3D-CONSOLE, AN APPLICATION INTERFACE BASED ON X3D

Brian Kristiansen

Brian.Kristiansen@oracle.com

Kristian Spangsege

Kristian.Spangsege@framfab.dk

EP-THESIS-2003-38

IMM

Printed by IMM, DTU

Abstract

The aim of this report is to describe the design and implementation of a system named "3D-Console". The ultimate goal for 3D-Console was to provide a 3-D user interface framework, which enabled users to work more efficiently. The user interface has been integrated into a proper 3-D environment. This report also discusses the rationale for such a system, as well as how to optimize interaction through hardware devices and graphical effects.

3D-Console also gives interested developers and designers a 3-D environment, an X3D compatible framework and an client API, which can be used for authoring applications with 3-D user interfaces.

The new idea is the combination of two technologies: independent applications and a 3-D environment.

3D-Console builds heavily on the new 3-D graphics standard, X3D.

Keywords

3-D, user interface, X3D, SAI, API.

Contents

Preface **xix**

1 Introduction **1**

2 Motivation **5**

2.1 How do we push the development of 3-D user interfaces onward 6

2.2 Ideal work situation . . . 7

2.3 A hypothetical use case - a File browser . . . 8

2.4 Back to reality . . . 9

2.5 Open Source . . . 10

3 Requirements for 3D-Console **11**

3.1 Quality parameters for 3-D environments . . . 11

3.2 3-D environment as a workspace . . . 12

3.3 Use of graphical effects for enhancing immersiveness . . . 13

3.4 Why is our “Ideal work situation” as it is? . . . 14

3.5 3-D environments as Operating Systems . . . 14

3.6 Entering text in a 3-D environment . . . 14

3.7 Applications in a shared 3-D environment . . . 15

3.7.1 Launching applications . . . 16

3.7.2 Sizing application models . . . 16

3.7.3 Navigation and scene setup . . . 16

3.7.4 Destructive behavior . . . 18

3.7.5 Bounding volumes . . . 19

3.7.6 Collision detection . . . 20

3.7.7 User permissions/rights . . . 21

3.8 System requirements . . . 21

3.8.1 Requirements imposed by client applications . . . 22

3.8.2 Requirements imposed by users . . . 24

3.9 Existing 3-D environments . . . 25

3.9.1 3Dwm . . . 26

3.9.2 DIVE . . . 27

3.9.3 How is 3D-Console different? . . . 27

4 Design background **29**

4.1 Scene representation . . . 29

4.1.1 The scene graph . . . 30

4.2 X3D . . . 33

4.2.1 What is X3D . . . 33

4.2.2 X3D is not a file format . . . 34

4.2.3 The specification - a quick tour . . . 36

4.2.4 The choice of X3D . . . 40

4.2.5 What is missing . . . 42

4.3 CORBA . . . 43

5 System design **45**

5.1 Client/server 45
5.2 X3D compliance 47
5.3 Type system 47
5.4 Multi-threading 48
5.5 Application resource accounting 49
5.6 Memory management 50
5.7 Application sessions 50
5.8 Application scenes 50
5.9 Client API 52
5.10 Additions to SAI 54
5.11 Abstraction layer 56

6 Implementation **59**

6.1 Use of 3rd party libraries 60
6.2 Overall structure 61
6.3 On-line project resources 62
6.4 Libraries 63
6.5 Node and field types 63
6.6 X3D components 69
6.7 Run-time type information 71
6.8 Server modules 76
6.8.1 The scene graph 76
6.8.2 Asynchronous content loading 77
6.8.3 Parsing X3D files 78
6.8.4 The renderer 78
6.8.5 Font server 79

6.8.6 Scripting 80
6.9 Server internals 80
6.10 The smart pointer 82
6.11 Multi threading and re-entrance 83
6.12 The client API 87
6.12.1 Value conversion 88
6.12.2 Node references 89
6.13 Class to file mapping 90

7 Client Applications **93**

7.1 Client Application examples 93
7.2 Dissecting a client application 95

8 Results **101**

9 Future Enhancements **103**

9.1 Linking applications 103
9.2 Application boundaries 105
9.3 Independent viewer 105
9.4 Collision detection 106
9.5 X Support 107
9.6 Connecting several 3-D environments together 108
9.7 Fine-grained scene graph locking 108
9.8 Concurrent Event Cascades 109
9.9 Non-conventional text 110
9.10 Files / Tools 112

10 Conclusion **113**

Bibliography **116**

Index **118**

A Project description **129**

B ToDo **131**

C Dependencies **137**

D X3D mailing list discussions **141**

E X3D specification obscurities **155**

F User Guide **161**

G Screen-shots **165**

H CORBA IDL for client API **177**

I Source code **183**

List of Figures

4.1 Example of a scene graph - button panel 31

4.2 X3D Architecture 35

5.1 Overview of 3D-Console 46

5.2 Application sessions 51

5.3 Application scenes 51

5.4 Established connection 54

6.1 Field class inheritance graph 72

6.2 Overview of system modules 76

6.3 Server objects 81

7.1 Screen-shot of the `archon_clock` application 95

List of Tables

6.1 Name spaces 61
6.2 Directory structure 62
6.3 Libraries 63
6.4 Field types 68
6.5 X3D components 69
6.6 List of supported X3D node types 70
6.7 Permitted field access capabilities during the node life cycle. 86
6.8 Class to file mapping. 91

Listings

4.1 Formal X3D node type specification 37
6.1 Formal definition of `Transform` 64
6.2 Implementation of `Transform` 64
6.3 Formal definition of `GroupingNode` 66
6.4 Implementation of `GroupingNode` 66
6.5 Run-time type initialization of `Transform` 73
6.6 Implementation of `NodeType` 74
6.7 Declaration of the `Node` class 84
6.8 Implementation of the `Node::add` field service 85
6.9 Implementation of value import/export in SAI 88
7.1 Setting up client application session and initial Transform . 95
7.2 Setting up clock dial plate 96
7.3 Setting up clock casing 96
7.4 Setting up client application close button 97
7.5 Defining generic clock pointer 98
7.6 Setting up clock hour/minute/second pointers 98
7.7 Launching the client application 99
7.8 Client application main loop 99
7.9 Functions handling external events 99

Preface

This M.Sc. project was carried out during a period from September 2002 through to July 2003 under the supervision of Associate Professor Niels-Jørgen Christensen at The Section for Image Analysis and Computer Graphics, under the department of Informatics and Mathematical Modelling (IMM), at the Technical University of Denmark (DTU).

For several years we have been discussing how to transform the interface used for the everyday work on the computer into a virtual reality environment. We have a firm belief that this could increase efficiency in the work process.

It is worth emphasizing that we envision the 3-D interface of "tomorrow" as a complete replacement of the conventional 2-D interface. The rationale behind this idea is that there is nothing to loose and a lot to win. That is, with the human/computer interfacing hardware of "tomorrow", all the types of work that are inherently 2-dimensional, such as writing in a word processor, can be performed just as well through a 3-dimensional interface.

Through the course, Computer Graphics, and later, Virtual Reality Systems, we were introduced to X3D, a 3-D graphics specification. We liked the concept in X3D, and found it fit as a foundation for a general purpose application interface formed as a 3-D environment.

Thus, the choice of X3D as a foundation for this project was made even before the project was started, and was also a consequence of a desire to base our system on an open standard. Even though the X3D standard is new, it may still be assumed to be widely known in that it bears close resemblance to VRML.

Lyngby, July 31, 2003

Brian Kristiansen
Kristian Spangsege

Chapter 1

Introduction

This report describes the design and implementation of a software system named “3D-Console”. 3D-Console is a system that maintains and renders a general purpose user interface for multiple applications in a shared three dimensional virtual space. The applications are arbitrary operating system processes, as we know them, and therefore, with 3D-Console, we can replace our two dimensional desktop with a 3-D environment.

The new idea presented in 3D-Console is to combine two well-known technologies: User interface frameworks for independent applications and 3-D environments. The design and implementation of 3D-Console builds heavily on the new Extensible 3-D Graphics specification, X3D.

