and use it to pass data between control and execution environments. Additionally, `ArrayManagerExecution` must provide the following elements.

`ValueType` The type for each item in the array. This is the same type as the first template argument.

`PortalType` The type of an array portal that can be used in the execution environment to access the array.

`PortalConstType` A read-only (const) version of `PortalType`.

`GetNumberOfValues` A method that returns the number of values stored in the array. The results are undefined if the data has not been loaded or allocated.

`PrepareForInput` A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a `bool` flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a `PortalConstType` that points to the data.

`PrepareForInPlace` A method that ensures an array is allocated in the execution environment and valid data is there. The method takes a `bool` flag that specifies whether data needs to be copied to the execution environment. (If false, then data for this array has not changed since the last operation.) The method returns a `PortalType` that points to the data.

`PrepareForOutput` A method that takes an array size and allocates an array in the execution environment of the specified size. The initial memory may be uninitialized. The method returns a `PortalType` to the data.

`RetrieveOutputData` This method takes a storage object, allocates memory in the control environment, and copies data from the execution environment into it. If the control and execution environments share arrays, then this can be a no-operation.

`CopyInto` This method takes an STL-compatible iterator and copies data from the execution environment into it.

`Shrink` A method that adjusts the size of the array in the execution environment to something that is a smaller size. All the data up to the new length must remain valid. Typically, no memory is actually reallocated. Instead, a different end is marked.

`ReleaseResources` A method that frees any resources (typically memory) in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying memory in methods such as `PrepareForInput` and `RetrieveOutputData`. This might require creating buffers in the control environment to efficiently move data from control array portals.

However, if the control and execution environments share the same memory space, the execution array manager can, and should, delegate all of its operations to the `Storage` it is constructed with. VTK-m comes with a class called `vtkm::cont::internal::ArrayManagerExecutionShareWithControl` that provides the implementation for an execution array manager that shares a memory space with the control environment. In this case, making the `ArrayManagerExecution` specialization be a trivial subclass is sufficient.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `ArrayManagerExecution`, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 20.6: Specialization of `ArrayManagerExecution`.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/internal/ArrayManagerExecution.h>
4  #include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>
5
6  namespace vtkm
7  {
8  namespace cont
9  {
10 namespace internal
11 {
12
13 template<typename T, typename StorageTag>
14 class ArrayManagerExecution<T, StorageTag, vtkm::cont::DeviceAdapterTagCxx11Thread>
15   : public vtkm::cont::internal::ArrayManagerExecutionShareWithControl<T, StorageTag>
16 {
17   using Superclass =
18     vtkm::cont::internal::ArrayManagerExecutionShareWithControl<T, StorageTag>;
19
20 public:
```

```
21    VTKM_CONT
22    ArrayManagerExecution(typename Superclass::StorageType* storage)
23      : Superclass(storage)
24    {
25    }
26  };
27
28  } // namespace internal
29  } // namespace cont
30  } // namespace vtkm
```

20.3.2 ExecutionPortalFactoryBasic

Example 20.7: Prototype for vtkm::cont::internal::ExecutionPortalFactoryBasic.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename T, typename DeviceAdapterTag>
9  struct ExecutionPortalFactoryBasic;
10 }
11 } // namespace cont
12 } // namespace vtkm
```

A device adapter must provide a partial specialization of vtkm::cont::internal::ExecutionPortalFactoryBasic for its device adapter tag. The implementation for ExecutionPortalFactoryBasic is capable of taking pointers to an array in the execution environment and returning an array portal to the data in that array. ExecutionPortalFactoryBasic has no state and all of its methods are static. ExecutionPortalFactoryBasic provides the following elements.

`ValueType` The type for each item in the array. This is the same type as the first template argument.

`PortalType` The type for the read/write portals created by the class.

`PortalConstType` The type for read-only portals created by the class.

`CreatePortal` A static method that takes two pointers of type `ValueType*` that point to the beginning (first element) and end (one past the last element) of the array to create a portal for the array. Returns a portal of type `PortalType` that works in the execution environment.

`CreatePortalConst` A static method that takes two pointers of type `const ValueType*` that point to the beginning (first element) and end (one past the last element) of the array to create a portal for the array. Returns a portal of type `PortalConstType` that works in the execution environment.

Specializations of this template typically take on one of two forms. If the control and execution environments share the same memory space, then the execution array manager can, and should, return a simple vtkm::cont::internal::ArrayPortalFromIterators. VTK-m comes with a class called vtkm::cont::internal::ExecutionPortalFactoryBasicShareWithControl that provides the implementation for a basic execution portal factory that shares a memory space with the control environment. In this case, making the ExecutionPortalFactoryBasic specialization be a trivial subclass is sufficient.

However, if the control and execution environments have separate memory spaces, then the typical vtkm::cont::internal::ArrayPortalFromIterators class might not work in the execution environment. In this case a custom array portal class will have to be implemented and created within the ExecutionPortalFactoryBasic.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `ExecutionPortalFactoryBasic`, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 20.8: Specialization of `ExecutionPortalFactoryBasic`.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>

namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename T>
struct ExecutionPortalFactoryBasic<T, vtkm::cont::DeviceAdapterTagCxx11Thread>
  : public vtkm::cont::internal::ExecutionPortalFactoryBasicShareWithControl<T>
{
};

} // namespace internal
} // namespace cont
} // namespace vtkm
```

20.3.3 ExecutionArrayInterfaceBasic

Example 20.9: Prototype for `vtkm::cont::internal::ExecutionArrayInterfaceBasic`.

```
namespace vtkm
{
namespace cont
{
namespace internal
{

template<typename DeviceAdapterTag>
struct ExecutionArrayInterfaceBasic;
}
} // namespace cont
} // namespace vtkm
```

A device adapter must provide a partial specialization of `vtkm::cont::internal::ExecutionArrayInterfaceBasic` for its device adapter tag. The implementation for `ExecutionArrayInterfaceBasic` is expected to allow allocation of basic C arrays in the execution environment and to copy data between control and execution environments. All implementations of `ExecutionArrayInterfaceBasic` are expected to inherit from `vtkm::cont::internal::ExecutionArrayInterfaceBasicBase` and implement the pure virtual methods therein. All implementations are also expected to have a constructor that takes a reference to a `vtkm::cont::internal::StorageBasicBase`, which should subsequently be passed to the `ExecutionArrayInterfaceBasicBase`. The methods that `vtkm::cont::internal::ExecutionArrayInterfaceBasic` must override are the following.

`GetDeviceId` Returns a `vtkm::cont::DeviceAdapterId` integer representing a unique identifier for the device associated with this implementation. This number should be the same as the `VTKM_DEVICE_ADAPTER_` macro described in Section 20.1.

`Allocate` Takes a reference to a `vtkm::cont::internal::TypelessExecutionArray`, which holds a collection of array pointer references, and a size of allocation in bytes. The method should allocate an array of the

given size in the execution environment and return the resulting pointers in the given TypelessExecutionArray reference.

`Free` Takes a reference to a vtkm::cont::internal::TypelessExecutionArray created in a previous call to `Allocate` and frees the memory. The array references in the TypelessExecutionArray should be set to `nullptr`.

`CopyFromControl` Takes a `const void*` pointer for an array in the control environment, a `void*` for an array in the execution environment, and a size of the arrays in bytes. The method copies the data from the control array to the execution array.

`CopyToControl` Takes a `const void*` pointer for an array in the execution environment, a `void*` for an array in the control environment, and a size of the arrays in bytes. The method copies the data from the execution array to the control array.

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by allocating arrays on the device and copying data between the main CPU and the device.

However, if the control and execution environments share the same memory space, then the execution array interface can, and should, use the storage on the control environment to allocate arrays and do simple copies (shallow where possible). VTK-m comes with a class called vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl that provides the implementation for a basic execution array interface that shares a memory space with the control environment. In this case, it is best to make the ExecutionArrayInterfaceBasic specialization a subclass of ExecutionArrayInterfaceBasicShareWithControl. Note that in this case you still need to define a constructor that takes a reference to vtkm::cont::internal::StorageBasicBase (which is passed straight to the superclass) and an implementation of the `GetDeviceId` method.

Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of ExecutionArrayInterfaceBasic, which by convention would be placed in the vtkm/cont/cxx11/internal/ArrayManagerExecutionCxx11Thread.h header file.

Example 20.10: Specialization of ExecutionArrayInterfaceBasic.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/internal/ArrayManagerExecutionShareWithControl.h>
4
5  namespace vtkm
6  {
7  namespace cont
8  {
9  namespace internal
10 {
11
12 template<>
13 struct ExecutionArrayInterfaceBasic<vtkm::cont::DeviceAdapterTagCxx11Thread>
14   : public vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl
15 {
16 public:
17   using Superclass =
18     vtkm::cont::internal::ExecutionArrayInterfaceBasicShareWithControl;
19
20   VTKM_CONT
21   ExecutionArrayInterfaceBasic(vtkm::cont::internal::StorageBasicBase& storage)
22     : Superclass(storage)
23   {
24   }
25
```

```
26    VTKM_CONT
27    virtual vtkm::cont::DeviceAdapterId GetDeviceId() const final override
28    {
29      return vtkm::cont::DeviceAdapterTagCxx11Thread();
30    }
31  };
32
33  } // namespace internal
34  } // namespace cont
35  } // namespace vtkm
```

20.4 Virtual Object Transfer

VTK-m defines a template named `vtkm::cont::internal::VirtualObjectTransfer` that is responsible for instantiating virtual objects in the execution environment. The `VirtualObjectTransfer` class is the internal mechanism that allocates space for the object and sets up the virtual method tables for them. This class has two template parameters. The first parameter is the concrete derived type of the virtual object to be transferred to the execution environment. It is assumed that after the object is copied to the execution environment, a pointer to a base superclass of this concrete derived type will be used. The second template argument is the device adapter on which to put the object.

