AN OBJECT-ORIENTED APPROACH TO DEVELOPING VISUALIZATION TOOLS PORTABLE ACROSS DESKTOP AND VIRTUAL ENVIRONMENTS

BY

PAUL JOHN RAJLICH

B.S., University of Illinois, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

Dedication

I would like to dedicate this work to my wife Cindy and our unborn child. The adventure is only beginning...

Acknowledgements

I would first like to acknowledge my advisor, Dr. M. Pauline Baker, for her strong support of my work and for giving me the opportunity to work for NCSA. I thank Prof. Richard M. Crutcher for his generosity in supporting this project.

I would also like to thank the members of the Visual Interactive Computing Environments group here at NCSA for their help, advice, and company.

Contents

Chapter 1

Introduction

Visualization tools come in a variety of different forms. Most scientists make use of visualization tools that run on their desktop graphics workstations. The popularity of the Web and the emergence of standard multimedia types such as VRML and MPEG have increased interest in providing visualization capabilities that can be accessed remotely using Web browsers. And new virtual reality facilities, such as the CAVE, invite exploration of how virtual environments can be utilized for visualization.

In current practice, visualization tools built for these different environments are platform specific. Although the tools are similar in functionality and use some of the same libraries, they often have separate code bases and still have to be maintained separately. CAVE applications are usually developed specifically for the CAVE. As a result, these applications are not easily transferred to the desktop. We decided to try to design a CAVE tool in such a way that it could be easily transferred to the desktop. The corresponding desktop tool would have many of the same objects as the CAVE tool. This would allow simultaneous development and maintenance of both tools without much added effort.

The bulk of the work deals with finding consistent, cross-platform ways to deal with the user interfaces. The interfaces to CAVE tools and desktop tools are inherently different. A lot of this is attributed to the fact that the CAVE has special input devices, including a motion tracker. In other research areas, such as 2D desktop gui frameworks, a lot of work has already been done so that interfaces are platform independent. Between the CAVE and the desktop, it's more complicated. It's no longer just a matter of different widgets, there are different forms of interaction in the two environments.

In order to accomplish this task, I am making use of object oriented techniques (design patterns). This essentially involves making abstractions for things that are common between different user interfaces and encapsulating the things that may change. Encapsulating things that may change facilitates reuse since the impact of any change will be localized.

This work presents the beginnings of an object oriented framework that facilitates the creation of portable and extensible custom visualization tools.

The framework emphasizes an approach to visualization that facilitates taking existing software and adapting it to different environments. It allows custom visualization tools to be built quickly by integrating components from different packages using object oriented techniques (design patterns). This is in contrast to the current trend of building custom visualization software from scratch for every new application.

Using this framework, a specific tool is created. This tool demonstrates how the framework architecture is applied to a particular application and the impact of the framework's design decisions. Designing a framework is an iterative process. More tools will be developed using this framework, helping it to evolve and become more reusable.

Chapter 2

Scientific Visualization

"The purpose of computing is insight, not numbers." [8]

Due to the rise of computational science [15], many of today's computational efforts produce enormous amounts of data. Humans are not capable of processing such quantities of numbers. This presents a new crisis: how can we produce insight rather then just numbers? [9] Visualization takes advantage of the highly developed human visual system. The human visual system is a massively parallel system that is an integral part of everyday life. Visualization works under the premise that data can best be translated to human insight by being cast as pictures. [9]

Scientific visualization is the use of data-driven computer graphics to aid in the understanding of scientific information. [15]. However, scientific visualization is more than just computer graphics. It is the process of selecting and combining appropriate graphical representations and packaging this information in understandable presentations. [15]

Given the benefits of scientific visualization, it's not surprising that many scientists are exploring all the different ways that they can visualize their data. With the advent of the CAVE, which was designed specifically with scientific applications in mind, additional opportunities have surfaced. The use of virtual reality facilitates more natural interaction techniques than are found on the desktop.[11][10][1] In addition, virtual reality enhances 3D understanding by allowing viewer-centered head-tracking perspective and stereopsis. Virtual reality also facilitates multisensory interfaces. These multisensory interfaces can strongly reinforce the visual illusions being presented. [9]

However, visualization is hard work [9]. It's hard enough to get visualization applications developed, let alone worry about portability and reusability issues. As a result, many of these issues have been overlooked in the past. This work takes a look at some of these issues and attempts to take a step in resolving them.

Chapter 3

Environment

We begin by introducing the environment in which this work was developed. The environment consists of the CAVE and two toolkits. The first toolkit is a visualization toolkit called VTK. The second toolkit is a rendering toolkit called Performer.

3.1 The CAVE Environment

This section describes the CAVE Virtual Reality environment here at NCSA. CAVE is a recursive acronym that stands for CAVE Automatic Virtual Environment. The CAVE is a projection-based VR display that was developed at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago. It premiered at the SIGGRAPH '92 conference.

In [5], the authors define a VR system as one which provides real-time viewer-centered head-tracking perspective with a large angle of view, interactive control, and binocular (stereo) display. Earlier VR systems, such as Head Mounted Displays (HMDs) and BOOMS (TM), achieved these features by using small display screens that move with the viewer, close to the viewer's eyes.

The CAVE was developed to overcome some of the limitations of HMDs and BOOMS (TM), especially for scientific applications. The CAVE makes use of large, fixed screens more distant from the viewer. This minimizes the encumbrances carried or worn by users and allows multiple people to share the VR experience. [6]

In addition to the ability to share the virtual environment among multiple users, the CAVE has presented other benefits. For instance, the CAVE is immersive, but it doesn't completely isolate the users from the real world. According to [4], real world isolation can be highly intrusive and disorienting. The viewer is still aware of the real world and may fear events such as running into a wall. In [5], the authors observe from their experience that seeing one's own body also decreases the chances of nausea. It has also been shown in [5] that tracking errors in the CAVE are less distracting then in other systems. This is due to the fact that the projection plane does not move with the viewer's position and angle as it does in a HMD or BOOM device.

The CAVE is made up of the following hardware components [5][6]. It is a 10x10x9 foot theater, with images rear-projected onto three of the walls which are screens. A fourth projector projects onto the floor from above. In the current NCSA configuration, the four displays are driven by one Silicon Graphics Onyx computer that has two

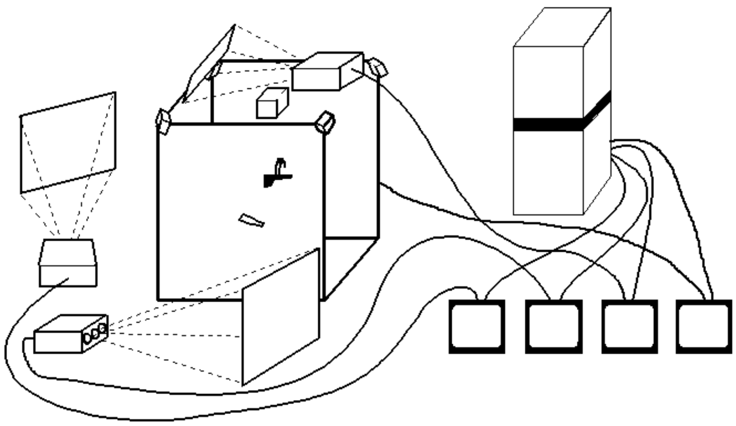

Figure 3.1: The CAVE layout

SGI Infinite Reality graphics pipelines. Each graphics pipe is split into two, driving two of the four displays. Each of the four displays has full workstation resolution (1024x768) and displays stereoscopic images at 96 Hz.

The 3D effect of the CAVE comes from the stereo projection. This stereo projection is achieved by projecting an image for the left eye followed by an image for the right eye. Viewers wear Stereographics' LCD Crystal Eyes stereo shutter glasses to view the stereoscopic images. The glasses are synchronized with the computer using infrared emitters mounted around the CAVE. As a result, whenever the computer is rendering the left eye perspective, the right shutter on the glasses is closed and vice-versa. This tricks the brain and gives the illusion that the left eye is seeing the left perspective and the right eye is seeing the right perspective.

Figure 3.2: LCD Stereo shutter glasses and infra-red emitter

In order for the perspective to be accurate, the user's head position and orientation has to be tracked. Although many people can be in the CAVE at once, only one user (the driver) is tracked. As a result, the other people in the CAVE should stand close to the driver to get the correct perspective. The tracking is achieved with a Ascension Flock of Birds electro-magnetic, 6 degree of freedom tracking system. The tracker senses a tethered electromagnetic sensor that is mounted on the driver's glasses.