The reader must understand that we have chosen to base our solution on X3D, and that this choice is made without a theoretical justification, that is, without a thorough consideration of alternatives. The main objective in this project is to design and implement a system that joins the benefits of the two technologies mentioned above, while being compliant with the X3D specification.

Since there is no suitable implementations of the specification from which we can derive our system, we will build it from the bottom and up. In our design we will apply a number of auxiliary constraints. The most important ones are robustness, scalability and performance. An actual performance optimization is not an issue in this report though.

With the system, we will include a number of sample applications. These applications serves as a demonstration of various aspects of the system.

We can identify no less than three distinct audiences for our system. Formally our target users must be the end users of applications written specifically for 3D-Console, but since 3D-Console is not yet completed to a level that would please any application end user, we will increase the emphasis on the two other groups. One of them consists of those who want to write X3D content and client applications for out system, and the other group is those who want to extend our server to make it more complete as well as to add new features.

To get the most out of this report, the reader must have knowledge about common concepts in the field of 3-D computer graphics and virtual environments. Furthermore the reader must have at least a superficial knowledge of X3D. To follow the text in the design, and in particular the implementation chapter, the reader must further have some knowledge in the fields of object oriented programming, parallel programming, distributed systems, operating systems and compiler techniques. Knowledge of operating systems is required to understand our concerns about system resources, application independence, application privacy, libraries, etc.. Knowledge of compiler techniques is required to understand the XML and VRML parsers.

Without any experience with C++ or a similar programming languages it will be hard to understand the implementation chapter, and probably impossible to gain anything from the source code appendix.

3D-Console was motivated by a belief that an immersive[1] 3-D user interface has a potential of allowing a user to perform certain kinds of work more efficiently than what is possible in the well known 2-D correspondent. The success would depend not only on 3D-Console, but also on the quality of the available human/computer interfacing hardware, and the applications themselves of course. An important point in the motivation for this project is that we need something better than the conventional monitor/mouse/keybord setup, that most people have today, to be successful.

It is important that the reader understands, that although we assume the existence of non-conventional interfacing hardware to justify our idea, this report does not present an implementation which utilizes such hardware. The system design takes care not to prevent or complicate the addition of

[1] For an explanation of “immersiveness” see section 3.1.

support for such hardware, but the implementation, as it is right now, only supports the conventional setup.

A thorough examination of the advantages and problems of interaction in a 3-D environment is not included in this report. We refer to other written works on this matter, e.g. the thesis of Karsten Bormann [Bor01]. Although his thesis discusses the matter from an "architectural design" point of view, we find that the arguments for and against are applicable to our environment too.

It has been clear from the start of this project that full immersiveness (through the use of non-conventional interfacing hardware) would not be something we could achieve within the 1 year scope. Instead we have focused on how to design the software system that renders and maintains the virtual user interface - the system we call "3D-Console". We expect the development to continue after the end of the project, therefore we will strive for a well structured, well designed, solution.

Two essential and somewhat similar phrases will need to be explained since they are used frequently in this report; *3-D user interfaces* and *3-D environments*. The two phrases are largely synonymous. We use both to refer to the system that we have made, but we will use the phrase *3-D user interface* when we want to emphasize the aspects which relate to interaction, and when we want to emphasize the aspects of presence and immersiveness we use the phrase *3-D environment*

To further clarify our distinction between the two phrases, it must be understood that a 3-D user interfaces need not be combined with a 3-D environment. There are many examples of so-called 3-D user interfaces which are really not much more than a 2-D user interface with some 3-D animations added on top. It is therefore time to emphasize that 3D-Console is as much a 3-D environment as it is a 3-D user interface.

Even shares of time has been spent on uncovering the requirements to the system, on designing the system, and on implementing it. Chapter 2 elaborates on some of the issues which motivated this project, as well as on our objectives, and finally on the limitation of the scope of this report.

In chapter 3 we discuss issues on the subject of 3-D environments and we attempt to uncover the requirements to our system with emphasis on the 3-D interface of "tomorrow". In chapter 4 we go through three relevant topics: scene representation, X3D and CORBA.

In chapter 5 we cover our design process. This includes the design of the X3D system, but more importantly it describes how we extend the X3D system while keeping compliance. Most of the extensions relate to the client API.

In chapter 6 we cover the implementation. This chapter takes the reader in and out of various areas of the implementation that we find interesting. The areas we deem most important are explained in detail and are backed up by code listings. The remaining areas of the implementation are treated more briefly.

We will give some examples of client applications in chapter 7. Chapter 8 summarizes on the results the we have achieved. Chapter 9 is a discussion on how to enhance the system in the future, and finally we summarize it all in the conclusion.

Chapter 2

Motivation

The original idea for this project was to study the possibilities and potential benefits of using three dimensional user interfaces for performing normal work on the computer. By "normal work" we mean, the 8 hours a day work - every weekday - done on computer screens today. Anything from basic tasks on the operating system level such as moving files around, to such tasks as maintaining a database and writing documents. Therefore we have our focus on functionality not 'pretty features'.

We wish to step from 2-D to 3-D because we believe, that a well designed and properly accessible 3-D environment allows one to work much more efficiently. It is in fact our long-term plan to show this, however it is not an objective for the present report. For a brief discussion on why to use a 3-D environment to work in - please see section 3.2. For 3-D environments to be useful, it has to be realistic. Not in the sense of realism/looking like the real world, but items and surroundings have to look like they are really there, and behave as if they are there. The environment has to be immersive. Because of this requirement we are forced to consider the use of non-conventional interaction hardware. Hardware which is not readily available today. For a further explanation see section 2.2.

We would like to create a foundation for making 3-D user interfaces. This foundation needs to be much like the Operating System. Several 3-D applications should be able to run in parallel.

We want to allow applications to set up their user interfaces in a shared

three dimensional space in much the same way as applications today are able to set up their user interface in the well known 2 dimensional graphical desktop such as the X Window System. If you do not know the X Window System, think of Microsoft Windows instead.

We wish to creating an open API. By creating an open API, we will give other creative developers a possibility to enhance and play around with 3-D environments. It is our hope, to attract application developers/designers into creating worlds without restricting them too much. Using an open API they can focus on creating applications/worlds in 3-D - and not have to develop a complete 3-D graphics engine themselves.

We would like to push the development of 3-D user interfaces even further.

2.1 How do we push the development of 3-D user interfaces onward

By making it more simple to develop applications which can display themselves in a 3-D environment, we hope to inspire other creative people to experiment with 3-D user interfaces. If it were easier for people to develop 3-D user interfaces, maybe more 3-D applications would be made, and 3-D more accepted as a alternative for the computer desktop of today.

It may not be all applications which are suitable for 3-D - but without doing experiments it is very hard to predict which applications are suitable or not. Maybe imagining another way of presenting an application could turn an application from being tedious and difficult to use in 3-D to be even smarter than the same application under 2-D.

One of the main driving forces which could kick-start development of 3-D user interfaces is the technological advancements of hardware interfaces i.e. image-on-eye projection, lightweight head mounted displays etc. [Ben03].

It is our opinion that a certain advancement in technology is needed before the switch to 3-D user interfaces will make sense. We call this technological setup the "Ideal work situation".

2.2 Ideal work situation

We imagine the user sitting in a chair at a desk in the work place. In front of the user there is no conventional monitor, but the user wears a lightweight head mounted display. The user puts on a pair of tracked gloves and sits in front of a tracked keyboard. The most important point to notice is, that everything is easy to wear and not uncomfortable to use 8 hours a day - everyday. It would also be nice if the equipment was easy to take on and off. The user must not feel trapped inside the equipment.

In the 3-D environment, the user is presented with a virtual keyboard represented at the position of the real keyboard. The user should be able to use the physical keyboard by believing that the virtual keyboard is used. Since entering text is still a major part of the work done on computers today, we suggest having a keyboard as a means to enter text into the 3-D environment. We still think a physical keyboard is of use, but we do not make any suggestions on how the design or even keyboard layout is done. We think that human sensory data e.g. when a key is pressed and we feel the impact, is very important for the efficiency. Human beings rely on the haptic feedback, when they type. e.g. Some persons type on a keyboard without even looking.