Example 20.11: Prototype for `vtkm::cont::internal::VirtualObjectTransfer`.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace internal
6  {
7
8  template<typename VirtualDerivedType, typename DeviceAdapter>
9  struct VirtualObjectTransfer;
10 }
11 } // namespace cont
12 } // namespace vtkm
```

A device adapter must provide a partial specialization of `VirtualObjectTransfer` for its device adapter tag. This partial specialization is typically defined in an internal header file with a prefix of VirtualObjectTransfer. The implementation for `VirtualObjectTransfer` can establish a virtual object in the execution environment based on an object in the control environment, update the state of said object, and release all the resources for the object. `VirtualObjectTransfer` must provide the following methods.

`VirtualObjectTransfer` (constructor) A `VirtualObjectTransfer` has a constructor that takes a pointer to the derived type that (eventually) gets transferred to the execution environment of the given device adapter. The object provide must stay valid for the lifespan of the `VirtualObjectTransfer` object.

`PrepareForExecution` Transfers the virtual object (given in the constructor) to the execution environment and returns a pointer to the object that can be used in the execution environment. The returned object may not be valid in the control environment and should not be used there. `PrepareForExecution` takes a single `bool` argument. If the argument is false and `PrepareForExecution` was called previously, then the method can return the same data as the last call without any updates. If the argument is true, then the data in the execution environment is always updated regardless of whether data was copied in a previous call to `PrepareForExecution`. This argument is used to tell the `VirtualObjectTransfer` whether the object in the control environment has changed and has to be updated in the execution environment.

`ReleaseResources` Frees up any resources in the execution environment. Any previously returned virtual object from `PrepareForExecution` becomes invalid. (The destructor for `VirtualObjectTransfer` should also release the resources.)

Specializations of this template typically take on one of two forms. If the control and execution environments have separate memory spaces, then this class behaves by copying the concrete control object to the execution environment (where the virtual table will be invalid) and a new object is created in the execution environment by copying the object from the control environment. It can be assumed that the object can be trivially copied (with the exception of the virtual method table).

Did you know?

For some devices, like CUDA, it is either only possible or more efficient to allocate data from the host (the control environment). To avoid having to allocate data from the device (the execution environment), implement `PrepareForExecution` *by first allocating data from the host and then running code on the device that does a "placement new" to create and copy the object in the pre-allocated space.*

However, if the control and execution environments share the same memory space, the virtual object transfer can, and should, just bind directly with the target concrete object. VTK-m comes with a class called `vtkm::cont::internal::VirtualObjectTransferShareWithControl` that provides the implementation for a virtual object transfer that shares a memory space with the control environment. In this case, making the `VirtualObjectTransfer` specialization be a trivial subclass is sufficient. Continuing our example of a device adapter based on C++11's `std::thread` class, here is the implementation of `VirtualObjectTransfer`, which by convention would be placed in the vtkm/cont/cxx11/internal/VirtualObjectTransferCxx11Thread.h header file.

Example 20.12: Specialization of `VirtualObjectTransfer`.

```
1  #include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>
2
3  #include <vtkm/cont/internal/VirtualObjectTransfer.h>
4  #include <vtkm/cont/internal/VirtualObjectTransferShareWithControl.h>
5
6  namespace vtkm
7  {
8  namespace cont
9  {
10 namespace internal
11 {
12
13 template<typename VirtualDerivedType>
14 struct VirtualObjectTransfer<VirtualDerivedType,
15                              vtkm::cont::DeviceAdapterTagCxx11Thread>
16   : VirtualObjectTransferShareWithControl<VirtualDerivedType>
17 {
18   VTKM_CONT VirtualObjectTransfer(const VirtualDerivedType* virtualObject)
19     : VirtualObjectTransferShareWithControl<VirtualDerivedType>(virtualObject)
20   {
21   }
22 };
23
24 } // namespace internal
25 } // namespace cont
26 } // namespace vtkm
```

20.5 Algorithms

A device adapter implementation must also provide a specialization of `vtkm::cont::DeviceAdapterAlgorithm`, which is documented in Section 8.4. 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 Section 13.11, 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 20.13: Minimal specialization of DeviceAdapterAlgorithm.

```
#include <vtkm/cont/cxx11/internal/DeviceAdapterTagCxx11Thread.h>

#include <vtkm/cont/DeviceAdapterAlgorithm.h>
#include <vtkm/cont/ErrorExecution.h>
#include <vtkm/cont/internal/DeviceAdapterAlgorithmGeneral.h>

#include <thread>

namespace vtkm
{
namespace cont
{

template<>
struct DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>
  : vtkm::cont::internal::DeviceAdapterAlgorithmGeneral<
      DeviceAdapterAlgorithm<vtkm::cont::DeviceAdapterTagCxx11Thread>,
      vtkm::cont::DeviceAdapterTagCxx11Thread>
{
private:
  template<typename FunctorType>
  struct ScheduleKernel1D
  {
    VTKM_CONT
    ScheduleKernel1D(const FunctorType& functor)
      : Functor(functor)
    {
    }

    VTKM_EXEC
    void operator()() const
    {
      try
      {
        for (vtkm::Id threadId = this->BeginId; threadId < this->EndId; threadId++)
        {
          this->Functor(threadId);
          // If an error is raised, abort execution.
          if (this->ErrorMessage.IsErrorRaised())
          {
            return;
          }
        }
      }
      catch (vtkm::cont::Error error)
      {
        this->ErrorMessage.RaiseError(error.GetMessage().c_str());
      }
      catch (std::exception error)
      {
        this->ErrorMessage.RaiseError(error.what());
      }
      catch (...)
      {
        this->ErrorMessage.RaiseError("Unknown exception raised.");
      }
    }
```

```
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 (vtkm::cont::Error error)
107       {
108         this->ErrorMessage.RaiseError(error.GetMessage().c_str());
109       }
110       catch (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 = static_cast<vtkm::Id>(std::thread::hardware_concurrency());
143     if (numThreads > numInstances)
144     {
145       numThreads = numInstances;
146     }
147     vtkm::Id numInstancesPerThread = (numInstances + numThreads - 1) / numThreads;
148
149     std::thread* threadPool = new std::thread[numThreads];
150     vtkm::Id beginId = 0;
151     for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
152     {
153       vtkm::Id endId = std::min(beginId + numInstancesPerThread, numInstances);
154       KernelType threadKernel = kernel;
155       threadKernel.BeginId = beginId;
156       threadKernel.EndId = endId;
157       std::thread newThread(threadKernel);
158       threadPool[threadIndex].swap(newThread);
159       beginId = endId;
160     }
161
162     for (vtkm::Id threadIndex = 0; threadIndex < numThreads; threadIndex++)
163     {
164       threadPool[threadIndex].join();
165     }
166
167     delete[] threadPool;
168
169     if (errorMessage.IsErrorRaised())
170     {
171       throw vtkm::cont::ErrorExecution(errorString);
172     }
173   }
174
175 public:
176   template<typename FunctorType>
177   VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id numInstances)
178   {
179     DoSchedule(ScheduleKernel1D<FunctorType>(functor), numInstances);
180   }
181
182   template<typename FunctorType>
183   VTKM_CONT static void Schedule(FunctorType functor, vtkm::Id3 maxRange)
184   {
185     vtkm::Id numInstances = maxRange[0] * maxRange[1] * maxRange[2];
```

```
186       DoSchedule(ScheduleKernel3D<FunctorType>(functor, maxRange), numInstances);
187     }
188
189     VTKM_CONT
190     static void Synchronize()
191     {
192       // Nothing to do. This device schedules all of its operations using a
193       // split/join paradigm. This means that the if the control threaad is
194       // calling this method, then nothing should be running in the execution
195       // environment.
196     }
197   };
198
199   } // namespace cont
200   } // namespace vtkm
```

20.6 Timer Implementation

The VTK-m timer, described in Chapter 9, 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 20.14: 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

TWENTYONE

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 `Invoke` method of the dispatcher is expected to support arguments that match these arguments, and part of the dispatching operation may require these arguments to be augmented before the worklet is scheduled. This leaves dispatchers 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`.

21.1 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 21.1: 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 provies 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 21.2: 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);
```

21.2 Parameters

One 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 21.3: Getting the arity of a FunctionInterface.

```
VTKM_STATIC_ASSERT(vtkm::internal::FunctionInterface<size_t(const char*)>::ARITY ==
                   1);

vtkm::IdComponent arity = strncpyInterface.GetArity(); // arity = 3
```

To get a particular parameter, FunctionInterface has the templated method GetParameter. The template parameter is the index of the parameter. 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.

