LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-224144

Architectural Visualization of C/C++ Source Code for Program Comprehension

T. Panas, T. G. W. Epperly, D. Quinlan, A. Saebjornsen, R. Vuduc

September 1, 2006

29th International Conference on Software Engineering
Minneapolis, MN, United States
May 20, 2007 through May 26, 2007

Architectural Visualization of C/C++ Source Code for Program Comprehension

Thomas Panas Tom Epperly Dan Quinlan Andreas Sæbjørnsen Richard Vuduc

Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory
{panas2|tepperly2|dquinlan|richie}@llnl.gov andsebjo@student.matnat.uio.no

Abstract

Structural and behavioral visualization of large-scale legacy systems to aid program comprehension is still a major challenge. The challenge is even greater when applications are implemented in flexible and expressive languages such as C and C++. In this paper, we consider visualization of static and dynamic aspects of large-scale scientific C/C++ applications. For our investigation, we reuse and integrate specialized analysis and visualization tools. Furthermore, we present a novel layout algorithm that permits a compressive architectural view of a large-scale software system. Our layout is unique in that it allows traditional program visualizations, i.e., graph structures, to be seen in relation to the application's file structure.

1 Introduction

Reverse engineering aids software systems comprehension by supporting developers, designers, maintainers, or managers in understanding the purpose, the structure, and the dynamics of such systems. For large-scale systems, program visualization is indispensable for program understanding. However, visualizing program structure and behavior requires the ability to read and meaningfully analyze program source code.

In this paper, we describe our efforts to visualize large-scale scientific C/C++ applications on an architectural level. By "architectural level," we mean an abstraction of a program model, in contrast to a detailed program representation such as an abstract syntax tree (AST). The goal of our current work is to create comprehensive program visualizations, following our findings on visualizing Java source code [11]. We have therefore developed a layout algorithm featuring i) layout predictability, meaning that different layout runs produce similar visualizations, thereby allowing the quick recognition of familiar visual structures, and ii) focus+context, a technique to illustrate visual detail without losing the visual context.

To retrieve, analyze, and visualize C/C++ source code, we have re-used, combined, and enhanced the following reverse engineering tools (see Section 6): ROSE [18], a C/C++ source retriever; ROSEVA, a program analyzer; Vizz3D [11, 14], a program visualization engine; and VizzAnalyzer [13, 23], a framework for reverse engineering tool integration. We use the VizzAnalyzer to combine the strengths of the separately developed ROSE, ROSEVA, and Vizz3D. For language interoperability between C/C++ (ROSE and ROSEVA) and Java (Vizz3D and VizzAnalyzer), we use Babel [1].

In Sections 2–3, we discuss the modelling assumptions and basic philosophy of our architectural visualization. Section 4 discusses our layout algorithm, describing our merge of a force directed layout algorithm with hierarchical information, and our representation of file information together with a programs graph representation. The application of our algorithm to current C/C++ projects and our findings are discussed in Section 5. In Section 6 we elaborate on the tools being re-used and merged. Finally, Section 7 concludes this paper.

2 The Program Model

Different types of entities and relations may be used to characterize a software system. For our architectural visualization we therefore use a general graph structure to capture the models. In this section, we discuss the model properties needed to view the architecture of a software system.

The graph representing the software system consists of i) nodes modelling entities, ii) edges modelling binary relations, iii) predefined "label" and "type" properties for the graph, nodes, and edges, and iv) arbitrary properties of the whole graph, its nodes and edges [11].

Depending on the software comprehension goal, different models of the software are appropriate. Since these goals may be quite diverse, the data needed for each model are also diverse. Our program model allows us to compactly represent a wide variety of aspects of a program, *e.g.*, call relationships between functions or files, inheritance relationships, or contains relationships such as functions belonging to a file or class.

2.1 Nodes and Edges

Our model contains the following nodes:

Free Function Definition node is a function definition in C or a non-member function in C++.

Member Function Definition node is a C++ class member function definition.

Class Definition node is a C++ class definition, including its variable and member function declarations (*i.e.*, forward declarations) and function definitions (*i.e.*, inline functions).

Header File node captures the C/C++ concept of a header file that contains common type and function declarations needed by multiple source files. In particular, a header file may contain multiple class declarations and definitions, forward declarations, function declarations and definitions, and variable declarations.

Source File node captures the C/C++ concept of a source file that typically implements member function declarations, but may also contain free function definitions.

Package node represents the directory path of a header or source file. Header and source files are typically grouped in a way that is reflected by their package structure.

Our model contains the following edges:

File Call edge encodes the calls between source files. This information is a result of aggregating calls between functions.

Header File Contains Class connects a header file and the class definitions it contains.

Header File Contains Definition connects a header file with its source files.

Source File Contains Function connects a source file with its free and member functions.

Class Call edge indicates the calls between class definitions. This information is a result of aggregating calls between member functions.

Class Hierarchy edge represents the inheritance relationship between classes.

Class Contains Function connects classes with their member functions.

Function Call represents the calls between functions.

Package Contains encodes the package and file hierarchy in a software system. A package node contains header and source files, and other packages (*i.e.*, subdirectories).

2.2 Generic Properties

Usually, a program model as described above is not directly useful for program comprehension. In general, further analyses are required in order to convey weaknesses and strengths directly. For our architectural visualization, we compute additional properties from the programs' AST and attach them to the nodes of our model. Our analyses collect a variety of such generic properties, including:

Global Variables. We traverse the AST to check for i) public declared variables and ii) global variables outside the scope of classes. Global variables are a bad programming style and should not appear in object-oriented code.