As the user is wearing tracked gloves, the user can see the virtual hands, if they are put in front of the eyes. The user can use the tracked gloves to manipulate objects by touching, grabbing, dropping etc.

The lightweight head mounted display presents the user with a view of the 3-D environment, which takes up the total field of view. Dependent on the technology, this view is tracked. By that the users head movement will change the field of view. It is difficult, if not impossible, to scry at the future, of course, but other ways of projecting an image to the user will be available. e.g. see MIT Media lab's Spatial Imaging Research group [Ben03].

In our initial approach to this ideal work situation, we did not imagine that the 3-D environment was using augmented reality, since we like to have as much flexibility to designing the user interface as possible. We feel that augmented reality could impose too many restrictions to the design, as augmented reality will merge the real physical boundaries, such as the walls, ceiling and furniture, with the virtual boundaries i.e. objects and walls rendered in the 3-D environment. On the other hand, isolation from

the real world, if the environment is not totally immersive, could lead to discomfort for the user. e.g. The user needs to have congruence between the virtual horizon and its physical counterpart, so no lag or other imprecision is allowed [Bri90].

The "Ideal work situation" is a prerequisite for this project, and when we talk about interaction with the environment in this report, we will be referring to this setup.

We acknowledge that our "Ideal work situation" may be far from ideal for physically challenged people, but in this report we will not discuss this matter further.

2.3 A hypothetical use case - a File browser

To illustrate the use of a 3-D user interface in an application, we imagine a hypothetical use case - a File browser.

The file browser as a functional utility, is not in itself a typical 3-D oriented application. Textual list of files and directories are normally related to 2-D applications. See section 9.9.

For the file browser to be efficient as a tool in a 3-D environment, it needs to utilize the potential abilities that makes human beings efficient.

Two handed interaction
As human beings are accustomed to use two hands for manipulating the physical world, it will be natural if the user's two hands are used to grab and hold items, i.e. files or folders, while copying or moving the files in the file browser. The use of two hands, instead of one, is more efficient as well as more natural. Imagine the left hand grabbing a file and holding it, while the right hands searches through other directories. When the correct directory is found, the user's left hands drop the file in the directory.

Another benefit of using both hands and arms, and being able to see them, is that "Watching your dynamic self-representation within the virtual world is convincing evidence that you're There" [Bri90].

Full field of view
Since we believe that using the full field of view is beneficial for the efficiency, we have chosen to include a lightweight head mounted display, or

something similar, capable of giving a full field of view display, in the description of the "ideal work situation". Through the human evolution we have learned to register visual input, even peripheral sight is acknowledged. Using our head to look around, and movement in cooperation with vision enhances our sense of depth and enhances the understanding of complex three dimensional objects.

To help the users feeling of orientation we suggest using a semi-transparent screen which surrounds the user. This screen is built of stacked belts of text columns. Each belt of text columns describes the directories and files in a directory. We believe that this semi-transparent screen gives the user a possibility to use the innate ability to comprehend the data more efficiently, by looking around. The semi-transparency gives the user a comfortable feeling of position as the horizon can be seen through the screen.

2.4 Back to reality

By now we hope that the reader understands our motivation for engaging in this project, but also that our motivation to some extent is based on a goal that cannot be reached within the scope of our project. The goal is clear to us, and hopefully to the reader. We want to be able to interact with the operating system and the applications in it, in the way explained in section 2.3. It is equally clear to us that a great distance will remain between the level of implementation and the "hypothetical use case" at the end of the project.

We do not even hope to be able to create a system that fully implements the "hypothetical use case". Not ever. That is why we call it, hypothetical. We believe that it is only achievable through lots and lots of experiments and evaluation by many different people. The point is, that to get a lot of people to do that, we need something that we do not have today. We need a software system that reduces the burden of creating 3-D applications. A system that lifts the level of abstraction to a higher ground. We need "3D-Console".

2.5 Open Source

We have chosen to donate our application to the Open Source community. This is partly an ideological decision, as well as a practical one. By making this application available to the public, we hope other developers will help us in the huge task it is to experiment and find the suitable 3-D user interface of "tomorrow".

Also, since we use open source libraries from other developers extensively, it is only fair to respond with a contribution to the open source community ourselves, e.g. SourceForge[1] have supplied us with disk space, website and CVS versioning system - for that we are grateful.

[1] SourceForge.net is the world's largest Open Source software development website, with a repository of approximately 66.000 available Open Source projects. SourceForge.net provides free services to Open Source developers.

Chapter 3

Requirements for 3D-Console

3.1 Quality parameters for 3-D environments

Human beings are used to rely on its senses - which is kind of obvious, since our well being, and maybe even our existence, is influenced by our ability to react on external stimuli e.g. if we touch something hot with a finger, then we remove our finger as a reaction. If we cheat the senses to believe something which is not real, we have created an illusion. The Illusion is the cornerstone of Virtual Reality.

A major part of the illusion is cheating the vision. For human beings, the visual illusion is quite strong if the screen goes to the edges of the the persons maximum peripheral vision or if the visual information comes only from the screen and everything else is masked out, as in some head mounted displays. This is why head mounted displays are popular in VR systems.

The interaction in 3-D environments is quite important as well. If the user of the environments are able to do things, which should clearly not be possible, e.g. breaks the laws of physics, such as put hands through walls etc, then this behavior breaks the illusion. This does not mean, that it has to be like the real world. i.e. In a cartoon world, cartoon physics will apply.

All senses needs to have feedback, if the illusion have to be complete. Some examples are:

- haptic feedback (sense of touch)
- olfactory nerves feedback (smelling)
- temperature feedback
- acoustic feedback

In the world of VR design, we talk a lot about immersiveness. Immersiveness is not a real word. If you look it up in a dictionary, it does not exist. But in the world of VR and computer game design, immersiveness is very important. Immersiveness is a measure of how good a VR environment, 3-D environment or computer game is to absorb and engross the user. So it is a kind of measure of how good the illusion is.

3.2 3-D environment as a workspace

The obvious question is now; why use a 3-D environment to work in?
We believe, that if we utilize the capabilities of the human being to its fullest it will only increase the efficiency, and we do think using 3-D environments as workspaces will utilize more of the capabilities.

Some of the human capabilities that we do not utilize in the 2-D computer workspace (desktop PC's), but we could use in 3-D environments are:

- An extra (3^{rd}) dimension. Human beings are built to relate ourselves to the physical world. e.g. placement of objects in space.
- More interaction resources are possible. Since we do not need to manipulate the world through a "window", but are able to manipulate the objects themselves in the 3-D environments, we get a more direct approach to solving the problems/doing the work.
- Perception of moving forms. Human beings are capable of understanding complex 3-D forms better, if we move the forms around. We get even more "feeling" of the object, if we are allowed to touch the item and move it around.

3.3 Use of graphical effects for enhancing immersiveness

Human beings use some graphical effects, specifically the information that the graphical effects give, to establish a perception on how to interpret the world. Some of these graphical effects will be described in this section. Those graphical effects chosen in this section, is not meant to be a complete list, neither are they ordered in any particular way.

Shadows

Shadows gives the user information on depth and position. It can be very difficult to see if an object is sitting on a table or hovering above the table, if its shadow is not seen. Human beings are used to rely on shadows to determine positions and depth of objects. If it is difficult to determine the correct position, then it is equally difficult to determine the size of the object - is the object near or far from you?

Transparency

Transparency enables the user to see more reference points, so position and orientation, with a horizon or floor, is easily determined. This could be in the form of windows in a wall. Removing the wall completely might give too much visual noise.

The benefit of using semi-transparency is illustrated by our hypothetical use case, the file browser. Fixating the text onto a semi-transparent screen as opposed to let the text float in "mid air" helps the user to understand the spatial organization of the text e.g. it provides a depth cue.

Horizon

By drawing a horizon or a floor, users can reassure themselves of their orientation, thereby they again can find their position more easily. The floor will also help users understand depth in the landscape, as well as the individual sizes of objects placed on the floor.