There are two ways to specify the index for GetParameter. The first is to directly specify the template parameter (e.g. GetParameter<1>()). However, note that in a templated function or method where the type is not fully resolved the compiler will not register GetParameter as a templated method and will fail to parse the template argument without a template keyword. The second way to specify the index is to provide a vtkm::internal::IndexTag object as an argument to GetParameter. Although this syntax is more verbose, it works the same whether the FunctionInterface is fully resolved or not. The following example shows both methods in action.

Example 21.4: Using FunctionInterface::GetParameter().

```
void GetFirstParameterResolved(
  const vtkm::internal::FunctionInterface<void(std::string)>& interface)
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>()) << std::endl;
}

template<typename FunctionSignature>
void GetFirstParameterTemplated(
  const vtkm::internal::FunctionInterface<FunctionSignature>& interface)
{
  // The following two uses of GetParameter are equivalent
  std::cout << interface.template GetParameter<1>() << std::endl;
  std::cout << interface.GetParameter(vtkm::internal::IndexTag<1>()) << std::endl;
}
```

Likewise, there is a SetParmeter method for changing parameters. The same rules for indexing and template specification apply.

Example 21.5: Using FunctionInterface::SetParameter().

```
void SetFirstParameterResolved(
  vtkm::internal::FunctionInterface<void(std::string)>& interface,
  const std::string& newFirstParameter)
{
  // The following two uses of SetParameter are equivalent
  interface.SetParameter<1>(newFirstParameter);
  interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
}

template<typename FunctionSignature, typename T>
void SetFirstParameterTemplated(
  vtkm::internal::FunctionInterface<FunctionSignature>& interface,
  T newFirstParameter)
{
```

```
15   // The following two uses of SetParameter are equivalent
16   interface.template SetParameter<1>(newFirstParameter);
17   interface.SetParameter(newFirstParameter, vtkm::internal::IndexTag<1>());
18 }
```

21.3 Invoking

FunctionInterface can invoke a functor of a matching signature using the parameters stored within. If the functor returns a value, that return value will be stored in the FunctionInterface object for later retrieval. There are several versions of the invoke method. There are always seperate versions of invoke methods for the control and execution environments so that functors for either environment can be executed. The basic version of invoke passes the parameters directly to the function and directly stores the result.

Example 21.6: Invoking a FunctionInterface.

```
1   vtkm::internal::FunctionInterface<size_t(const char*)> strlenInterface;
2   strlenInterface.SetParameter<1>("Hello world");
3
4   strlenInterface.InvokeCont(strlen);
5
6   size_t length = strlenInterface.GetReturnValue(); // length = 11
```

Another form of the invoke methods takes a second transform functor that is applied to each argument before passed to the main function. If the main function returns a value, the transform is applied to that as well before being stored back in the FunctionInterface.

Example 21.7: Invoking a FunctionInterface with a transform.

```
1  // Our transform converts C strings to integers, leaves everything else alone.
2  struct TransformFunctor
3  {
4    template<typename T>
5    VTKM_CONT const T& operator()(const T& x) const
6    {
7      return x;
8    }
9
10   VTKM_CONT
11   vtkm::Int32 operator()(const char* x) const { return atoi(x); }
12 };
13
14 // The function we are invoking simply compares two numbers.
15 struct IsSameFunctor
16 {
17   template<typename T1, typename T2>
18   VTKM_CONT bool operator()(const T1& x, const T2& y) const
19   {
20     return x == y;
21   }
22 };
23
24 void TryTransformedInvoke()
25 {
26   vtkm::internal::FunctionInterface<bool(const char*, vtkm::Int32)>
27     functionInterface = vtkm::internal::make_FunctionInterface<bool>(
28       (const char*)"42", (vtkm::Int32)42);
29
30   functionInterface.InvokeCont(IsSameFunctor(), TransformFunctor());
31
32   bool isSame = functionInterface.GetReturnValue(); // isSame = true
33 }
```

As demonstrated in the previous examples, FunctionInterface has a method named GetReturnValue that returns the value from the last invoke. Care should be taken to only use GetReturnValue when the function specification has a return value. If the function signature has a void return type, using GetReturnValue will cause a compile error.

FunctionInterface has an alternate method named GetReturnValueSafe that returns the value wrapped in a templated structure named vtkm::internal::FunctionInterfaceReturnContainer. This structure always has a static constant Boolean named VALID that is false if there is no return type and true otherwise. If the container is valid, it also has an entry named Value containing the result.

Example 21.8: Getting return value from FunctionInterface safely.

```
template<typename ResultType, bool Valid>
struct PrintReturnFunctor;

template<typename ResultType>
struct PrintReturnFunctor<ResultType, true>
{
  VTKM_CONT
  void operator()(
    const vtkm::internal::FunctionInterfaceReturnContainer<ResultType>& x) const
  {
    std::cout << x.Value << std::endl;
  }
};

template<typename ResultType>
struct PrintReturnFunctor<ResultType, false>
{
  VTKM_CONT
  void operator()(
    const vtkm::internal::FunctionInterfaceReturnContainer<ResultType>&) const
  {
    std::cout << "No return type." << std::endl;
  }
};

template<typename FunctionInterfaceType>
void PrintReturn(const FunctionInterfaceType& functionInterface)
{
  using ResultType = typename FunctionInterfaceType::ResultType;
  using ReturnContainerType =
    vtkm::internal::FunctionInterfaceReturnContainer<ResultType>;

  PrintReturnFunctor<ResultType, ReturnContainerType::VALID> printReturn;
  printReturn(functionInterface.GetReturnValueSafe());
}
```

21.4 Modifying Parameters

In addition to storing and querying parameters and invoking functions, FunctionInterface also contains multiple ways to modify the parameters to augment the function calls. This can be used in the same use case as a chain of function calls that generally pass their parameters but also augment the data along the way.

The Append method returns a new FunctionInterface object with the same parameters plus a new parameter (the argument to Append) to the end of the parameters. There is also a matching AppendType templated structure that can return the type of an augmented FunctionInterface with a new type appended.

Example 21.9: Appending parameters to a FunctionInterface.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

using InitialFunctionInterfaceType =
  FunctionInterface<void(std::string, vtkm::Id)>;
InitialFunctionInterfaceType initialFunctionInterface =
  make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

using AppendedFunctionInterfaceType1 =
  FunctionInterface<void(std::string, vtkm::Id, std::string)>;
AppendedFunctionInterfaceType1 appendedFunctionInterface1 =
  initialFunctionInterface.Append(std::string("foobar"));
// appendedFunctionInterface1 has parameters ("Hello World", 42, "foobar")

using AppendedFunctionInterfaceType2 =
  InitialFunctionInterfaceType::AppendType<vtkm::Float32>::type;
AppendedFunctionInterfaceType2 appendedFunctionInterface2 =
  initialFunctionInterface.Append(vtkm::Float32(3.141));
// appendedFunctionInterface2 has parameters ("Hello World", 42, 3.141)
```

`Replace` is a similar method that returns a new `FunctionInterface` object with the same paraemters except with a specified parameter replaced with a new parameter (the argument to `Replace`). There is also a matching `ReplaceType` templated structure that can return the type of an augmented `FunctionInterface` with one of the parameters replaced.

Example 21.10: Replacing parameters in a `FunctionInterface`.

```
using vtkm::internal::FunctionInterface;
using vtkm::internal::make_FunctionInterface;

using InitialFunctionInterfaceType =
  FunctionInterface<void(std::string, vtkm::Id)>;
InitialFunctionInterfaceType initialFunctionInterface =
  make_FunctionInterface<void>(std::string("Hello World"), vtkm::Id(42));

using ReplacedFunctionInterfaceType1 =
  FunctionInterface<void(vtkm::Float32, vtkm::Id)>;
ReplacedFunctionInterfaceType1 replacedFunctionInterface1 =
  initialFunctionInterface.Replace<1>(vtkm::Float32(3.141));
// replacedFunctionInterface1 has parameters (3.141, 42)

using ReplacedFunctionInterfaceType2 =
  InitialFunctionInterfaceType::ReplaceType<2, std::string>::type;
ReplacedFunctionInterfaceType2 replacedFunctionInterface2 =
  initialFunctionInterface.Replace<2>(std::string("foobar"));
// replacedFunctionInterface2 has parameters ("Hello World", "foobar")
```

It is sometimes desirable to make multiple modifications at a time. This can be achieved by chaining modifications by calling `Append` or `Replace` on the result of a previous call.

Example 21.11: Chaining `Replace` and `Append` with a `FunctionInterface`.