A second electromagnetic sensor is attached to a device called the wand. The wand is a hardware device that can be thought of as a 3D equivalent of a mouse. In addition to being tracked, the wand has three buttons and a pressure-sensitive joystick. The wand is the primary input device in the CAVE, but certain applications may integrate their own special hardware. Since the wand is tracked, it facilitates various interaction techniques that are not found on the desktop [11][10][1]. More natural interaction techniques is one of the motivations for using virtual environments in the first place. For instance, an object in the virtual world can be picked by pointing at it with

Figure 3.3: The wand

the wand. Typically, in such a technique, a virtual beam is emitted from the wand, allowing the user to intersect that beam with the desired object and selecting it with a wand button.

Typically, the virtual world in a CAVE application may be a lot larger then the physical size of the CAVE. In order to be able to explore this world, some form of navigation has to be encorporated into the CAVE application. This navigation can come in many forms, including flying or walking and may be controlled in various ways [11] [1].

Software support for the CAVE comes in the form of the CAVE library. Applications are built on top of the CAVE library, which controls the display, tracking, and input devices. The CAVE hides a lot of the device specific details. It generates the correct perspective, synchronizes the screens, etc. The library provides callback routines that execute the application code. In the application specific code, the programmer can make use of CAVE library calls to poll tracker states for the wand or the glasses, poll for joystick and button states, etc.

The CAVE library also provides a CAVE simulator that lets the developer test applications on the desktop. The simulator runs the same executables as the CAVE using different configuration files. The configuration file specifies whether or not to use simulator mode. It also specifies the layout of the walls in the CAVE. This makes the CAVE library flexible and allows it to support different projection based setups other then the CAVE, such as the ImmersaDesk (IDesk) or the Infinity Wall (IWall). These systems are covered in detail in [6].

The standard CAVE library makes use of OpenGL (a standard graphics API) and requires CAVE applications to be written using OpenGL. In addition to the standard CAVE library, there is a small interface library called pfCAVE that allows users to use Performer in the CAVE. Performer is a software toolkit for 3D rendering that is based on OpenGL. Performer is discussed in detail in a later section.

3.2 The Visualization Toolkit (VTK)

The Visualization toolkit (VTK) is a freely available C++ class library for 3D graphics and visualization. The toolkit was developed at GE Corporate Research and Development. The authors of the toolkit had a lot of experience with a proprietary system that they had developed for a number of years. [13] Using their experience, they were able to design a simple, portable, powerful, and flexible toolkit that is widely used today.

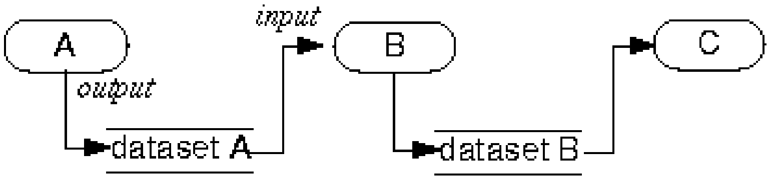

Figure 3.4: VTK model. Process objects A, B, and C are source, filter, and mapper objects, respectively.

The VTK model is based on the data-flow paradigm [13]. In this paradigm, modules are connected to form a network which is referred to as a VTK pipeline. The modules perform algorithmic operations on the data as it flows through the pipeline. In the VTK model, there are two basic types of objects: process objects and data objects. Process objects are the modules in the pipeline. Data objects represent the data at various stages in the pipeline. Essentially, the developer is only aware of the process objects, the data objects "live" in-between the process objects.

Process objects can be one of three types: sources, filters, and mappers. Source objects are found at the beginning of the pipleline. They generate one or more output datasets. A source object may be a reader for a particular file type or it may even generate its own data, such as a sphere source. The output of this source object is then connected to the input of another process object. The act of connecting the input of one process object to the output of another process object is how the pipeline is built. For instance, to connect the output of filter A to the input of filter B, a construct of the following form is used:

```
B->SetInput( A->GetOutput() );
```

Each process object has only one output. However, fanning of the output is allowed since multiple process objects can set their input to be the output of the same process object. The pipeline terminates with mappers. A mapper "maps" its input to the screen (renders it). At this stage of the pipeline, the input to the mapper is the output of some filter that outputs geometry data (vtkPolyData). Geometry data (vtkPolyData) is a data object, just like any other data object. In fact, there are filters that work on geometry data such as a decimation algorithm (vtkDecimate) or a triangulation algorithm (vtkTriangleFilter).

The mapper itself is one component of an object called a vtkActor. A vtkActor represents a geometrical object and its attributes. Other information in a vtkActor includes the object's appearance attributes (vtkProperty), and its location in space. As a result, the developer will instantiate a vtkActor for each mapper in the VTK pipeline. The developer will then set any attributes of each vtkActor and add each vtkActor to what is called a vtkRenderWindow. The vtkRenderWindow will then display all of its actors in a window on the screen.

In this work, however, we do not make use of VTK's rendering capabilities since they are insufficient for VR environments like the CAVE. Rather then putting the vtkActors into a vtkRenderWindow, the vtkActors are translated to Performer structures and rendered using Performer. This process is discussed in detail in later sections.

Building VTK pipelines is simple, but one also needs to understand how a VTK pipeline executes. VTK uses a model that has an implicit control of execution [14]. Execution only occurs when output is requested from an object (i.e. demand-driven). This scheme is implemented with two key methods: `Update()` and `Execute()`.

Each process object keeps track of its last modification time. If output is requested from the object and the object or its input has been modified since it last executed, then the process object should re-execute. The process of executing the pipeline begins with a call to a process object's `Update()` method. This typically happens when it is time to render and the mapper wants to update itself before rendering. The process object that receives the `Update()` message will recursively invoke the method on its input. This continues until a source object is encountered. The source object compares its modification time to the last time it executed. If it has been modified, then it will re-execute itself using the `Execute()` method. As mentioned previously, data objects "live" between process objects. If the source object has not been modified, then its output (a data object) is still valid. The recursion then backtracks with each successive process object comparing its input time and its own modification time with the time that it last executed. If necessary, the process object re-executes itself using the `Execute()` method. This process ends when control is returned to the object that initiated the `Update()`.

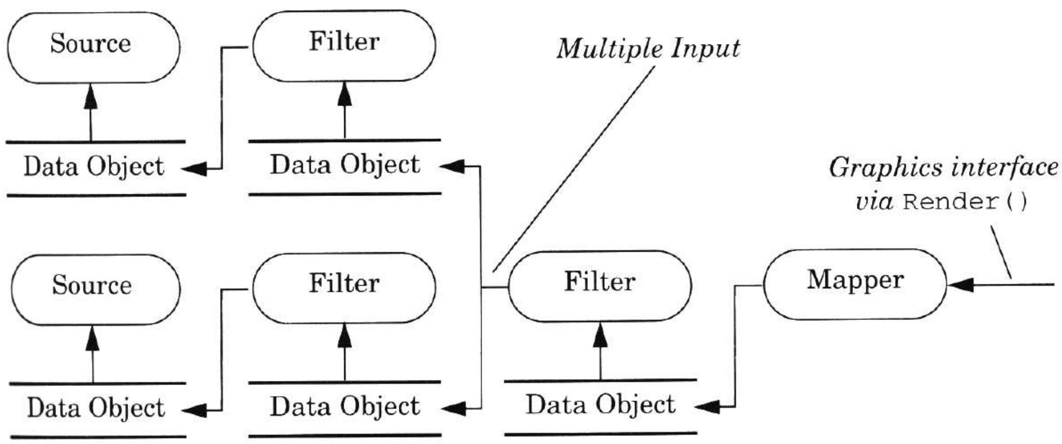

Figure 3.5: VTK model. Arrows indicate direction of Update() process. Data flows back to mapper via Execute().

The process is simple, but it requires that modification times and execution times are kept track of properly. This is especially important if the developer chooses to write custom process objects. The benefits of this approach are that segments of the pipeline are only re-executed when their output is requested and only if they have changed since the last execution.

Overall, the VTK toolkit is simple to use and very powerful. The power comes from the fact that VTK comes with more than 300 classes. Also, since it is freely distributed, users contribute code and bug fixes. The library also comes with wrappings that allow it to be used with interpreted languages such as Tcl, Java, and Python. These languages offer greater flexibility and cut down on development time [13].

3.3 IRIS Performer

IRIS Performer is a SGI (Silicon Graphics) software toolkit for 3D rendering. Performer plays a role in this work because VTK (visualization toolkit) wasn't designed for rendering to multiple channels (views) simultaneously. Rendering to multiple channels simultaneously is needed in multi-headed displays such as the CAVE here at NCSA.