3.4 Why is our "Ideal work situation" as it is?

We believe a certain level of immersiveness is needed if our 3-D environment should be effective to use. The effectiveness is related to how immersive the 3-D environment is, because we trust our senses so much, that if the world reacts accordingly to what our senses tells us, then we do not need to spend so much time, thinking on how to do things - it becomes an autonomous reaction. If the users have to fight to get the job done, just because they use our 3-D environment, then we have failed.

We see the ideal work situation as a description of the minimum level of technology for our 3-D environment to make sense and to be efficient. The definition of the ideal work situation is not meant to suggest, that this exact technology is used.

This "ideal work situation" is almost realistic today. We still need to see a truly lightweight head mounted display, as light as a pair of glasses or even lighter. Other ways of projecting an image onto the eye could be the solution of the future, thereby eliminating a head mounted display.

3.5 3-D environments as Operating Systems

It is possible to look at 3-D environments as Operating Systems. Not to be confused with operating systems as 3-D environments! Client application will start a session in the 3-D environment server. Just like an operating system, the server will administer all the nodes in the scene graph that the application creates. The server should be able to remove all the nodes, and underlying resources, when an application is closed down or dead.

Since client applications are independent operating system processes as well, traditional communication between applications (and to the operating system) is still possible i.e. the 3-D environment server should only administer resources specific to the 3-D environment.

3.6 Entering text in a 3-D environment

Since text is a very important and effective way of entering and displaying information in any system, it is of course important in a 3-D environment, which main purpose it is to be an effective daily workspace.

So how can we enter text in 3-D environment without a keyboard? We will give some examples here:

- Speech Recognition. Dictating instead of writing is of course a possibility. The speech recognition engines, need to be very intelligent so the right words are chosen, dependent on the situation. In the word processor natural language is needed, but in the system console technical commands are needed.
- Using unistroke pen movement. Similarly to hand gestures, unistroke pen movement can be used to enter letters or whole words. Unistroke pen movements are a type of gesture where you, in one fluent sweep, makes the equivalent motion with your hand as if you where writing.
- Virtual keyboard. A virtual keyboard could be presented in front of the user. The keys on this virtual keyboard can be pressed, in the way that the user normally uses a keyboard. Our enhancement to this approach is, that the real physical keyboard is tracked and shown as a virtual keyboard. In this way the user still gets the haptic feedback from the real keyboard.

In our implementation text is supported through an X3D Text node, see section 6.8.5. This enables users to experiment with text in 3-D, as the Text node will show you the text in the 3-D environment.

3.7 Applications in a shared 3-D environment

By starting several client applications inside a shared 3-D environment we, in a sense, creates an autonomous environment. Thus by making the environment dynamic, we enhance the immersiveness. Allowing several applications to exist in the same shared 3-D environment, will introduce a wast array of issues/problems. We will discuss some of these issues in this section.

Multiple applications, which shares the 3-D environment, can enable users to collaborate. Collaborating in a shared environment is an interesting topic, which we will not go into.

3.7.1 Launching applications

First of all, how can we handle the launching of applications? The actual launching mechanism that launches an application i.e. the hand gesture, speech command or button to press etc, is a high level design decision. It is one of the design choices, which we hope users of our 3-D environment, will experiment with.

When the application launch has been triggered, several approaches to presenting an application is possible. We will give one example of such approach. Only experiments will show what the best approach will be.

We imagine, that a wire frame box appears, if the application wishes to present itself. The user can interpret this wire frame box as the minimal bounded box for the application. The user, who started this application - the owner of the application, will have the possibility to position, orient and scale this wire frame box, before settling it. Further discussion on application boundaries is found in 9.2.

3.7.2 Sizing application models

Since the client application developers are free to make models in any scale they want, sizing of the application models when launching is important. We would not like to have a watch to be as big as the house next to it in the scene.

The X3D standard defines 1 unit to correspond to 1 meter. So 1 unit = 1 meter should be respected by client applications. The server will respect this, so client applications can be assured, that if they setup a model, as described in the X3D standard, then the server will not rescale it by default. Similarly viewing systems must respect it as well, so viewed in 3-D environments, models which is supposed to be a certain measure, is in fact that, when checked. This is very important for augmented reality viewers - of course.

3.7.3 Navigation and scene setup

It is possible to use many different hardware interfaces to navigate in a 3-D environment. Using one of these interfaces you can navigate, by actual movement of the body i.e. walking in the laboratory will navigate (walk) in 3-D environments. In other types of interfaces, the user will be more constrained in the movement. Navigating in 3-D environments could be initiated by a hand gesture - so no actual physical walking.

In 3-D environments we distinguish between four different types of navigation, which are the most common:

- Walk. Is as the word says used for navigation using walk. The user will follow the terrain and gravity is in effect.
- Fly. Like walk but without the gravitational and terrain constraints.
- LookAt. Using this type of navigation, the user can jump from waypoint to waypoint by selecting objects in the scene. Selecting an object will navigate to it.
- Examine. This type of navigation is used for viewing objects. This is done by spinning or orbiting around the center of the object.

The two basic forms of navigation, LookAt and Examine, is often used in conjunction. When selecting a new waypoint using the LookAt navigation type, this also sets the center of rotation for the Examine navigation type.

It is very important that the navigation in the 3-D environment is not cumbersome and inefficient. It must be easy and fast to navigate around even large scenes. Therefore some additional navigational elements could be of use, such as portals or telepods. These additional elements could be used for quick transportation from one location in the 3-D environment to another. The benefit of the portal is, that it is possible to see the destination before entering.

Care must be taken to design the 3-D environment scene in a sensible and effective way, this includes placements of client application models. If the client application models are placed in reasonably orderly fashion, then another navigational short-cut could be to navigate the user, using high speed, up into the air, from where the user have an overview of all the application models. The user then pick the model to navigate to. This kind of quick navigation is best used if the scene is built in an open landscape. One of the problems in using open landscapes as scenes, is that nearly all the scene is visible most of the time. If the complete scene is visible, then the viewer will have to render it all.

To limit the scenes complexity, it could be a good idea to design the scene as an indoor scene, such as a room. In this way only a limited part of the scene needs rendering by the viewer. The downside is that most users

could loose their orientation, if they are lead through hallways and rooms etc. Adding windows, so users were able to look out at the landscape and recognize landmarks, could help solving this problem. Similarly in open landscapes where landmarks, trees or other recognizable items, can help the user to position and reorient. It is worth noticing that the recognizable objects, do not need to be realistic objects, it could be a grid as well.

3.7.4 Destructive behavior

Client applications are capable of several types of destructive behavior. By destructive behavior by a client application, we mean any type of behavior, which can annoy other applications, the user or the system itself.

Bothering other applications

A client application is capable of annoying other applications by colliding into its geometry, this can destroy the geometry of the annoyed application, by drawing overlapped geometry. A side effect of this is, that the underlying geometry can not be picked by a pointing device. If collision detection is enabled, then it could be very annoying, if collision events flood the system.

Another way to alter the visual interpretation of another application is, if it is possible to create new light sources. Light sources normally have effects on all geometry - not only the scene setup by the application.

Annoying the user/viewer

Concerns for the user/viewer must be taken, when designing applications, not only when the model is designed, but also for the behavior of the application while running. It is very easy to create applications which annoy users. Annoying behavior could be:

- Placing applications in front of the user/viewer, disabling the users possibility of picking other objects.
- Moving applications around in an irritating intrusively manner.

Abusing the 3-D environment server

Since the client applications are given access to use a lot of resources, abuse of these resources could of course be a problem for the 3-D environment server. Some resources which could diminish the performance of the server are:

- Light sources. Adding more light sources will degrade the performance, since calculations on multiple light sources in a scene is time consuming. Besides that, new light sources will illuminate all scenes, and not only on the scene, where the light source is placed.
- Polygon count. Abusive use of polygons i.e. excessive use of polygons in a too detailed model, will degrade the performance as well, since calculation of the light model etc. will use up the GPU's processing capabilities.
- Vertex count. Similar to Polygon count. see above.
- Texture size and count. Allocating very large textures or many small different textures can use up the texture memory.
- Memory usage. Allocating very many nodes can fill up the memory and maybe provoke the operating system to swap. Swapping degrades the overall performance.