Lines of Code (LOC). We measure the number of lines of code of each free and member function.

Cyclomatic Complexity (CC). We compute the complexity of each function. CC indicates how much effort is required to maintain a function. We implemented CC according to Li and Henry [10], where functions are weighted according to McCabe's Cyclomatic Complexity Metric. Our implementation counts the possible execution branches in a function for the following branching statements: if, for, while, do-while, and (switch-)case.

Unsafe Function Calls. Due to some aspects of C++ (*e.g.*, unchecked array access, raw pointers), programming errors can lead to low-level buffer overflows, page faults, and segmentation faults. In this analysis, we detect calls to "unsafe" functions, such as sprintf, scanf, strcpy.

Arithmetic Complexity. For each function, this analysis counts the number of arithmetic operations on float, int, float pointer, and int pointer types. Thus, functions and classes with large arithmetic operation counts can be detected. This property is important in scientific code, since such functions should be the most robust and reliable pieces of the software.

Run-Time. This property is attached to functions to reveal information about its actual behavior during execution. Our purpose is to detect functions with high execution times to be more thoroughly inspected.

2.3 Relationship Properties

Relationship properties are analysis results attached to nodes and edges, just like generic properties. However, generic properties express information about a single node (or edge), whereas relationship properties contain information about how nodes relate or interact with each other. Relationship properties we have implemented include:

Class Membership. This analysis annotates member functions with a property about the class it belongs to and the source file it is implemented in. It discovers fragmented member functions, *i.e.*, member functions declared in the same class but defined in different files. The result may indicate bad coding styles or discover refactoring efforts that were applied to split large source files.

Strongly Connected Components (SCC). We detect cyclic dependencies between functions, classes or files. In general, nodes in a cyclic dependency may be merged to reduce call dependencies, and hence to reduce the structural complexity of the system.

Clustering. We detect nodes with similar generic properties. For instance, the user may wish to see all nodes with similar arithmetic and cyclomatic complexity. Our implementation is based on the k-means [5] algorithm.

Note that our program model does not distinguish between generic and relationship properties. Properties are merely a result of program analyses. The mapping between analysis results (properties) and the visualization (properties) is part of the VizzAnalyzer [13] framework (see Section 6). The mapping between program analysis information and information visualization is described elsewhere [11].

3 The Visualization

In a program visualization, the final image should be easy to comprehend. Program comprehension, however, is an individual interactive process of building a mental model of the program to be understood. It is therefore difficult or even impossible to define this process or the visualization necessary to support rapid comprehension.

Many reverse engineering approaches aim to support of improved program comprehension [20, 21, 3], including bottom-up, top-down, and integrated program comprehension approaches. To support all of these approaches and hence the individual construction of a mental model, the visualization tool must aid the user with adequate navigation and interaction capabilities. Therefore, we require our visualization tool to possess these features:

Zoom, Rotate and Pan. These features are traditional techniques. Zooming can take two forms. *Geometric zooming* simply provides a blow up of the graph content, while *semantic zooming* means that the information content changes and more details are shown when approaching a particular area of the graph [6].

Select and Filter. These features allow the user to interactively select and hide (single or groups of) nodes and edges from the current view, or filter nodes and edges by type. Users can thereby reduce the amount of information represented at once (cognitive load) and also focus visually on specific problems.

Single View. Presenting a single visual representation is likely to be better than presenting two visualizations of the same program [16, 17]. A single representation uses less screen space, avoids problems of switching from one representation to the other and of finding the right place in each one. We therefore aim for a single view visualization.

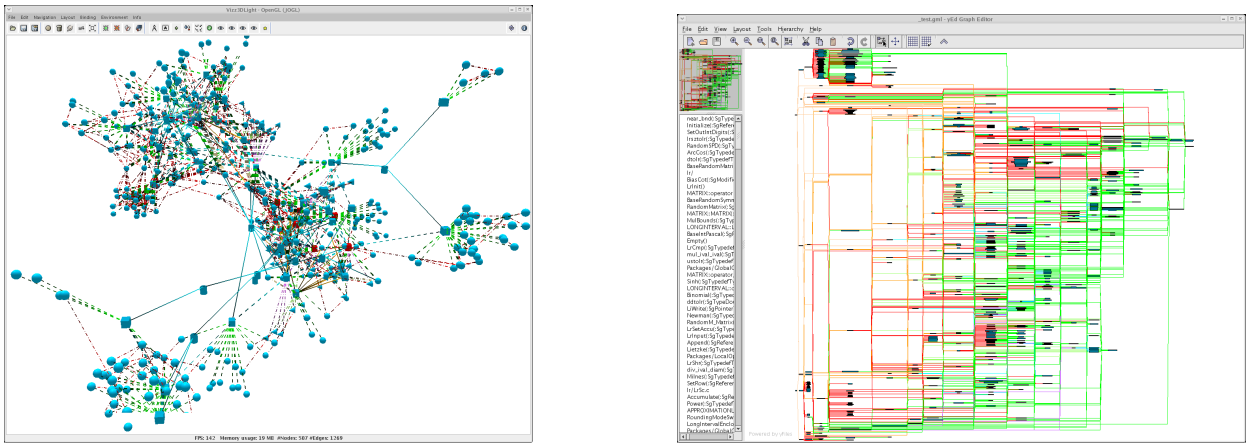
Changeable Metaphor. Metaphors are families of visual objects fitting together. Metaphors, when depicting real worlds and establishing social interaction [4], especially in virtual reality [22], become very important. Therefore, the choice of metaphor is essential to improving the usability of a system. Our tool should allow the user to change metaphors flexibly and easily.

