2007

Visualization `a la Unix

Hans-Peter Bischof

Tim Peterson

Visualization à la Unix™

Hans-Peter Bischof (hpb [at] cs.rit.edu)
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
One Lomb Memorial Drive
Rochester, NY 14623-5603, USA

Tim Peterson (tjp4434 [at] cs.rit.edu)
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
One Lomb Memorial Drive
Rochester, NY 14623-5603. USA

Abstract ***A visualization framework can be seen as a solution to a specialized data flow problem. Spiegel[1] is a visualization framework which uses the Unix pipeline model to execute programs that visualize scientific data. A visualization program in Spiegel is constructed out of simple components with communication endpoints which can be connected together. Spiegel provides a graphical programming environment, which can be used to write programs using these components. This paper describes the language used to define a Spiegel program, the graphical programming environment, and the Spiegel architecture.***

Keywords: Visualization framework, graphical programming environment, data flow

[1]Spiegel is the German word for mirror. Like the mirror in a telescope helps to observe the universe, Spiegel helps to observe the simulation of objects in space.

1 Visualization Environments

Visualization systems can be broadly categorized as closed systems, like IRIS Explorer [**?**], or toolkits, like that described in [**?**]. Applications in the first group can be extremely powerful and can produce excellent visualizations. However, it is difficult to impossible to add desired functionality that is not already available. For example, it is very difficult to follow a specific particle in IRIS Explorer with a camera.

Toolkits address the final step in the visualization process, rendering output, but they are of no help as a framework to create complete visualizations, which require operations such as data pre-processing and changing the phase space of data.

Most applications use one visualization toolkit, which is sometimes not extensible, such as OpenGL [**?**] or VTK [**?**]. As a consequence of this, toolkits which are better suited for the problem cannot be used.

In this paper we will present an extensible visualization framework, which allows using and combining the best of both worlds. The framework allows the

use of the best suitable visualization toolkit and it provides an environment which allows creating highly specialized visualizations.

2 Introduction

Unix™ [?] was born in 1969, and became an immediate success amongst developers and researchers. One of the main reasons for the success of Unix amongst developers was the fact that powerful new commands could easily be developed by piping data through a set of commands. Also, adding new components was easy, as no special code must be developed in order for a new component to participate in the pipeline stream. The operating system takes care of the communication mechanism between the components. The trick now became to write components on the one hand small enough to solve a specific problem, but on the other hand configurable enough so that they could be used in multiple ways. Command line options were used to influence the behavior of the commands; for example, *sort*'s $-n$ option forces the output be sorted in numeric order rather than in lexicographic order. Shell scripts can be stored in files, and, if written correctly, they can be executed on every Unix operating system which has the required shell and commands. Neither the shell nor the programs used in the shell script need to know how to communicate or how to be executed. The environment, i.e. the Unix operating system, takes care of such details as $fork$, *pipe*, signals, etc. [?]

A visualization program in Spiegel is constructed out of small, simple components whose communication endpoints are connected together. Each component provides one or more typed communication endpoints. The functionality of each component can be fine tuned using arguments. New components can be easily added to the pool of usable components by creating a Java class in an appropriate directory, similarly to adding a component to a shell search path. The program is then read by a runtime environment and executed, similarly to how a Unix shell executes a Unix pipeline. The result of a visualization done with Spiegel is a series of two or three dimensional images that can be combined into a movie, a process that can also be done within the Spiegel runtime system.

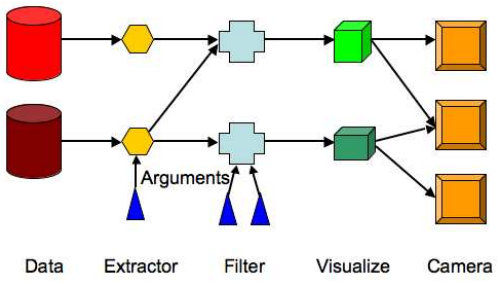

Figure 1: Spiegel Pipeline Architecture

The next chapters describe the concept of the Spiegel pipeline architecture and the programming language it implements, followed by a chapter describing how new components can be added to the system. The paper concludes with a comparison of Spiegel with other existing systems.