So what can we do about all these kinds of destructive behavior? Beside the destructive behaviors mentioned above, it will of course also be possible for applications to hack the server or use exploits. We do not at the moment have the illusion, that systems like this is totally secure.

Stopping some of the abusing of the server can be done by introducing limits on polygon count per application, node count per application and ban light source creation from application. It will of course help a bit if we restrict the graphically possibilities of the client applications, only to render inside their own bounding box.

3.7.5 Bounding volumes

The bounding volume is the smallest simple shape which encloses a geometric shape, this geometric shape can be a complex compound shape. The bounding volumes are typically boxes or spheres. Some geometric shapes are easily bounded by a volume, others are not. A large thin disk is not effectively bounded by a sphere, but a large thin box will be more effective. Bounding volumes can then be grouped into more and more complex objects. Ultimately a whole client application model, could be defined in one bounding volume.

Bounding volumes are used to check for collision between geometric shapes. As mentioned, these bounding volumes are typically boxes or spheres, since collision detection between boxes or spheres are simple and fast.

The method is quite logical, if there is a collision between two compound shapes, then their bounding volumes must collide as well. So if the bounding volumes does not collide, then the shapes does not collide either. If the bounding volumes collide, then the shapes does not necessarily collide, but the actual collision between the shapes, will have to be checked.

3.7.6 Collision detection

Collisions between geometry in a application scene, and even between geometry from two different applications is most likely, if several applications must share the same 3-D environment. Several different collision detection strategies could be applied.

Collision detection is complicated by the fact that the scene is very dynamic. Well known methods such as BSP-trees (Binary Space Partitioning) is not applicable since its efficiency relies on the scene being static.

For fast object to object collision detection, we will probably have to turn to libraries such as MathEngine [Mat], I-Collide [URGoMA95] or SWIFT [Ehm01]. The common denominator between those systems is an assumption that one does not ask it to check for collisions between every object in the scene, unless the scene is very small, but instead choose a limited set of objects. The reason is of course that the complexity of the problem grows tremendously with the number of objects. Say we want to check for collision between 100 objects, then our system would need to check the 4950 possible combinations individually.

Because of this restriction in applicability we still need to find ways to reduce the number of objects to check. One way is to pass the problem onward to the applications, giving them the responsibility of choosing which objects which needs to be checked for collision. Another, and probably better approach would be to use structural information from the scene graph such as the hierarchy of bounding volumes. By doing a smaller number of collision detections between these bounding volumes many object combinations may be filtered out. This is often called a “broad phase” collision detection because it reduces the problem at a coarser level and then passes the remaining combinations on to the narrow phase, which could be handled by a system like SWIFT. The concept of confining applications to bounding volumes is discussed further in section 9.2.

See more about a possible implementation of collision detection in section 9.4.

3.7.7 User permissions/rights

Some of the global resources are limited. These limitations are not always fixed, but exhaustive use of a resource could diminish the performance of the 3D-Console as a whole, therefore it will be necessary to administer some of the resources. Some resources it will even be necessary to deny access to in general e.g. the applications own scene graph sub-tree is private to the applications. For more information about resources see section 3.7.4.

Therefore some system administration will be mandatory. Some of these rights/permissions could be setup as limits e.g. maximum polygon count, others could be granted or denied as a whole. Designers/developers of this feature could maybe get some inspiration from UNIX system administration.

We imagine the permissions in some way should be negotiated at the time the application connects to the server. The application will supply a set of credentials, and the server will then know which permissions that should be granted and which that must be denied.

3.8 System requirements

In this section we will unveil a number of requirements that would generally be imposed on any 3-D user interface by both the client applications and the users of the applications and the system as a whole. It is not to be understood as an exhaustive list of all such requirements. Instead it is here as a means of gaining an idea of what kind of features 3D-Console needs to support. In particular we will use these results to help choose a suitable scene representation, see section 4.1 and 4.2.

3.8.1 Requirements imposed by client applications

The most immediate requirement from the client applications side would probably be the ability to construct models of their own choice. That is, a text viewer application wishes to construct a model of a piece of paper, and a clock application wants to model the housing and the pointers. Ergo, there is a need for the ability to construct models from some set of geometric primitives. The primitives could be just simple polygons, or it could be a set of more high-level building blocks, such as boxes, spheres, and spline surfaces. Obviously it will be relevant to have the ability to assign specific material properties to the surfaces such as color, shininess and even texture. Without this possibility it will be difficult for the viewer to distinguish the various parts of the model from each other.

But the modeling capability must be accompanied with a means to display text as well as image. Although neither text nor image has anything new to offer by themselves in a 3-D environment compared to their role in a 2-D environment, they cannot be ignored in the context of 3D-Console. Text is important since it is a precise and compact way of presenting information. Also, text plays a major role at the operating system level today (and probably will continue to do so for a long time), but see section 9.9 for a discussion of whether there is an alternative to conventional text. Images are important since they assist in delivering information to the user. Images represent an easy way to improve the amount of detail in the model. The same level of detail may be extremely cumbersome or even impossible to provide with the geometric primitives mentioned above.

So far we have touched only upon the ways the application can send information to the user (output). Any user interface must also provide for ways in which the user can send information back to the application (input). First there is the trivial, but important, message saying that the application should close down right now. The system can of course not rely on the application doing that, so the message would only serve as a friendly notification before its interface is forcefully removed by the server.

Now, before we can discuss other types of input, we need to know which kind of hardware input devices are available to the user. We will limit ourselves to the interaction scenario outlined in section 2.2. In this scenario there are two kinds of input devices. One is a pair of tracked gloves and the other is a keyboard with the well known purpose of typing in text. This requires that the system has a way of sending text to the application.

The gloves are assumed to be tracked for both position and orientation. For now we will think of the tracked gloves as being able to deliver three different kinds of input. One is a gesture interpreted by the system. Gestures may be targeted at specific applications, requiring a way for the system to send gesture events to the applications. Another function of the glove is as a pointing device. As a pointing device the glove may be used to press buttons or to indicate movement. For the case of buttons there is a requirement by the application to be able to register certain surfaces to be sensitive to touch and to have a way of being notified from the system about a touch event.

Indication of movement is best done by projecting the movement onto a surface. This will aid in converting the rather imprecise/instable movement of the glove to a constrained path on the surface. Again, applications need a way to register certain surfaces to be sensitive to movement and to be notified when it happens.

One of the main purposes of having multi-application user interfaces is to provide a standardized way to transfer information from one application to the other and to link and combine applications in various ways. In the UNIX shell this is done with pipes expressed as vertical bars. In many windowing systems there are ways to "drag and drop" both files and pieces of text. In a 3-D user interface we should have similar ways of linking applications. It is far from obvious, though, how this is going to look and work. See section 9.1 for a suggestion of how to deal with this.

Since our applications share the same 3-D space, there is of course the possibility of collision or overlap unless there is some part of the system that prevents it. Assuming that the user interface is able to detect collisions between geometric objects, it would be reasonable to require that an application could get a notification from the server indicating that a request to move a geometric objects has failed due to collision. Together with this notification we should expect some information about the impact such as the direction of reflection. The issues of collision detection are discussed further in section 3.7.6.

Due to the fact that we are dealing with a general purpose user interface, we can make no assumptions about quality and good intentions of the applications. Just like applications are protected from each other by the operating system kernel, we must give our applications a guarantee that their model is not trifled with by other applications. But privacy has another edge to it, and that is about preventing the clients from inspecting each others

model. Altogether this means we should require that application has access to inspect or update nothing but their own model. Anything else than that must be mediated by the system.

Applications need to have control over location. That is, the application must be able to position objects relative to each other in the way that it desires. On the other hand, applications should not concern themselves with absolute positions in space. Instead the server must assign a local coordinate system for an application to work in. This has at least two benefits. One is that it frees the application from dealing with absolute coordinates. The other benefit is that it allows the server (and thus, a user) to translate, rotate and scale the applications at any time while they are running. Note that this is in close accordance with the concept of windows in a two-dimensional user interface like X Windows. In fact, it is an example of a more fundamental but widely uses design principal which we could call "virtualization" or "indirection". e.g. "virtual memory".