Flexible Representation of Analysis Results. A visualization tool must be able to convey i) generic analysis results, and ii) relationship analysis results. Analyses may result in tremendous amounts of information that must be represented. Our tool should be capable to compress that information and, if necessary, show all results together with the structural program representation.

The most essential part of a program visualization tool, however, is the layout algorithm applied. We seek a layout algorithm with the following qualities:

Focus+Context is a technique allowing a user to focus on a visual detail without losing the visual context. Zooming on a focus, for instance, causes the loss of contextual information and can become a considerable usability obstacle [6]. To simulate focus+context effects, a distortion may be implemented into a layout algorithm itself. User interaction with the view or the layout algorithm parameters may trigger a distortion at a certain view position at run-time.

Predictability means that two different runs of a layout algorithm, involving the same graphs, should not lead to radically different visual representations. This property is also referred to as "preserving the mental map" of the user [12]. Spring embedders are usually not predictable layout algorithms. While hierarchical algorithms improve the situation, they do not scale very well, *cf.* Figure 1.



(a) (b)

Figure 1. Program Visualization (a) SpringEmbedder Layout (b) Hierarchical Layout.

4 SpringCity Visualization

For the visualization of our C/C++ scientific applications, we have chosen the Vizz3D framework, a highly configurable, open-source information visualization engine that was originally developed for the VizzAnalyzer framework (see Section 6).

Vizz3D supports by default i) zoom, rotate and pan - all based on OpenGL, ii) select and filter, iii) a single view, iv) changeable metaphors and v) user defined layout algorithms.

To reduce a user's cognitive load, Vizz3D has a variety of operations, such as geometric zoom, and aggregation and filtering of nodes based on the selected nodes. In addition, users may at run-time filter all nodes and edges of certain types. For instance, just by interactively selecting only certain edges of the program model to be visualized, we can represent a function-call graph, a class-call graph, a file-call graph, a class-inheritance graph, a file-contains graph, a class-contains graph, *etc*., individually or all at once.

With a single-view visualization, we must preserve the users' mental model when interacting with a source code visualization. Therefore, we have applied two essential techniques: a proper metaphor and layout.

4.1 The Metaphor

Many graphic designs lack an intuitive interpretation, requiring that a user be trained to understand them. Metaphors found in nature or in the real world avoid this problem by providing a graphic design that the user already understands. When illustrating a reverse engineered architecture, it is important for the understanding of a program that the final picture is adjusted for the individual [8, 19]. Therefore, we have chosen the city metaphor [9, 15] to increase individual program understandability.

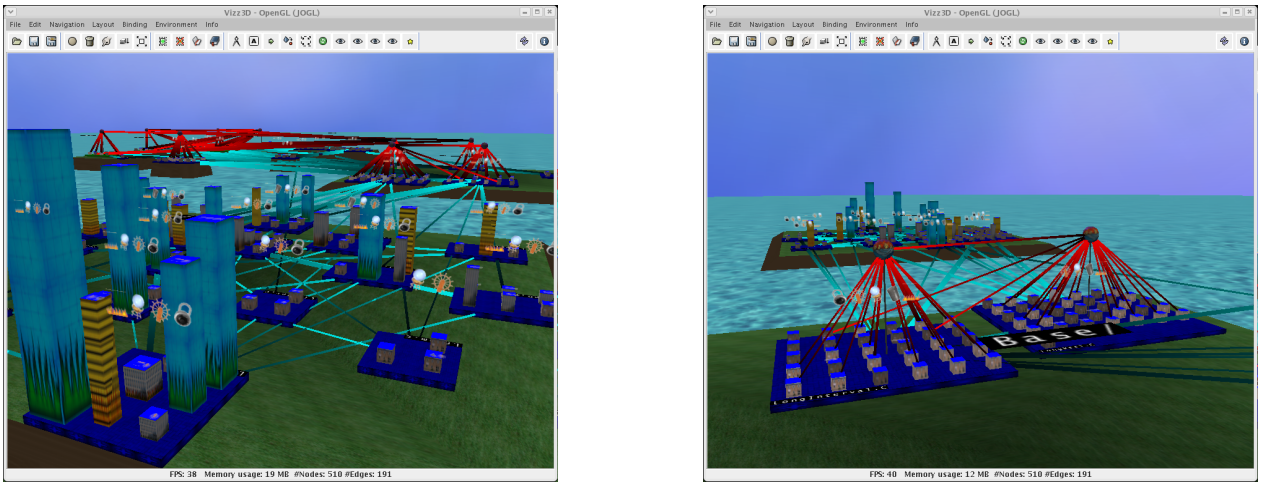
We map program model entities to visual entities as follows: *Buildings* represent functions. The type of building (texture) depends on the LOC of each function. To improve visual contrast, we distinguish four different textures as shown in Figure 2 a). Free functions have a slightly lighter texture tone than member functions. *Pillars* represent class definitions as shown in Figure 2 b). *Water Towers* (spheres), being held by pillars, represent header files. *Cities* (plates) indicate source files. *Landscapes* carrying cities and water towers represent packages; see Figure 5.

