Dynamic Scene Generation and Software Parallel Rendering of Anatomical Structures

Evan Albright

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2000

Program Authorized to Offer Degree: Electrical Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a master's thesis by

Evan Albright

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

____

Linda Shapiro

____

Jim Brinkley

Date: ____________________________

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purpose or by any means shall not be allowed without my written permission.

Signature ______________________

Date ______________________

University of Washington

Abstract

Dynamic Scene Generation and Software Parallel Rendering of Anatomical Structures

by Evan Albright

Chair of Supervisory Committee

Professor Linda Shapiro
Computer Science/Electrical Engineering

This thesis discusses two related topics, a system for generating scenes of 3-D anatomical structures, and the implementation of parallel algorithms for 3-D rendering. In an educational setting, 3-D scenes help students discover relationships between various anatomical structures. In a clinical environment, physicians can use 3-D scenes to examine how anatomical structures are effected by disease or injury.

In order for a scene generation system to be usable, scenes must be rendered as fast as possible. One approach to increase rendering performance is to use parallel algorithms to take advantage of multi-processor or multi-workstation architectures.

TABLE OF CONTENTS

List of Figures **iv**

List of Tables **vi**

Chapter 1: Overview **1**

1.1 Digital Anatomist Information System 1

1.2 Contributions 4

1.3 Thesis Overview 5

Chapter 2: The Dynamic Scene Generator and Graphics Server **6**

2.1 Graphics Server 6

2.1.1 Common Classes and Interfaces 7

2.1.2 Scene Renderer Module 8

2.1.3 FM Interface 8

2.1.4 ModelDB Interface 9

2.1.5 Scene Generator 9

2.1.6 Client vs. Server Rendering 10

2.2 Interfaces 11

2.2.1 Dynamic Scene Generator 11

2.2.2 Scene Manager 16

2.2.3 Dynamic Scene Explorer 19

2.2.4 Interface Implementation Highlights 22

Chapter 3: Parallel Rendering Techniques **24**

3.1 Mesa and Software Rendering Overview 24

3.2 An Approach to Increasing Software Rendering Performance: Parallel Rendering 27

3.3 Sort-Last Algorithm: Parallel Execution Using Processes and/or Threads 30

3.3.1 Parallel Virtual Machine (PVM) 31

3.3.2 Threads 32

3.3.3 System V IPC 32

Chapter 4: Results And Discussion **34**

4.1 Interface Critique 34

4.2 Parallel Rendering Performance 36

4.2.1 Pthreads Performance 38

4.2.2 System V IPC 39

4.2.3 Parallel Virtual Machine Performance 41

4.2.4 Parallel Processing Architecture Comparison 44

Chapter 5: Conclusion **47**

5.1 Interfaces 47

5.1.1 Future Work 48

5.2 Parallel Rendering Techniques 49

5.2.1 Future Work 49

Bibliography **51**

Appendix A: UW Digital Anatomist (UWDA) Dynamic Scene Generator Tutorial **53**

A.1 SCENE GENERATOR 53

A.2 SCENE MANAGER 56
A.3 SCENE EXPLORER 56

Appendix B: Dynamic Scene Generation Tools Feedback **59**
B.1 Scene Generator 59
B.2 Scene Manager 59
B.3 Scene Explorer 60

LIST OF FIGURES

1.1 Digital Anatomist framework diagram. 2

2.1 Block diagram of the components that interact with the Graphics Server 7

2.2 The initial state of the Dynamic Scene Generator in Netscape (the viewport was reduced in order to display all of the controls). 13

2.3 The Eighth Thoracic Vertebra added to an empty scene using the Dynamic Scene Generator 14

2.4 The *parts of* the *Vertebral Column* have been *highlighted*, resulting in the Diaphragm becoming transparent. 16

2.5 Initial appearance of the Scene Manager using Netscape 18

2.6 Adding the *om esophagus* add-on to the Add-On Display Panel . . . 19

2.7 The initial scene for the group *CR's Scenes* is displayed on the left, while the scene *Branches of Ascending Aorta* is being viewed as a potential replacement. 20

2.8 The group *CRexercise1* is previewed by clicking on the link *View CRexercise1* in the Scene Manager, resulting in the Dynamic Scene Explorer being launched in a new browser 21

3.1 Sort-last implementation data-flow diagram for N_T=3 children 29

4.1 VRML 2.0 scenes used for performance evaluations. Triangle counts are 36535 (top left), 118638 (top right), 171515 (lower left), and 414710 (lower right) 38

4.2 Speed-up ratios of each parallel processing implementation compared against an ideal ratio for a triangle count of 36535. 45

4.3 Speed-up ratios of each parallel processing implementation compared against an ideal ratio for a triangle count of 414710. 46

5.1 Brain Mapper Java Swing Applet interacting with the Skandha4 brain server. Advanced GUI features include sliders, a split panel, and a tabbed panel. The split panel contains the viewport in the left panel and a tabbed panel containing two sets of controls in the right panel. 48

LIST OF TABLES

4.1 Results of Pthread 3-D rendering test cases. Values are shown as a sum of rendering and occlusion test time in seconds. 39

4.2 Results of Pthread 3-D rendering test cases. Values are shown as a speed-up ratio, using the single Pthread time as a reference 40

4.3 Results of System V IPC 3-D rendering test cases. Values are shown as a sum of rendering and occlusion test time in seconds. 41

4.4 Results of System V IPC 3-D rendering test cases. Values are shown as a speed-up ratio, using the single process time as a reference 41

4.5 Results of PVM 3-D rendering test cases over a 10 MB/s hub connected network using a dual 550 MHz Pentium III Xeon, 500 MHz Pentium III Xeon, and quad 550 MHz Pentium III Xeon. Values are shown as a sum of rendering and occlusion test time in seconds. 43

4.6 Results of PVM 3-D rendering test cases on a quad 550 MHz Pentium III Xeon. Values are shown as a sum of rendering and occlusion test time in seconds. 44

Chapter 1

OVERVIEW

1.1 Digital Anatomist Information System

The Digital Anatomist Project[3] is an attempt to create an anatomy information system that is available from any computer that has access to the Internet. Typical uses of the Digital Anatomist include queries of the knowledge base for specific anatomic questions, retrieving dynamically generated 3-D scenes corresponding to a query, and using the retrieved information to access other databases and image repositories that fall under the scope of the Digital Anatomist. Figure 1.1 shows the various elements of the Digital Anatomist system and how they interact. This thesis is concerned with four elements of this system: the 3-D Scene Generator, Scene Manager, and Scene Generator (collectively referred to as the *Dynamic Scene Generator*), along with the Graphics Server, a program for rendering 3-D scenes as images on a high-performance server. The Scene Generator and Explorer allow a user to interact with the Graphics Server and view the rendered image.

Prior to this work, web access to 3-D scenes was through the Digital Anatomist Project Interactive Atlases. The Interactive Atlases allow the user to navigate through various groups of pre-generated 3-D anatomical scenes by following links to a desired group of images. These scenes are not dynamically rendered, they are simply retrieved from a static image database based on user input.

The Dynamic Scene Generator and Graphics Server developed in this thesis add another level of interaction to the Interactive Atlases, by dynamically rendering each

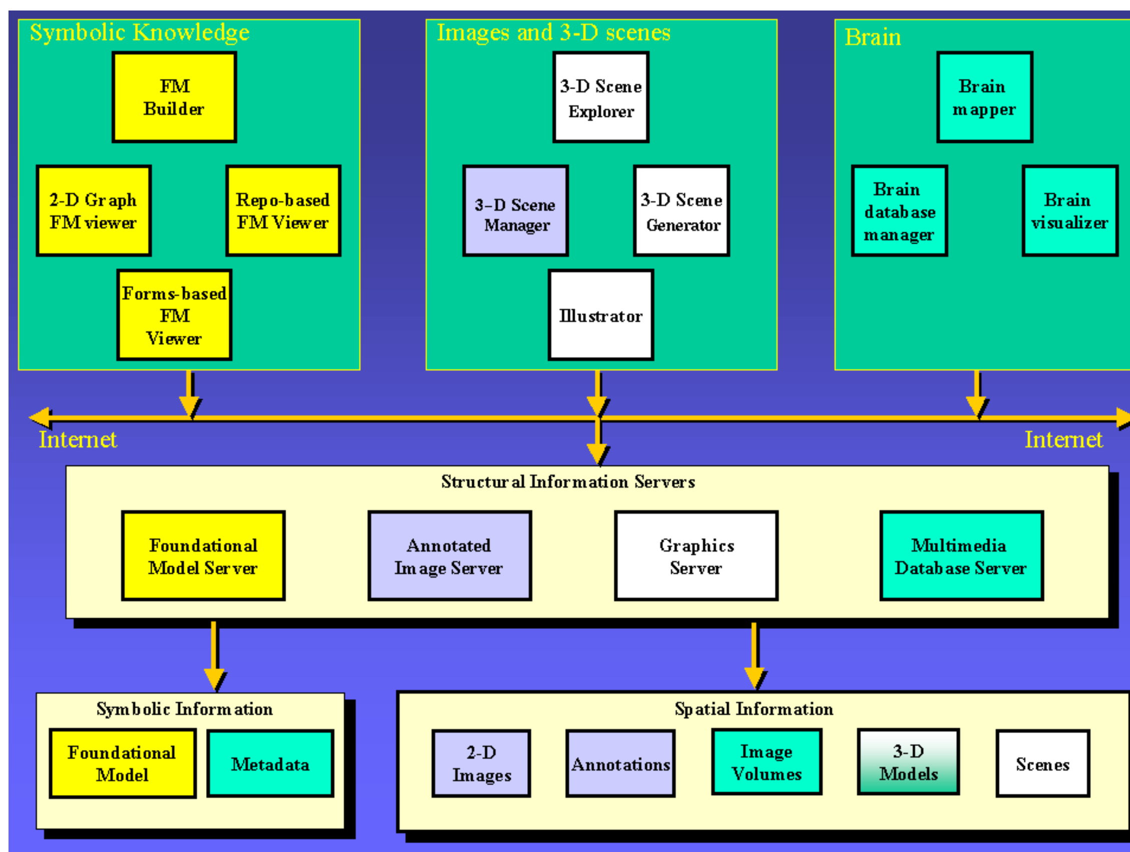

Figure 1.1: Digital Anatomist framework diagram.

scene and allowing the user to control how a scene is presented. Various architectures and levels of control are available to allow scene authoring and hierarchical scene viewing suitable for users with fast as well as slow clients. The Dynamic Scene Generator and Graphics Server make use of two pre-existing resources in the Digital Anatomist Information System (AIS): the Foundational Model and a set of 3-D models of anatomical structures.

The Foundational Model (FM) Database is a collection of anatomical terms that are grouped in hierarchies giving an anatomist a complete description of the human body. The hierarchies currently consist of *part-of*, *is-a*, *branch-of*, *tributary-of*, and

contained-in. A spatial hierarchy is currently being developed which will include spatial information such as boundaries of 3-D entities and adjacencies of structures. Each hierarchy describes a tree consisting of a single root term (e.g. *Human Body* for the *part-of* hierarchy) and each successive layer of children consisting of a finer level of detail of terms. The Foundational Model will eventually contain information down to the molecular level, but currently the detail ends at the smallest visible structures. The FM is accessed by the Foundational Model Server (FMS), which in turn can be accessed over the Internet by other applications.

Queries to the FMS allow a human or computer to obtain different types of knowledge on a particular term. The user can discover the *branches of* the Ascending Aorta (i.e. the arteries that carry blood away from the Ascending Aorta), the *tributaries of* the Pulmonary Trunk (i.e. the veins carrying blood to the Pulmonary Trunk), what is *contained in* the Mediastinum, and a multitude of other queries.

The 3-D models (meshes) were created by tracing particular organs on image slices from a human body, and using these contours to create a 3-D surface. In the AIS the models are saved as 3-D mesh *primitives*, each corresponding to an individual structure part in the FM. The correspondence between 3-D primitive file names is managed by a 3-D model database, which for the moment is just a flat file. It is the task of the Dynamic Scene Generator and Graphics Server to interact with the user, the FM, and the 3-D model database to build, color, and render a 3-D scene for viewing on the web.

This task involves two major subtasks: creating and generating the scenes, the subject of chapter two, and fast rendering of the scenes, the subject of chapter three. This thesis makes contributions in both of these areas.

1.2 Contributions

Prior to development of the Dynamic Scene Generator interfaces, there existed a simple web interface and the Graphics Server. The web interface could be thought of as a Foundational Model query visualizer, allowing the user to add models to a scene based on queries to the FMS. The code base of the Graphics Server consisted of a *structure* class to store 3-D models, an *xsks-global-camera* module for controlling camera parameters, a *modeldb* module for simulating a 3-D model database, and a *scene-generator* module for generating a scene based on FMS queries. The Graphics Server was essentially complete, but contained some bugs that needed to be tracked down. The majority of the contributions to the Graphics Server were in the *scene-generator* module, in order to allow scene authoring.

Work for this thesis began with the creation of a suite of tools for Dynamic Scene Generation. A Scene Manager application was developed for organizing these scenes into exercises. Dynamic Scene Generator and Dynamic Scene Explorer interfaces were created for use by an instructor, in the former, and a student, in the latter. Both interfaces were implemented using HTML Frames, allowing an anatomy list and a currently selected structure frame to co-exist with a frame resembling the original interface. The Dynamic Scene Explorer interface exchanged the FM interface for a series of up to twelve *add-ons* shown as icons depicting structures that can be added/removed/highlighted in the scene, removing the necessity of the user being familiar with the Foundational Model system.

Additional FM capabilities were added to the scene generator, including highlighting and removing a hierarchy from a scene. The creation and loading of add-ons were added to the server as well as a *get anatomy* function that returns a list of structures that are currently in the scene.

The parallel rendering methods discussed in chapter three were implemented completely by the author, including a VRML 2.0 renderer, parallel processing algorithms,

and parallel processing evaluation functions. These were implemented by the author using a combination of Lex/Yacc, Mesa, Pthreads, System V IPC, and PVM libraries. The ultimate goal is to incorporate the most effective parallel implementation into the Graphics Server.

1.3 Thesis Overview

The thesis consists of two main topics: web interfaces for rendering 3-D anatomical structures, and parallel software 3-D rendering. Chapter two details the Digital Anatomist Dynamic Scene Generator Interfaces, as well as discussing the various servers that are used. Parallel Software Rendering Techniques are discussed in Chapter three. User evaluations of the Dynamic Scene Generator Interfaces and performance evaluations of the various 3-D parallel rendering implementations are discussed in Chapter four. Chapter five concludes the thesis and indicates potential future work.