```
template<typename FunctionInterfaceType>
void FunctionCallChain(const FunctionInterfaceType& parameters, vtkm::Id arraySize)
{
  // In this hypothetical function call chain, this function replaces the
  // first parameter with an array of that type and appends the array size
  // to the end of the parameters.

  using ArrayValueType =
    typename FunctionInterfaceType::template ParameterType<1>::type;

  // Allocate and initialize array.
  ArrayValueType value = parameters.template GetParameter<1>();
```

```
13    ArrayValueType* array = new ArrayValueType[arraySize];
14    for (vtkm::Id index = 0; index < arraySize; index++)
15    {
16      array[index] = value;
17    }
18
19    // Call next function with modified parameters.
20    NextFunctionChainCall(parameters.template Replace<1>(array).Append(arraySize));
21
22    // Clean up.
23    delete[] array;
24  }
```

21.5 Transformations

Rather than replace a single item in a FunctionInterface, it is sometimes desirable to change them all in a similar way. FunctionInterface supports two basic transform operations on its parameters: a static transform and a dynamic transform. The static transform determines its types at compile-time whereas the dynamic transform happens at run-time.

The static transform methods (named StaticTransformCont and StaticTransformExec) operate 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 21.12: Using a static transform of function interface class.

```
1   struct ParametersToPointersFunctor
2   {
3     template<typename T, vtkm::IdComponent Index>
4     struct ReturnType
5     {
6       using type = const T*;
7     };
8
9     template<typename T, vtkm::IdComponent Index>
10    VTKM_CONT const T* operator()(const T& x, vtkm::internal::IndexTag<Index>) const
11    {
12      return &x;
13    }
14  };
```

```
15
16 template<typename FunctionInterfaceType>
17 VTKM_CONT typename FunctionInterfaceType::template StaticTransformType<
18   ParametersToPointersFunctor>::type
19 ParametersToPointers(FunctionInterfaceType& functionInterface)
20 {
21   return functionInterface.StaticTransformCont(ParametersToPointersFunctor());
22 }
```

There are cases where one set of parameters must be transformed to another set, but the types of the new set are not known until run-time. That is, the transformed type depends on the contents of the data. The `DynamicTransformCont` method achieves this using a templated callback that gets called with the correct type at run-time.

The dynamic transform works with two functors provided by the user code (as opposed to the one functor in static transform). These functors are called the transform functor and the finish functor. The transform functor accepts three arguments. The first argument is a parameter to transform. The second argument is a continue function. Rather than return the transformed value, the transform functor calls the continue function, passing the transformed value as an argument. The third argument is a `vtkm::internal::IndexTag` for the index of the argument being transformed.

Unlike its static counterpart, the dynamic transform method does not return the transformed `FunctionInterface`. Instead, it passes the transformed `FunctionInterface` to the finish functor passed into `DynamicTransformCont`.

In the following contrived but illustrative example, a dynamic transform is used to convert strings containing numbers into number arguments. Strings that do not have numbers and all other arguments are passed through. Note that because the types for strings are not determined till run-time, this transform cannot be determined at compile time with meta-template programming. The index argument is ignored because all arguments are transformed the same way.

Example 21.13: Using a dynamic transform of a function interface.

```
1  struct UnpackNumbersTransformFunctor
2  {
3    template<typename InputType, typename ContinueFunctor, vtkm::IdComponent Index>
4    VTKM_CONT void operator()(const InputType& input,
5                              const ContinueFunctor& continueFunction,
6                              vtkm::internal::IndexTag<Index>) const
7    {
8      continueFunction(input);
9    }
10
11   template<typename ContinueFunctor, vtkm::IdComponent Index>
12   VTKM_CONT void operator()(const std::string& input,
13                             const ContinueFunctor& continueFunction,
14                             vtkm::internal::IndexTag<Index>) const
15   {
16     if ((input[0] >= '0') && (input[0] <= '9'))
17     {
18       std::stringstream stream(input);
19       vtkm::FloatDefault value;
20       stream >> value;
21       continueFunction(value);
22     }
23     else
24     {
25       continueFunction(input);
26     }
27   }
28 };
```

```
29
30 struct UnpackNumbersFinishFunctor
31 {
32   template<typename FunctionInterfaceType>
33   VTKM_CONT void operator()(FunctionInterfaceType& functionInterface) const
34   {
35     // Do something
36   }
37 };
38
39 template<typename FunctionInterfaceType>
40 void DoUnpackNumbers(const FunctionInterfaceType& functionInterface)
41 {
42   functionInterface.DynamicTransformCont(UnpackNumbersTransformFunctor(),
43                                          UnpackNumbersFinishFunctor());
44 }
```

One common use for the `FunctionInterface` dynamic transform is to convert parameters of virtual polymorphic type like `vtkm::cont::DynamicArrayHandle` and `vtkm::cont::DynamicPointCoordinates`. This use case is handled with a functor named `vtkm::cont::internal::DynamicTransform`. When used as the dynamic transform functor, it will convert all of these dynamic types to their static counterparts.

Example 21.14: Using `DynamicTransform` to cast dynamic arrays in a function interface.

```
1  template<typename Device>
2  struct ArrayCopyFunctor
3  {
4    template<typename Signature>
5    VTKM_CONT void operator()(
6      vtkm::internal::FunctionInterface<Signature> functionInterface) const
7    {
8      functionInterface.InvokeCont(*this);
9    }
10
11   template<typename T, class CIn, class COut>
12   VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<T, CIn>& input,
13                             vtkm::cont::ArrayHandle<T, COut>& output) const
14   {
15     vtkm::cont::DeviceAdapterAlgorithm<Device>::Copy(input, output);
16   }
17
18   template<typename TIn, typename TOut, class CIn, class COut>
19   VTKM_CONT void operator()(const vtkm::cont::ArrayHandle<TIn, CIn>&,
20                             vtkm::cont::ArrayHandle<TOut, COut>&) const
21   {
22     throw vtkm::cont::ErrorBadType("Arrays to copy must be the same type.");
23   }
24 };
25
26 template<typename Device>
27 void CopyDynamicArrays(vtkm::cont::DynamicArrayHandle input,
28                        vtkm::cont::DynamicArrayHandle output,
29                        Device)
30 {
31   vtkm::internal::FunctionInterface<void(vtkm::cont::DynamicArrayHandle,
32                                          vtkm::cont::DynamicArrayHandle)>
33     functionInterface = vtkm::internal::make_FunctionInterface<void>(input, output);
34
35   functionInterface.DynamicTransformCont(vtkm::cont::internal::DynamicTransform(),
36                                          ArrayCopyFunctor<Device>());
37 }
```

21.6 For Each

The invoke methods (principally) make a single function call passing all of the parameters to this function. The transform methods call a function on each parameter to convert it to some other data type. It is also sometimes helpful to be able to call a unary function on each parameter that is not expected to return a value. Typically the use case is for the function to have some sort of side effect. For example, the function might print out some value (such as in the following example) or perform some check on the data and throw an exception on failure.

This feature is implemented in the for each methods of `FunctionInterface`. As with all the `FunctionInterface` methods that take functors, there are separate implementations for the control environment and the execution environment. There are also separate implementations taking `const` and non-`const` references to functors to simplify making functors with side effects.

Example 21.15: Using the `ForEach` feature of `FunctionInterface`.

```
1  struct PrintArgumentFunctor
2  {
3    template<typename T, vtkm::IdComponent Index>
4    VTKM_CONT void operator()(const T& argument, vtkm::internal::IndexTag<Index>) const
5    {
6      std::cout << Index << ":" << argument << " ";
7    }
8  };
9
10 template<typename FunctionInterfaceType>
11 VTKM_CONT void PrintArguments(const FunctionInterfaceType& functionInterface)
12 {
13   std::cout << "( ";
14   functionInterface.ForEachCont(PrintArgumentFunctor());
15   std::cout << ")" << std::endl;
16 }
```

CHAPTER

TWENTYTWO

WORKLET ARGUMENTS

From the `ControlSignature` and `ExecutionSignature` defined in worklets, VTK-m uses template metaprogramming to build the code required to manage data from control to 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 13 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 23.

The management of data in worklet arguments is handled by three classes that provide type checking, transportation, and fetching. 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 10.2.11), 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 23.

22.1 Type Checks

Before attempting to move data from the control to the execution environment, the VTK-m dispatchers 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::exec::ExecutionObjectBase and must be copyable through memcpy or similar mechanism.

vtkm::cont::arg::TypeCheckTagArray True if the type is a vtkm::cont::ArrayHandle. TypeCheckTagArray also has a template parameter that is a type list. The ArrayHandle must also have a value type contained in this type list.

vtkm::cont::arg::TypeCheckTagAtomicArray Similar to TypeCheckTagArray except it only returns true for array types with values that are supported for atomic arrays.

vtkm::cont::arg::TypeCheckTagCellSet True if and only if the object is a vtkm::cont::CellSet or one of its subclasses.

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 dispatcher code, so these examples are for demonstration only.

Example 22.1: 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::TypeCheckTagArray;
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 = // true
18     TypeCheck<TypeCheckTagArray<vtkm::TypeListTagField>, ArrayType>::value;
19   bool check4 = // false
20     TypeCheck<TypeCheckTagArray<vtkm::TypeListTagIndex>, ArrayType>::value;
21   bool check5 = TypeCheck<TypeCheckTagExecObject, ArrayType>::value; // false
22 }
```

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 Invoke 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 22.2: Defining a custom TypeCheck.

```
1  namespace vtkm
2  {
3  namespace cont
4  {
5  namespace arg
6  {
7
```

```
8  struct TypeCheckTag2DCoordinates
9  {
10 };
11
12 template<typename ArrayType>
13 struct TypeCheck<TypeCheckTag2DCoordinates, ArrayType>
14 {
15   static const bool value = vtkm::cont::arg::TypeCheck<
16     vtkm::cont::arg::TypeCheckTagArray<vtkm::TypeListTagFieldVec2>,
17     ArrayType>::value;
18 };
19
20 } // namespace arg
21 } // namespace cont
22 } // namespace vtkm
```

Did you know?