In Vizz3D, metaphors either by using binding functions to change how program model information maps to visual information [11], or directly within the layout algorithm.

4.2 The Layout

The Vizz3D framework permits easy extension of layout algorithms. We have used this facility to develop a new layout algorithm that we refer to as *SpringCity*.

Our layout algorithm combines a spring embedder algorithm with a hierarchical visualization. Initially, because force-directed layout algorithms can in general be rather slow, we calculate the forces for coarse-grained nodes first, *i.e.*, for header files, source files, and packages. Figure 3 a) shows the layout of 46 files (including the packages). The edges represent Package Contains relationships. Entities belonging to the same package, representing compo-



(a) (b)

Figure 2. Program Visualization (a) C Program (b) C++ Program.

nents by design, are laid out close together. The size of the source files reflects the number of functions defined within them. Invisible in the image are the call edges between files that are also used for the force directed layout. Figure 3 b) shows the same graph, except that functions (buildings) are made visible. Buildings are laid out compactly next to each other, *i.e.*, they are not part of the force directed layout.

We modified the spring embedding layout algorithm by Huang and Eades [7]. Our alteration is based on the C++ types of nodes and edges described in Section 2. The layout itself is performed within 2D space.

The second step is to apply the landscapes (package structure) for the cities (files), *cf.* Figure 3 c) and Figure 3 d). The height of the landscape (y-axis) is represented by the depth of the package path. Therefore, cities or files in a deeper directory structure are represented on a higher hierarchical level. As a result of the spring embedder, packages containing sub-packages are laid out closely. As sub-packages are on a higher hierarchical level, sub-packages produce "visual mountains". This approach can be compared with 3D treemaps [2] where usually directory structures are represented to the user in a hierarchical way.

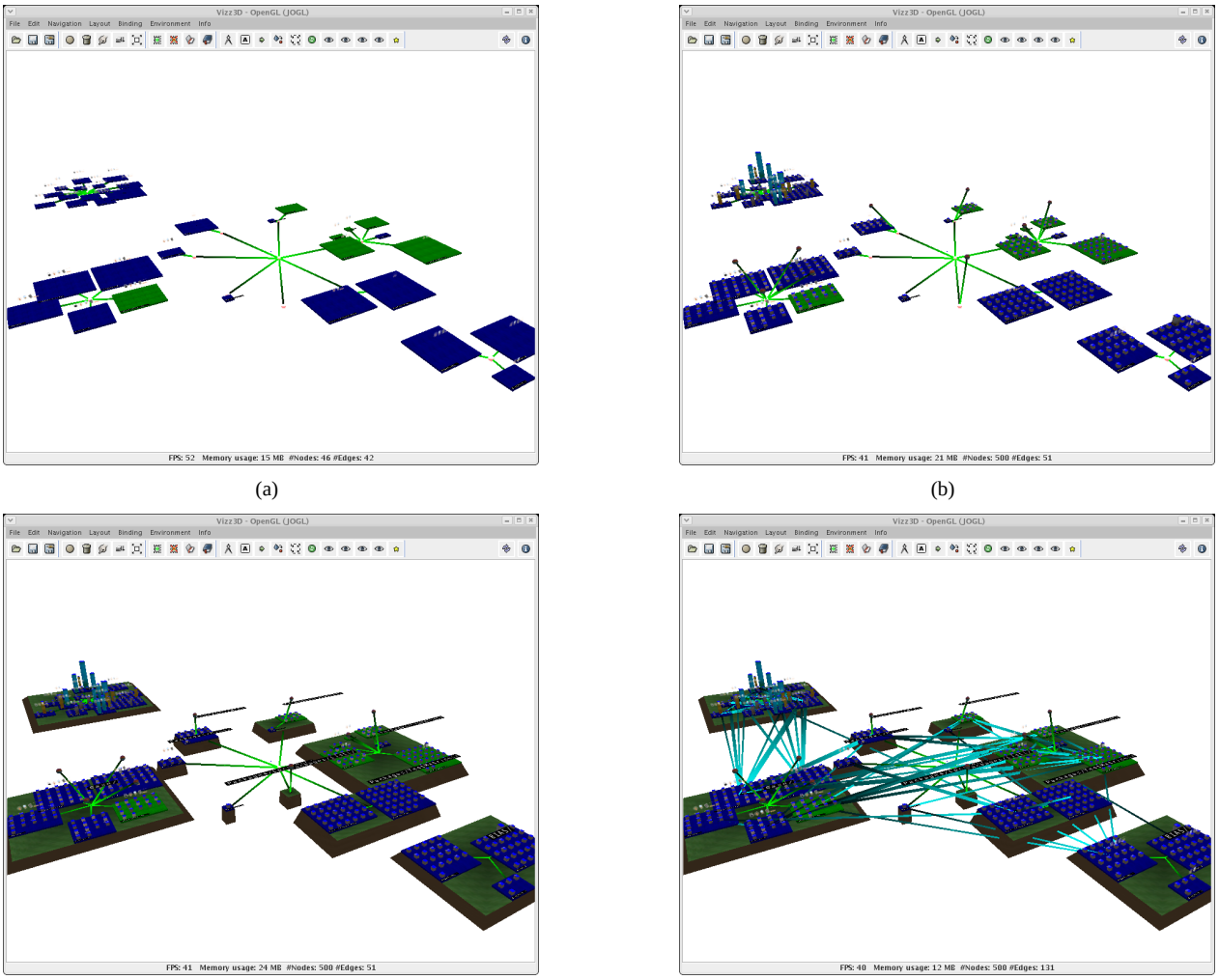
As an optional third step, the user may layout the entire visualization at a finer granularity. In particular, after the first two algorithm have been performed, we allow the application of a force based layout algorithm on all pre-calculated function definitions. This approach spreads the layout space out, *cf.* Figure 5. However, this visualization results in overlapping nodes, since the forces are chosen to be small in order to keep a compact view of the entire source. Figure 4 a), for instance, represents the center city of Figure 5. Although one might think that all functions are represented from this city, Figure 4 b) shows that they are not. Indeed, Figure 4 b) shows our application of focus+context implemented within the layout algorithm. The layout is recalculated for the nodes that are close to the camera and hence focus+context is user interactive.