By using VTK and Performer, we are able to make use of all the visualization algorithms that VTK provides and make use of the rendering capabilities of Performer. However, some work had to be done in order to get these toolkits to work together. This work is described in the vtkActorToPF section.

The main concept behind Performer is the notion of a scene graph. Essentially, the use of a scene graph emphasizes 3D objects rather than drawings. [17] The scene graph is a database which stores all the information about the objects in the scene. The information in the database can be used in a variety of ways, the most common being rendering (displaying an image of the 3D objects on the screen). Other actions that can be applied to a scene graph include picking, calculating bounding boxes, writing to a file, etc.

Performer uses OpenGL for rendering. OpenGL is a standard API for graphics programming. However, in OpenGL, rendering is explicit. OpenGL code describes the primitives that are to be drawn. In Performer, the scene is organized into higher level objects rather then just individual primitives. This makes it easier to manipulate individual objects within the scene. During rendering, the scene is traversed and each object knows how to render itself (using OpenGL). As a result, working with scene graphs is a lot easier than working directly with OpenGL.

The structure of the Performer scene graph is a tree structure which permits shared instancing. Shared instancing means that multiple parents can share the same child. Due to the shared instancing, a reference counting mechanism is provided to ensure that deleting objects is not a dangerous operation. [3] Each node in the tree is an object. Typically, leaves in the scene graph represent geometry objects and interior nodes represent transformation objects. The shared instancing mechanism offers database and program economy [17]. A subtree in the scene graph may represent an object in the scene (the object itself may be made of subcomponents). If multiple copies of that object are going to populate the scene, then it is efficient to reuse the same subtree in different parts of the graph. Also, any change in that subtree will be reflected in all instances of it.

For the purposes of this work, only two types of Peformer nodes are utilized, pfGeode and pfDCS. The pfGeode is a geometry node that contains a list of geometry structures called pfGeoSets [3]. pfGeoSets encapsulate state and geometry. Each individual pfGeoSet holds one type of primitive. Altogether a pfGeode describes the attributes and primitives of a geometrical object in the scene. The pfDCS node is a branch node (interior node) that represents a dynamic coordinate system [3]. This type of node is used to place the geometry in 3D space.

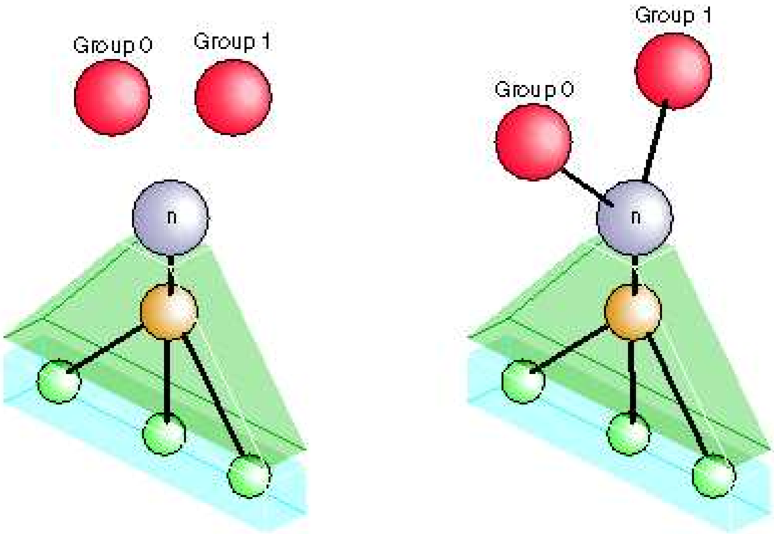

Figure 3.6: Shared Instances

As detailed in the vtkActorToPF section, VTK is used for its visualization algorithms. The resulting geometry is stored in VTK objects called vtkActors. A vtkActor is the equivalent of a pfGeode. However, since these are two different toolkits, the geometry representation is different. In order to be able to render the geometry using Performer, each vtkActor is translated into an equivalent pfGeode and inserted into the Performer scene graph. Whenever a vtkActor changes, the corresponding pfGeode must be retranslated.

3.4 IRIS Performer in the CAVE

As mentioned in the section describing the CAVE, in addition to the standard CAVE library, there is a small interface library called pfCAVE that allows users to use Performer in the CAVE. [12] This library takes care of initializing Performer to work with the CAVE's displays. This involves setting up the Performer pipes and channels (views). However, since all of this work is done automatically by this library, the details of this setup will not be covered. The pfCAVE library also provides some functions to translate between the two different coordinate systems used by Performer and the CAVE.

Once this initialization is performed, the application need only to be concerned about the contents of the scene-graph. The pfCAVE library will take care of all of the details of synchronizing the displays, maintaining the correct perspective, and rendering in stereo.

Chapter 4

Object Oriented Design

4.1 Object Oriented Programming

A basic knowledge of object oriented programming is assumed. There are many books on the subject. A popular book is *Object Oriented Analysis and Design with Applications* by Grady Booch. There's also *The C++ Programming Language* by Bjarne Stroustrup.

In addition to reading about the basics of object oriented programming, I strongly recommend *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

4.2 Design Patterns

Simply put, design patterns are techniques for solving common problems that have been proven to work well. Experienced programmers reuse solutions that have worked for them in the past rather then trying to solve problems from first principles. This is true in other fields as well. Experts rely on proven approaches to guide their decision making. In object oriented programming, design patterns refer to "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." [7] Since patterns are well established, each pattern's constraints and tradeoffs are known ahead of time, making it easier to avoid pitfalls.

Using design patterns not only makes it easier to get a design "right" faster, it helps improve communication. If proven techniques are identified as design patterns and people learn about them, it makes it easier for those people to communicate about object oriented design in general. In a sense, it adds to their vocabulary, allowing them to express themselves more succinctly and clearly.

4.3 Object Composition vs. Inheritance

Object composition and inheritance are two techniques for reusing functionality in object-oriented systems [7]. Class inheritance, or subclassing, allows a subclass' implementation to be defined in terms of the parent class' implementation. This type of reuse is often called **white-box reuse**. This term refers to the fact that with inheritance, the parent class implementation is often visible to the subclasses.

Object composition is a different method of reusing functionality. Objects are composed to achieve more complex functionality. This approach requires that the objects have well-defined interfaces since the internals of the objects are unknown. Because objects are treated only as "black boxes," this type of reuse is often called **black-box reuse**.

Each of these two methods have advantages and disadvantages. The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that the subclass becomes dependent on the parent class implementation. This makes it harder to reuse the subclass, especially if part of the inherited implementation is no longer desirable. According to [16], "inheritance breaks encapsulation." One way around this problem is to only inherit from abstract classes. Another problem with class inheritance is that the implementation inherited from a parent class cannot be changed at run-time.

In object composition, functionality is acquired dynamically at run-time by objects collecting references to other objects. The advantage of this approach is that implementations can be replaced at run-time. This is possible because objects are accessed only through their interfaces, so one object can be replaced with another just as long as they have the same type. In addition, since each object is defined in terms of object interfaces, there are less implementation dependencies [7]. The disadvantage of object composition is that the behavior of the system may be harder to understand just by looking at the source code. A system using object composition may be very dynamic in nature so it may require running the system to get a deeper understanding of how the different objects cooperate.

In general, object composition should be favored over inheritance. It promotes smaller, more focused classes and smaller inheritance hierarchies. Most designers overuse inheritance, resulting in large inheritance hierarchies that can become hard to deal with. A design based on object composition will have less classes, but more objects. The behavior of such a system depends more on the interrelationships between objects rather then being defined in a particular class [7].

However, inheritance is still necessary. You cannot always get all the necessary functionality by assembling existing components. This is because "the set of components is never quite rich enough in practice" [7]. Inheritance can be used to create new components that can be composed with old components. As a result, the two methods work together.

Due to the flexibility and power of object composition, most design patterns emphasize object composition over inheritance whenever it is possible. Many times, a design pattern shows a clever way of solving a common problem through the use of object composition rather then a standard, less flexible, inheritance based solution.

4.4 Delegation

Although object composition is very useful, one may question whether it is as powerful for reuse as inheritance. Specifically, using inheritance, an inherited method can always refer to the receiving object using the `this` member variable in C++. In delegation, the receiving object delegates operations to a delegate (one of the objects it is composed of). The same effect (as `this`) is achieved by having the receiving object pass a reference to itself to the delegate. Using delegation, a method can always refer to the original recipient of the message, regardless of the number of indirections due to object composition.

Delegation allows object composition to be as powerful for reuse as inheritance [7]. Several design patterns make use of this strategy.