Chapter 2

THE DYNAMIC SCENE GENERATOR AND GRAPHICS SERVER

2.1 Graphics Server

The Graphics Server consists of a *Scene Renderer* module, a *Scene Generator* module, and a *3-D Model Database* module. The purpose of the Graphics Server is to allow a remote user to generate an image of a 3-D scene. All of the modules are implemented using *Skandha4* (an in-house graphics manipulation toolkit[2]) in server mode, which allows Internet communication to one of several Skandha4 child processes through a socket. Figure 2.1 illustrates the relationship between the various server modules and the user interface.

Skandha4 is a modified version of Xlisp[1] incorporating a high level interface for OpenGL-style rendering. Current implementations are on Irix, which uses the actual OpenGL library, and Linux, using the freely available Mesa library. Skandha4 combines the flexibility of Lisp with powerful 2-D and 3-D rendering capabilities. Typical algorithm development involves initial implementation at the Lisp level, followed by computationally intense algorithms being moved to the C level to improve performance. Lisp allows algorithms to be developed and debugged quickly, reducing the amount of time spent in debugging. Xlisp adds object classes to Lisp, allowing the ability to create user defined types complete with methods and constructors. For example, the Xlisp *structure* class of the Graphics Server encapsulates several of the common actions performed on 3-D models in Skandha4. The next section discusses classes and modules used by both the Scene Generator and Scene Renderer.

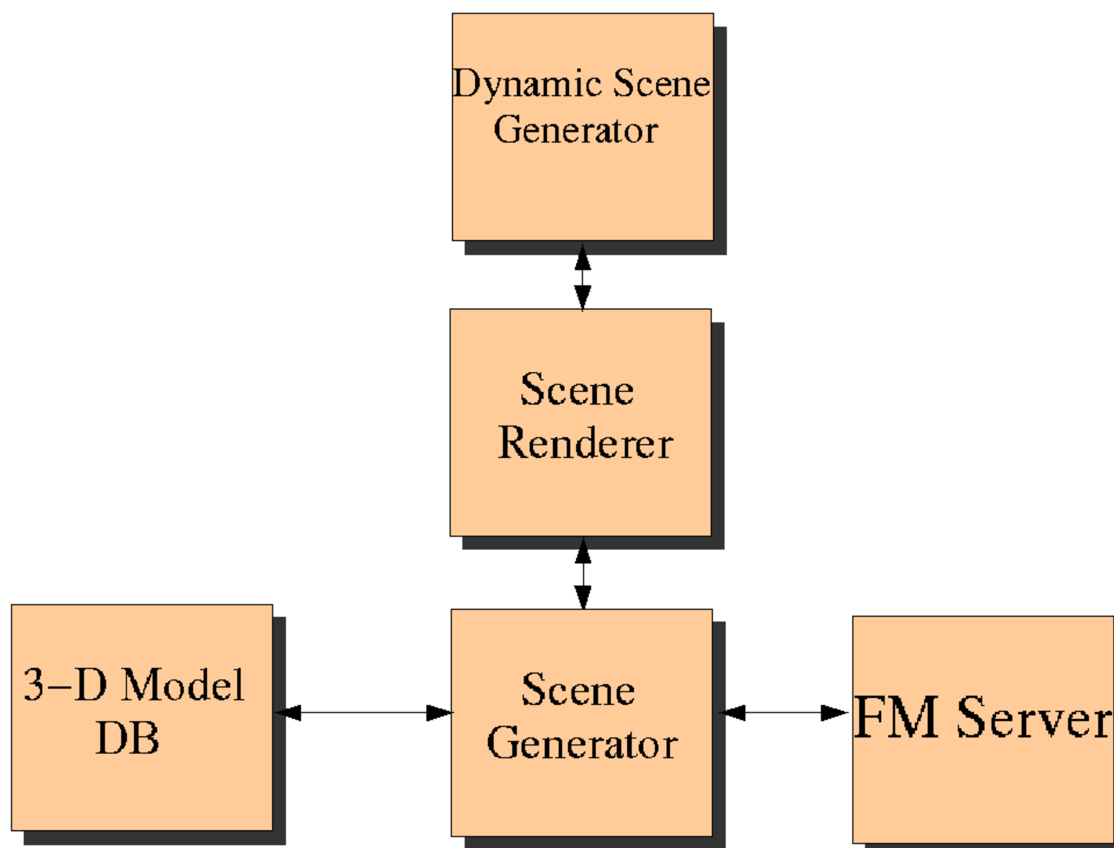

Figure 2.1: Block diagram of the components that interact with the Graphics Server

2.1.1 Common Classes and Interfaces

The primary purpose of the *structure* class is to allow the creation of a scene graph, represented as a tree of *structure* objects with a single root object. The *xsks-global-camera* module in the Graphics Server contains several functions for changing the orientation of the camera used to view the structures. The *Structure Interface* Skandha4 module provides wrapper functions for the various methods in the custom *structure* class and *xsks-global-camera* module. Typical functions include changing the camera orientation, adding/removing lights, adding/removing *structure* objects from the tree, finding a particular object in the tree, and creating a list of all the objects beneath a

certain node.

2.1.2 Scene Renderer Module

The *Scene Renderer* module calls on several of the functions described in the previous section to load and render a scene, as well as saving state between successive web browser connections. Saving state is only necessary when a client is used that fails to maintain state, such as CGI scripts. The function for saving state, *gs-save-state*, calls the *structure* method *Save-Yourself*, returning a list of structures complete with attached lights and materials. A file is then created which stores all of the commands necessary to recreate the current scene, followed by the commands to restore the camera to its current state. The function for loading state, *gs-load-state*, simply loads the state file that was saved by the corresponding save state call. The state functions are not needed for persistent connections, such as when a Java Applet connects to the Graphics Server.

2.1.3 FM Interface

The Graphics Server is connected to the Foundational Model Server (FMS) through a socket, allowing access to the several FMS API calls. Communication through a socket with Skandha4 requires calling the *Net-Eval* function, with the message to be sent through the socket as the argument to the function. The most common FMS API calls used by the Graphics Server are *kb-get-descendants* and *kb-get-ancestors*, each of which require a term name and a hierarchy to search. Kb-get-descendants is used when the user wishes to perform an action on a group of structures with respect to the current scene. Kb-get-ancestors is used on the *is-a* hierarchy to find a more general description of the particular structure (e.g. to find out that the Ascending Aorta is an artery).

2.1.4 ModelDB Interface

The ModelDB Interface simulates the presence of a 3-D Model database server which will be developed in the future. The model database consists of a hash table of pointers to *structure* objects containing 3-D models. The Foundational Model preferred term is used as the key for the hash table. The hash table is populated when the Graphics Server starts up by calling the function *modeldb-connect* which reads a text file that associates the preferred term name with the actual name of the model.

The function *modeldb-find-node* is called when a *structure* needs to be appended to the scene tree. Modeldb-find-node is also used to filter out all of the terms that don't have models from a FMS query.

2.1.5 Scene Generator

The Scene Generator is defined by an API that is used to create a new scene, or augment the current scene. The Scene Generator is a temporary solution that will eventually be replaced by an intelligent agent. There is currently a tentative plan to replace the Scene Generator with a remote server incorporating a natural language reasoning engine implemented in Prolog. One weakness of the current Scene Generator is that it is extremely dependent on the Foundational Model Server. If a term name is changed in the FM, the Scene Generator is unable to automatically adjust to the change. Although the Scene Generator lacks intelligence, it is adequate for developing the related systems as well as determining which features should be implemented in the next generation of intelligent agents.

The Scene Generator contains a group of functions used by a save state function in the Scene Renderer Module, which is used both to maintain state and save an authored scene. For example, the *load-anat-list* function generates a scene from a file containing a list of structure names. The Scene Generator also allows the ability to add, remove, or highlight a group of structures based on an FM query result. One of

the original scene generator functions is *get-material*, although extensive updating was required to provide the correct color properties to the various anatomical structures.

The function *cgi-action* finds all of the models that exist beneath a parent in a specific hierarchy and performs a specific action on the scene using the queried models. Actions can be *highlight*, *add*, or *dissect*. An example use of cgi-action would be to *highlight* all of the models that are a *branch of* the *Ascending Aorta*. The list of term names returned by the FM query is then pruned to contain a list of terms corresponding to 3-D models that exist with the help of the function *modeldb-find-node*. A helper function called *create-str-file* takes the pruned list and generates all of the commands necessary to perform a specific action on the scene and save them in a cache file with an 'str' extension. Additional queries are made to the FM Database while the str file calls *get-material* on each of the structure names. Queries to the FM Database are time consuming, so each query is cached, eventually reducing the amount of delays communicating to the FM Database.

Get-material contains a table of general anatomical terms and the corresponding material properties. The ancestors of a provided term are traversed in the *is-a* hierarchy until an ancestor matches one of the general terms in the material table. If the recursive function reaches the root node of the hierarchy without finding a material, no material is returned and an error log file is generated listing the term name. If a model does not have a corresponding material, the material table is in error and must be manually edited.

2.1.6 Client vs. Server Rendering

High performance computers are becoming more affordable by the common user, motivating the development of computationally intense 3-D client-rendering applications through the Internet. Examples of such applications are VRML and the use of the Java3D API. With today's computers, complex 3-D scenes are still difficult to realize through the less efficient client-rendering applications. The Graphics Server was

implemented to allow users with any computer, regardless of performance or operating system, to gain access to high-quality 3-D images. The Digital Anatomist Atlas models represent the threshold of usable performance for a single user with current computers. In order to accommodate several users concurrently, faster 3-D rendering techniques need to be employed.

The upper limit on the number of polygons in a usable client-rendering application scene will increase in the future, motivating concurrent development of a Java3D solution within our group[14]. Until Java3D is supported on all platforms and rivals the performance of Skandha4, the Graphics Server will be used for the primary source of dynamically generated 3-D images. The Pendragon[6] illustration package is an example of the flexibility and potential of applications developed with Skandha4 (e.g. the Graphics Server).

2.2 Interfaces

The previous section discussed the implementation and philosophy behind the Graphics Server. This section will shift the focus to the interfaces that allow the user to effectively utilize the various Graphics Server API to create anatomical *scenes* and *add-ons*. An important distinction exists between scenes and add-ons. A scene contains a group of structures, color properties, and camera orientation in order to create a specific image. An add-on, on the other hand, simply stores a list of structures that are present to be used to add, remove, or highlight groups of structures in future scenes. The rest of this section illustrates how a scene and add-on are created, as well as how they are used differently.

2.2.1 Dynamic Scene Generator

The Dynamic Scene Generator interface is a forms based CGI web application implemented in Perl. The interface is intended to be used by anatomists familiar with

the Foundational Model system and by anatomy educators. Typical uses would be to simply create high-quality anatomy images, or to build a group of add-ons to be used by a student constructing an elaborate anatomical scene with the Dynamic Scene Explorer.

The layout of the interface consists of three frames, as shown in Figure 2.2. The *FM Navigator* frame labeled in Figure 2.2 contains an interface to the Foundational Model *part-of* and *contained-in* hierarchy. The user clicks on a term to display a list of children in the part of hierarchy. If the term is a leaf node (i.e. has no children), the *contained-in* hierarchy is checked. The top of the term list displays the possible parent terms to ascend to. Navigating the FM interface allows the user to become familiar with how terms are organized as well as discover which terms can be represented as 3-D models. If a term has a corresponding 3-D model, a box appears next to the term name. A solid box indicates that the model is not currently in the scene, while a box with an 'X' in it denotes a model that exists in the scene.

By clicking on an anatomical term's box, the model becomes *selected*. A selected model triggers the top frame, labeled as *Current Structure and Scene* in Figure 2.2, to display the model name and which actions can be performed on it. If the model is not in the scene, it can be added, as shown in Figure 2.3, otherwise it can be *hidden*, *dissected*, *highlighted*, or *looked at*.

Highlighting a structure causes all other structures in the scene to become a grey transparent color. Technically, a lowlight is being performed on all but the selected structure, but most users are more familiar with the term highlight. Highlighting can be particularly useful for finding structures that are occluded by surrounding structures. The *look-at* feature alters the orientation of the scene in order to zoom up close to the selected structure. The user has the option to undo the *highlight* and *look-at* operations. If the highlighted structure is removed from the scene the highlighting is turned off.

The *hide* and *dissect* options effect the models available to the Dynamic Scene

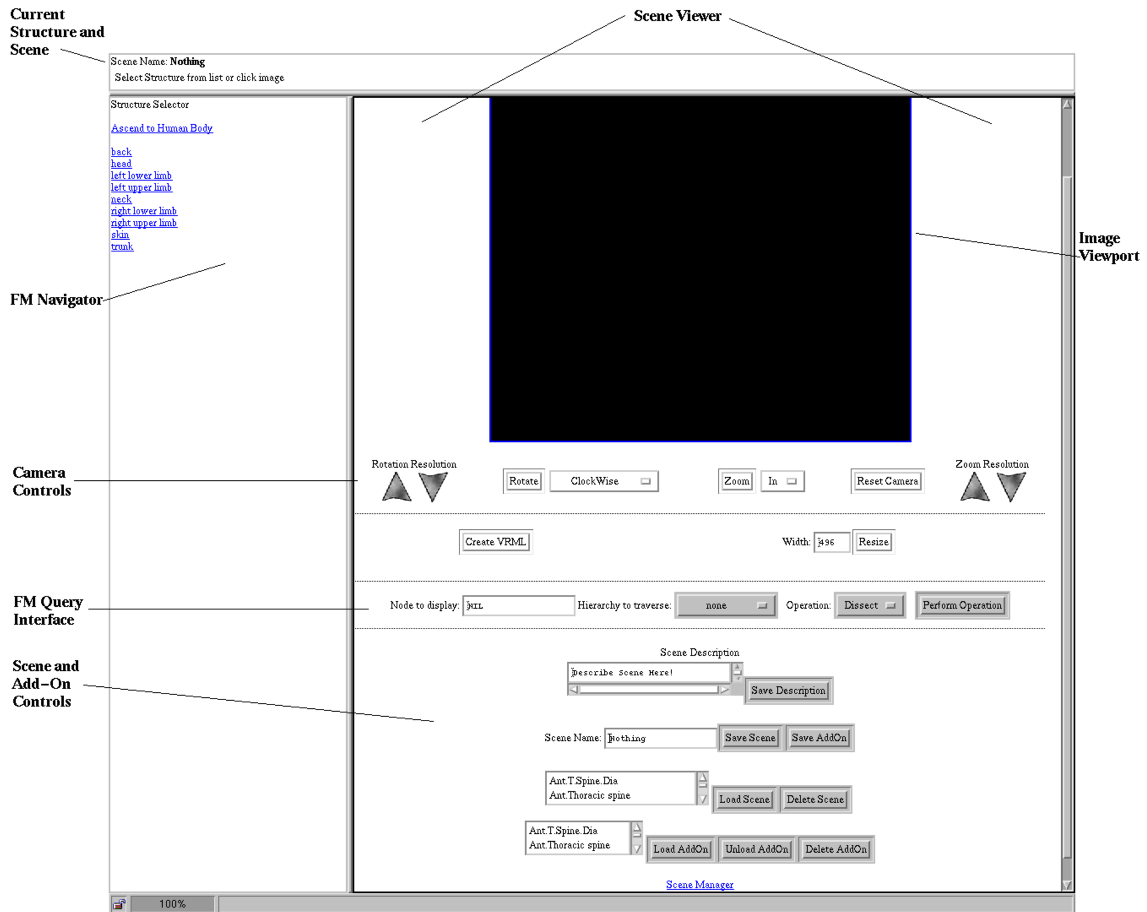

Figure 2.2: The initial state of the Dynamic Scene Generator in Netscape (the viewport was reduced in order to display all of the controls).