The type check defined in Example 22.2 could actually be replaced by the more general `TypeCheckTagArray` *that already comes with VTK-m (and, in fact, the implementation uses this type check internally for simplicity). This example is mostly provided for demonstrative purposes. In practice, it is often useful to use* `std::is_same` *or* `std::is_base_of`*, which are provided by the standard template library starting with C++11, to determine* `value` *in a* `TypeCheck`*.*

22.2 Transport

After all the argument types are checked, the base dispatcher 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 4 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, and the size of the output domain and 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` Simply returns the given execution object, which should be ready to load onto the device.

`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::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 13.5.2 for a description of "from" and "to" topologies.) The returned execution object is a connectivity object (as described in Section 13.8).

`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 base VTK-m dispatcher code, so these examples are for demonstration only.

Example 22.3: Behavior of `vtkm::cont::arg::Transport`.

```
using ArrayType = vtkm::cont::ArrayHandle<vtkm::Id>;

void DoTransport(ArrayType inArray, ArrayType outArray)
{
  using Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG;

  using vtkm::cont::arg::Transport;
  using vtkm::cont::arg::TransportTagArrayIn;
  using vtkm::cont::arg::TransportTagArrayOut;
  using vtkm::cont::arg::TransportTagWholeArrayInOut;

  // The array in transport returns a read-only array portal.
  using ArrayInTransport = Transport<TransportTagArrayIn, ArrayType, Device>;
  ArrayInTransport::ExecObjectType inPortal =
    ArrayInTransport()(inArray, inArray, 10, 10);

  // The array out transport returns an allocated array portal.
  using ArrayOutTransport = Transport<TransportTagArrayOut, ArrayType, Device>;
  ArrayOutTransport::ExecObjectType outPortal =
    ArrayOutTransport()(outArray, inArray, 10, 10);

  // The whole array in transport returns a read-only array portal wrapped in
  // a vtkm::exec::ExecutionWholeArrayConst.
  using WholeArrayTransport =
    Transport<TransportTagWholeArrayInOut, ArrayType, Device>;
  WholeArrayTransport::ExecObjectType wholeArray =
    WholeArrayTransport()(inArray, inArray, 10, 10);
}
```

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 22.4: Defining a custom `Transport`.

```
namespace vtkm
{
namespace cont
{
namespace arg
{

struct TransportTag2DLineSegmentsIn
{
};

template<typename ContObjectType, typename Device>
struct Transport<vtkm::cont::arg::TransportTag2DLineSegmentsIn,
                 ContObjectType,
                 Device>
{
  VTKM_IS_ARRAY_HANDLE(ContObjectType);

```

```
19    using GroupedArrayType = vtkm::cont::ArrayHandleGroupVec<ContObjectType, 2>;
20
21    using ExecObjectType =
22      typename GroupedArrayType::template ExecutionTypes<Device>::PortalConst;
23
24    template<typename InputDomainType>
25    VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
26                                        const InputDomainType&,
27                                        vtkm::Id inputRange,
28                                        vtkm::Id) 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());
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 for debugging purposes in case there is a problem with the* `TypeCheck` *or this* `Transport` *is used with an unexpected* `TypeCheck`*.*

22.3 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 four 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 a type of thread indices object, which manages the indices and other metadata associated with the thread for which the `Fetch` operator gets called. 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 23.2.

The fourth template parameter to a `Fetch` `struct` is the type of the execution object that is created by the `Transport` (as described in Section 22.2). This is generally where the data are fetched from.

A Fetch contains a type named ValueType that is the type of data that is passed to and from the worklet function. 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.

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::AspectTagFromCount and vtkm::exec::arg::AspectTagFromIndices 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 ValueType type and a pair of Load and Store methods that get a value out of the data and store a value in the data, respectively.

Example 22.5: Defining a custom Fetch.

```
1  namespace vtkm
2  {
3  namespace exec
4  {
5  namespace arg
6  {
7
8  struct FetchTag2DLineSegmentsIn
9  {
10 };
```

```
11
12 template<typename ThreadIndicesType, typename ExecObjectType>
13 struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
14              vtkm::exec::arg::AspectTagDefault,
15              ThreadIndicesType,
16              ExecObjectType>
17 {
18   using ValueType = typename ExecObjectType::ValueType;
19
20   VTKM_SUPPRESS_EXEC_WARNINGS
21   VTKM_EXEC
22   ValueType Load(const ThreadIndicesType& indices,
23                  const ExecObjectType& arrayPortal) const
24   {
25     return arrayPortal.Get(indices.GetInputIndex());
26   }
27
28   VTKM_EXEC
29   void Store(const ThreadIndicesType&, const ExecObjectType&, const ValueType&) const
30   {
31     // Store is a no-op for this fetch.
32   }
33 };
34
35 } // namespace arg
36 } // namespace exec
37 } // namespace vtkm
```

Did you know?

The fetch defined in Example 22.5 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 13.10 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::AspectTagFromCount` 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::AspectTagFromIndices` 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 22.6: Defining a custom `Aspect`.

```
namespace vtkm
{
namespace exec
{
namespace arg
{

struct AspectTagFirstPoint
{
};

template<typename ThreadIndicesType, typename ExecObjectType>
struct Fetch<vtkm::exec::arg::FetchTag2DLineSegmentsIn,
             vtkm::exec::arg::AspectTagFirstPoint,
             ThreadIndicesType,
             ExecObjectType>
{
  using ValueType = typename ExecObjectType::ValueType::ComponentType;

  VTKM_SUPPRESS_EXEC_WARNINGS
  VTKM_EXEC
  ValueType Load(const ThreadIndicesType& indices,
                 const ExecObjectType& arrayPortal) const
  {
    return arrayPortal.Get(indices.GetInputIndex())[0];
  }

  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
```

22.4 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 dispatcher's `Invoke` method 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 22.7: 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 22.8: Using a custom `ControlSignature` tag.

```
1  using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
2                                FieldOut<Vec2> vecOut,
3                                FieldIn<Index> index);
```

22.5 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 22.3, 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 22.7).

Example 22.9: 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 22.10: Using a custom ExecutionSignature tag.

```
1  using ControlSignature = void(LineSegment2DCoordinatesIn coordsIn,
2                                FieldOut<Vec2> vecOut,
3                                FieldIn<Index> index);
4  using ExecutionSignature = void(FirstPoint<_1>, SecondPoint<_1>, _2);
```

CHAPTER

TWENTYTHREE

NEW WORKLET TYPES

The basic building block for an algorithm in VTK-m is the worklet. Chapter 13 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.

23.1 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 23.1.

The fractal is formed by iteratively replacing the curve's lines with this basic shape. Figure 23.2 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 23.1: Basic shape for the Koch Snowflake.

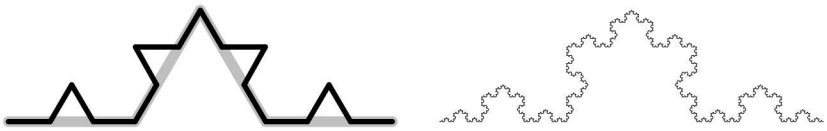


Figure 23.2: The Koch Snowflake after the second iteration (left image) and after several more iterations (right image).

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 23.6 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 23.3.

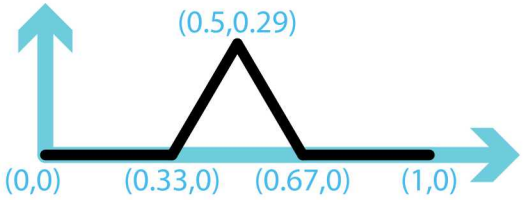


Figure 23.3: Parametric coordinates for the Koch Snowflake shape.

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 23.4.

To implement the application of the line fractal demonstrated in Figure 23.4, 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 23.1

Example 23.1: A support class for a line fractal worklet.

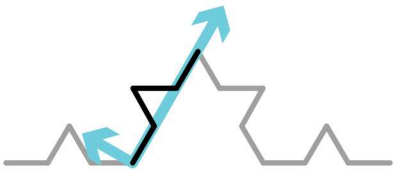


Figure 23.4: Applying the line fractal transform for the Koch Snowflake.