3 Data Flow Architecture

In a Unix environment, the output of a program can only be piped to one other program, and a component has typically one input channel (*stdin*) and one output channel (*stout*). *popen*() connects the *stdout* stream with *stdin* and the Unix operating system takes care of the buffering and the execution of the processes.

In a visualization system it is very often the case that output of one component must become the input of many components and that a component can combine multiple inputs. Therefore, a Spiegel program can be represented as a directed acyclic graph. A data flow graph in Spiegel can be divided into four functionality groups: extracting of data, filtering of data, converting data into visual components, and finally displaying or storing the data.

Figure 1 shows the overall data flow structure.

The four data flow layers:

- Extraction Layer: An *extractor* component reads the data from the source, which is typically a file or a network communication end-

point. An *extractor* knows the structure of the input stream.

- Filter Layer: A filter *component* filters only the objects out of the pipeline stream which are to be considered in the visualization. Typically several specialized filters are used to extract the desired information. A filter can also convert the data into a different phase space.

- Visualization Layer: A *visualizer* component converts the data stream from the filters into a structure which can be visualized.

- Display Layer: A *display* component finally creates a visual. The visual can be a shown on a 2D monitor or in a 3D projection system, or stored to disk for later processing.

The data may flow through many components in each category in order, but data typically does not flow backwards. Unlike Unix, each input and output channel has a data type assigned with it, and only matching communication channels can be connected.

3.1 Hello World

The graphical programming environment makes it easy to create programs. The resulting program can be stored and executed later on. A basic "Hello World" program may look like Figure 2. The Extractor *Stars* reads the data from the file *theSimulation.sim* and sends its result to the visualizer *Stars3D*. *Stars3D* converts the data into a visual scene, which is then sent to the display object *Camera3D*. *Camera3D* shows a visual representation of the scene on a screen. The clock sends out a time value to the extractor, which triggers the evaluation of *Stars*, which results in an update of each component in turn, until finally *Camera3D* displays the resulting image.

3.2 Extracting and Using Data

In a Unix environment each component evaluates its arguments before any data processing occurs. A simulation, which creates the input for a visualization,

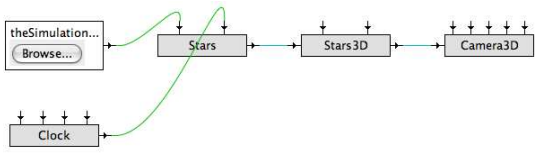

Figure 2: Hello World à la Spiegel

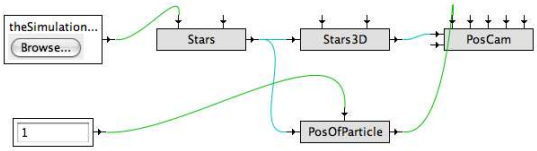

Figure 3: Arguments reevaluated

has a notion of time. The data for each time step represents the state of the model at a given moment in time. The the extractor for a visualization has to read the data for a given moment in time and send it along its output channels. This invalidates the receiver's input data, which then triggers a re-evaluation of the receivers arguments, before the receiver processes the incoming data. For example, if a camera should follow a specific particle, we need to know where the particle is, and send the position of the particle to the camera before the camera renders a new image.

This technique is used in Figure 3 to center the camera at the position of particle 1. The extractor *Stars* send the data to the visualizer *Stars3D* and the *PosOfParticle* component. *PosOfParticle* reads its argument, which is 1, and sends out the position of particle 1 as an argument to *PosCam*. This centers the scene at the position of particle 1.

4 Distributed Computing

The visualization process of scientific data can be very CPU- and/or memory-intensive. For example, the density distribution calculation of galaxies has the complexity of $O(n^2)$ [**?**]. Simulations of merging galaxies used in this project contain up to 10^6 particles, which means that on the order of 10^{12} calculations are needed to determine the density distribution

```
cat bigFile | ssh biggie /bin/sort |
ssh biggie /bin/compress > x.z
```

Figure 4: Using *ssh* in a Unix Pipe

for one time step. It is obvious that a visualization system would greatly benefit if the components could be distributed over a network.