4.5 Frameworks

A framework is a set of classes that make up a reusable design for a particular domain [7]. An application is developed by creating application specific subclasses of abstract classes in the framework. The emphasis is on design reuse rather then code reuse. For the most part, a framework captures the common design decisions for the application domain that it was built for. By making the design decisions for the developer, it allows the developer to concentrate on the application specific details. Not only do frameworks make for quicker application development, applications built with the same framework are consistent and easier to maintain.

However, since frameworks make many design decisions and assumptions about the domain, good frameworks are very hard to develop. A framework has to be flexible enough to meet the demands of new applications. A design has to be reused several times before it can truly be called a framework. Many so-called frameworks break down when somebody tries to reuse them. It is almost always impossible to predict the needs and trends of future applications. As a result, building a framework is an iterative process. The framework evolves as new applications are built and weaknesses in the framework are discovered and fixed.

4.6 Object Oriented Design and Visualization Tools

The focus of this thesis work has been to apply object-oriented design knowledge to the domain of visualization tools. Our Visual Interactive Computing Environments group develops custom visualization tools for a variety of visualization applications. Rather then viewing each of these applications as an independent project, we envision them as being pieces of an overall picture. Our ultimate goal is to take our visualization experiences and put them together so that we can better serve our customers in the future.

A good framework captures the knowledge and the experience of the designer. It encompasses what the designer has learned about the domain. This work represents a lot of what I have learned over the last year working in the domain of visualization tools. It is by no means the ideal framework. Rather, it is a learning experience and possibly the beginnings of a visualization tool framework. Hopefully, elements or ideas from this framework will be reused and will help our group achieve its goals.

Chapter 5

Model/Framework

The design involves breaking the visualization tool into components. The three major components are called the VisGen (visualization generator), the Application, and the Interface. The VisGen can be thought of as a wrapper for a visualization package that generates the geometry. In our case, we are using the freely available VTK library (visualization toolkit). The Application maintains the state of the tool, such as what dataset is being visualized, and it tells the VisGen what to generate. The interface component manipulates the state of the tool by sending messages to the Application. As a result, the VisGen is general purpose and can be used in any tool. The Application is application specific, but platform independent. The interface is platform specific, but even there some objects are shared between the CAVE and desktop version. Specifically, both interfaces make use of Command objects. Each gui component has a Command object that executes whenever the component is changed

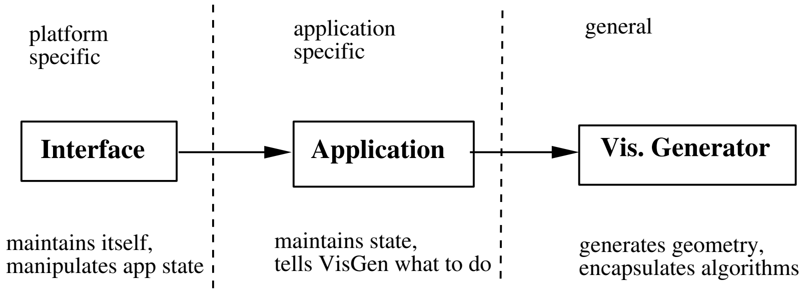

Figure 5.1: Model

by the user. There are concrete subclasses of Command so that a gui component can be "bound" to a particular action at run-time. For instance, a CAVE slider widget can be attached to a Command that changes the isosurface value or the same slider can be attached to a Command that changes the zoom factor. Similarly, the desktop slider widget can be "attached" to the same Command objects.

Each component maintains itself and delegates whatever work it can to the more general component. The next sections describe each of the three components in greater detail.

5.1 VisGen

The VisGen (visualization generator) component is responsible for generating the geometry. It is a general purpose component that can be used across visualization applications. The reason for the abstraction is so that the visualization applications do not become dependent on a particular visualization package or toolkit. At the same time, it allows the component to be interchanged with another component supporting the same interface. For instance, the regular VisGen component could be replaced with another VisGen component that acts like a collaborative visualization server, sending the geometry across the network to all the participants in the session.

Another useful feature of having a VisGen component is that it acts as a builder, creating the internal visualization package objects dynamically. In our case, since we are using VTK, the VisGen facilitates building dynamic VTK pipelines. Normally, VTK pipelines are developed statically by putting them in the C++ code and then compiling. Using the VisGen, the pipeline can be built during run-time. The VisGen allows the user to add datasets, change filters, remove and add segments of the pipeline, etc.

One other feature of the VisGen is that it provides a higher level interface to visualization algorithms. The person that writes the VisGen must know a great deal about the visualization package (VTK), but anybody can use the VisGen with minimal instruction. In a sense, the VisGen hides some of the ugly details of the visualization package being used.

In the end, the VisGen has to produce some "standard product" that describes the geometry that was generated. One possible "standard product" could be VRML. For now, the "standard product" is a collection of vtkActors. These can be rendered on the desktop very simply using VTK. In the CAVE, they are translated to equivalent Performer structures since VTK cannot handle rendering to multiple displays simultaneously. Basically, another small component is attached to the output of the VisGen that automatically takes care of the translation. See the section about vtkActorToPF.

5.2 App

The Application component maintains the state of the application. It gets input from the interface and it tells the VisGen what geometry to generate. This is where a lot of the application dependent code is stored. This helps make the interface component simpler and at the same time facilitates reuse since both the CAVE tool and the Desktop tool use the same Application component.

Since the Application component maintains the state of the application, it can also be made to store that state. This would allow the user to "save" the state in some session file and restore that state at a later time.

5.3 Interface

The interface component is the platform specific component that maintains a collection of interface widgets. However, instantiating a collection of widgets is not the only task that the interface is responsible for. Often, there are dependencies between widgets. For instance, there may be a reset button that resets the other widgets.

The naive way to implement dependencies would be to have each widget keep track of its dependents. Then each widget could notify its dependents when a dependency applied. Using this type of approach makes the interface less reusable. The more dependencies there are, the less likely it is that you can remove or add widgets or even pull out widgets and use them elsewhere. Instead, you have a bunch of very specialized widgets that work with a particular interface setup.

The solution to this problem is to use an interface mediator. The interface mediator is a central object that instantiates the widgets and maintains all of the dependencies between them. The widgets do not know about each other, they only know about a mediator. Whenever a widget changes, it only notifies the mediator. The mediator then decides what should happen to any other widgets if anything at all. As a result, the widgets are very reusable. Any of the ugliness specific to a particular interface is hidden within the interface mediator. This makes the interface mediator specific to a particular interface setup. There is a interface mediator abstraction, but each new interface must implement a concrete interface mediator.

Using a mediator promotes loose coupling between the user interface objects. It localizes behavior in a central object. However, the drawback is that often the mediator itself becomes a complicated object that is hard to maintain.

5.4 Widget

The interface component is made up of multiple widgets. The user can manipulate these widgets which in turn manipulate the state of the application component. An abstraction for widgets is made that is used across platforms. Each widget has a graphical representation, a mediator that it notifies, and a command object. The widget is bound to a particular action at run-time by instantiating the appropriate command object. Whenever the value of the widget changes, the command object is invoked, affecting the state of the application. The widget also notifies the mediator that it has changed.

From this abstraction, concrete subclasses of Widget can be created for different platforms. Although widgets on different platforms may look quite different and interact with the user in different ways, they make use of the same commands and so affect the application in the same way. For instance, a desktop slider might be used to modify the isovalue for an isosurface that is being visualized for a particular dataset. The corresponding CAVE application might have a corresponding CAVE widget that is a 3d knob floating in space. The user can select it with a virtual beam coming out of the wand and then "twist" it to change the isovalue. Although these two widgets are quite different in their look and interaction, they can both be attached to the same command object. Each widget maintains a value and whenever that value is changed by interaction, the command for changing the isovalue is invoked. That command can get the new value from the widget that invoked it and send the appropriate message to the application component.

5.4.1 Widget Values

The easiest and most consistent way to represent the value of a widget is with a floating point number between 0 and 1. For "boolean" widgets, 0.0 is false and 1.0 is true. For continuous widgets, 0.0 is the minimum value and 1.0 is the maximum value. When a command is invoked, it is the command that decides what to do with the value. The command knows about the application component, so it can do the appropriate transformations. For instance, if it is an isovalue command, then the command might get the high and low values from the dataset by asking the application and then interpolate that range using the value of the widget. If the user desires different behavior, then it is only a matter of creating a new command and instantiating it in place of the old command.

A more general method of representing the value would be to use strings. Changing the widget abstraction to do this would be easy. This would allow string as well as numerical values since the numerical values could be stored as strings and converted back to numerical values when necessary. As of right now, none of the widgets take string values as input, so this wasn't done. In the future, this may be done to facilitate a speech interface in the CAVE. There's no easy way to input text with the CAVE input devices, so speech is the natural alternative to typing at the desktop.