Now let us summarize all the requirements we have talked about so far:

- Modeling with geometric primitives.
- Display of text.
- Display of image.
- Shut down event.
- Input of text.
- Receiving of gesture events.
- Touch sensitivity on specific surfaces.
- Movement sensitivity on specific surfaces.
- Inter-application communication.
- Colliding object notification.
- Privacy, no interference with model from other applications.
- Locality, application models in local coordinates.

3.8.2 Requirements imposed by users

There are, of course, requirements which has nothing to do with the applications. We could say that the total list of requirements must be those implied by applications together with those implied by the users. Many of the requirements mentioned above relates to both parties, but in this section we will list a few which are exclusively related to the users.

Since 3-D is all about viewing things from various directions, the system must support some kind of navigation. As described in section 2.2 the users head/eyes are tracked, and this, of course, makes up for part of the navigation. But there is a need for more than that. The user must be able to travel to more distant locations without moving him/her physical self too far. Such navigation could e.g. be base on gestures. The topic of navigation is explained in more general terms in section 3.7.3.

Navigation is connected with collision detection. That is, as the user moves along, it is important he/she is stopped by geometrical obstacles. This is a very important requirement for immersiveness (see 3.1). Further more it is reasonable to require that the user is able to move along on a ground-like surface, which in turn implies the presence of gravity.

The user, of course, needs to be able to start new applications, and when he does he should also have some control over where the application is placed in space. The application might appear on a location chosen by the system, but the system should then let the user move the application around. It may also be relevant for the user to re-orient and scale the application, and it will certainly be relevant for him to be able to forcefully terminate it. How these manipulations are carried out is not clear yet, but it probably could be linked to the concept of a bounding volume as described below.

The final consideration is that of boundary. In a 2-D user interface most applications are bounded by the borders of their window. But how should we deal with this in 3-D? One approach, of course, is to not impose any boundary onto applications. A problem with this, however, is that it will be hard for the user to know which geometric elements that belong to which applications. The application geometry might get mixed up in way that confuses the user. Another approach would be to confine an application to a certain bounding volume. Apart from limiting the spatial extent of the application, this volume may also help the user to manipulate the application. See section 9.2 for a more concrete consideration.

3.9 Existing 3-D environments

We have chosen to describe two existing 3-D environments. There are many others such as MASSIVE, CVE, COVEN and MiMaze, but we will limit the discussion to the former two, and since none of the systems mentioned

above are based on X3D, we will not go into a thorough comparison. The mentioned 3-D environments is not meant to be an exhaustive list.

3.9.1 3Dwm

3Dwm (three-dimensional workspace manager) is a research project at Chalmers Medialab. They have more or less the same goal as we do: "... the mission is to build a solid research platform with the necessary primitives to support just about any kind of user interface. In other words, 3Dwm is not the ultimate 3D-UI, but it may very well be instrumental in finding it. At least we hope so." [Med].

In general the 3Dwm consists of a central display server, which handles the rendering of the scene graph and handles I/O interfaces. The central display server is also the server for all the applications.

The application management in 3Dwm is built around a concept they call spaces. A space is the equivalent of a window in a windows based system e.g. Microsoft Windows or X Windows. As an application in a windows based system is bounded by its window, applications in 3Dwm is bounded by a volume, called "space". An important reason for this is, that it allows the server to perform some important optimizations when performing collision detection.

3Dwm uses a distributed scene graph, which in principle allows part of the scene graph to be distributed to the clients but the 3Dwm development team suggests the scene graph should reside on the server alone. Applications maintain their own part of the scene graph on the server and requests redrawing whenever needed. The 3Dwm uses a central display server for rendering the scene graph.

The scene graph has a self-invented structure, and is thus something the application developer will have to gain knowledge of by reading the 3Dwm documentation. The application development API is developed by the 3Dwm developers and is called Nobel API.

One of the many advantages of 3Dwm is its platform abstraction. 3Dwm is capable of running on a Linux desktop PC, as well as in the Chalmers Medialab 3D-CUBE. For more detailed information on 3Dwm please see: [Med].

3.9.2 DIVE

SICS Distributed Interactive Virtual Environment (DIVE) is another application, whose aim is to experiment with 3-D environments. DIVE is a peer-to-peer based application, thus no central server is needed. Applications connect to each other on the network e.g. on the Internet. Users navigate in a shared virtual space where they meet, collaborate or just explore the world.

Users explore the world by using a rendering application, called a visualizer. The visualizer handles the rendering and all I/O interfaces.

In a typical DIVE world, the user can enter and leave the environment at all times. Since all objects are replicated, they are independent and typically outlive their creators. All object are able to live, as long as one client is still running. Objects get their dynamics by interpretative scripts in Dive/Tcl.

For more information on DIVE, please visit their homepage [pd].

3.9.3 How is 3D-Console different?

3Dwm

Even though we share the same vision as 3Dwm, we must also emphasize, that we have a different approach than the 3Dwm project, since we build our design on the existing X3D standard.

We look at the server and renderer/viewer as two different things - which 3Dwm does not. We have not implemented the viewer as a separate entity yet.

3Dwm's client API (Nobel) is a CORBA level API. Therefore CORBA will always be a required part of 3Dwm, and client application developers need to be familiar with CORBA.

In 3D-Console we also use CORBA. The reason for this is explained in section 5.1. We have gone a great distance to design our client API, in such a way that CORBA is hidden. Hiding CORBA from the client applications makes it possible to substitute CORBA with another technology if needed. A further benefit is that the client application developer does not need to have knowledge about CORBA. If the client application developer does not

wish to use our client API, it is still possible to code an client application against CORBA directly.

DIVE

DIVE is a totally different approach to developing a 3-D environment. The developers, at DIVE platform development at SICS, have another motivation than us. They built a multi-user virtual environment, where it is possible to meet and interact with other users and applications.

Our focus is not on the multi-user collaboration 3-D environment, but on the daily work situation using applications or the operating system. We aim at maximizing the efficiency for the user of our system, by enabling the users innate abilities to comprehend and manipulate the 3-D environment.

Chapter 4

Design background

4.1 Scene representation

A “scene” is a commonly used term in anything that has to do with 3-D computer graphics, and this project is no exception. We define this term to mean a description of a spatial structure of surfaces and light sources through which one can model elements from the real world such as a room with a gallery of paintings or all the “wheels a gears” of a nuclear power plant control room, and which lends itself to being rendered on standard hardware through OpenGL a similar graphics API. For more information about OpenGL, see [SG].

So, the primary job for 3D-Console is to manage and render such a scene. 3D-Console must honor requests from applications to add new elements to the scene and to update it in a variety of other ways. But how would one go about representing it? Which data structure will support the needs of 3D-Console best? Out of the many different ways scenes are represented, two stand out as being more frequently used than the others. The first one is a low level representation which consists merely of a list of polygons. The polygons are described as lists of vertexes and the vertexes are associated with a set of properties, usually coordinates, color, normal and texture coordinates. This representation is sometimes referred to as “polygon soup” since it possesses no inter-polygon structure.

4.1.1 The scene graph

The other typical representation is the scene graph. The scene graph is a high level representation compared to “polygon soup”. It features a hierarchy of coordinate system transformations to help represent the scene. The graphical elements and the coordinate system transformations are both nodes in the scene graph, and the key feature of this representation is that the absolute coordinates of a graphical node is given only indirectly as the combination of all the coordinate system transformations which are crossed on the path back to the root of the graph.

A scene graph is always a directed acyclic graph (DAG), and as far as we know, it always has just a single root. This restriction leads to the important concept of containment. That is, any node except the root node must be thought of as a child of another node which is closer to the root, and may itself have a number of other nodes as children. Because this is a graph, and not a tree structure, a node may be a child of several other nodes simultaneously. This brings us to the concept of reuse, which is another very important property of the scene graph. A graphical node has a distinct appearance for each distinct path that leads back to the root. This, of course, is most interesting when the paths go through different coordinate transformations such that the appearances gets separated in space.