Note that Figure 3 represents the same program as Figure 1. One can argue which image is easier to understand. However, we favor Figure 3 because it uses a familiar metaphor, and hence remembering certain structures within different layout runs of the same program is simpler. The hierarchical visualization of packages, in our opinion, is a major aid to rediscover certain elements of the visualized program.

4.3 Representation of Analysis Results

Our visualization also supports the flexible representation of analysis results, for both generic and relationship results. Since we support only one view, we prefer to display all analysis results within that view. Information overflow is an immediate concern. To overcome this problem, we display generic program analyses with 2D *semiotics* [16] within our 3D scene, a common convention in game development.

To visualize the generic properties of our program model, we illustrate 2D icons on top of each building and above each city (to indicate buildings with a certain property exist in this city). Our icons are, *cf.* Figure 2: *Wheel Icon* indicates that global variables are being accessed from a function (building). *Globe Icon* indicates that global variables are being accessed within a source or header file. *Lock Icon* shows that unsafe function calls are being used within a function (or source file). *Sad Smiley Icon* indicates cyclo-



(a) (b) (c) (d)

Figure 3. SpringCity Layout Algorithm: (a) layout algorithm between files in progress (b) displaying in addition all functions and member functions (c) displaying the package structure of the files (d) displaying the file call graph.

matic complexity exceeds some threshold. This icon is represented above cities (source files) and in addition as a texture on houses (functions). The threshold can be adjusted at run-time with the layout dialog. *Math Icon* indicates arithmetic complexity of a function above a threshold.

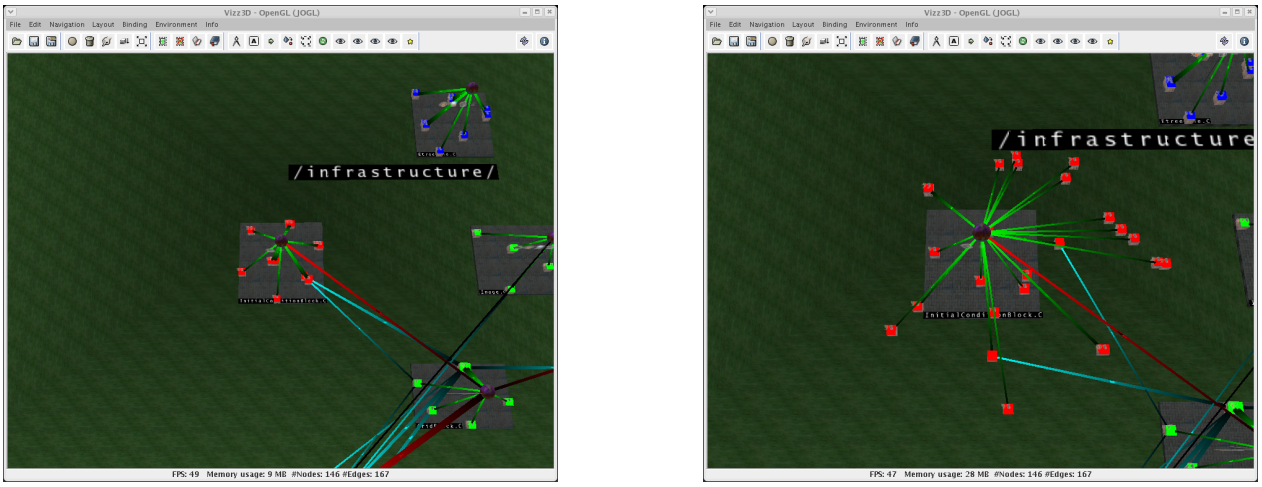
Some generic properties are better represented as visual properties such as height, width, depth, texture, color, among others. Many variations are possible and Vizz3D allows for a rapid re-mapping of visual properties through the specification of mappings defined in XML.

We map the remaining generic properties as follows: *Height* of buildings represents the LOC of a function. *Width and Depth* of buildings indicates the amount of time spent in each function. The wider a building is, the more frequent a function is called at run-time.

So far, we have not used the visual property color. Color is a very strong visual property, which we reserve for indicating relationship properties. Hence, only one relationship property can be shown at a time. We discuss analysis and visualization of relationship properties in Section 5.

5 Experimental Results

We visualized several scientific C/C++ codes developed at the Lawrence Livermore National Laboratory. In the following, we discuss the visualization and interpretation of relationship analysis results for these codes using our



(a) (b)

Figure 4. Illustrating Focus+Context (a) Simple Layout (b) Layout with Focus+Context.