```
namespace vtkm
{
namespace exec
{

class LineFractalTransform
{
  using VecType = vtkm::Vec<vtkm::FloatDefault, 2>;

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
  {
    VecType ppointCast(ppoint);
    VecType 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:
  VecType Offset;
  VecType UAxis;
  VecType 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 23.2 demonstrates how we intend to use this worklet type.

Example 23.2: 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);
    }
  };

  template<typename Device>
  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 2>> Run(
    vtkm::IdComponent numIterations,
    Device)
  {
    using VecType = vtkm::Vec<vtkm::Float32, 2>;

    vtkm::cont::ArrayHandle<VecType> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, VecType(0.0f, 0.0f));
    points.GetPortalControl().Set(1, VecType(1.0f, 0.0f));

    vtkm::worklet::DispatcherLineFractal<KochSnowflake::FractalWorklet, Device>
      dispatcher;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<VecType> outPoints;
      dispatcher.Invoke(points, outPoints);
      points = outPoints;
    }

    return points;
  }
};
```

23.2 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 22.3, a thread indices object is given to the `vtkm::exec::arg::Fetch` class 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, the execution object for the input domain, and an offset of the current index of the local invoke to a global indexing (used for streaming).

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 23.4).

Example 23.3: Implementation of `GetThreadIndices` in a worklet superclass.

```
1  VTKM_SUPPRESS_EXEC_WARNINGS
2  template<typename OutToInPortalType,
3           typename VisitPortalType,
4           typename InputDomainType>
5  VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
6    vtkm::Id threadIndex,
7    const OutToInPortalType& outToIn,
8    const VisitPortalType& visit,
9    const InputDomainType& inputPoints,
10   vtkm::Id globalThreadIndexOffset) const
11 {
12   return vtkm::exec::arg::ThreadIndicesLineFractal(
13     threadIndex, outToIn, visit, inputPoints, globalThreadIndexOffset);
14 }
```

As we can see in Example 23.3, 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 23.4: 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::Vec<vtkm::FloatDefault, 2>;

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OutToInPortalType,
           typename VisitPortalType,
           typename InputPointPortal>
  VTKM_EXEC ThreadIndicesLineFractal(vtkm::Id threadIndex,
                                     const OutToInPortalType& outToIn,
                                     const VisitPortalType& visit,
                                     const InputPointPortal& inputPoints,
                                     vtkm::Id globalThreadIndexOffset = 0)
    : Superclass(threadIndex,
                 outToIn.Get(threadIndex),
                 visit.Get(threadIndex),
                 globalThreadIndexOffset)
  {
    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
```

23.3 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 22 describes in detail the process of defining new worklet types and the associated code to manage data from an argument to the dispatcher's `Invoke` to the data that are passed to the worklet operator. Rather than repeat the discussion, readers should review Chapter 22 for details on how custom arguments are defined for a new worklet type. In particular, we use the code from Examples 22.2 (page 280), 22.4 (page 283), and 22.5 (page 285) 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 23.5: Custom `ControlSignature` tag for the input domain of our example worklet type.

```
1  struct SegmentsIn : 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  };
```

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 23.1), so it makes sense to define our own custom `ExecutionSignature` tag to provide this object.

Chapter 22 gives an example of a custom `ExecutionSignature` tag that modifies what information is fetched from an argument (Examples 22.6 and 22.9). 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 23.6: A `Fetch` for an aspect that does not depend on any control argument.

```
1  namespace vtkm
2  {
3  namespace exec
4  {
5  namespace arg
6  {
7
8  struct AspectTagLineFractalTransform
9  {
10 };
11
12 template<typename FetchTag, typename ExecObjectType>
13 struct Fetch<FetchTag,
14              vtkm::exec::arg::AspectTagLineFractalTransform,
15              vtkm::exec::arg::ThreadIndicesLineFractal,
16              ExecObjectType>
17 {
18   using ValueType = LineFractalTransform;
19
20   VTKM_SUPPRESS_EXEC_WARNINGS
21   VTKM_EXEC
22   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 23.7: 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 23.8: 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 23.8 relies on a custom transport named `TransportTag2DLineSegmentsOut`. There is nothing particularly special about this transport, but we provide the implementation here for completeness.

Example 23.9: 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::template ExecutionTypes<Device>::Portal;

  template<typename InputDomainType>
  VTKM_CONT ExecObjectType operator()(const ContObjectType& object,
                                      const InputDomainType&,
                                      vtkm::Id,
                                      vtkm::Id outputRange) const
  {
    GroupedArrayType groupedArray(vtkm::cont::make_ArrayHandleGroupVec<2>(object));
    return groupedArray.PrepareForOutput(outputRange, Device());
  }
};

} // 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 23.10: Implementing a `FieldIn` tag.

```
  template<typename TypeList = AllTypes>
  struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray<TypeList>;
    using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
  };
```

23.4 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 23.3) and the `GetThreadIndices` method (as discussed in Section 23.2. 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).

Example 23.11: Superclass for a new type of worklet.

```
namespace vtkm
{
namespace worklet
{

class WorkletLineFractal : public vtkm::worklet::internal::WorkletBase
{
public:
  /// Control signature tag for line segments in the plane. Used as the input
  /// domain.
  ///
  struct SegmentsIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTag2DCoordinates;
    using TransportTag = vtkm::cont::arg::TransportTag2DLineSegmentsIn;
    using FetchTag = vtkm::exec::arg::FetchTag2DLineSegmentsIn;
  };

  /// Control signature tag for a group of output line segments. The template
  /// argument specifies how many line segments are outputted for each input.
  /// The type is a Vec-like (of size NumSegments) of Vec-2's.
  ///
  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;
  };

  /// Control signature tag for input fields. There is one entry per input line
  /// segment. This tag takes a template argument that is a type list tag that
  /// limits the possible value types in the array.
  ///
  template<typename TypeList = AllTypes>
  struct FieldIn : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray<TypeList>;
    using TransportTag = vtkm::cont::arg::TransportTagArrayIn;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectIn;
  };

  /// Control signature tag for input fields. There is one entry per input line
  /// segment. This tag takes a template argument that is a type list tag that
  /// limits the possible value types in the array.
  ///
  template<typename TypeList = AllTypes>
  struct FieldOut : vtkm::cont::arg::ControlSignatureTagBase
  {
    using TypeCheckTag = vtkm::cont::arg::TypeCheckTagArray<TypeList>;
    using TransportTag = vtkm::cont::arg::TransportTagArrayOut;
    using FetchTag = vtkm::exec::arg::FetchTagArrayDirectOut;
  };

  /// Execution signature tag for a LineFractalTransform from the input.
  ///
  struct Transform : vtkm::exec::arg::ExecutionSignatureTagBase
```

```
  {
    static const vtkm::IdComponent INDEX = 1;
    using AspectTag = vtkm::exec::arg::AspectTagLineFractalTransform;
  };

  VTKM_SUPPRESS_EXEC_WARNINGS
  template<typename OutToInPortalType,
           typename VisitPortalType,
           typename InputDomainType>
  VTKM_EXEC vtkm::exec::arg::ThreadIndicesLineFractal GetThreadIndices(
    vtkm::Id threadIndex,
    const OutToInPortalType& outToIn,
    const VisitPortalType& visit,
    const InputDomainType& inputPoints,
    vtkm::Id globalThreadIndexOffset) const
  {
    return vtkm::exec::arg::ThreadIndicesLineFractal(
      threadIndex, outToIn, visit, inputPoints, globalThreadIndexOffset);
  }
};

} // namespace worklet
} // 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.*

23.5 Dispatcher

The final element required for a new worklet type is an associated dispatcher class for invoking the worklet. As documented in Chapter 13, each worklet type has its own associated dispatcher object. By convention, the dispatcher is placed in the `vtkm::worklet` and has the same name as the worklet superclass with the `Worklet` replaced with `Dispatcher`. So since the worklet superclass for our motivating example is named `WorkletLineFractal`, we name the associated dispatcher `DispatcherLineFractal`.

Also by convention, a dispatcher is a templated class. The first template argument should be the type of the worklet (which should be a subclass of the associated worklet superclass). The last template argument should be a device adapter tag with a default value set to `VTKM_DEFAULT_DEVICE_ADAPTER_TAG`. Other template arguments that the dispatcher might need should be placed in between these two.

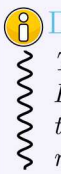
Example 23.12: Standard template arguments for a dispatcher class.

```
template<typename WorkletType, typename Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG>
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 23.13: Subclassing `DispatcherBase`.