Every program in a Unix pipeline reads from *stdin* and writes to *stdout*, even if the process is executed on a remote machine via *ssh*. The operating system takes care of all the communications needs, making the actual data transport mechanism transparent to the programs. Pipeline sequences like those seen in Figure 4 are used quite often.

Spiegel uses a similar idea. Let's take a look at figure Figure 5. $BottomHalf$ gets as arguments values for which remote host to use ($fast.cs.rit.edu$) and then a volume description ([0-1][0-2][0-2]) that specifies that it is to compute on only half of the volume (0-1). The execution of the component takes place on the remote host, and the underlying communication details are taken care of by the Spiegel runtime environment. The second part of the cube is used on $TopHalf$ and calculated on $fix.cs.rit.edu$. The results are combined on the local machine. The developer of a component does not know nor care where the component is actually executed.

The Spiegel runtime environment takes care of this in a similar way to how Unix does, with one exception. If the sender and the direct receiver are executed on the same remote host, then the communication stays completely on the remote host. The pipeline in Figure 4 would send the data back from *biggie* and then to *biggie* again. This would be very inefficient for a visualization system. The complete graph needs to be known before execution in order to optimize the data flow.

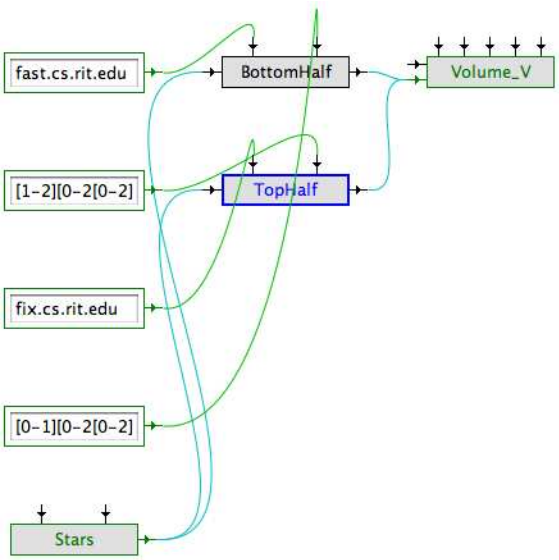

Figure 5: Spiegel Components are executed Remotely

5 Graphical Programming Environment

The graphical programming environment (GPE) is the part of Spiegel which is used to create visualization systems. The GPE allows one to select components and connect matching input and output communication endpoints. The GPE also allows configuration of component-specific parameters, such as the position of a camera, the visible size of a star, etc. The constructed program can be stored in a file. A video of how to use the GPE can be found on the web [**?**].

Every useful programming environment must be capable of creating abstractions, i.e. creating procedures and using them. Spiegel's GPE allows the creation of functions which can be reused later on. A procedure can be created using the GPE and then stored as a file. Figure 6 shows the function $PosCam$. The components appear inside the box representing $PosCam$, with marks indicating the available connections to outside components. It is important to notice that the input arguments and the input and output connections are connected to the inner compo-

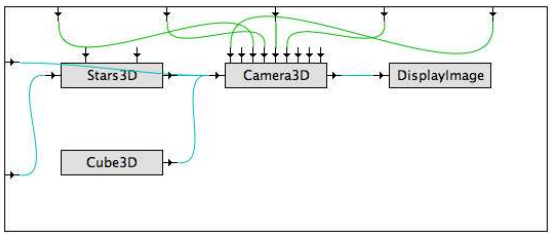

Figure 6: Procedures

nents, exposing arguments of the inner components to outside configuration. This allows the user to tailor procedures for a very specific use.