5.4.2 Distinction between CAVE and Desktop Widgets

A distinction between a CAVE widget and a Desktop widget is necessary because some attributes are specific to CAVE widgets and some attributes are specific just to desktop widgets. For instance, during placement, a CAVE tool will have rotation and 3D translation. So, even though there is a tool abstraction, handles to the actual specific tool are needed to do platform specific operations. All of these platform specific operations are hidden in the interface mediator. As mentioned before, the interface mediator hides all of the interface ugliness so that the other components of the interface are flexible and reusable.

Chapter 6

Implementation

In this chapter, we present some of the more important implementation issues for this Model/Framework.

As mentioned in the previous chapter, each widget in the interface has a graphical representation, a reference to the interface mediator, and a command object. This composite component is called a `pjrWidget`. The graphical representation for a `pjrWidget` is an instance of a `Tool`. The reference to the mediator is a pointer to an instance of the `InterfaceMediator` class. The command object is an instance of `Command`.

The interface for the widget abstraction is:

```
class pjrWidget {
  Tool *tool    // graphical representation (Tool is abstract class)
  InterfaceMeditor *mediator
  Command *command
  float prev_val
```

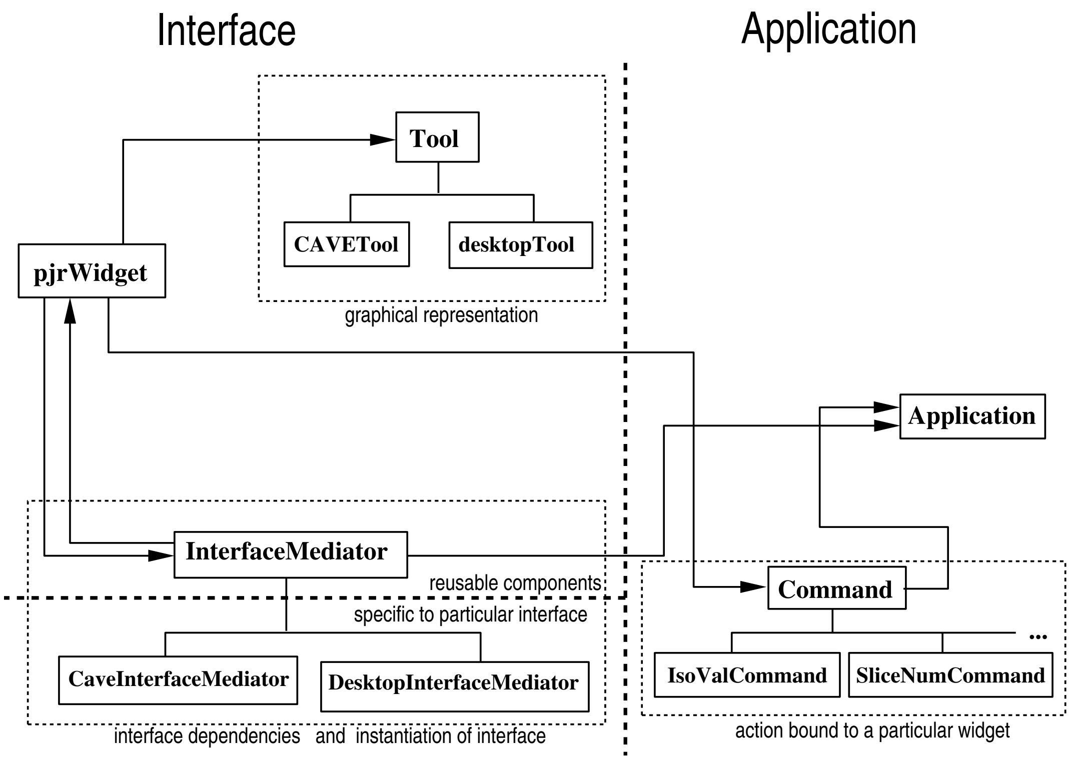

Figure 6.1: Diagram of pjrWidget

```
  ...
  value()
  setValue();
}
```

6.1 Graphical Tool Implementation

The `Tool` class is an abstract class. There are abstract subclasses for CAVE tools (`CaveTool`) and desktop tools (`DesktopTool`). Each of these abstract classes has

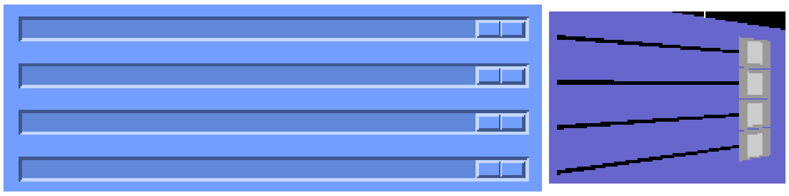

Figure 6.2: Graphical widgets example: Four Motif sliders. Four Costigan virtual sliders

concrete subclasses for specific widgets.

A `CaveTool` object represents a virtual widget. For convenience, rather then developing my own virtual widgets, I made use of Jim Costigan's CaveTooLs widgets [2]. Jim Costigan works for EVL at UIC, where the CAVE was developed. In order to use these widgets, they had to be wrapped to conform to the `Tool` interface. Originally, the widgets were not polymorphic. They were independent classes. Since the widgets were wrapped to the same interface (`CaveTool`), they are now polymorphic. The process of wrapping these widgets is described in more detail in the Adapter section.

The concrete desktop tools were adapted from X/Motif widgets. In Motif applications, user defined callbacks define the behavior of the widgets. In order to preserve the object-oriented design of the model, each instance of a `DesktopTool` instantiates a Motif widget and registers a generic callback. The generic callback simply calls the `update()` method on the desktop tool.

The generic callback:

```
void genericCB(Widget w, XtPointer client_data, XtPointer call_data) {
  Tool *tool = (Tool *) client_data;
  tool->update();
}
```

Adding the generic callback from within a class method:

```
XtAddCallback(w, XmNactivateCallback, genericCB, (XtPointer) this);
```

Using this generic callback mechanism preserves encapsulation. The widget specific code in response to a user event is kept within the class.

6.2 Patterns Used

This section describes how certain patterns from [7] were implemented in this work.

6.2.1 Command

The command pattern encapsulates a request as an object [7]. Essentially, this pattern creates an abstraction for an action that is to be taken. In the context of this work, whenever a widget changes value, it invokes its command object. The abstraction allows the widget to be bound to a particular action by instantiating the appropriate command object at run-time.

Invoking a command object form within a `pjrWidget` method:

```
  this->command->Execute(this);
```

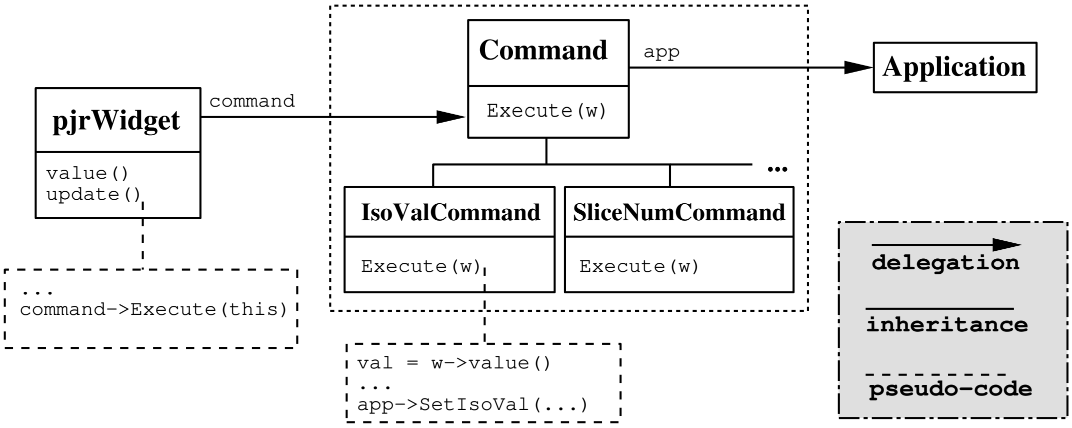

Figure 6.3: Partial diagram of Command pattern. Concrete commands invoke methods on Application.

The instance of `pjrWidget` passes a reference to itself to the command. This allows the command to get information about the state of the widget that invoked the command. The command object also keeps a reference to the `Application` component. It can get information from the application and together with the information from the widget, invoke a method in the `Application` component, changing the application's state.

6.2.2 Mediator

The mediator pattern encapsulates how a set of objects interact [7]. Rather then keeping track of its own dependents, each widget notifies the mediator whenever it changes. The mediator then determines how this will affect any of the other components of the interface. The pattern allows widgets to be more flexible and reusable.