Explorer for the current scene. *Hidden* indicates that the model is not initially contained in the scene, but will be listed in the left frame. A *dissected* model will not be available to the user in the Dynamic Scene Explorer for this particular scene. If a model has been removed from the scene, the option to add it back into the scene is displayed.

The bottom right frame contains the focus of the interface, the *scene viewer*, as shown in Figure 2.2. The scene viewer consists of an image view port, camera controls,

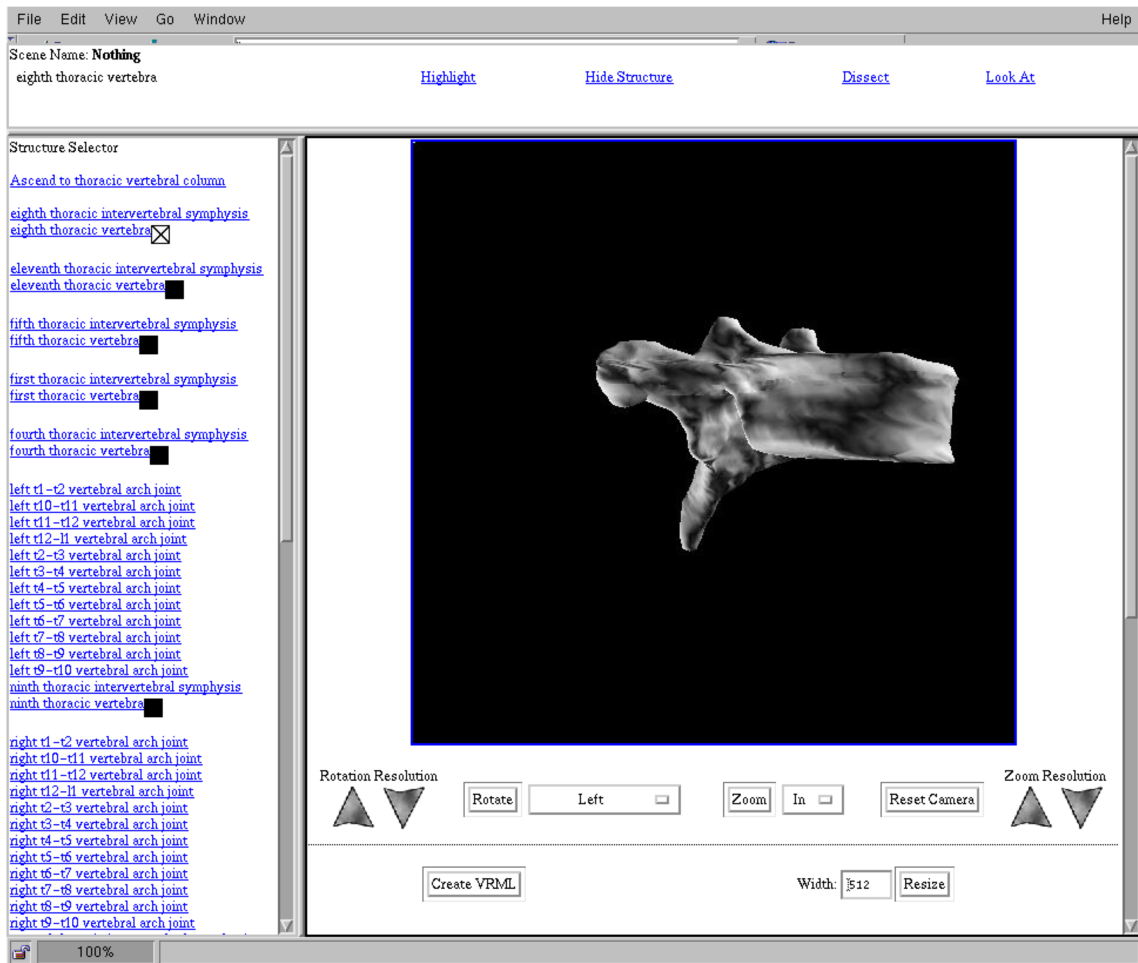

Figure 2.3: The Eighth Thoracic Vertebra added to an empty scene using the Dynamic Scene Generator

a Foundational Model query form, a view port size control, a scene description form, and an interface for saving and loading scenes or add-ons. The camera controls allow the scene to be rotated about the X, Y, or Z axis, as well as zooming in or out. All of the rotation and zooming controls use default increment values, which can be altered by adjusting the *rotation* or *zoom resolution*. A *reset* button allows the server to attempt to orient the camera in such a way as to fit all of the models tightly in the view port. Reset is a quick way to fix the camera when it is pointing in the wrong

direction. The view port begins with a blank image, so 3-D models must be added to the scene before altering the camera properties.

The scene viewer adds some powerful methods of creating scenes. After perusing the Foundational Model navigator in the left frame to become familiar with which anatomical structures have models, the Foundational Model query interface (bottom row of controls in the bottom right frame of Figure 2.3) can be used in the scene viewer to perform actions on complete hierarchies of models. Actions that can be performed are *Add*, *Dissect*, *New*, and *Highlight*, while the hierarchies that are currently available are *branch-of*, *is-a*, *part-of*, *tributary-of*, and *contained-in*. Selecting the *New* action will create a scene that contains only the models that were returned from the query. The *Dissect* action will remove the models returned by the query from the current scene. *Adding* is the reverse of *Dissecting* models. Highlighting a query results in an entire group of structures being highlighted. An example of the convenience of using the FM query interface is to *highlight* the *parts of* the *Vertebral Column* by setting the specific values in the *Node To Display* controls, as shown in Figure 2.4.

With a single command several structures are now highlighted, with all other structures transparent. Clicking on a particular structure in the view port will cause the structure name to appear in the top frame, as if it was selected from the FM navigator. The user can only click on structures that are visible in the view port, while any structure can be selected by clicking on the term name in the structure list.

Even with the power of the FM query interface, it is often desirable to create a scene that requires multiple FM queries. Custom lists of structures can be created by saving a current scene as an add-on. Complex scenes can be built by repeatedly loading different add-ons to the scene. If the author wishes to illustrate a specific aspect of the scene, either by adjusting the camera parameters or highlighting specific structures, the *save scene* button can be selected. If a scene is *loaded*, all structures previously in the scene are removed. The difference between loading a saved scene and loading an add-on is analogous to writing to a file or appending to a file. The

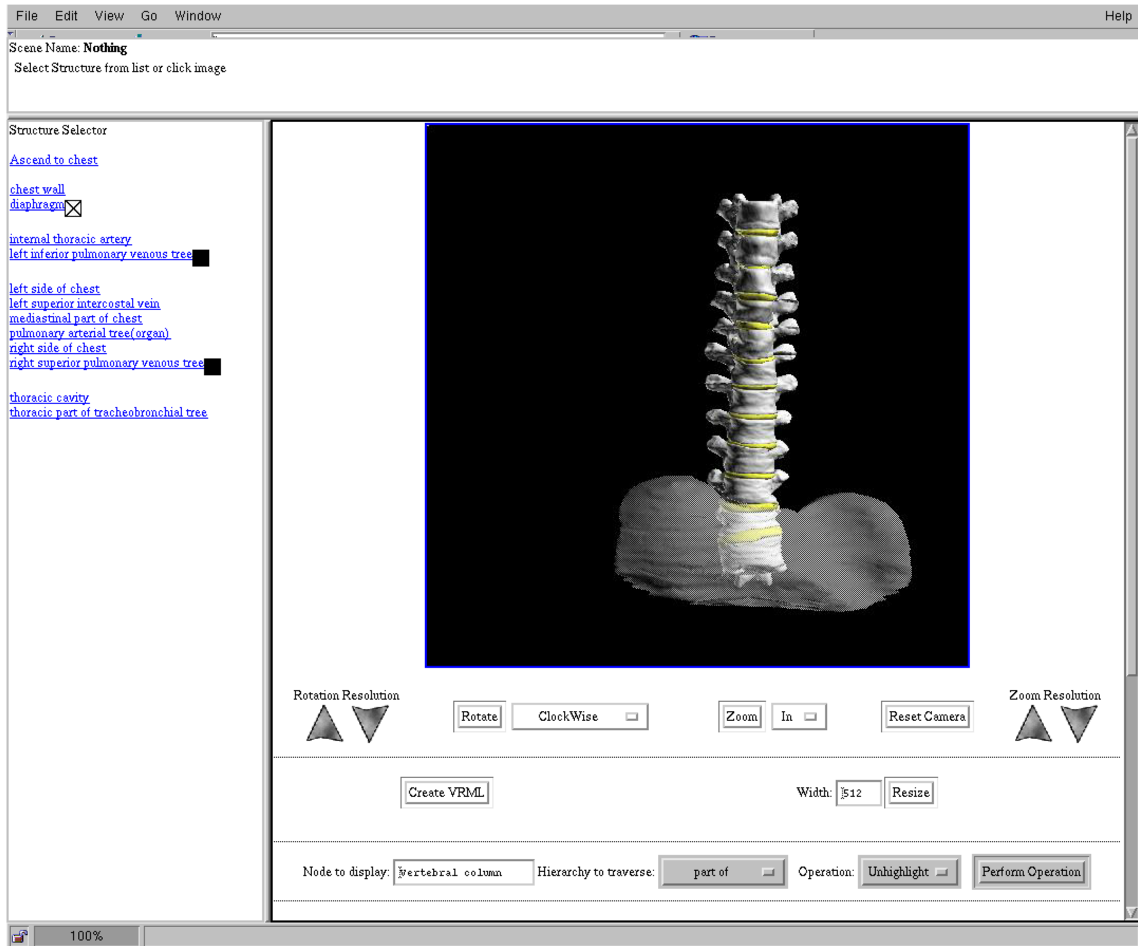

Figure 2.4: The *parts of* the *Vertebral Column* have been *highlighted*, resulting in the Diaphragm becoming transparent.

author can add an add-on to the current scene, but can not add a saved scene to the current scene.

2.2.2 Scene Manager

After a scene and a number of add-ons have been created, but before the end user can view the scene with the Dynamic Scene Explorer, the author must create a *group* using the Scene Manager. The Scene Manager is accessed by clicking on the *Scene*

Manager link at the bottom of the Dynamic Scene Generator(Figure 2.2). The Scene Manager is loaded into a separate browser window, allowing the user to interact with both interfaces at once. A group consists of an initial scene and up to twelve add-ons. Groups can be recursive, allowing a group to contain multiple groups, which each contain an initial scene and set of add-ons.

The Scene Manager interface, shown in Figure 2.5, represents scenes and add-ons by the names with which they were saved, plus a static image that was generated by the Dynamic Scene Generator upon saving the scene or add-on. The top row of the interface contains scrolling lists of scenes, add-ons, and groups. A new group can be inserted in the current group listing by using the *Add Group* field. The next row of the interface contains an area that shows the current initial scene for the selected group, an area for viewing snapshots of other scenes, and a text box for displaying and writing a group description. The third row contains several buttons for manipulating the Add-On Display Panel and Initial Scene Display Panel, navigating through the different groups, and launching the Dynamic Scene Explorer to preview the current group.

The controls for the Add-On Display Panel include *Add*, *Remove*, and *Save All*. By clicking *Add*, as shown in Figure 2.6, the add-on that is currently selected in the add-on scroll box will be added to the Add-On Display Panel, showing the name and snapshot of the add-on. Add-ons can be removed from the Add-On Display Panel in a similar manner using the Remove button. *Save All* will place all of the add-ons in the Add-On Display Panel into the selected group. Add-ons can individually be added to the selected group by clicking on the corresponding snapshot in the Add-On Display Panel.