The current developed GPE supports:

- Layout: All components are automatically placed on the screen in an optimal position. We use the Sugiyama [**?**] layout algorithm to produce readable graphs. This algorithm works great, if the components and connections are constant. This is not the case during the creation of a program, as the user will add and remove components and connections until the program is complete. Therefore, this was provided for by making changes in component positions be done smoothly so that the user can follow the changes.
- Hiding: Components that are not being actively worked with can be hidden, resulting in a less cluttered screen.
- Grouping: Components which make only sense as group can be grouped together as one screen object in order to save screen space.
- Functions: The GPE can be used to define functions which can later be used in other programs. The system supports defining which communication endpoints of the inner components are visible outside the function. Functions can also have components which are themselves functions.

Each graphical Spiegel program has a textual representation. This program can be stored as a ASCII text file and modified with a text editor. It is also possible to create a visualization system using a text editor or other tools.

A Spiegel program can be loaded during startup of the system and immediately executed. The runtime system creates the required components and communication links.

6 Components

A Spiegel component has input and output communication endpoints, argument inputs, and it defines a set of commands it understands.

More generally, a component for the Spiegel visualization framework has

- k typed data input communication endpoints, 0 ¡= k
- l typed data output communication endpoints, 0 ¡= l
- m typed argument inputs
- 1 command channel endpoint

The simulation providing input can create a stream of data representing the state of the system as a function of time, and it can create a stream of data representing interesting events. We call the data stream d-stream and the event stream e-stream. A component typically receives data from one or more input streams, processes the data, and sends the data out on one or more output streams.

Commands are sent to each component, for example to change a location, through the command channel. Each component can process the command, or discard it if it is not required. This allows, for example, to focus all cameras with one command onto a specific location. The command channel can be used by a user, the simulation, or the components, to modify the behavior of the visualization system. Figure 7 show a schematic representation of the communication paths.

7 Language

A program for the Spiegel visualization framework consists of creating and connecting components, as

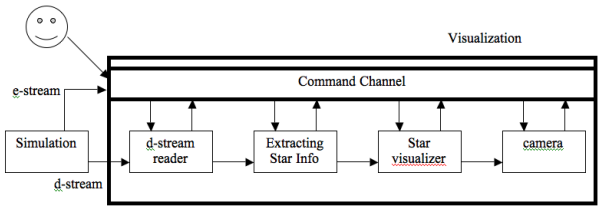

Figure 7: Spiegel Pipeline Architecture

well as commands to send to the components. We preferred to simplicity over complexity as we designed the program.

For example, the program in Figure 6 can be described using the program shown in 8

A procedure is defined in a very similar way as the program shown in 6. Components are created and connected in the same way, but additionally we have to define which connections on the internal components can be connected to externally by the user of the component. This is done by the use of the keywords *connect Container*. The statement *Container output size to Camera3D input size* creates a input communication endpoint to the procedure, named *size*, which is connected to the *Camera3D* input named *size*. This technique allows the use of procedures within procedures.

8 Conclusion

Spiegel is a visualization framework that provides the means for communication in a data-flow approach to data visualization. A language defines a visualization program, which is then executed by the runtime environment. It is very easy to add new components or use different visualization toolkits. Bethel states in [**?**] that interoperability of visualization software and data models is not an achievable goal. We believe have shown that it is.

Acknowledgments.

We would like to thank Dr. David Merritt and his research team and Dr. Campanelli and her research

```
   # a new object of type File will
   # be created, and the object
   # is called File0
new spiegel.plugins.input.File File0
   # the command set will be
   # sent to File0 with the
   # filename  theSimulation.sim
File0 set file theSimulation.sim

   # a new object of type Stars3D0
   # will be created, and the
   # object is called Stars3D0
new spiegel.plugins.visual.Stars3D0 Stars3D0

   # a new object of type Stars3D
   # will be created, and the
   # object is called Stars3D0
new spiegel.plugins.extractor.Stars Stars

   # a new object of type Camera3D
   # will be created, and the object
   # is called Camera3D0
new spiegel.plugins.util.UserFunction Camera3D0

   # the output named stars of
   # the object Stars0 is
   # connected with the input
   # named stars of the Stars0 object
connect Stars0 output stars to Stars3D0
input stars

..
```

Figure 8: Textual Program

team for their support of our research efforts.