SpringCity layout algorithm and metaphor.

5.1 Class Membership

This analysis discovers fragmented member functions, *cf.* Section 2.3. Our aim is to investigate how developers use header and source files when developing code. Indicators of poor class membership are:

Spread Member Functions. Member functions are spread throughout various source files. A good coding style would define all member functions in the same source file. C++ does not enforce this rule. It may even make sense to break this rule, *e.g.*, when refactoring is necessary. Nevertheless, visualizing spread member functions allows a reverse engineer to understand design decisions and detect bad coding styles. Spread Member Function detection helps also to improve the componentization of software systems. Figure 5 shows an example of this analysis. The *Class Membership* relationship property assigns the same color to each member function of a class. In the lower right part of Figure 5, the package "/wpp3D/" contains four source files and one header file (sphere). The header file contains one pillar (class) that contains a couple of member functions. All member functions of that class are colored red (red roof). We see one member function is defined in a different source file. Most likely, this function has been refactored out.

Multiple Class Declarations. In C++, multiple classes may be declared within one header file. Visually, this is indicated by multiple pillars below a water tower. In future work, we will study the advantages or disadvantages of this coding style for the componentization of source code.

5.2 Strongly Connected Components

Cyclic dependencies between files are shown in Figure 6. The dependencies are colored in green (blue is default). Interestingly, all but one source files are in the same package, suggesting the outlier may be in the wrong package. This suspicion is even stronger when the file call edges of the outlying source file are observed. The image reveals, however, no cyclic dependencies between functions (buildings).

5.3 Clustering

We also clustered functions according to different properties, such as arithmetic complexity, cyclomatic complexity, and LOC. Functions with similar properties, *i.e.*, in the same cluster, were colored the same [1] Clustering suggests ways to re-engineer a system that could lead to a better componentized structure.

6 Tool Setup

We built our visualization system by combining the tools shown in Figure 7. We use ROSE [18] as the C++ front-end. ROSEVA extends the ROSE API for use within the VizzAnalyzer framework [13, 23]. ROSEVA has two main interfaces, one for the parsing of C/C++ code (source retrieval and AST construction) and one for high level analyses. Since ROSE and ROSEVA are developed in C/C++

[1] We chose only four clusters, and so did not run out of colors.

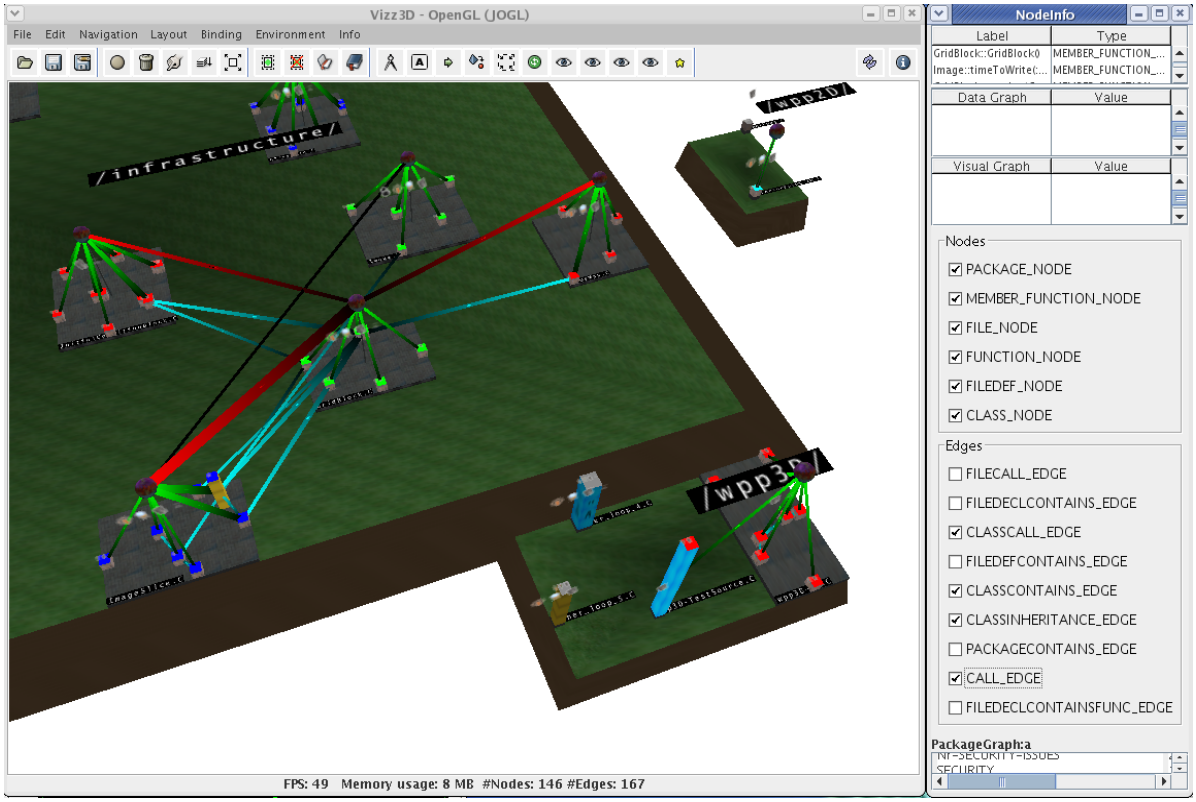


Figure 5. Class Membership Visualization.