The real strength of indirect coordinates and multiple appearances emerges from the logical concept of compound objects. Imagine that we need to model a panel with 3 rotary buttons. We could build one rotary button as a compound object, then we could insert it as a child in 3 different transformation nodes which could then be added as children to the panel, see fig. 4.1. Each button may be turned to a custom angle without affecting the other buttons. Further more the panel may be moved to any location and the buttons will move with it. The panel with its buttons may itself be thought of as a compound object. Note that a compound objects are themselves scene graphs. That is, they have a single root, and as a set of nodes (in a mathematical sense) they are closed over the parent to child relation.

Finally it should be noted that scene graph systems often deliver a comprehensive set of higher level geometric shapes and components, such as spheres, cylinders, boxes, elevation grids and spline surfaces. Scene graph systems tend to encapsulate most of the low level details of rendering, such

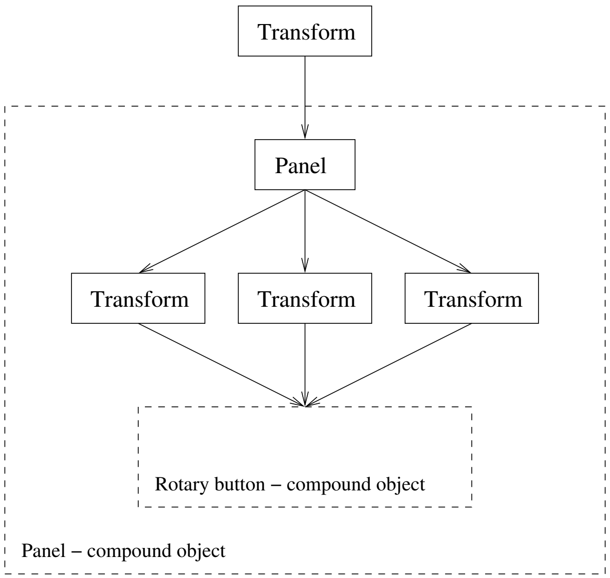

Figure 4.1: Example of a scene graph - button panel. This figure illustrates the concept of compound objects, how compound objects may be nested, how they may be reused and how they may be relocated as an integral entity.

as surface tessellation and various performance tricks. On the other hand there is no inherent reason why scene graphs could not at the same time offer a type of node which represents a bunch of low-level polygons.

Now, without regard to specific needs by 3D-Console let us summarize the advantages of the two different kinds of scene representation described above:

Low-level polygon representation (e.g. OpenGL):

- Simple to implement, less code is needed.
- Fast, since data may be streamed directly to the hardware.
- Flexible, since it generally resembles the abilities of the rendering hardware more directly.

High-level scene graph representation (e.g. X3D/VRML):

- Easy coordinate manipulation of compound objects.
- Reuse of compound objects.
- Easy to author and read, more intuitive, less verbose.
- Independent of many rendering/performance tricks.
- Rendering quality independent.
- Functional as opposed to visual.

It may come as no surprise that we have decided to go with the scene graph representation in 3D-Console. This decision was primarily a consequence of one of our stated goals: "To ease the burden of creating 3-D applications by lifting the level of abstraction to a higher ground" (see section 2.4). There are also a number of more concrete reasons to choose the scene graph over a low level "polygon soup" type of representation. We stated that applications need to work in a private local coordinate system, and that users need to be able to manipulate an application as a combined entity. This requirement seems to go well with the concept of compound objects in the scene graph.

We acknowledge the fact that the flexibility of the low-level representation would allow the applications to fiddle with details and tweak the abilities of the hardware to improve their visual appearance, but we do not think this is an important attribute in the context of the 3D-Console. We ascribe a much higher importance to functionality which we think is supported better by the scene graph representation.

4.2 X3D

By choosing to base our 3D-Console on the scene graph representation we have also accepted the fact that our user interface needs to support a comprehensive set of features (see section 4.1.1). So the question of which set of features to support has now become urgent. We knew X3D was based on a scene graph, and it was thus natural for us to start looking more into the world of X3D. We quickly realized that in fact the X3D concept supported many of our ideas for 3D-Console well.

In the remainder of this section we will give a brief introduction to X3D and round off with an discussion of why we chose X3D as a foundation of 3D-Console. It is immediately obvious though, by choosing to incorporate the X3D concept, we relieve our selves from a major part of the burden of determining the set of primitives to offer to the applications.

4.2.1 What is X3D

X3D is a brand new Extensible 3-D Graphics specification [Grob]. In fact it is still "work in progress". It is being designed and maintained by The X3D Graphics Working Group. It builds heavily on the (now aging) VRML 97 (Virtual Reality Modeling Language) specification.

The primary benefits of X3D compared to VRML is that X3D is designed for extensibility, that the X3D specification contains much fewer ambiguities on behavioral topics than did the VRML specification and finally that the specification of the SAI[1] is much more mature and in the line of our 3D-Console than its predecessor, EAI, from the VRML specification.

As stated, the specification has not been finalized yet. There has been approximately 5 releases of drafts since the beginning of the project up to the time of this writing. The currently latest draft is from December 26, 2002. A new Committee Draft is planned for submittal to ISO[2] in October/November this year. The latest draft may be found on the following address:

`http://www.web3d.org/technicalinfo/specifications/ISO_IEC_19775`

[1] SAI is described in 4.2.3.

[2] *ISO* (International Organization for Standardization) is the world's largest developer of standards.

The fundamental parts of the specification, of course, have not changed a lot. Still, we have had some challenges due to vague or conflicting statements in the early drafts on such subjects as the naming of node instances, the relationship between nodes and execution contexts, and the propagation of events. In several cases the interpretation we chose initially turned out to be wrong with respect to the latest specification draft which has become less ambiguous on these matters. In most of these cases we have changed our code to regain compliance. See chapter 6 for more details.

So, why did we stick with X3D regardless of these problems? There are several reasons for this. The first one is that we find X3D to be very suitable as a foundation for 3D-Console - as described above - despite the immaturity of the specification. The second reason is, that it is a chance for us to be among the first to deliver an X3D compliant system. It should be noted here that at the time of writing there are, to our knowledge, no reasonably complete open implementations of the X3D specification. The last reason for sticking with X3D, is that it gave us a chance to influence the development of the specification. We have actively utilized this opportunity by submitting comments and suggestions to the X3D contribution mailing list `x3d-contributors@web3d.org`. We have received positive and helpful responses, and the latest specification does in fact reflect our feedback in several places. For the full listing of the correspondence with the editors of the X3D specification, see appendix D. We have also been maintaining a list of specification obscurities. This list is found in appendix E.

4.2.2 X3D is not a file format

Though many people think of it as such, X3D is not a file format. Parts of the specification is devoted to specifying various alternative file formats for X3D. The most obvious being XML (Extensible Markup Language) and the traditional VRML notation. But what is more important to us is that the X3D specification is a specification of a software system. A software system that can handle a three-dimensional scene graph, render it, propagate field values from node to node via events and routes and communicate with external applications.

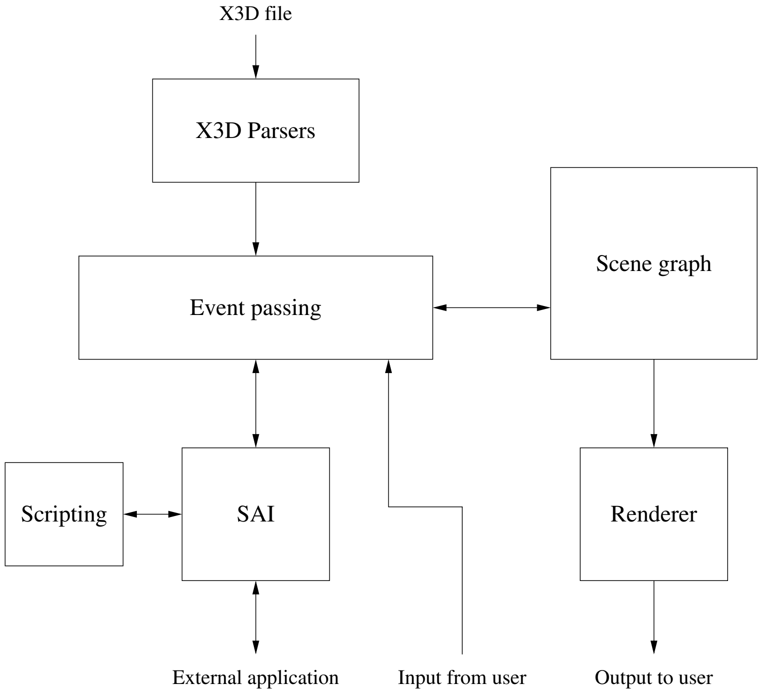

Figure 4.2: X3D Architecture. The scene graph is the central data structure of an X3D system. The scene graph is presented to the user by the renderer. Most other access with the scene graph is handled through passing of events. The X3D parser builds scene graphs from external files. Scripts and external applications access the scene graph through a Scene Authoring Interface (SAI). User input (e.g. touch) is available as events.