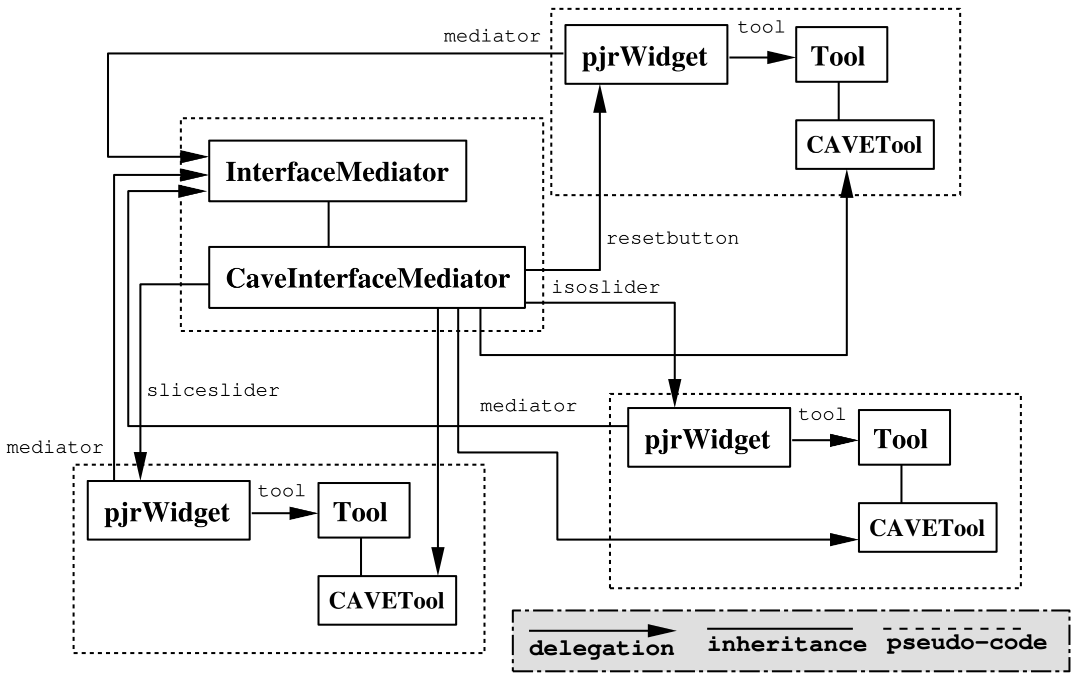

Figure 6.4: Partial diagram of mediator. Concrete mediator has handles to platform specific tools (graphical representation of widgets) for platform specific operations.

All of the code specific to a particular interface is encapsulated within a concrete mediator class.

The interface mediator abstraction:

```
class InterfaceMediator {

  CreateInteface();
  ToolChanged(pjrWidget *theWidget)
  ...
}
```

When a widget changes:

```
this->mediator->ToolChanged(this);
```

The interface mediator instantiates all the widgets in the interface. It keeps handles to the specific graphical widgets used so that platform specific operations can be done to the widgets. Outside of the interface mediator, all references to widgets are done through the abstract `pjrWidget` interface. Any of the ugly details of initializing and maintaining the interface are hidden within the mediator.

Even though a concrete mediator has to be made for each new interface setup, each mediator conforms to the abstract mediator interface, allowing other classes to be independent of any particular mediator.

6.2.3 Adapter

According to [7], the adapter pattern is used to adapt the interface of a class into another interface which clients expect. In order to be able to use Jim Costigan's widgets, they had to conform to the `CaveTool` abstraction. This was accomplished by implementing each cave tool in terms of an instance of one of Jim Costigan's widgets. For instance, one of the concrete subclasses of `CaveTool` is `SliderTool`. A `SliderTool` instantiates a `slider` object (a Jim Costigan widget). The `SliderTool` conforms to the `CaveTool` interface, but it delegates all the work to the `slider` object. The methods in `SliderTool` simply call methods in `slider`.

The desktop widgets also make use of the Adapter pattern. X/Motif widgets are "adapted" to the `DesktopTool` interface which is a subclass of `Tool`.

As a result, the developer can program for the most part using the `Tool` interface.

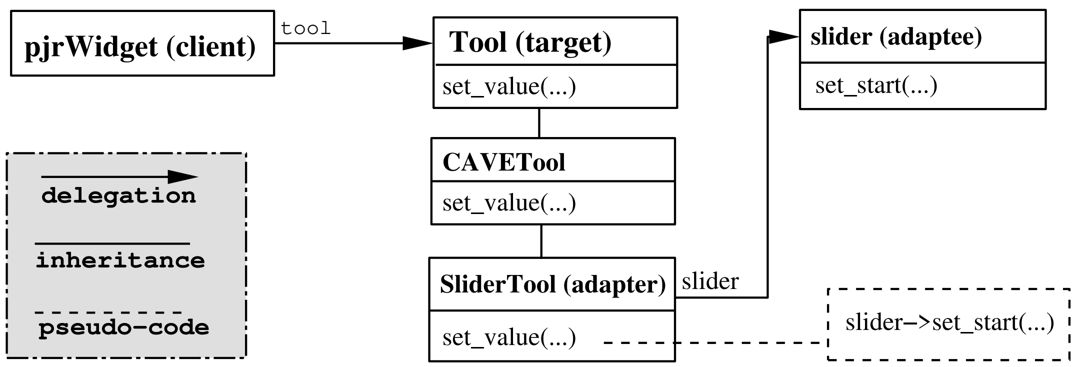

Figure 6.5: Partial diagram of Adapter pattern. The client (pjrWidget) expects a Tool object. The adapter (SliderTool) defines a SliderTool that conforms to the Tool interface and is implemented in terms of a slider object.

This allows the code to be portable across the CAVE and the desktop. However, within the interface mediator, handles to either `DesktopTools` and `CaveTools` are kept. As mentioned in the last chapter, this is necessary in order to do some of the platform specific operations for these widgets.

6.3 vtkActorToPF

In the process of integrating VTK with the CAVE environment, we found that VTK was not designed to be able to render the same geometry multiple times simultaneously to different channels (views). At first, I addressed this problem by modifying the source code for certain VTK classes in order to get them to work in the CAVE environment. This worked, but it was not a good design decision. Whenever the VTK library would change, as it did, I would have to redo the whole process. The solution was also very inefficient. It produced four identical yet separate VTK pipelines, one for each wall.

This initial experience taught me the importance of encapsulation. By breaking encapsulation and changing some of the VTK implementation, I lost some of the benefits of object oriented design.

We found that a better solution was to make use of another package to do the rendering. SGI distributes a toolkit called Perfomer that is widely used for 3D graphics (See Performer section). There is even a Performer version of the CAVE library (See pfCAVE section). Using VTK and Performer together allows the developer to get all of the visualization algorithms from VTK and all of the rendering advantages of Performer, such as multi-process rendering to multiple channels (views).

Plugging these two components together is not trivial. On the one hand, there is a VTK pipeline that generates geometry and on the other hand there is a Performer scene graph which is traversed whenever it is time to render. These two structures know nothing about each other. Then there is another issue about which component should control execution.

The solution is to develop a translator that translates the VTK geometry and creates a corresponding Performer structure and attaches it to the scene-graph. Vtk geometry is organized into an object called a vtkActor. Performer geometry is stored in a pfGeode. As for the issue of control of execution, VTK has an implicit control model, meaning that it updates the pipeline only when it is time to render and only re-executes the segments of the pipeline that are no longer valid. Since VTK is no longer doing the rendering itself, the Performer component has the control of execution and it is responsible for telling the VTK pipeline to update itself.

One approach is to use a function that explicitly translates a vtkActor to a pfGeode. While this works well, the function must be called explicitly whenever the Performer geometry needs to be updated in order to make it consistent with the VTK geometry.

A better approach is to create a translator object. This translator object knows about a particular vtkActor and a particular pfGeode. Since it is an object rather than a function, it can store state. In this case, the time of the last translation is stored so that it can be compared with the modification time of the vtkActor. Whenever the vtkActor changes, the pfGeode is automatically retranslated. The way this is accomplished is that the translator registers a callback with Performer that is called (on behalf of the pfGeode) before rendering. The callback checks to see if in fact the vtkActor did change since the last rendering, and retranslates if this is the case.

Using this approach preserves VTK's implicit model of execution. For instance, you can simply change the iso value of an isosurface filter in the pipeline and you know that the change will cause the VTK pipeline to update itself and re-execute the isosurface algorithm. Then, before rendering, the new VTK geometry will automatically be translated and plugged into the Performer scene-graph.