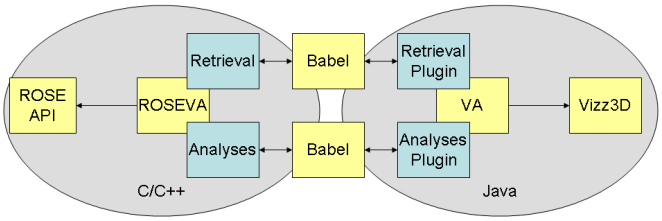


Figure 7. Architectural Setup

and VizzAnalyzer in Java, we use Babel [1] to connect these worlds. Finally, we use Vizz3D [11] to visualize the results.

6.1 ROSE

ROSE is an open infrastructure for building compiler-based source-to-source analysis and transformation tools. For C and C++, ROSE fully supports all language features, preserves all source information for use in analysis, and permits arbitrarily complex source-level translation via its rewrite system. Although research in the ROSE project emphasizes performance optimization, ROSE contains many of the components common to any compiler infrastructure, and thus supports the development of general source-based analysis and transformation tools.

6.2 ROSEVA

ROSEVA has been designed as a library using the ROSE API. ROSEVA has two interfaces that VizzAnalyzer accesses via Babel: *retrieval* and *analysis*. The retrieval takes the files to be parsed by ROSE as input and constructs internally a AST representation of the C/C++ source files. On user interaction, selected graphs are returned to the VizzAnalyzer [2]. These results can be visualized directly or further inspected with additional analyses.

Therefore, ROSEVA provides a second interface allowing VizzAnalyzer to access a variety of analyses on any graph produced by the frontend. Analyses results, such as CC or LOC, are fed back directly as properties to the VizzAnalyzer. These properties are used by the VizzAnalyzer to flexibly visualize any kind of program information.

[2] One such graph is the program model described in Section 2.

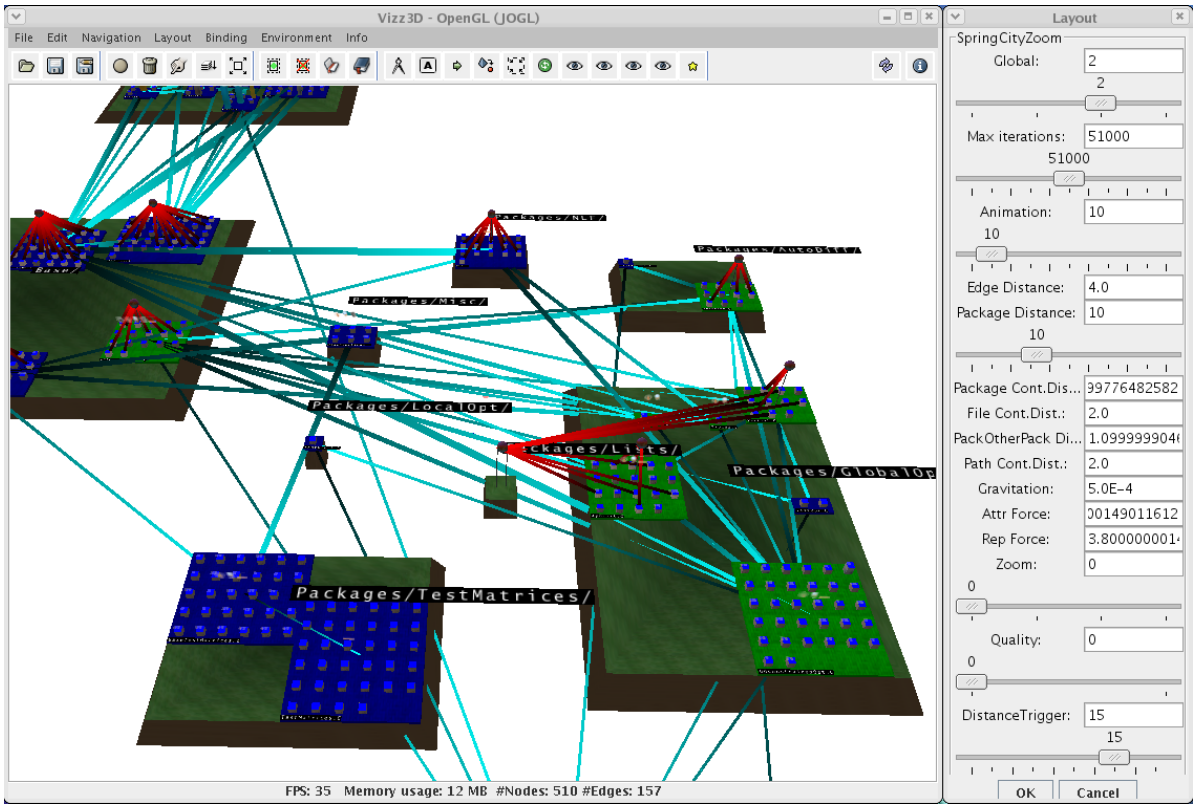


Figure 6. Strongly Connected Components Visualization.

6.3 Babel

Babel is a tool for mixing C, C++, Fortran77, Fortran90, Python, and Java in a single application. Babel is the foundation for a multi-language scientific component framework. Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques. In particular, Babel uses a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components.