The Initial Scene Display Panel in Figure 2.7 shows the scene that is currently the default for the selected group and allows a different snapshot to be viewed simultaneously by clicking the *View Init* button. Clicking the *Save Init* button or the previewed scene snapshot sets the previewed scene to become the initial group

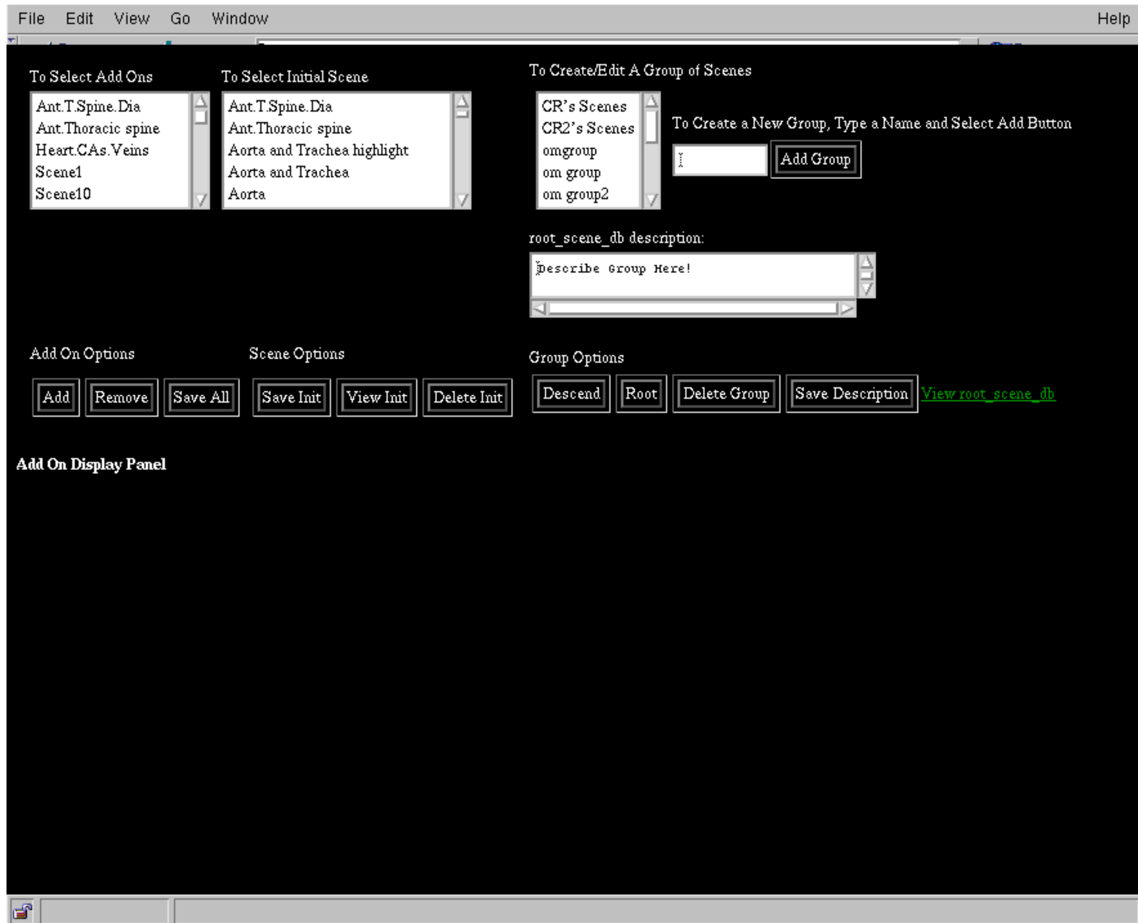

Figure 2.5: Initial appearance of the Scene Manager using Netscape

scene, while *Delete Init* removes the initial scene and image from the file system. The Group Options allow the user to *Descend* within the current selected group to view the groups or add-ons contained within. The *Root* button returns to the topmost group. *Delete Group* removes the selected group. The *Save Description* button saves the group description contained in the text area above. Once the author is satisfied with the initial scene and the selection of add-ons for a particular group, the scene can be evaluated using the Dynamic Scene Explorer.

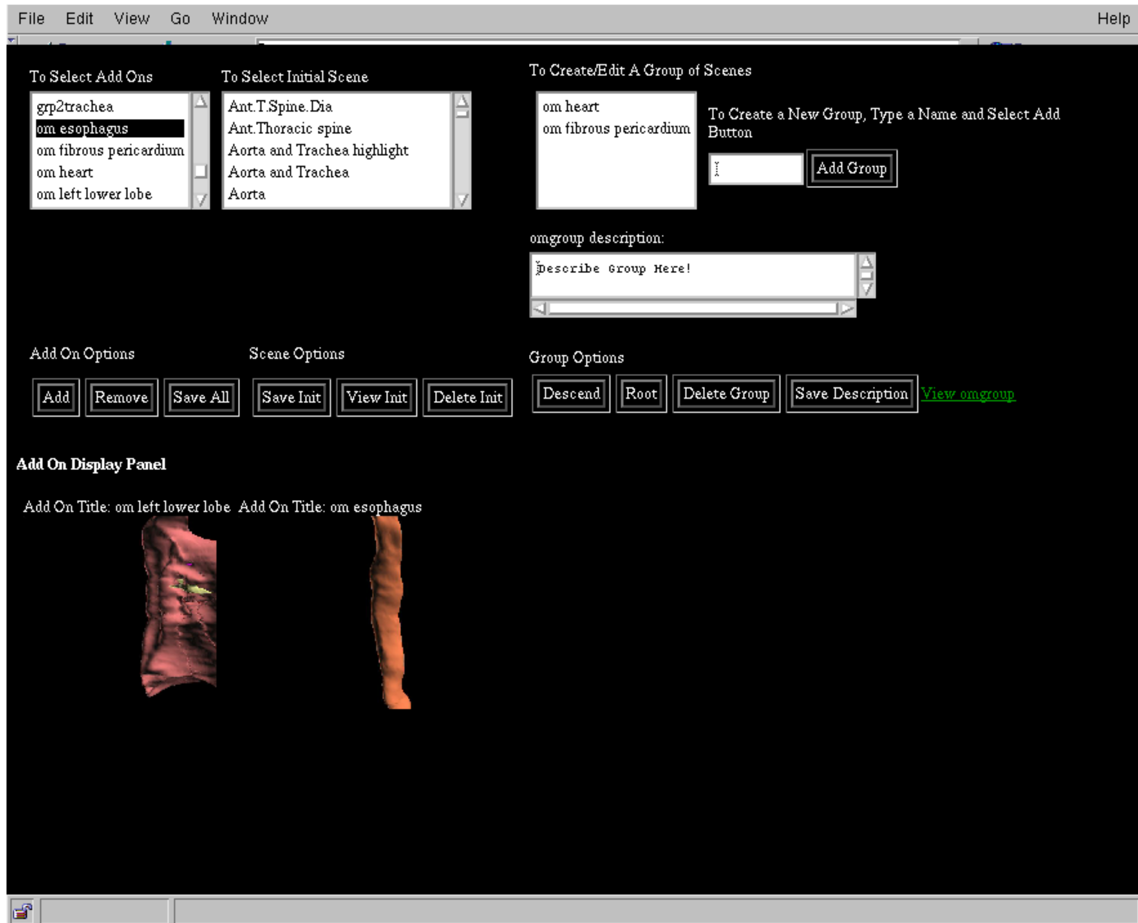

Figure 2.6: Adding the *om esophagus* add-on to the Add-On Display Panel

2.2.3 Dynamic Scene Explorer

The Dynamic Scene Explorer is designed to be used by people who are not acquainted with the Foundational Model. The interface is a stripped-down version of the Dynamic Scene Generator, as illustrated in Figure 2.8. The lower-left frame simply displays a flat list of models available for *adding* or *hiding* from the scene, instead of an FM browser. The top frame is essentially the same, although the user can not *dissect* structures. If a model was to be *dissected*, the user would be unable to add it back to the scene, since it would be removed from the structure list in the lower-left frame.

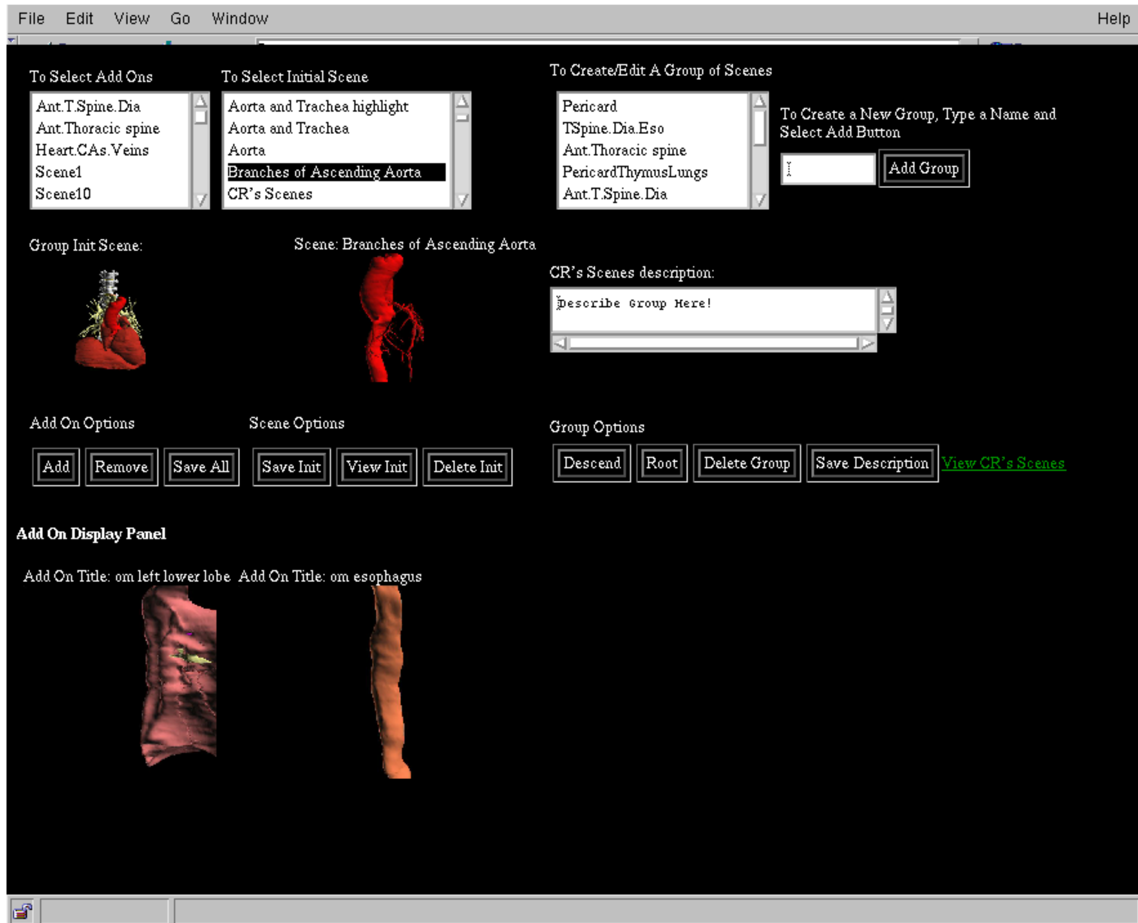

Figure 2.7: The initial scene for the group *CR's Scenes* is displayed on the left, while the scene *Branches of Ascending Aorta* is being viewed as a potential replacement.

Hiding a model simply removes it from the scene and changes the 'X' box to an empty box in the structure list. The user can add a *hidden* structure by selecting it, which will transfer it to the top frame, and clicking on *Add Structure* in the top frame.

The scene viewer in the lower-right frame consists only of the view port, camera controls, view port resize button, a *Submit Result* button, a collection of add-on snapshot icons surrounding the view port, and a list of add-on actions above the view port. The default add-on action is *Preview*, which causes a new instance of the

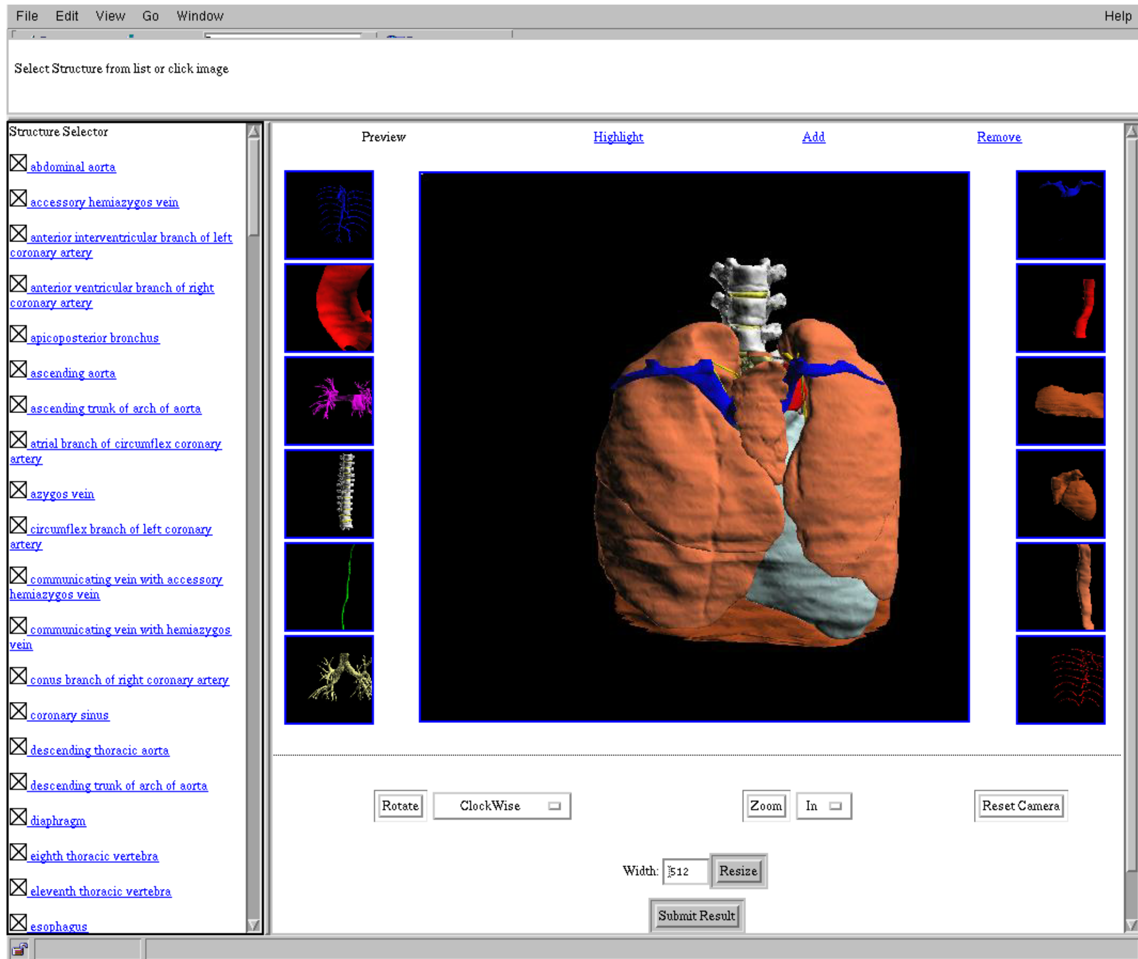

Figure 2.8: The group *CRexercise1* is previewed by clicking on the link *View CRexercise1* in the Scene Manager, resulting in the Dynamic Scene Explorer being launched in a new browser

Dynamic Scene Explorer to be executed in a new browser window in order to closely examine the add-on that was selected. If *Highlight* is selected, all of the structures listed in the add-on that are in the current scene will be highlighted. An add-on can be added to the current scene by first selecting the *Add* action, then clicking on the desired add-on icon. Removal of an add-on can be accomplished in a similar manner by selecting the *Remove* action.