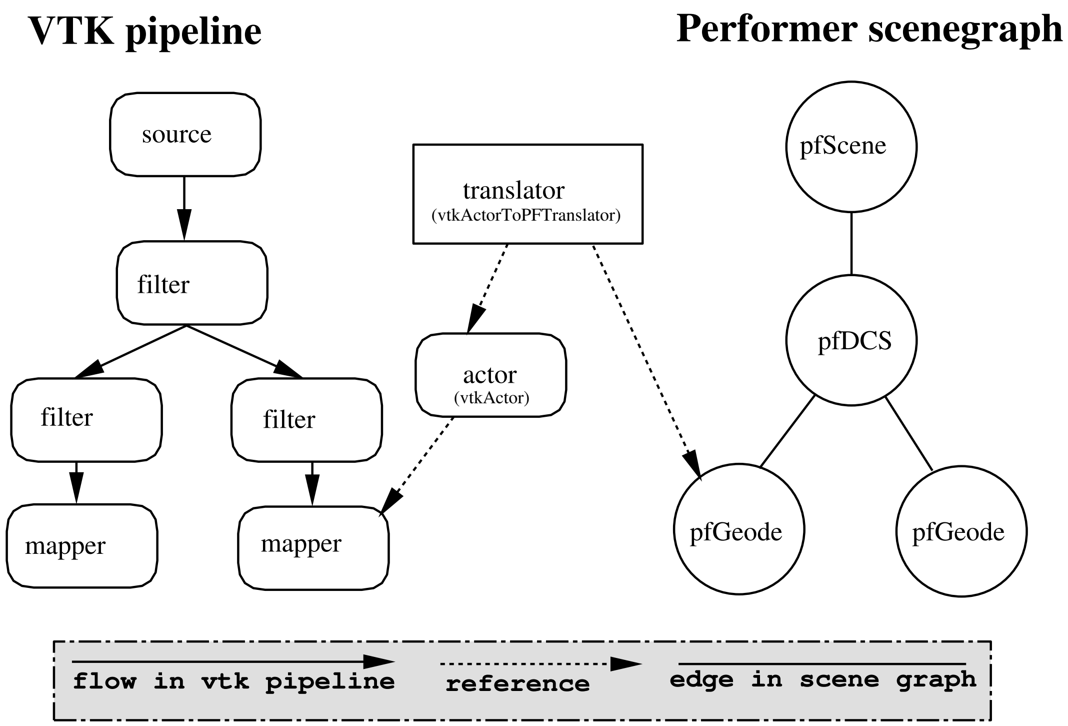

Figure 6.6: Diagram of how vtkActorToPFTranslator object works. The object has access to a particular vtkActor and a particular pfGeode. The translator accesses the VTK geometry through the actor's mapper. It deposits geometry into the pfGeoSets stored in the geode.

This approach emphasizes object composition, making this a very flexible and reusable solution. The two toolkits work together simply by creating new types of objects (translators) that bridge the gap. After creating these objects, the toolkits work together and the translation is transparent.

The source code and examples can be obtained at the following web address:

`http://hoback.ncsa.uiuc.edu/group/vtkActorToPF/`

6.4 Putting It All Together

Assuming that the `VisGen` component and the various interface widget components are already developed, a new application is developed by creating a new `Application` component, creating new subclasses of `Command` that manipulate the new `Application` component, and creating a new concrete `InterfaceMediator` that instantiates the interface components (`pjrWidgets`). These interface components are then bound to actions by instantiating the appropriate `Command` objects.

To port the application to the other platform (from desktop to CAVE or vice versa), all that needs to be done is that a second concrete `InterfaceMediator` needs to be written. The two separate interface mediators are required since they are responsible for platform specific operations that need to be done. In addition, the two interfaces may be set up differently or may have different dependencies.

It may also be necessary to have different main programs. However, these tend to be short and platform specific initializations could actually be moved into the interface mediator if necessary. The main program will instantiate the `VisGen`, `Application`, and `InterfaceMediator` components. It then enters the main control loop. On the desktop, this is the X/Motif loop and in the CAVE this is the Performer loop. Once this done, the program is running and waiting for user input. Actions by the user that affect the interface will cause commands to be invoked that will change the state of the `Application` component. In response, the `Application` component may cause the `VisGen` to generate new geometry.

An important implication of the design is that the `Application` and `VisGen` components can be reused, even without an interface. This allows server applications or commandline applications to be developed reusing the same components.

Chapter 7

Visualization Tools

Using this framework, a cave tool and a corresponding desktop tool were created. The two tools have a common code base. They compile within the same directory and make use of a lot of the same object files. In addition, a VRML server was created for web-based visualization. It makes use of the same `Application` and `VisGen` components that are in the two visualization tools. Developing this VRML server was simple and it demonstrated that components of the framework can be re-used. It also demonstrates that the visualization components of the framework are independent of the interface. The VRML server can be run from the commandline or over the web from a Java Applet interface.

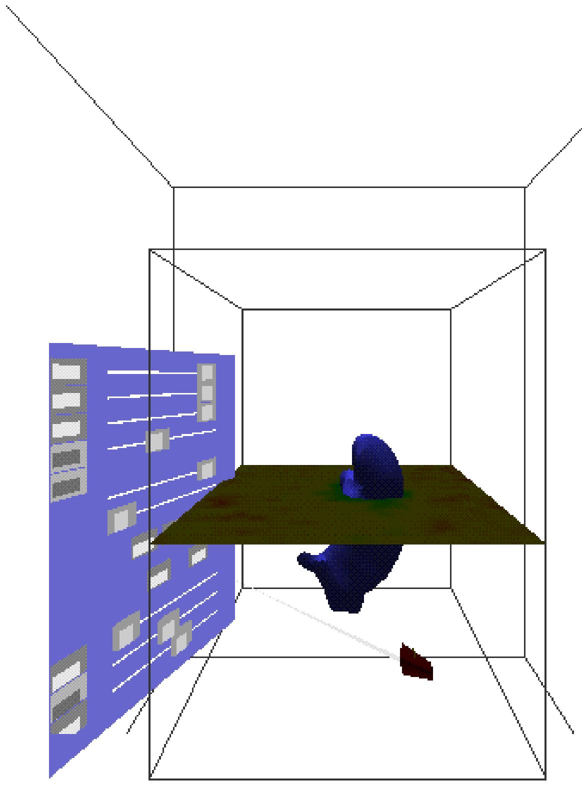

Figure 7.1: Snapshot from CAVE simulator. Visualization includes an isosurface and a slice.

7.1 CAVE Tool

The CAVE tool allows the user to view multiple, semi-transparent isosurfaces as well as slices. The controls in the interface allow the user to control visualization parameters such as visibility, isovalues, opacity, and slice numbers. In addition, the interface allows the user to rotate the volume.

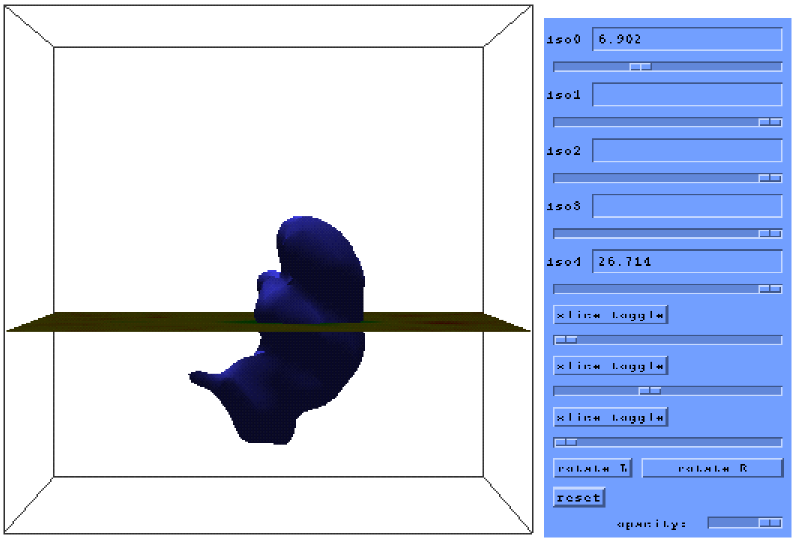

Figure 7.2: Snapshots from Desktop Tool (window borders removed). Same visualization as seen in CAVE Tool figure.

7.2 Desktop Tool

The Desktop Tool corresponds to the CAVE tool and provides the same functionality. It makes use of a lot of the same objects, including command objects. As a result, if a command's behavior is changed, this change will be reflected in both tools. One difference is that the Desktop Tool has text fields. The text fields are connected to the same command as the sliders. In fact, the dependencies in the interface mediator tie each slider to a textfield. If the textfield is changed, the slider moves to the new position. Similarly, if the slider is moved, the textfield's value is updated.