6.4 VizzAnalyzer

The VizzAnalyzer framework is our composition system for reverse engineering, supporting the rapid composition of individual software reverse engineering tools by reusing arbitrary reverse engineering components. VizzAnalyzer distinguished two domains: Program retrieval and analysis are part of the *software analysis domain* and program visualization is part of the *information visualization domain* [13]. Each domain operates on its own program model. For instance, a model for software analysis may contain information about a program's clusters, metrics, cycles, *etc*., while a model for information visualization contains information about the visualization of a program, such as the position or color of entities. VizzAnalyzer allows the merging of tools from both domains.

6.5 Vizz3D

Vizz3D is a 3D information visualization system. It presents system structure and quality information to a user in a comprehensible way and leverages the understanding of that system. Vizz3D is highly flexible and allows users to define and re-assign layout algorithms and metaphors at run-time. Hence, visualizations can be online-configured. This enables also an interactive and iterative software analysis, where appropriate views are created on demand.

7 Conclusion

In this paper, we present our efforts to visualize large-scale scientific C/C++ applications. We have developed the

SpringCity layout algorithm for the Vizz3D visualization framework supporting the interactive visualization of different levels of details within one view, layout predictability, and compressive visualization of analysis results. For architectural representation, we have chosen the city metaphor, allowing us to visualize program information together with its file structure information.

We believe that our visualization approach allows software developers, designers, maintainers and managers to get a rapid and comprehensive overview of a C/C++ software system. Certain weaknesses of a software may be discovered right away, such as unsafe function calls or complex implementations, other weaknesses must be explored interactively, *e.g.*, good componentization and file affiliation.

We are extending ROSE to support a whole-program representation, to better support visualization of truly large-scale C/C++ applications. ROSE itself can compile several million line applications on a file-by-file basis. However, Vizz3D requires a complete AST representation of an entire software system to create its program model. We are enhancing ROSE to support a memory efficient whole-program AST representation. This is our current limitation to visualize software systems with more than approximately 1.000.000 AST nodes or 50 source files (not counting the header files) or 10.000 LOC.

References

[1] Babel. Available at: `http://www.llnl.gov/casc/components/babel.html`, July 2006.

[2] T. Bladh, D. Carr, and J. Scholl. Extending tree-maps to three dimensions: a comparative study. In M. Masoodian, S. Jones, and B. Rogers, editors, *6th Asia-Pacific Conference on Computer-Human Interaction (APCHI 2004), New Zealand*, June 2004.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann Publishers, 2003.

[4] C. R. dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. Paris. Metaphor-aware 3d navigation. In *IEEE Symposium on Information Visualization*, pages 155–65. Los Alamitos, CA, USA, IEEE Comput. Soc., 2000.

[5] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Arnold, 2001.

[6] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.

[7] M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *6th Int. Symposium on Graph Drawing*, pages 374–383. Springer LNCS 1547, 1998.

[8] J.F.Hopkins and P.A.Fishwick. The Rube Framework for Personalized 3D Software Visualization. In *Software Visualization. International Seminar. Revised Papers (Lecture Notes in Computer Science Vol.2269). Springer Verlag, pages 368-380. Berlin, Germany*, 2002.

[9] C. Knight and M. C. Munro. Virtual but visible software. In *IV00*, pages 198–205, 2000.

[10] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proceedings of the 1st International Software Metrics Symposium*, May 1993.

[11] W. Löwe and T. Panas. Rapid Construction of Software Comprehension Tools. *International Journal of Software Engineering and Knowledge Engineering*, December 2005.

[12] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Tree visualisation and navigation clues for information visualisation. *J. of Visual Languages and Computing*, 6:183–210, 1995.

[13] T. Panas. *A Framework for Reverse Engineering*. PhD thesis, Department of Computer Science, Växjö University, Sweden, December 2005.

[14] T. Panas. Vizz3D. Available at: `http://vizz3d.sourceforge.net`, July 2006.

[15] T. Panas, R. Berrigan, and J. C. Grundy. A 3d metaphor for software production visualization. In *IV03*, London, UK, June 2003. IEEE.

[16] G. Parker, G. Franck, and C. Ware. Visualization of large nested graphs in 3d: Navigation and interaction. *Journal of Visual Languages and Computing*, 9(3):299–317, 1998.

[17] M. Petre, A. Blackwell, and T. Green. Cognitive questions in software visualization. *Software Visualization: Programming as a Multimedia Experience*, pages 453–480, January 1998.

[18] D. Quinlan, S. Ur, and R. Vuduc. An extensible open-source compiler infrastructure for testing. In *Proc. IBM Haifa Verification Conference*, volume LNCS 3875, pages 116–133, Haifa, Israel, November 2005.

[19] S.North. Procession: Using Intelligent 3D Information Visualization to Support Client Understanding during Construction Projects. In *Proceedings of Spie - the International Society for Optical Engineering, vol. 3960, p. 356-64. USA*, 2000.

[20] M.-A. D. Storey, F. D. Fracchia, and H. A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 17, Washington, DC, USA, 1997. IEEE Computer Society.

[21] S. Tilley. A Reverse Engineering Environment Framework. Technical report, cmu/sei-98-tr-005, Software Engineering Institute–Carnegie Mellon University, Pittsburg, PA 15213, April 1998.

[22] K. Vaananen and J. Schmidt. User interfaces for hypermedia: how to find good metaphors? In *CHI '94 conference companion on Human factors in computing systems*, pages 263–264. ACM Press, 1994.

[23] VizzAnalyzer. Available at: `http://www.arisa.se/`, 2006.