```
template<typename WorkletType, typename Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG>
class DispatcherLineFractal
  : public vtkm::worklet::internal::DispatcherBase<
      DispatcherLineFractal<WorkletType>,
      WorkletType,
      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 23.14: 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 23.15: 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 21.) 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 23.16: Checking the input domain tag and type.

```
1   // Get the control signature tag for the input domain.
2   using InputDomainTag = typename Invocation::InputDomainTag;
3
4   // If you get a compile error on this line, then you have set the input
5   // domain to something that is not a SegmentsIn parameter, which is not
6   // valid.
7   VTKM_STATIC_ASSERT(
8     (std::is_same<InputDomainTag,
9                   vtkm::worklet::WorkletLineFractal::SegmentsIn>::value));
10
11  // This is the type for the input domain
12  using InputDomainType = typename Invocation::InputDomainType;
13
14  // If you get a compile error on this line, then you have tried to use
15  // something that is not a vtkm::cont::ArrayHandle as the input domain to a
16  // topology operation (that operates on a cell set connection domain).
17  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 23.17: 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 23.18: Implementation of a dispatcher for a new type of worklet.

```
namespace vtkm
{
namespace worklet
{

template<typename WorkletType, typename Device = VTKM_DEFAULT_DEVICE_ADAPTER_TAG>
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:
  // 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)
  {
  }

  template<typename Invocation>
  VTKM_CONT void DoInvoke(Invocation& invocation) const
  {
    // 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));
```

```
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
```

23.6 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.

23.6.1 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 23.5.

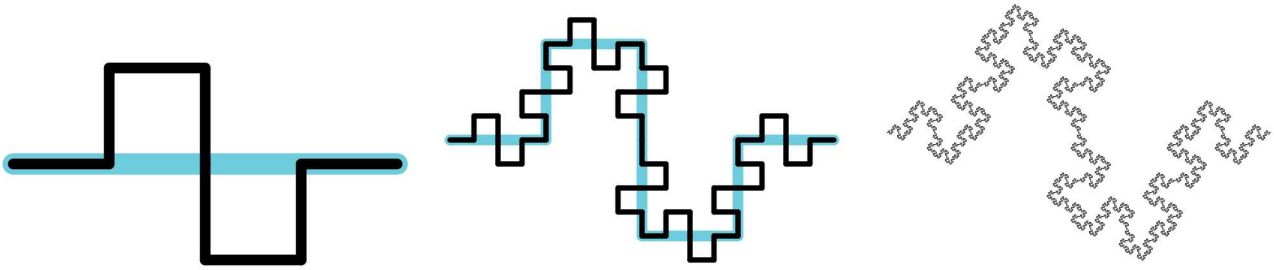


Figure 23.5: The quadratic type 2 curve fractal. The left image gives the first iteration. The middle image gives the second iteration. The right image gives the result after a few iterations.

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 23.19: A worklet to generate a quadratic type 2 curve fractal.

```
1  struct QuadraticType2
2  {
```

```
3   struct FractalWorklet : vtkm::worklet::WorkletLineFractal
4   {
5     using ControlSignature = void(SegmentsIn, SegmentsOut<8>);
6     using ExecutionSignature = void(Transform, _2);
7     using InputDomain = _1;
8
9     template<typename SegmentsOutVecType>
10    void operator()(const vtkm::exec::LineFractalTransform& transform,
11                    SegmentsOutVecType& segmentsOutVec) const
12    {
13      segmentsOutVec[0][0] = transform(0.00f, 0.00f);
14      segmentsOutVec[0][1] = transform(0.25f, 0.00f);
15
16      segmentsOutVec[1][0] = transform(0.25f, 0.00f);
17      segmentsOutVec[1][1] = transform(0.25f, 0.25f);
18
19      segmentsOutVec[2][0] = transform(0.25f, 0.25f);
20      segmentsOutVec[2][1] = transform(0.50f, 0.25f);
21
22      segmentsOutVec[3][0] = transform(0.50f, 0.25f);
23      segmentsOutVec[3][1] = transform(0.50f, 0.00f);
24
25      segmentsOutVec[4][0] = transform(0.50f, 0.00f);
26      segmentsOutVec[4][1] = transform(0.50f, -0.25f);
27
28      segmentsOutVec[5][0] = transform(0.50f, -0.25f);
29      segmentsOutVec[5][1] = transform(0.75f, -0.25f);
30
31      segmentsOutVec[6][0] = transform(0.75f, -0.25f);
32      segmentsOutVec[6][1] = transform(0.75f, 0.00f);
33
34      segmentsOutVec[7][0] = transform(0.75f, 0.00f);
35      segmentsOutVec[7][1] = transform(1.00f, 0.00f);
36    }
37  };
38
39  template<typename Device>
40  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 2>> Run(
41    vtkm::IdComponent numIterations,
42    Device)
43  {
44    using VecType = vtkm::Vec<vtkm::Float32, 2>;
45
46    vtkm::cont::ArrayHandle<VecType> points;
47
48    // Initialize points array with a single line
49    points.Allocate(2);
50    points.GetPortalControl().Set(0, VecType(0.0f, 0.0f));
51    points.GetPortalControl().Set(1, VecType(1.0f, 0.0f));
52
53    vtkm::worklet::DispatcherLineFractal<QuadraticType2::FractalWorklet, Device>
54      dispatcher;
55
56    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
57    {
58      vtkm::cont::ArrayHandle<VecType> outPoints;
59      dispatcher.Invoke(points, outPoints);
60      points = outPoints;
61    }
62
63    return points;
64  }
65 };
```

23.6.2 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 23.6. Iterative replacing using this "Y" shape produces a bushy tree shape.

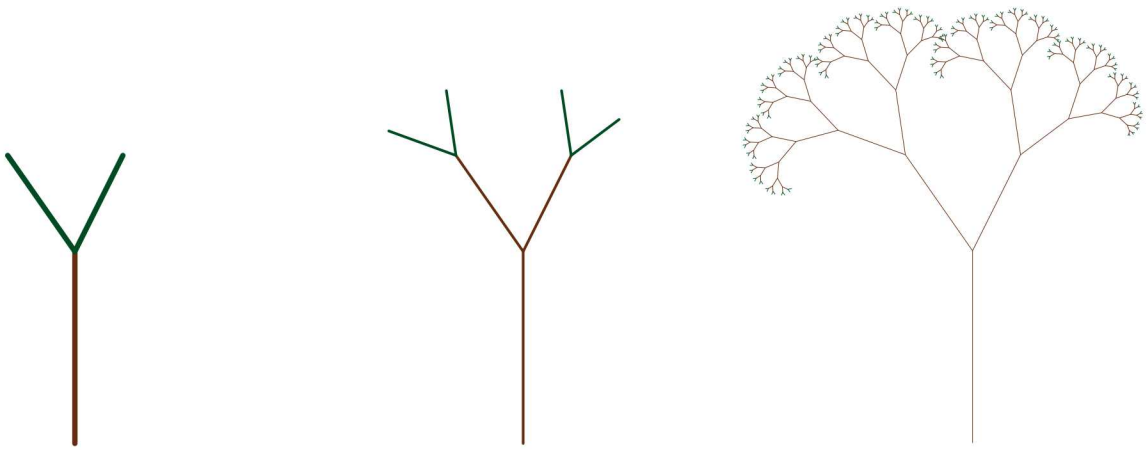


Figure 23.6: The tree fractal replaces each line with the "Y" shape shown at left. An iteration grows branches at the end (middle). After several iterations the tree branches out to the bushy shape at right.

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 13.10.) 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 23.20: A worklet to generate a tree fractal.

```
1  struct TreeFractal
2  {
3    struct FractalWorklet : vtkm::worklet::WorkletLineFractal
4    {
5      using ControlSignature = void(SegmentsIn,
6                                    SegmentsOut<1>,
7                                    FieldOut<> countNextIteration);
8      using ExecutionSignature = void(Transform, VisitIndex, _2, _3);
9      using InputDomain = _1;
10
11     using ScatterType = vtkm::worklet::ScatterCounting;
12
13     template<typename Storage, typename Device>
14     VTKM_CONT static ScatterType MakeScatter(
15       const vtkm::cont::ArrayHandle<vtkm::IdComponent, Storage>& count,
16       Device)
17     {
18       return ScatterType(count, Device());
19     }
20
```