The *Submit Result* button is used for academic exercises where evaluation of the user's performance is desired. When the user clicks the Submit Result button, a unique file is created storing all of the structures in the scene. An example of how this would be used is to construct a scene consisting of all of the anatomical structures that are in contact with the Vertebral Column. When the user is satisfied that his/her scene contains all of the anatomy in contact with the Vertebral Column, the Submit Result button is pressed. The instructor could then run a grading script which compares the result submitted by the student against the result generated by the instructor.

The next section delves beneath the user interfaces, describing some implementation details. Since the Dynamic Scene Generator and Dynamic Scene Explorer use the same infrastructure, their implementation shall be discussed as a single application.

2.2.4 Interface Implementation Highlights

The Dynamic Scene interfaces each consist of two CGI scripts implemented in Perl. The top level CGI script contains code for defining the HTML frames, as well as the actual code for the lower-left and top frames. The second CGI script contains the much more complex scene viewer code, used for viewing and transforming the scene. The *skandha module* is a collection of functions used for communicating between the Graphics Server and the CGI scripts, which is included in each of the CGI scripts by using the *require* keyword. The skandha module uses the Perl Telnet module for TCP/IP communication. Skandha4 outputs images in the TIFF image format, requiring the use of the PerlMagick module within the CGI scripts to convert the images to a format supported by web browsers (e.g. Jpeg or PNG).

Another module that is used throughout the interfaces is the *DB File* implementation of Berkeley DB. DB File allows the developer to create, edit, save, and retrieve various Perl data structures from a corresponding file. DB Hash is part of the DB File module, for the specific case of tying a hash reference to a file. By tying a DB Hash,

hash data is loaded from a file and stored into a hash reference to be manipulated. Untying the hash causes the updated hash to be written out, allowing a more efficient construct when compared to using simple files. All of the anatomy lists are the keys of tied DB Hashes, with a value of '1' indicating that the structure is presently in the scene (i.e. 'X' box) and '0' indicating that the structure is currently *hidden* (i.e. empty box). If the structure is not in the DB Hash, then the structure is only available through the Dynamic Scene Generator (i.e. solid box in the Dynamic Scene Generator).

Maintaining state between CGI script executions is accomplished by using Forms, which store the client state, and informing the Graphics Server which user's state to load with *gs-load-state*. The user ID is set by the process ID of the initial invocation of the interface script. The Form variable *Action* will for the most part contain the command to be executed by the client. The majority of commands on the client involve interacting with the Graphics Server API, unless the scene does not need to be rendered again, resulting in the CGI script loading the previously rendered image in order to reduce the server workload. If a scene or add-on is saved, the current structure list and view port image are copied to the specified name with file extensions '.anat' and '.jpeg', respectively.

The Scene Manager is a simpler interface that relies primarily on DB Hashes and a file naming convention to store and retrieve data. All scenes and add-ons are stored in a single directory with the file extension '.str' and '.aon', respectively. The corresponding scroll lists are populated by using Perl to open the directory and *grep* the scene or add-on file names. The group hierarchies are stored in a collection of DB Hashes with '.sdb' file extensions, while each group description is stored in a simple text file with a '.txt' file extension.

Chapter 3

PARALLEL RENDERING TECHNIQUES

An advantage of the server-based renderer described in the previous chapter is that the server can be run on high performance hardware, allowing almost anyone with a web browser to have access to dynamically-generated 3-D scenes. One way to increase the server performance even further is to develop parallel rendering methods. This chapter describes parallel rendering algorithms that could be incorporated into the Graphics Server.

3.1 Mesa and Software Rendering Overview

OpenGL [13] is a portable, interactive 2-D and 3-D development environment created by Silicon Graphics Incorporated (SGI). It defines a set of functions, or *application programming interface* (API), that allow the programmer to create graphics applications without requiring any knowledge of the underlying hardware that is being used to display the graphics. Mesa [9] is a freely available development environment that closely resembles OpenGL and is available on several different platforms, including Linux. All of the experiments conducted in this paper were implemented using Mesa 2.6 or Mesa 3.1 on Debian Linux machines. A 3-D API transforms an area of a 3-D *world* into a corresponding 2-D image, similar to how a camera takes a photograph.

A 3-D application developer needs to create a world, consisting of various light sources, 3-D objects, and an eye. The eye, or view point, corresponds to the Cartesian point in the world from which the viewing is taking place. The 3-D objects are created from a set of polygons, each of which are defined by a set of Cartesian points

representing its vertices. Each polygon also has a *material* associated with it, defining such properties as color and shininess of the polygon. The majority of the models used in this evaluation were constructed using triangles. The position of the lights as well as the *target* that they are aimed at and the *normal vector* of each polygon are used to alter the appearance of the object being displayed due to such factors as shininess and shadowing. The normal vector can easily be calculated by creating two vectors from three of the polygon vertices and determining their cross product. Once the world has been defined, the 3-D API can be used to define the *transformation* between 3-D and 2-D coordinates.

Before a 3-D scene is to be displayed in 2-D (i.e. *rendered*), either a perspective or orthogonal transformation must be defined. A perspective transformation is similar to how the eye observes objects in the natural world, where distant objects appear smaller than objects in the foreground. Painters began using perspective transformations in the 16th century (e.g. *The School of Athens*) to represent the third dimension of depth in a two dimensional canvas. An orthogonal projection transformation would cause two triangles of equal dimensions, but different depth values, to appear identical, while a perspective projection transformation would render the closer of the two triangles larger than the other. A special 2-D orthographic projection transformation for rendering images exists which limits all depth values to lie within the range of -1 and 1. Each transformation defines a volume, rectangular for orthographic projections, pyramid shaped for perspective transformations, within which all 3-D polygons will be rendered. Any polygons which lie outside of this bounding box will be excluded from the final 2-D image.

One other transformation of note is the *viewing transform.* The viewing transform is used to alter the view point by performing such operations as rotating about an axis and zooming. An entire 3-D world can appear to be rotating by moving the view point about an axis that intersects the center of the world, just as the Sun appears to rotate about the Earth to someone who does not know otherwise. Applying various

transformations to a set of polygons can drastically effect the resulting 2-D rendered image.

The 2-D image is stored in a *frame buffer*, which simply holds the color value for each *pixel* in the image. An additional buffer, called the *depth* or *Z buffer*, contains a spatial depth value for each corresponding pixel in the image. If a polygon is rendered into a set of pixels, each rendered pixel's depth is compared against the depth of the previously rendered pixel occupying the same slot in the frame buffer. The relation between the depth and frame buffers allow for the proper representation of occlusion between one part of an object and another. The final 2-D image can be contained in a window on the display of a computer, or (exclusively with Mesa) can be stored into a file.

OSMesa (off-screen Mesa) is an extension that was added to the Mesa 3-D rendering environment, allowing the developer the ability to create applications that render images without requiring a display, or even a graphics card. Web applications become more efficient by not having to add the extra functionality of controlling a display and rendering an image which, in addition, must then be dumped into an image file. The bulk of the applications developed in this paper rely on the ability of OSMesa to generate an image file without the use of a display.

Since OSMesa does not use a graphics card during the rendering process (i.e. software rendering), the server's CPU is left with the task of rendering the image. CPU's are general by design, with little or no optimizations for 3-D rendering. Graphics cards, on the other hand, can contain processors and support hardware specifically designed for the single task of rendering 3-D objects. Currently in the Linux world, 3-D hardware rendering is in development, forcing most Linux environments to use software rendering regardless of whether off-screen or on-screen rendering is being performed. Creating a method of increasing software rendering performance is essential for current 3-D applications, until a stable and complete Linux 3-D hardware acceleration library becomes the default development environment.

3.2 An Approach to Increasing Software Rendering Performance: Parallel Rendering

Several forms of image processing and volume rendering have benefitted from dividing the desired task into sub-tasks and executing each sub-task in parallel. A good candidate for parallelism can be assessed by determining whether the output from an operation on part of the input depends on any of the outputs from other parts of the input set. 3-D polygon rendering can be described as a sorting problem [11], with three general categories of sort-first, sort-middle, and sort-last.

A sort-first algorithm divides the viewport into several subregions. Each processor renders the entire list of polygons, but the majority of polygons are clipped due to the much smaller size of the subregion. An alternate implementation can involve a pre-processing stage where only the polygons contained within a subregion are sent to the corresponding processor to be rendered. A robust load-balancing algorithm is usually employed to assure that each processor performs an equal amount of work.

Sort-middle algorithms divide the list of polygons between different geometry processors, which are used to generate screen coordinates for each polygon, resulting in a set of screen-space primitives. The screen-space primitives are then sorted and distributed to a corresponding rasterizer. Each rasterizer is responsible for a specific subregion of the viewport, as in the sort-first algorithm. The sort-middle implementation for software rendering requires advanced knowledge of OpenGL to gain access to the middle of the rendering pipeline.

The algorithm selected for this work was the sort-last algorithm, which divides the polygon list equally between different processors, each of which renders an entire output image. The resulting images are then combined in a final stage by performing occlusion tests. The sort-last algorithm is easily implemented in Mesa, due to the ability to access the depth buffer through the OSMesa API. The limitation of the sort-last algorithm is the large amounts of data that must be examined in the final

stage, specifically an entire image and corresponding depth buffer for each processor. If a cluster of workstations is used to implement the sort-last algorithm, a high-speed network must be employed. The decision to implement the sort-last algorithm was based on two factors, the first of which is the affordability of multi-processor workstations and high-speed networks. The second factor for implementing the sort-last algorithm relies on the ease of implementation due to the accessibility of the frame and depth buffers through the OSMesa API. All other classes of parallel rendering require extensive work developing additional API calls in order to access the internal rendering pipeline of Mesa.

OSMesa includes additional API functions that allow access to the depth buffer corresponding to a particular frame buffer, as well as *context* switching. A typical procedure for performing a sort-last operation using OSMesa involves defining one context for each segment, rendering each segment, and combining each context's frame buffer by comparing depth buffer values. The motivation for the sort-last approach is based on the fact that smaller lists of polygons require less time to render. In addition, merging the context frame buffers will not depend on the number of polygons rendered, but simply the size of the viewport (i.e. constant, assuming the viewport size remains fixed). The following example illustrates the sort-last algorithm in more detail. Initially, a set of OSMesa contexts, frame buffers, and depth buffers are allocated. Each context and pair of buffers correspond to a separate parallel thread of execution. The parallel execution mechanism (e.g. threads, processes, etc.) is executed to spawn N_T children for parallel rendering, where N_T is usually the number of processors. Figure 3.1 illustrates the basic interaction between a master and children for the case where N_T is three.

The ith child checks to see which rendering pipeline it is in charge of and makes its context current by making an OSMesa API call. Setting the context indicates, among other things, which frame buffer and depth buffer is to be used, preventing other children from interfering with the rendering process. Initial scene settings are

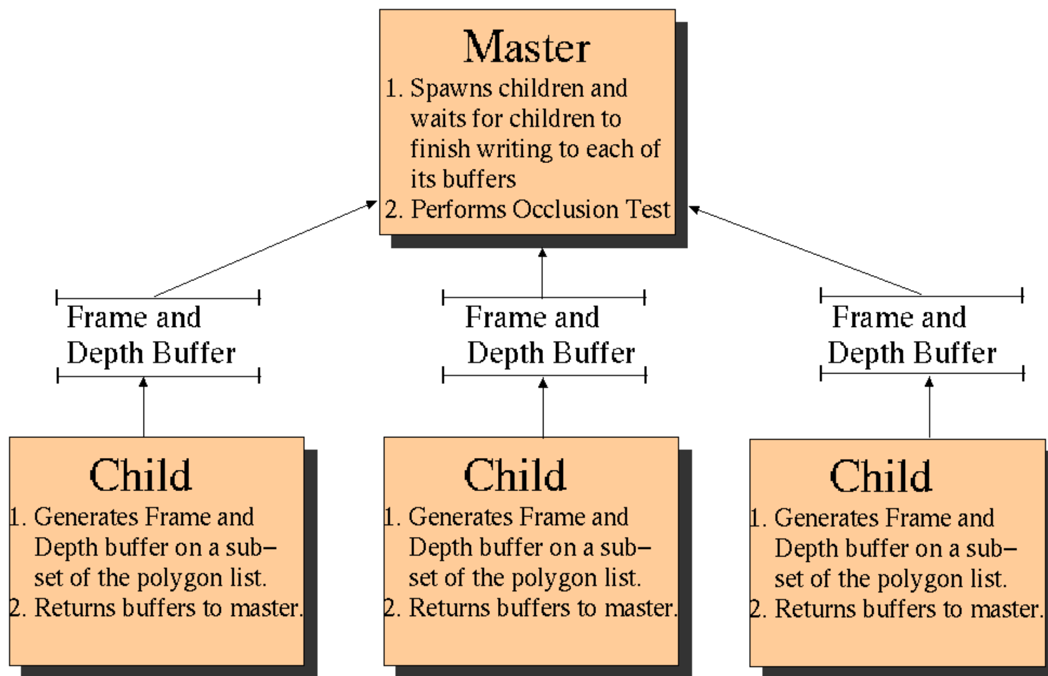

Figure 3.1: Sort-last implementation data-flow diagram for N_T=3 children

configured (i.e. lights, view point, etc.). The slice of the polygon list is determined by dividing the total number of polygons into N_T partitions and choosing the ith one.

After the ith partition is rendered, an OSMesa API call is made to fetch the depth buffer for this context. At this point, each child has a frame buffer of RGBA (Red Green Blue Alpha) pixel values representing a partition of the polygons, and a depth buffer for each pixel value indicating the corresponding Z coordinate value. The children may now exit and return control to the parent.

When the parent thread has detected that all of the children have completed execution, an arbitrary frame buffer is stepped through to test for occlusion. For

each index in the frame buffers, the depth buffer values for the corresponding pixel index are examined. When a depth buffer value is determined to be in front of all other depth buffer values for the specific pixel location, the pixel value of the corresponding frame buffer is selected as the final value. The execution time of the occlusion tight loop is directly proportional to the amount of parallel threads executed.