7.3 VRML Server

The VRML server is used for web-based visualization. The server itself is a stand alone executable with only a commandline interface. However, it reuses the `Application` and `VisGen` components from the other two tools. This made the development of the VRML server almost trivial. The code only has to parse the commandline, instantiate the correct objects, and call the appropriate methods.

The VRML server is run over the web from a Java Applet interface. Using this interface, the user selects datasets and the appropriate visualization parameters. The user then chooses to generate VRML. This causes the VRML server executable to be run on the server side as a CGI (Common Gateway Interface) program using the parameters selected from the applet interface. The VRML server generates the VRML and stores it in a file on the server side. It then notifies the user that the VRML has been generated and how large the file is. If the user deems that the size of the resulting visualization is acceptable, the user can click on a button and transfer the contents of the VRML file to the user's web browser (the client). Upon downloading the VRML, the web browser will start up the appropriate VRML plugin or helper application and display the geometry that was generated. This process allows the user to avoid downloading VRML files that are either too small (uninteresting) or files that are too large for the client side web browser to handle.

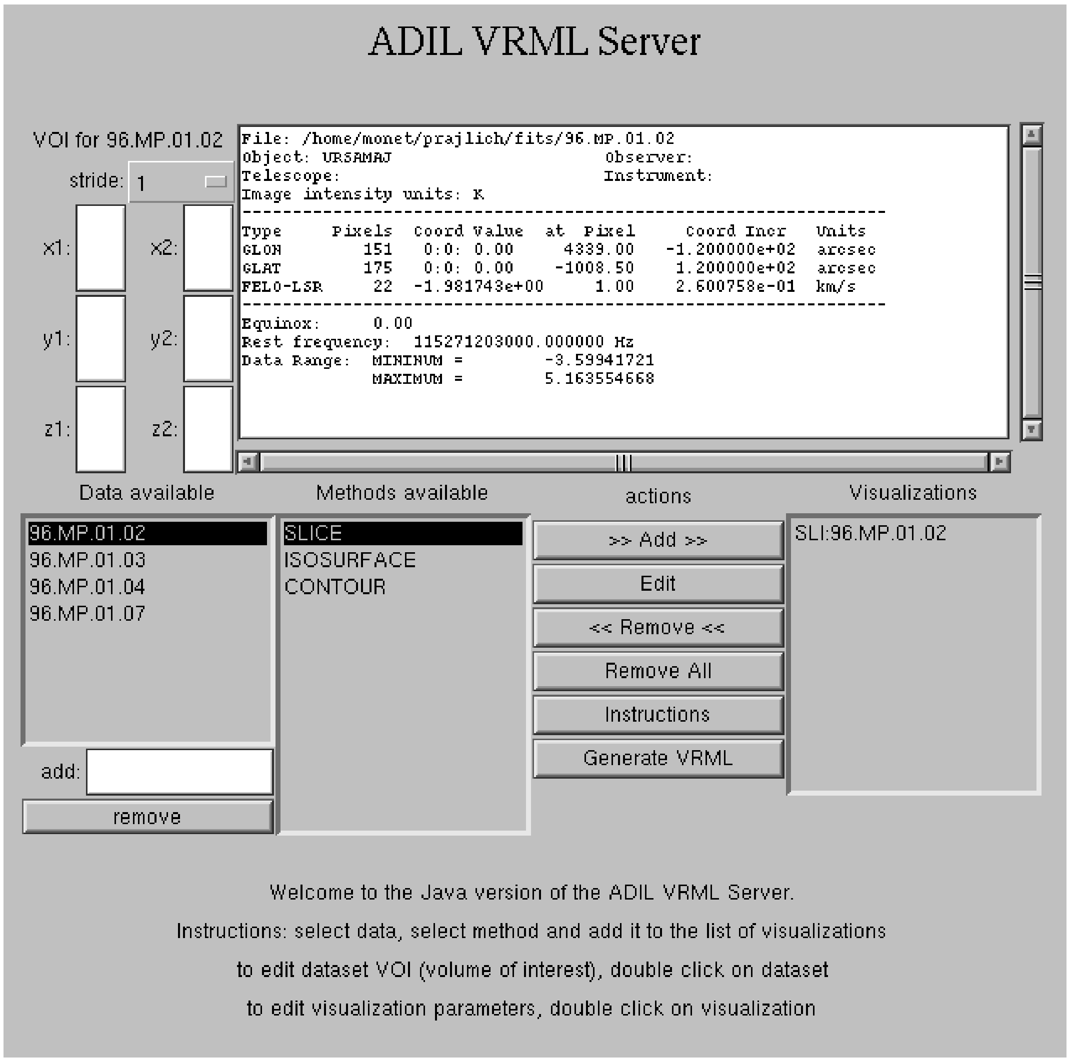

Figure 7.3: Snapshot from the VRML Server Java Applet Interface

Chapter 8

Conclusion

The original goal of this work effort was to try to design a CAVE tool in such a way that it could be easily transferred to the desktop. The motivation was that such a design would allow quicker development as well as simpler maintenance. Essentially, both tools would share the same code base.

The bulk of the work dealt with finding consistent, cross-platform ways to deal with the user interfaces. This was achieved through the use of object oriented techniques, commonly referred to as design patterns.

From these design patterns, a prototype framework emerged. This framework is a step in the direction of allowing portable and extensible custom visualization tools to be developed more easily.

In addition to developing the framework itself, further work had to be done in order to integrate the VTK visualization library with the Performer rendering library.

Since these are two popular toolkits, the work done to integrate them will be useful for other developers in the field of visualization.

Using the prototype framework, a CAVE tool and a desktop tool were developed. The two tools share the same code base and many of the same objects. In addition, a VRML server was created with components from the framework, facilitating web-based visualization. This demonstrated that elements of the framework are reusable.

Bibliography

[1] M. Pauline Baker and Christopher D. Wickens. Human factors in virtual environments for the visual analysis of scientific data. Technical report, National Center for Supercomputing Applications, 1995.

[2] Jim Costigan. Cave tools. http://www.evl.uic.edu/costigan/Tools3.html.

[3] Patricia Creek, Jed Hartman, and Gary Kuhn. Iris performer programmer's guide, 1995.

[4] C. Cruz-Neira, J. Leigh, M. Papka, C. Barnes, S. Cohen, S. Das, R. Engelmann, R. Hudson, T. Roy, L. Siegal, C. Vasilakis, T. DeFanti, and D. Sandin. Scientists in wonderland: A report on visualization applications in the cave virtual reality environment. *IEEE Computer Graphics*, pages 59–66, 1993.

[5] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. Defanti. Surround-screen projection based virtual reality: The design and implementation of the cave. In *Proceedings of ACM SIG Graphics*, 1993.

[6] Marek Czernuszenko, Dave Pape, Daniel Sandin, Tom DeFanti, Gregory L. Dawe, and Maxine D. Brown. The immersadesk and infinity wall projection-based virtual reality displays. *Computer Graphics*, May 1997.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[8] R.W. Hamming. *Numerical Methods for Scientists and Engineers.* McGraw-Hill, 1962.

[9] Frederick P. Brooks Jr. Grasping reality through illusion - interactive graphics serving science. In *ACM SIG Computer Human Interaction*, pages 1–11, 1988.

[10] Tom Meyer and Al Globus. Direct manipulation of isosurfaces and cutting planes in virtual environments. Technical report, Brown University, 1993.

[11] Mark R. Mine. Virtual environment interaction techniques. Technical report, University of North Carolina, 1995.

[12] Dave Pape. pfcave cave/performer library (cavelib version 2.6), 1997.

[13] William J. Schroeder, Kenneth M. Martin, and William E. Lorenson. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *IEEE Visualization '96*, pages 93–100, 1996.

[14] William J. Schroeder, Kenneth M. Martin, and William E. Lorenson. *The Visualization Toolkit.* Prentice Hall PTR, 1996.

[15] William R. Sherman, Alan B. Craig, M. Pauline Baker, and Colleen Bushell. Scientific visualization. In Allen B. Tucker Jr., editor, *The Computer Science and Engineering Handbook*, chapter 35, pages 820–846. CRC Press, 1997.

[16] Alan Synder. Encapsulation and inheritance in object-oriented languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 38–45, 1986.

[17] Josie Wernecke and The Open Inventor Architecture Group. *The Inventor Mentor*. Addison Wesley, 1994. has better description of scene graphs then Performer documentation.