```
21      template<typename SegmentsOutVecType>
22      void operator()(const vtkm::exec::LineFractalTransform& transform,
23                      vtkm::IdComponent visitIndex,
24                      SegmentsOutVecType& segmentsOutVec,
25                      vtkm::IdComponent& countNextIteration) const
26      {
27        switch (visitIndex)
28        {
29          case 0:
30            segmentsOutVec[0][0] = transform(0.0f, 0.0f);
31            segmentsOutVec[0][1] = transform(1.0f, 0.0f);
32            countNextIteration = 1;
33            break;
34          case 1:
35            segmentsOutVec[0][0] = transform(1.0f, 0.0f);
36            segmentsOutVec[0][1] = transform(1.5f, -0.25f);
37            countNextIteration = 3;
38            break;
39          case 2:
40            segmentsOutVec[0][0] = transform(1.0f, 0.0f);
41            segmentsOutVec[0][1] = transform(1.5f, 0.35f);
42            countNextIteration = 3;
43            break;
44          default:
45            this->RaiseError("Unexpected visit index.");
46        }
47      }
48    };
49
50    template<typename Device>
51    VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 2>> Run(
52      vtkm::IdComponent numIterations,
53      Device)
54    {
55      using VecType = vtkm::Vec<vtkm::Float32, 2>;
56
57      vtkm::cont::ArrayHandle<VecType> points;
58
59      // Initialize points array with a single line
60      points.Allocate(2);
61      points.GetPortalControl().Set(0, VecType(0.0f, 0.0f));
62      points.GetPortalControl().Set(1, VecType(0.0f, 1.0f));
63
64      vtkm::cont::ArrayHandle<vtkm::IdComponent> count;
65
66      // Initialize count array with 3 (meaning iterate)
67      count.Allocate(1);
68      count.GetPortalControl().Set(0, 3);
69
70      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
71      {
72        vtkm::worklet::DispatcherLineFractal<TreeFractal::FractalWorklet, Device>
73          dispatcher(FractalWorklet::MakeScatter(count, Device()));
74
75        vtkm::cont::ArrayHandle<VecType> outPoints;
76        dispatcher.Invoke(points, outPoints, count);
77        points = outPoints;
78      }
79
80      return points;
81    }
82  };
```

23.6.3 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 23.7. As you can see by the fourth iteration a more complicated pattern starts to emerge. Figure 23.8 shows the twelfth iteration a demonstrates a repeating spiral.

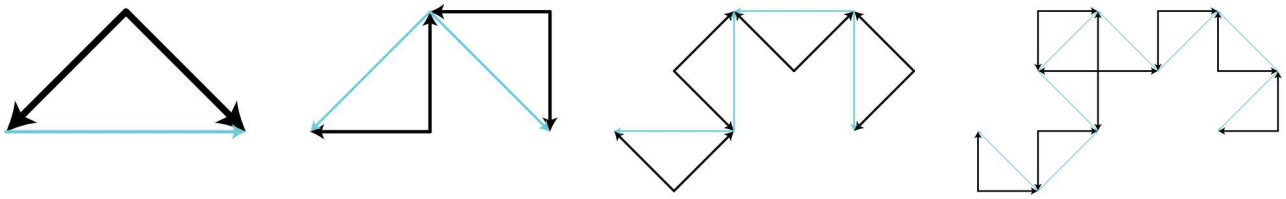


Figure 23.7: The first four iterations of the dragon fractal. The cyan lines give the previous iteration for reference.

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 23.7 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 23.7 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.

Example 23.21: A worklet to generate the dragon fractal.

```
1  struct DragonFractal
2  {
3    struct FractalWorklet : vtkm::worklet::WorkletLineFractal
4    {
5      using ControlSignature = void(SegmentsIn, SegmentsOut<2>);
6      using ExecutionSignature = void(Transform, _2);
7      using InputDomain = _1;
8
9      template<typename SegmentsOutVecType>
10     void operator()(const vtkm::exec::LineFractalTransform& transform,
11                     SegmentsOutVecType& segmentsOutVec) const
12     {
13       segmentsOutVec[0][0] = transform(0.5f, 0.5f);
14       segmentsOutVec[0][1] = transform(0.0f, 0.0f);
15
16       segmentsOutVec[1][0] = transform(0.5f, 0.5f);
17       segmentsOutVec[1][1] = transform(1.0f, 0.0f);
18     }
19   };
20
21   template<typename Device>
22   VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 2>> Run(
23     vtkm::IdComponent numIterations,
```

Figure 23.8: The dragon fractal after 12 iterations.

```
24      Device)
25    {
26      using VecType = vtkm::Vec<vtkm::Float32, 2>;
27
28      vtkm::cont::ArrayHandle<VecType> points;
29
30      // Initialize points array with a single line
31      points.Allocate(2);
32      points.GetPortalControl().Set(0, VecType(0.0f, 0.0f));
33      points.GetPortalControl().Set(1, VecType(1.0f, 0.0f));
34
35      vtkm::worklet::DispatcherLineFractal<DragonFractal::FractalWorklet, Device>
36        dispatcher;
37
38      for (vtkm::IdComponent i = 0; i < numIterations; ++i)
39      {
40        vtkm::cont::ArrayHandle<VecType> outPoints;
41        dispatcher.Invoke(points, outPoints);
42        points = outPoints;
43      }
44
45      return points;
46    }
47  };
```

23.6.4 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 23.9 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.)

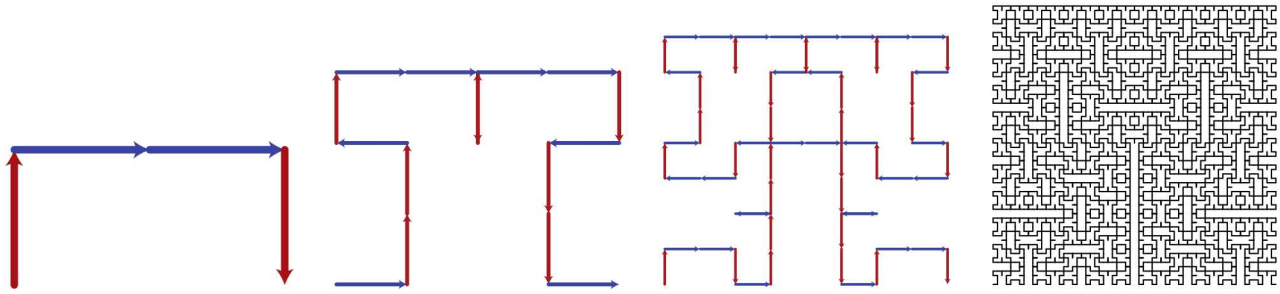


Figure 23.9: The first, second, third, and sixth iterations, respectively, of the Hilbert curve fractal.

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 23.9, is drawn to the "left" of the initial line along the horizontal axis. The next iteration, the second image in Figure 23.9, is created by drawing the shape to the "right" of the vertical line segments but to the left of the horizontal segments.

Section 23.6.3 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 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 23.9), then the shape is drawn to the "left." If the field value is set to -1 (represented by the red line segments in Figure 23.9), 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 23.22: A worklet to generate the Hilbert curve.

```
struct HilbertCurve
{
  struct FractalWorklet : vtkm::worklet::WorkletLineFractal
  {
    using ControlSignature = void(SegmentsIn,
                                  FieldIn<> directionIn,
                                  SegmentsOut<4>,
                                  FieldOut<> directionOut);
    using ExecutionSignature = void(Transform, _2, _3, _4);
    using InputDomain = _1;

    template<typename SegmentsOutVecType>
    void operator()(const vtkm::exec::LineFractalTransform& transform,
                    vtkm::Int8 directionIn,
                    SegmentsOutVecType& segmentsOutVec,
                    vtkm::Vec<vtkm::Int8, 4>& directionOut) const
    {
      segmentsOutVec[0][0] = transform(0.0f, directionIn * 0.0f);
      segmentsOutVec[0][1] = transform(0.0f, directionIn * 0.5f);
      directionOut[0] = -directionIn;

      segmentsOutVec[1][0] = transform(0.0f, directionIn * 0.5f);
      segmentsOutVec[1][1] = transform(0.5f, directionIn * 0.5f);
      directionOut[1] = directionIn;

      segmentsOutVec[2][0] = transform(0.5f, directionIn * 0.5f);
      segmentsOutVec[2][1] = transform(1.0f, directionIn * 0.5f);
      directionOut[2] = directionIn;

      segmentsOutVec[3][0] = transform(1.0f, directionIn * 0.5f);
      segmentsOutVec[3][1] = transform(1.0f, directionIn * 0.0f);
      directionOut[3] = -directionIn;
    }
  };

  template<typename Device>
  VTKM_CONT static vtkm::cont::ArrayHandle<vtkm::Vec<vtkm::FloatDefault, 2>> Run(
    vtkm::IdComponent numIterations,
    Device)
  {
    using VecType = vtkm::Vec<vtkm::Float32, 2>;

    vtkm::cont::ArrayHandle<VecType> points;

    // Initialize points array with a single line
    points.Allocate(2);
    points.GetPortalControl().Set(0, VecType(0.0f, 0.0f));
    points.GetPortalControl().Set(1, VecType(1.0f, 0.0f));

    vtkm::cont::ArrayHandle<vtkm::Int8> directions;

    // Initialize direction with positive.
    directions.Allocate(1);
    directions.GetPortalControl().Set(0, 1);

    vtkm::worklet::DispatcherLineFractal<HilbertCurve::FractalWorklet, Device>
      dispatcher;

    for (vtkm::IdComponent i = 0; i < numIterations; ++i)
    {
      vtkm::cont::ArrayHandle<VecType> outPoints;
      vtkm::cont::ArrayHandle<vtkm::Int8> outDirections;
      dispatcher.Invoke(points,
                        directions,
```

```
65                          outPoints,
66                          vtkm::cont::make_ArrayHandleGroupVec<4>(outDirections));
67        points = outPoints;
68        directions = outDirections;
69      }
70
71      return points;
72    }
73  };
```

Part V

Appendix

INDEX