If bandwidth is an issue, a data structure can be implemented that holds only the pixels that are associated with a 3-D object. This can drastically reduce data transfer between the child and its parent, unless the scene fills the entire viewport of each child. The implementations evaluated in this thesis do not attempt to reduce the amount of data being transferred from each child to the parent.

3.3 Sort-Last Algorithm: Parallel Execution Using Processes and/or Threads

Parallel execution of a program on the Linux platform typically involves the use of spawning child processes or separate threads to execute an algorithm on a subset of the input data. A *process* can be thought of as an instance of an executing program which must share execution time with other processes, as determined by the operating system [16]. When a child process is spawned, the operating system must allocate the necessary resources, and any data needed from the parent process must be copied to the child process. The Parallel Virtual Machine (PVM) library is a special process API that allows the developer to spawn processes on other machines across a network in addition to the machine that the parent process is executing on. Communication between the child processes across the network is achieved using TCP/IP, the same mechanism used for communicating across the Internet. The advent of PVM has resulted in the creation of parallel processing systems consisting of large clusters of medium performance PC's.

Threads are similar in concept and application to processes, but threads inherently

share all of the data of the parent thread as well as requiring little or no operating system intervention for creating a new thread[12, pg. 25]. The thread interface translates into a more efficient method of dividing tasks within a process. The relatively new Linux Pthreads library was used for evaluation on this project.

3.3.1 Parallel Virtual Machine (PVM)

The PVM library can be used to distribute work between clusters of workstations on a local network, across the Internet, or any other medium in which TCP/IP is available. An effective cluster will be limited to being linked together with high speed Ethernet (100 MB/s). Unfortunately, the environment available for development is limited to standard Ethernet (10 MB/s), which may result in a decrease in performance due to data transfer overhead through the network.

PVM is also designed to work across heterogeneous networks, allowing several radically different architectures to work in parallel. But cross-platform compatibility does not come without its price. The PVM API is somewhat more complex than the standard single machine parallel processing mechanisms, and adds an additional amount of latency when packaging data to be transmitted to another process. Latency penalties can be reduced by sending a single large package of data rather than several small packages. A PVM-implemented distributed Graphics Server would consist of a master workstation, and several slave workstations. The Graphics Server would be loaded on the master workstation, while rendering processes would be running on each slave workstation. The rendering processes would each have a subset or all of the model data loaded into memory. The master would be responsible for load-balancing the slaves, which involves dispatching rendering jobs only to slaves that are idling. Threads and Processes naturally execute within the confines of a single workstation, resulting in a simpler architecture than the PVM parallel renderer.

3.3.2 Threads

The Linux Pthreads library is used to evaluate the performance of multi-threaded rendering. Linux Pthreads use a *lightweight process* [4] implementation. Lightweight processes exist in kernel space rather than user space, requiring only slightly less overhead than standard processes[12, pg. 199]. Although user threads require less overhead, they are unable to communicate across multiple processors, drastically limiting performance on multi-processor architectures.

The current release of Mesa is Mesa 3.1, which includes a significant performance increase over Mesa 2.6, largely due to the addition of Intel MMX instruction support. Regrettably, Mesa 3.1 is not thread safe, resulting in random core dumps during the execution of the rendering pipeline. As a result, the threads evaluation is forced to use the slower Mesa 2.6 library.

3.3.3 System V IPC

Parallel rendering using the process mechanism has the advantage of being able to harness the additional performance of using the Mesa 3.1 library over the slower Mesa 2.6 library. In order to assess the overhead differences between processes and threads, an additional evaluation of processes using Mesa 2.6 will be examined.

The System V Interprocess Communication (IPC) is required for transferring data generated in a child process back to the parent. The *fork* operation in Linux treats the data in the parent process as *copy-on-write* with respect to the child process (i.e. all of the parent's data is readable by the child and a private copy of the data is created for the child if a write is attempted). When a child process attempts to write into a section of memory inherited from the parent process, only the local child's copy of the memory is altered. After the child halts, all data written to memory within the child is lost. As a result, the buffers in the parent process are never modified, producing an empty image. By creating a shared memory segment, the child and

parent process share the same section of memory for a particular buffer. After the child process halts, the values written into the shared buffer are still present, since the parent process buffer points to the same location as the child process's buffer.

The shared memory segment was created and destroyed as part of the performance measurement, although a practical implementation would reuse the same shared memory segments throughout its lifetime, slightly reducing overhead. The performance evaluation will show that the additional overhead involved in managing shared memory segments is insignificant.

Chapter 4

RESULTS AND DISCUSSION

4.1 Interface Critique

The layout and features of the Dynamic Scene Generator (DSG) tools were developed with considerable input from Dr. Jim Brinkley, Dr. Cornelius Rosse, and Dr. Sara Kim. An initial evaluation was conducted by Dr. Onard Mejino, an anatomist involved in the Foundational Model project. Appendix B contains a complete account of Dr. Mejino's experience with the DSG tools. This section is devoted to summarizing the evaluation of the DSG tools.

The evaluation of the Dynamic Scene Generator interface revealed three shortcomings. The first request is to remove the *Rotation Resolution* buttons, replacing them with a form to supply a specific rotation amount. Instead of having to push the *Rotation Resolution* buttons three times to increment from the default rotation amount of 45 degrees to a new amount of 90 degrees, the user could simply input the value of 90 into a *Rotation Resolution* form.

The next request is to allow the user to click on a term in the FM navigator and add all of the models beneath the selected node. The current implementation only allows adding a single model through the FM navigator.

The final request is to have the camera in the Scene Renderer positioned in a way that results in the scene always being centered within the viewport. The camera is currently aimed at the origin of Cartesian space. Each 3-D model is loaded into the coordinates that they would occupy within the Thorax, resulting in several structures being centered about a point other than the origin. The camera can be sent a *frame-*

things message, informing it to try and fit the scene within the viewport, although the camera position will not be modified severely. As a result, the initial camera position should be a good distance away from the Thorax. If the camera is positioned inside or close to the surface of the Thorax, the *frame-things* message will not alter the camera orientation enough to capture a small scene of structures that are located near the boundaries of the Thorax (e.g. a scene consisting of only the Hemiazygos Vein).

The critique of the Scene Manager revealed several layout alterations and convenience options that should be incorporated into the interface. Changes to the layout include using shorter labels in the viewing areas and swapping the location of the *Scene* scroll box with the *Add-On* scroll box.

The convenience options consist of shortcuts that allow the user to work more efficiently. A *Reset* button would clear all of the viewing areas with a single command, instead of the user having to select each image and remove it. Adding the ability to select multiple items in the scroll boxes (i.e. holding *Shift* or *CTRL* while selecting items) would reduce the amount of time spent selecting add-ons for a scene. Using a button instead of a hyper-link would make the *view scene* command more visible. Some additional features requested can be implemented in Java, but not HTML, such as double clicking on an item to trigger an action.

The Scene Explorer comments dealt primarily with the add-on interface of thumbnail images surrounding the viewport. Suggestions included the ability to reverse the effects of highlighting an add-on. An indication of which add-on was used last, perhaps by displaying an image icon with a white background instead of a black background, would help the user keep track of the current state of the scene and what should be done next.

4.2 Parallel Rendering Performance

Rather than spending several days incorporating and debugging various parallel mechanisms into Skandha4, a simple environment was created to simulate Skandha4's rendering process. The simulation environment consisted of a VRML 2.0 renderer, implemented with Lex, Yacc, and Mesa. The VRML 2.0 renderer allows the user to define, by setting an environment variable, whether the rendering method uses System V IPC, Pthreads, PVM, or on-screen rendering. The term *threads* will be used to refer to all of the parallel mechanisms in general, while Pthreads will be reserved for specific references. Performance results will determine which, if any, of the rendering methods should be incorporated into Skandha4.

All VRML rendering performance measurements were calculated based on the amount of time spent converting a list of polygons to a final array of pixel values. The Graphics Server stores all of the 3-D anatomical structure models in memory, eliminating load time from the rendering process. Equation 4.1 illustrates the major components that determine the parallel rendering time (T_P).

$$T_P = \frac{T_N}{N_T} + T_B + T_O \tag{4.1}$$

$$T_B = \frac{(DepthBufferBPP + FrameBufferBPP) * N_T * N_P}{DataRate} \tag{4.2}$$

$$T_O = MinTestTime * N_P * N_T \tag{4.3}$$

$$N_P = PixelSizeofImage$$

$$N_T = NumberofThreads$$

The value T_N represents the amount of time the specific scene takes to render, without using any parallel mechanisms. The amount of time required to transfer the data from the threads back to the parent process is represented by T_B. For the Pthreads and System V IPC cases, the *Data Rate* will be much greater than amount of data being transferred, resulting in a T_B value approaching zero. The T_O represents

the time required for performing the final occlusion tests, and is measured after the children have transferred the frame and depth buffers to the parent. The occlusion test consists of testing for the minimum value of each of the N_T depth buffers at each of the N_P pixel locations. An implementation must be *execution-bound* in order for improved parallel performance (i.e. the data-transfer time, T_B, must be less than the normal tendering time, T_N, for a single thread), as equation 4.4 shows. If the I/O Time dominates equation 4.4, the implementation is classified as *I/O bound.* An I/O bound implementation will decrease in performance as more threads are used.

$$N_T * (I/OTime) < \frac{ExecuteTime}{N_T} \tag{4.4}$$

All evaluations were conducted on a quad 550 MHz Pentium III Xeon server running Debian Linux Potato. The PVM evaluations added a dual 550 MHz and a single 500 MHz Pentium III Xeon workstation. The computers are connected to the lab network via a 10 MB hub. The following sections discuss the performance of each method using one, two, four and ten threads with a set of four images of various amounts of triangles, shown in Figure 4.1. The ten thread case is used to examine the effects of more threads than processors for each of the implementations.

Each test scene was generated by using the *Create VRML* feature of the Dynamic Scene Generator. The VRML parser extracts a vertex list, facet index list, and material list from the VRML file. The camera and two lights are hard-coded into the rendering algorithm. The tight-loop within the renderer checks if a new material needs to be set for each triangle, as well as calculating each normal vector.

The VRML 2.0 renderer was not intended to be an optimal implementation, but rather an environment that closely gauges the trade offs between the various parallel mechanisms. Any additional optimization to the rendering algorithm should be magnified by the effect of the particular parallel implementation.

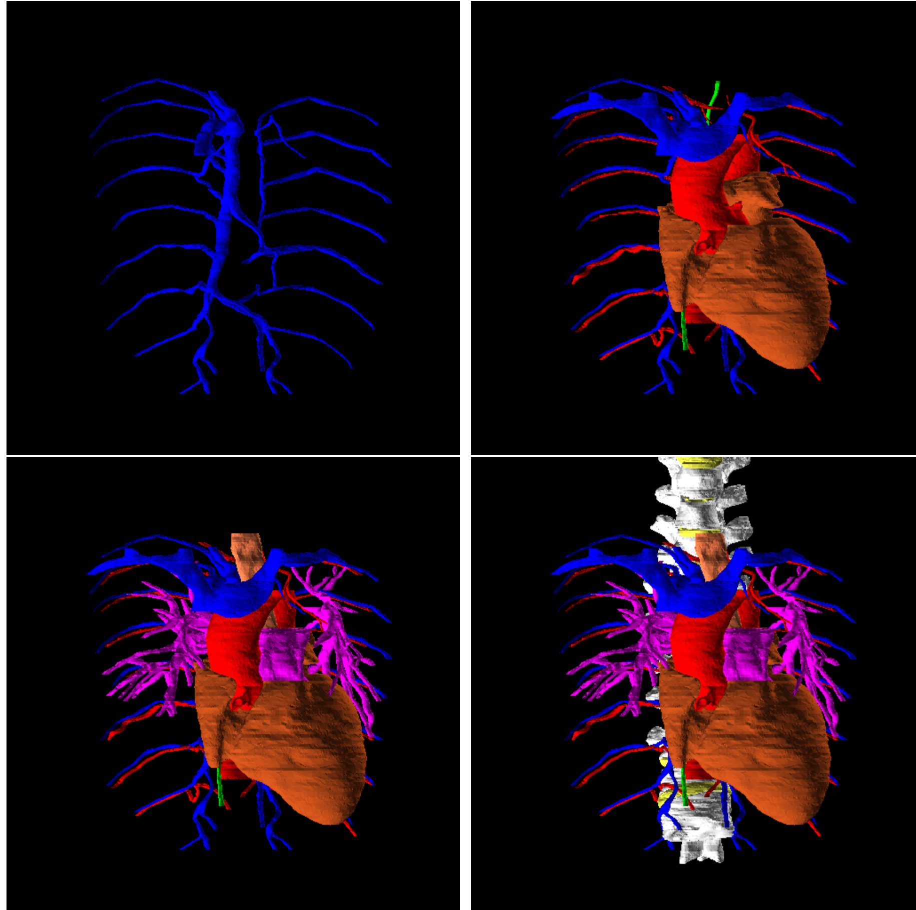

Figure 4.1: VRML 2.0 scenes used for performance evaluations. Triangle counts are 36535 (top left), 118638 (top right), 171515 (lower left), and 414710 (lower right)

4.2.1 Pthreads Performance

Due to Mesa 3.1 not being Pthread-safe, the Pthreads implementation was not able to benefit from the performance increase of the MMX instructions. Table 4.1 shows the amount of time required to render each of the separate images, as well as the time required for occlusion testing needed to produce the final image.

The occlusion test time is relatively insignificant for larger amounts of polygons. Even with ten threads, the occlusion test time is only about ten percent of the best

Table 4.1: Results of Pthread 3-D rendering test cases. Values are shown as a sum of rendering and occlusion test time in seconds.

Pthreads 3-D System Rendering Time in Seconds				
	No. of Threads			
Triangles	1	2	4	10
36535	0.27	0.16+0.01	0.11+0.02	0.27+0.07
118638	0.87	0.46+0.01	0.28+0.02	0.42+0.07
171515	1.25	0.65+0.01	0.36+0.02	0.59+0.07
414710	2.91	1.53+0.01	0.79+0.02	0.96+0.07

time for rendering the largest test scene, indicating that the Pthreads implementation is execution-bound. For all of the cases where the number of threads is less than or equal to the number of processors on the workstation, the parallel rendering performance exceeds the normal rendering performance. The actual performance increase over normal rendering is examined in Table 4.2. Ideal ratios should equal the number of threads used for the particular test case.

The performance ratio is further away from the ideal as the number of threads increases, indicating an increase of overhead. The extreme case of ten threads still performs well on larger sets of triangles, although the quad processor performs best with one thread per processor.

4.2.2 System V IPC

The System V IPC performance evaluation uses Mesa 3.1, resulting in improved rendering times. Each test attaches the children to a newly created shared memory segment. The performance benefit of keeping a shared memory segment active over several renderings is negligible, although if memory is abundant it may be more

convenient.

The results in Table 4.3 illustrate the benefit of combining the MMX enabled Mesa 3.1 with parallel rendering. By comparing the single process times against the single thread times, performance increases approximately 25% by using Mesa 3.1. As with the Pthreads evaluation, the ten threads case performs worse than the four threads case, indicating that the resources required to manage the threads and memory segments increase with the number of threads spawned.

The Mesa dependency can effectively be eliminated by calculating the speed-up ratio, shown in Table 4.4. A comparison between the System V IPC speed-up ratio versus the Pthreads speed-up ratio indicates that Pthreads are slightly more efficient than processes.

A direct comparison using Mesa 2.6 yielded one tenth of a second faster performance of Pthreads over System V IPC. Until Mesa 3.1 becomes Pthread safe, the slightly more efficient Pthreads are no competition for the more robust System V IPC implementation.

Table 4.2: Results of Pthread 3-D rendering test cases. Values are shown as a speed-up ratio, using the single Pthread time as a reference

Pthreads 3-D System Rendering Speed-up Ratio				
	No. of Threads			
Triangles	1	2	4	10
36535	N/A	1.59	2.08	0.79
118638	N/A	1.85	2.90	1.78
171515	N/A	1.89	3.29	1.89
414710	N/A	1.89	3.59	2.88

Table 4.3: Results of System V IPC 3-D rendering test cases. Values are shown as a sum of rendering and occlusion test time in seconds.

System V IPC 3-D System Rendering Time in Seconds				
	No. of Processes			
Triangles	1	2	4	10
36535	0.22	0.13+0.01	0.11+0.02	0.21+0.07
118638	0.67	0.36+0.01	0.22+0.02	0.33+0.07
171515	0.96	0.51+0.01	0.29+0.02	0.43+0.07
414710	2.21	1.17+0.01	0.62+0.02	0.73+0.07

Table 4.4: Results of System V IPC 3-D rendering test cases. Values are shown as a speed-up ratio, using the single process time as a reference

System V IPC 3-D System Rendering Speed-up Ratio				
	No. of Processes			
Triangles	1	2	4	10
36535	N/A	1.57	1.69	0.79
118638	N/A	1.81	2.79	1.68
171515	N/A	1.85	3.10	1.92
414710	N/A	1.87	3.45	2.76

4.2.3 Parallel Virtual Machine Performance

While the previous parallel mechanisms investigated deal with dividing a task into separate threads to execute on multiple processors within a single workstation, PVM allows threads to execute on multiple workstations through a network.

To simulate the distributed server architecture of PVM, a master process was

created. The master process spawns slave processes, which consist of the VRML 2.0 renderer. The slave task sends a message to the master after parsing the VRML model. The master begins timing after all slaves have indicated they are ready to render. Each slave renders the subset of the VRML model that it is responsible for, and sends the depth and frame buffers back to the master process for occlusion testing.

The limiting factor with the PVM architecture is the network bandwidth, represented by T_B for the specific case of the parallel renderer. Several bandwidth testers are available, allowing the developer to determine the most effective encoding technique for packing PVM messages. *In Place* encoding places the data in PVM packets without modification, as the name implies. Several other encoding techniques exist, the majority of which are designed to allow communication between different types of machines. Although In Place encoding was determined to be the most efficient, producing an average bandwidth of 7.86 MB/s across the 10 MB network used for the evaluation. In Place encoding is only guaranteed to work with homogeneous clusters, since transferring data between two different machine architectures may corrupt the data (e.g. big endian versus little endian).

A typical transmit time for a four bytes per pixel frame buffer and a two bytes per pixel depth buffer corresponding to a 512x512 pixel image is 1.6 seconds, based on the empirical bandwidth of 7.86 MB/s. An I/O time of almost two seconds is much higher than the execution time required for rendering. A network consisting of at least 100 MB Ethernet connected by switches could be predicted to yield 78.6 MB/s, using the empirical PVM bandwidth and assuming performance would improve by a factor of ten. If the PVM bandwidth would in fact increase to 78.6 MB/s, the transmit time required for a set of buffers would decrease to the more reasonable time of 0.16 seconds.

Two sets of evaluations were performed with the PVM library, an evaluation where the quad server was linked to a dual and single processor workstation, and an

evaluation on the quad server alone. Examining the performance of PVM on a single multi-processor machine may give insight as to how effective PVM might be on a high performance cluster. Although the data transfer rate between processes on a single machine using PVM will most likely be faster than any network available, memory access will be delayed. This is a result of all the processes on the same machine having to share a single memory bus, while processes running on different machines have their own memory busses.

Table 4.5 compares the performance of PVM on a network versus the standard rendering time. The average rendering time increases with the number of processes, regardless of the number of triangles. According to equation 4.4, the data indicates that the PVM implementation is I/O bound.

Table 4.5: Results of PVM 3-D rendering test cases over a 10 MB/s hub connected network using a dual 550 MHz Pentium III Xeon, 500 MHz Pentium III Xeon, and quad 550 MHz Pentium III Xeon. Values are shown as a sum of rendering and occlusion test time in seconds.

PVM 3-D System Rendering Time in Seconds				
	No. of Processes			
Triangles	1	2	4	10
36535	0.27	3.18+0.01	4.74+0.02	11.20+0.07
118638	0.87	3.45+0.01	4.82+0.02	10.99+0.07
171515	1.25	3.62+0.01	4.73+0.02	11.00+0.07
414710	2.91	2.60+0.01	5.06+0.02	11.20+0.07

Due to the primitive network used for evaluation, Table 4.5 is not a clear indication of PVM's potential. Table 4.6 shows the performance of PVM localized to just the quad server.

The results of Table 4.6 compared against the System V IPC performance (Ta-

Table 4.6: Results of PVM 3-D rendering test cases on a quad 550 MHz Pentium III Xeon. Values are shown as a sum of rendering and occlusion test time in seconds.

PVM 3-D System Rendering Time in Seconds				
	No. of Processes			
Triangles	1	2	4	10
36535	0.27	0.23+0.01	0.32+0.02	0.69+0.07
118638	0.87	0.46+0.01	0.43+0.02	0.78+0.07
171515	1.25	0.61+0.01	0.51+0.02	0.85+0.07
414710	2.91	1.22+0.01	0.77+0.02	1.05+0.07

ble 4.3) demonstrate the latency of passing messages with PVM. The PVM latency impairs the scalability of a distributed system, even with an ideal network. As the number of processes increase, the performance gain decreases, indicating that the I/O time is slightly less than the execution time (see equation 4.4).

4.2.4 Parallel Processing Architecture Comparison

To summarize the results obtained from the timing measurements in the previous sections, a direct comparison of speed-up ratios between the different parallel processing architectures is presented. The effect of the number of triangles in a scene on performance can be examined by comparing the next two graphs. Figure 4.2 shows the speed-up ratios of each rendering technique for the small test case of 36535 triangles.

Notice that the System V IPC speed-up ratio mirrors the Pthreads ratio, with just the slightest degradation in performance. The next graph, shown in Figure 4.3, shows the same comparison, but with a much larger scene consisting of 414710 triangles.

Figure 4.3 shows larger speed-up ratios versus the graph in Figure 4.2, indicating that as the triangle count increases, the execution time dominates over the amount of

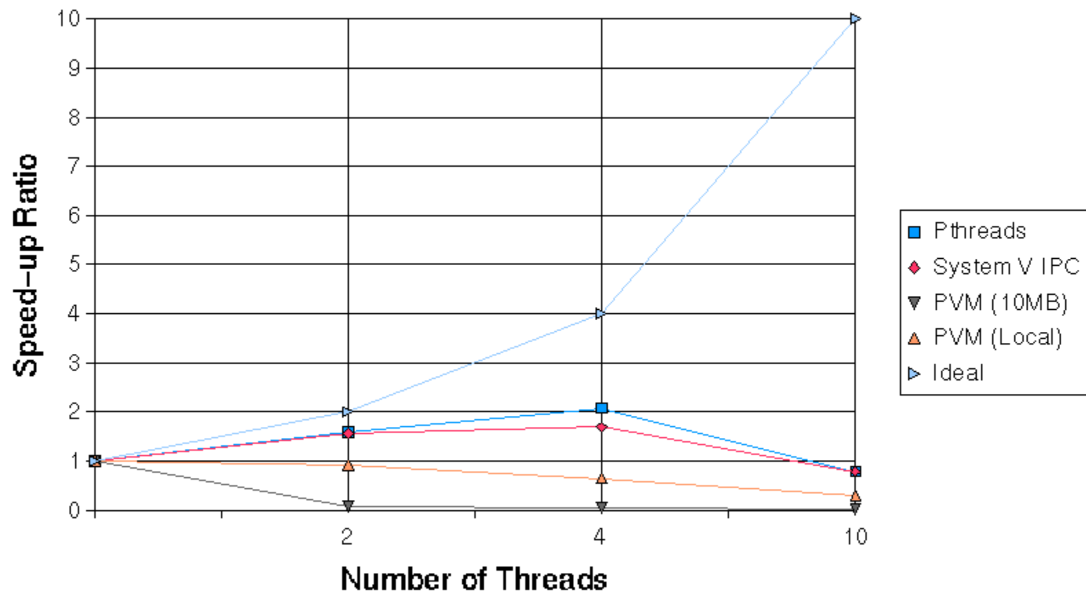

Figure 4.2: Speed-up ratios of each parallel processing implementation compared against an ideal ratio for a triangle count of 36535.

time required to spawn and manage children. Another useful observation from Figure 4.2 is that for the local machine implementations using ten threads on a small scene, the speed-up ratio approaches one, indicating that the execution and I/O times are nearly equal.

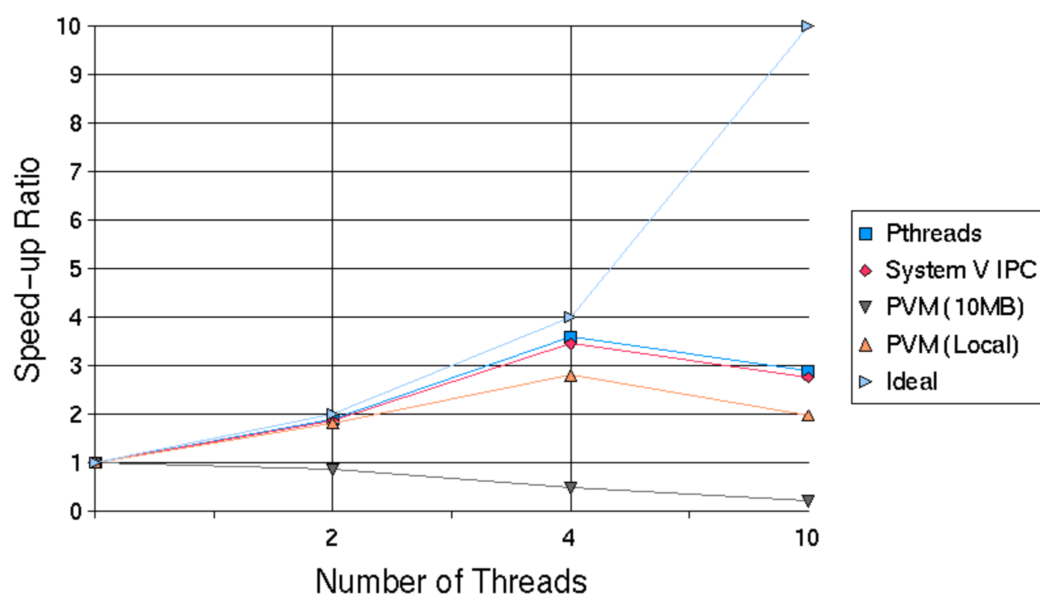

Figure 4.3: Speed-up ratios of each parallel processing implementation compared against an ideal ratio for a triangle count of 414710.

Chapter 5

CONCLUSION

This thesis has described a complex client-server architecture for dynamic scene generation and rendering over the web. The previous two chapters discussed work on both the interfaces, and on the server. The following two sections present conclusions and future work for each of these.

5.1 Interfaces

The Dynamic Scene Generator (DSG) tools evaluated in chapter five demonstrate that an anatomist familiar with web browsers can successfully create educational anatomical exercises. In order to reduce the amount of time required to generate multiple exercises, several shortcut commands should be added. Typical shortcut commands include removing all of the selected add-ons and scenes from the Scene Manager viewing panels, and using the FM navigator to add an entire subtree of models into a scene. But interface development is ultimately limited by the development environment, HTML and CGI in this case. More advanced interface development environments include Java-script, for augmenting HTML web pages, and Java Applets, applications that run within a browser by using a Java Virtual Machine Plug-in. Java Applets are the more powerful of the options described, although only more recent web browsers provide adequate support to run most Applets. Future versions of the DSG tools should be implemented as Java Applets, allowing the developer the ability to select from a wide range of GUI and image manipulation libraries.

5.1.1 *Future Work*

A Java interface to the Skandha4 based *Brain Mapper*[10] application, shown in Figure 5.1, is currently being developed, allowing a finer level of interaction between the client and server. The success of the Brain Mapper Interface has motivated porting the Dynamic Scene Generator Interfaces into a suite of Java Applets. Java allows the developer to use more powerful GUI components, maintain state on the client, and perform powerful 2-D and 3-D image operations.

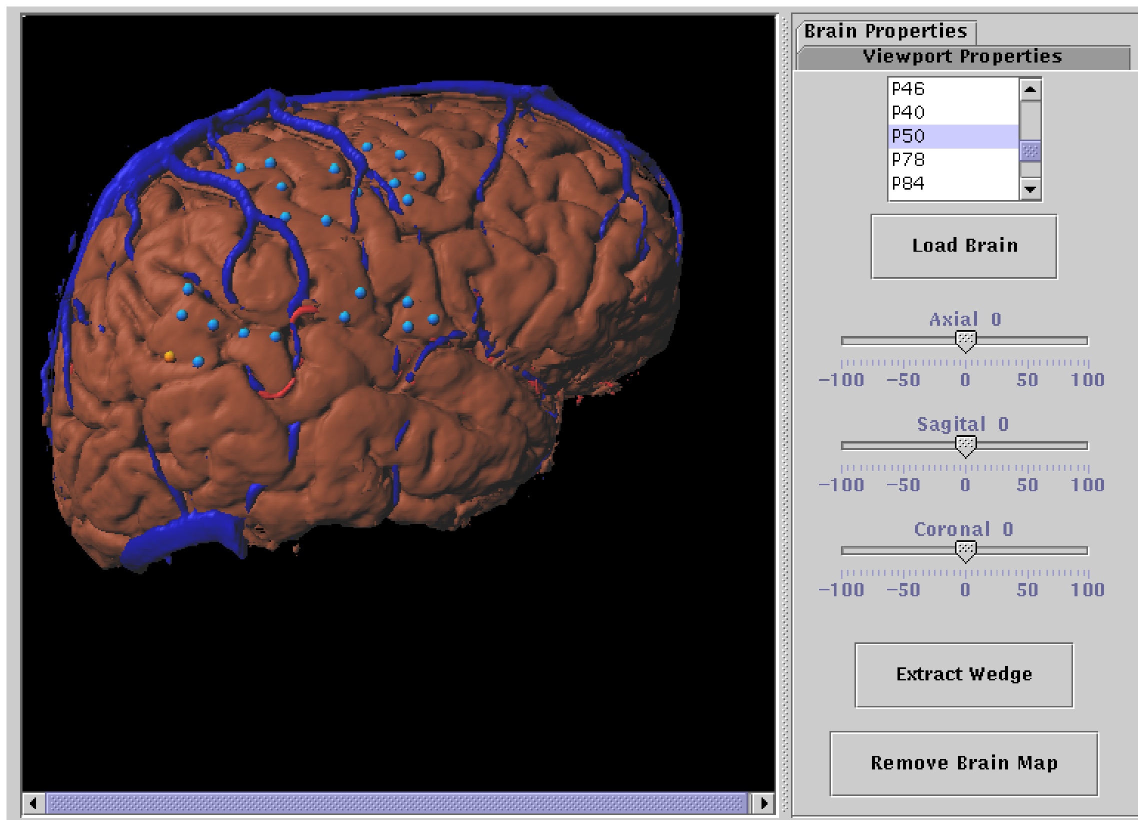

Figure 5.1: Brain Mapper Java Swing Applet interacting with the Skandha4 brain server. Advanced GUI features include sliders, a split panel, and a tabbed panel. The split panel contains the viewport in the left panel and a tabbed panel containing two sets of controls in the right panel.

The Brain Mapper Interface would require slight modifications in order to interact with the Graphics Server. Any information that needs to be written to disk must be handled by the Graphics Server, due to Java Applet security issues. The Scene Manager would need to interface with a more elaborate database, since the current database is a collection of references to Berkeley DB within CGI scripts.

5.2 Parallel Rendering Techniques

Three parallel processing mechanisms were evaluated for parallel rendering, Pthreads, System V IPC, and PVM. System V IPC clearly produced the fastest evaluation times, although primarily due to the ability to use Mesa 3.1 (Pthreads can only be used with Mesa 2.6) and minimal I/O time (PVM inserts latency and local network delay).

Pthreads showed promise, exhibiting slightly lower overhead than System V IPC. The Pthreads API is simple to use, due to the thread managing shared memory rather than the programmer. Pthreads should be used as the parallel rendering architecture if future versions of Mesa become Pthread safe. Currently it is unclear as to the future of Mesa with respect to Pthreads.

The most complex API of the three evaluated is PVM. Several forms of data encoding and packing exist for transmitting data between processes. If a high performance network of multi-processor workstations is available, PVM could be combined with System V IPC to possibly outperform a single multi-processor server.

5.2.1 Future Work

The performance of PVM and System V IPC should be evaluated by connecting several dual processor workstations in the lab to the quad server with high speed Ethernet. Each multi-processor workstation will spawn two processes, or four in the case of the quad server, to perform parallel rendering. A single set of frame and depth buffers will be transmitted from each dual processor workstation to the quad server

for a final set of occlusion tests.

Incorporating System V IPC into Skandha4 should be done while waiting for access to a high speed cluster. Once System V IPC is incorporated into Skandha4, other parallel rendering mechanisms could be added easily. The *Mosix* kernel patch might be an alternative to PVM, allowing a master server move processes to different cluster machines based on the work-load.

BIBLIOGRAPHY

[1] Betz, D. Xlisp: An Object-Oriented Lisp. Unpublished reference manual, 1989.

[2] Brinkley, J.F. and Prothero, J.S. Slisp: A Flexible Software Toolkit for Hybrid, Embedded and Distributed Applications. *IEEE Software – Practice and Experience*, 27(1):33-8, 1997.

[3] Brinkley, J.F., Wong, B.A., Hinshaw, K.P., and Rosse, C. Design of an Anatomy Information System. *IEEE Computer Graphics and Applications*, 19(3):38-48, 1999.

[4] Garcia, F. and Fernandez, J. POSIX Thread Libraries. *Linux Journal* 70 http://www.linuxjournal.com, Feb, 2000.

[5] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine A Users's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Scientific and Engineering Computation, Cambridge, Massachessets, 1994.

[6] Hinshaw, K. *Seeing Structure: Using Knowledge to Reconstruct and Illustrate Anatomy.* Ph.D. Dissertation, Department of Computer Science and Engineering, University of Washington, 2000.

[7] Levine, J., Mason, T., and Brown, D. *Lex & Yacc.* O'Reilly & Associates, Inc, Sebastopol, California, 1995.

[8] McLendon, P. *Graphics Library Programming Guide.* 7.1-7.12, Volume I Silicon Graphics, Inc, Mountain View, California, 1992.

[9] *The Mesa 3d Graphics Library.* http://mesa3d.sourceforge.net/Systems, March 23, 2000.

[10] Modayur, B.R., Prothero, J.S., Ojemann, G.A., Maravilla, K. , and Brinkley, J. Visualization-Based Mapping of Language Function in the Brain. *Neuroimage* 6:245-258, 1997.

[11] Molnar, S., Cox, M., Ellsworth, D., and Fuchs, H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics And Applications*, 23-30, July, 1994

[12] Nichols, B., Buttlar, D., and Farrell, J.P. *Pthreads Programming.* 1-58,194-200, O'Reilly & Associates, Inc, Sebastopol, California, 1998.

[13] *OpenGL Overview.* http://www.opengl.org/About/About.html, March 23, 2000.

[14] Schwartz, P., Bricker, L., Campbell, B., Furness, T., Inkpen, K., Matheson, L., Nakamura, N., Shen, L., Tanney, S., and Yes, S.. Virtual Playground Architectures for a Shared Virtual World. *Proceedings of the ACM Symposium on Virtual Reality Software and Technology: Association for Computing Machinery*, 43-50, 1998.

[15] Stevens, Richard W. *Unix Network Programming Interprocess Communications.* 343-9, Volume II second edition Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1999.

[16] Wall, K. *Linux Programming By Example.* pg. 62, Que Corporation, Indianapolis, Indiana, 2000.

Appendix A

UW DIGITAL ANATOMIST (UWDA) DYNAMIC SCENE GENERATOR TUTORIAL

The UWDA Dynamic Scene Generator is a suite of tools designed for scene generation of 3-D models of anatomical structures. It consists of the Scene Generator, the Scene Manager and the Scene Explorer.

A.1 SCENE GENERATOR

The dynamic SCENE GENERATOR is a program that calls up a 3-D model of an anatomical structure from the 3-D model database, displays that model on the screen and allows for its manipulation. It is designed for authors or teachers to create or develop anatomy exercises for the students. The objective is to create a final scene that consists of one or more structures. From this initial scene, different exercises can then be developed.

A. The SCENE GENERATOR consists of 3 frames, the Scene Viewer panel on the right, the Structure Selector on the left and the Scene Name frame on top. The default screen is blank and the Scene Name is Nothing. There are two ways to add a 3-D model to the view port.

1. Node to Display.
   - Type in the term name of the structure in the *Node to display* box in the Scene Viewer panel.
   - After typing in the term name, go to *Operation* tab and select *New*.

- Click on the *Perform Operation* button and the image of selected structure is displayed on the black screen.

2. Structure Selector.

   - Click on a parent node in the hierarchy and search up and down the tree to find the target term. If the image model for the structure is available, there is a black flag at the end of the term. Click on this flag.
   - On the top panel, click on *Add structure* to display the image on the black screen.

B. Once the structure is loaded, it can be viewed in a number of ways.

- The *Zoom in/out* button controls image size while *Zoom Resolution* arrows control defined increments of view distance.
- The *Rotate* button allows for rotation of the scene in different directions in degree increments defined by the *Rotate Resolution* control.
- The pull down *Rotate* menu displays a selection of rotation directions or orientation: Clockwise, Counterclockwise, Right, Left, Forward and Back.

C. After desired image size and scene orientation has been selected, type in the description of the scene in the *Scene Description* box and click on *Save Description.*

D. There are two ways to save the image, as a Scene or as an AddOn.

1. A Scene is used to generate the starting image for the exercise and can consist of a single structure or a group of structures. A scene also contains specific information, such as the spatial orientation of the structures and the colors of each structure. To create a Scene, add a structure, following

step A above, and then type in the name of that scene and click on *Save Scene*.

2. An AddOn is a list of structures that make up a Scene, and does not contain any spatial orientation or structure color information. AddOns are used to apply a specific operation to a group of structures, (e.g. an AddOn consisting of the respiratory system could be used to highlight all of the structures within the current scene that belong to the respiratory system). To save a group of structures as an AddOn, follow step A above to add a structure, type in name and click on *Save AddOn*.

3. Additional structures can be incorporated in the current Scene either by:

   a. adding a new structure, following step A. Click on *peration* tab and select or highlight *Add*. Then click on the *Perform Operation* button and the image of the selected structure is added onto the current Scene.

   b. loading an already existing AddOn. Select and highlight the AddOn name in the *AddOns* menu box in the Main Display module and click on *Load AddOn*. That automatically loads the selected AddOn onto the current Scene. Repeat the process if you want to add more structures to your Scene. Remember to save the Scene before moving on to the next task.

E. You can delete Scenes or AddOns by clicking on their respective *Delete* buttons. You can also unload an AddOn from the current scene by clicking on *Unload AddOn* button, resulting in all of the structures that belong to the AddOn being removed.

F. After saving scenes and AddOns, click on SCENE MANAGER to proceed to the next step.

A.2 SCENE MANAGER

The SCENE MANAGER organizes saved Scenes and AddOns according to the purpose or demand of the exercise. An exercise group consists of a Scene and a group of AddOns.

A. Select a scene from the *To Select Initial Scene* menu and click on *Add* button under *Add On Options*. An image of that selected scene appears on the left-hand side of the page. Then click on *Save Init* to save that image as the starting or initial scene of the exercise.

B. From the *To Select Add Ons* menu, select the AddOns that should be associated with the initial Scene. Highlight each, one by one, and click *Add* after each selection. The selected AddOns are displayed under *Add On Display Panel*. Then click on *Save All* to save the AddOns for the exercise.

C. Give the exercise associated with the Scene and the AddOn images a name by typing in the group name in the *To Create a New Group, Type a Name and Select Add Button* and then click on *Add Group*. Highlight the given name from the scene group list and click on *Descend* under *Group Options* to select the exercise and view the AddOns of the scene for that group.

D. Click on *View 'Group Name'* to link to Scene Explorer.

A.3 SCENE EXPLORER

SCENE EXPLORER is the interface to manipulate the Scene. The program allows the user to dissect, add or highlight parts of the Scene, as required by the particular exercise.

A. The SCENE EXPLORER, like the SCENE GENERATOR, has the Main Display, Structure Selector and Scene Name frames.

B. The selected initial Scene is displayed on the black view port. Lined up along both sides of the screen are thumbnail images of selected AddOns. On the bottom panels are controls to adjust the size and to rotate the Scene in different directions.

C. There are three functions operable on the Scene: HIGHLIGHT, ADD and REMOVE, and there are two ways to implement those functions.

1. Scene Viewer Controls. Above the Scene Viewer panel are controls to highlight, add or remove a part of the Scene. For example, to remove or dissect a structure from the Scene, click on *Remove* (term becomes unhighlighted) and then move the cursor to the AddOn thumbnail image of the structure to be removed and click. That part is removed from the Scene. Conversely, to add a structure to the Scene, click on *Add* and then on the thumbnail image of that structure which is then automatically added to the Scene. To highlight a part, click on *Highlight* and then on the thumbnail image. The target part retains it color while the rest of the Scene becomes unhighlighted (ghosting).

2. Scene Name Controls. When you click on a part of a Scene, the term name of that structure is displayed in the top frame. There are three functions executable on that selected structure in relation to the Scene. You can highlight it by clicking on *Highlight*, or remove it from the scene by clicking on *Hide Structure*. Clicking on *Unhighlight* or *Unhide Structure* reverses the process, respectively. To zoom up close to the structure, click on *Look At*, and click on *Unlook At* to return to previous view. In this method the term names of all the structures in the Scene are listed in the

Structure Selector, proceeded by a check box. Boxes are X-marked if the corresponding structures are in the scene, and open-ended if available to be added to the scene.

Appendix B

DYNAMIC SCENE GENERATION TOOLS FEEDBACK

B.1 Scene Generator

1. Search or find term box for FM hierarchy in the selector module.

2. Ability to retrieve image by clicking on term itself in the hierarchy, whether its a single structure or a set or group of structures.

3. Reset button to switch back to default.

4. Rotation resolution degree counter where one can directly select the degree of rotation without having to go through all increments to get to desired position.

5. Have image centered on the screen at any zoom position.

6. Clicking on *Scene Manager* brings up Scene Manager page.

B.2 Scene Manager

1. Create shorter names for the window boxes.

2. Scene box should be the first box on the left, then followed by AddOn box.

3. Reset button to switch back to default.

4. Double clicking on selection executes the addition function.

5. Multiple selection ability to select more than one AddOn using either *Shift* and *arrow* keys or *Control* and *arrow* keys.

6. A separate and more conspicuous button for *view scene*.

7. *Return to scene generator* button.

B.3 Scene Explorer

1. Add, remove, or highlight function is executed when structure is clicked on.

2. Back and forward buttons, either in text or in arrow forms.

3. Reset button to switch back to default.

4. Image icon should change in color or become highlighted when selected.

5. Top module should display the last structure selected.

6. Label appears when structure is clicked on.

7. Unhighlight button to cancel out AddOn highlighting.