4.2.3 The specification - a quick tour

As revealed, X3D uses a scene graph to represent the scene, see fig. 4.2. The scene graph is indeed the central axis around which the entire specification revolves. As nodes we find everything which we would expect. This includes geometric nodes such as `Sphere`, `IndexedFaceSet` and `NurbsSurface`. In X3D, text is also considered a geometric primitive and is displayed with the `Text` node. New local coordinate systems are introduced by the `Transform` node. There is a node, `Group`, for grouping other nodes. `Transform` is also a grouping node.

Input

There is a variety of nodes which deals with input. They are all called sensor nodes in X3D. `TouchSensor` registers touches by the pointing device. `PlaneSensor` registers movement and `StringSensor` can be used to input text. In X3D timing is also considered as input. The `TimeSensor` can be programmed to generate events at regular intervals, but has a lot of other possibilities. The notion of events in X3D is explained later.

Inheritance

A fundamental part of the X3D specification is the inheritance hierarchy of X3D nodes. There is currently 161 distinct node types in the X3D specification of which 115 are concrete (i.e., not abstract). The abstract node types exists merely to group other node types with common features. At the root of this hierarchy is the abstract node type, `Node`. At the leaves are all the concrete node types of which some are mentioned above. For the complete inheritance diagram, see the X3D Specification [Grob] part 1, section 4.4.2.3.

Fields

In X3D all nodes are specified as a collection of fields plus a description of their meaning and behavior. The fields of a node represents its interface to its surroundings. This includes the other nodes in the scene graph, scripts and external applications. External applications are further described below. Fields are inherited from ancestor nodes according to the hierarchy described above.

It is important to understand exactly what a field is capable of doing in X3D. First of all a field has a type associated with it. The type determines which kind of values that may be read or written to it. The possible types include integers, `SFInt32`, floating point numbers, `SFFloat`, three component vectors, `SFVec3f`, strings, `SFString` and references to nodes, `SFNode`. All field types have “multiple value” variant which starts with an “M” instead of an “S”. For the complete list, see the X3D Specification [Grob] part 1, section 5.1.

Regardless of the type of the field, there are two properties which a fields may or may not have. One is the input capability and the other is, of course, the output capability. This gives rise to four classes of fields. The “input only” fields, the “output only” fields, the “input output” fields and the “initialize only” fields. The four classes bear resemblance to well-known concepts in classes of ordinary programming languages and should in general be understood like this in X3D. “input output” fields does in this respect represent public variables. That is, variables which may be read and written by the outside world at any time, but has an influence on the behavior of the object. “initialize only” fields represent private variables as a means of customizing the object. “input only” fields represent “setter” methods which may do other things than just assign the argument to an internal property. Finally, as you might have guessed, “output only” fields represent “getter” methods.

Listing 4.1: Formal X3D node type specification. Both inheritance and fields are specifed. For fields the left most column is the field type, then comes the input/ouput capabilities, then comes the name of the field, then comes its default value, and finally the value domain is specified.

```
Transform : X3DGroupingNode, X3DBoundedObject {
  MFNode      [in]      addChildren                  [X3DChildNode]
  MFNode      [in]      removeChildren               [X3DChildNode]
  SFVec3f     [in,out]  center           0 0 0       (-∞,∞)
  MFNode      [in,out]  children         []          [X3DChildNode]
  SFRotation  [in,out]  rotation         0 0 1 0     [-1,1] or (-∞,∞)
  SFVec3f     [in,out]  scale            1 1 1       (0,∞)
  SFRotation  [in,out]  scaleOrientation 0 0 1 0     [-1,1] or (-∞,∞)
  SFVec3f     [in,out]  translation      0 0 0       (-∞,∞)
  SFVec3f     []        bboxCenter       0 0 0       (-∞,∞)
  SFVec3f     []        bboxSize         -1 -1 -1    (0,∞) or -1 -1 -1
}
```


Now we know enough to understand the formal notation used in the X3D specification to define node types. In listing 4.1 we have shown you an example of such a definition.

Events

In X3D, the notion of events is the key to making dynamic scenes. An X3D event is generated by a field having the output capability, whenever the value of that field changes. Through the use of routes, the event can be directed to any other field having the input capability and the same field type as the source field. The purpose of an event is to carry the new field value along the route. The effect is that the new value of the source field will be assigned to the target field. When routes are chained through a sequence of “input output” fields the new value will propagate to all the affected fields immediately. Both “fan-ins” and “fan-outs” are allowed. This means that any number of routes may be sourced from a single field, and any number of routes may be targeted to a single field. This generality gives rise to a number of problems such as propagation order and cycle breaking. These matters are all treated by the specification under the subject of “The event cascade”. See the X3D specification part 1, 4.4.8.3 Execution model.

In general field values do not change by themselves. “input output” fields only change when an event is sent to it. It is in general only the “output only” fields which can be used as the ultimate source of events. “output only” fields are found in all sensor nodes, since their purpose is to translate external physical events to X3D events.

SAI

The X3D specification offers a Scene Authoring Interface (SAI). The SAI is a language neutral specification of an interface (API) between the server and various forms of custom code. The term *server* is not used by the X3D specification. It uses the term `Browser` to describe the X3D system as a whole. In the remainder of this report we will use the words *server*, *engine* and `Browser` interchangeably to refer to the same thing. In X3D, custom code can exists in one of two forms. The form that has most relevance to us is external applications. The other form is scripts, which are embedded within the scene graph. The specification uses the phrases "internal interaction" and "external interaction" to distinguish between the two alternative forms. There is a subset of the SAI which is available only to "external interaction". The specification of the SAI is found in part 2 of the X3D specification.

The SAI allows an external application to manipulate the current scene graph of the Browser, create new nodes, and to combine them together. Through the SAI, an application can read the value of any field having the output capability, and write to any field having the input capability. Of course, to access the fields of a node, it must first obtain a reference to that node. Reading a field value is a concept that is unique to the SAI. Writing a value, on the other hand, is just like sending an event to that field through a route. This means that if the field is an "input output" field, it will generate an event carrying the new value.

The SAI also allows an application to register interest in a field with the output capability. When doing this, the server will automatically route change events to the application. Which means the application has to register a handler for the event. How this is done is not specified, since it will depend of the choice of programming language.

Execution context

An execution context (mostly expressed as an `ExecutionContext` object in the specification) is a concept of importance to 3D-Console, and this is because it represents a way of enclosing or "fencing" a part of the scene graph. To understand the execution context we must first understand the notion of a scene as defined by the specification. Simple scene graphs consists of a single scene, but X3D supports the concept of sub-scenes. That is, one scene can be inserted into another scene with the help of the `Inline` node. But even though several scenes may be combined to construct something bigger, the individual sub-scenes are still complete scenes by themselves. This implies that a sub-scene in no way can refer to elements of the scene which it is embedded into. More specifically, a scene is a closed name space (or scope) for node names, and consequently there is no way for one scene to refer to outside nodes. This e.g. implies that it is not possible to place routes between nodes from two different scenes. All these restrictions are, of course, only fortunate, since they allow the individual sub-scenes to be authored without caring about name clashes with nodes in the other scenes.

The concept of prototypes in X3D is similar to that of sub-scenes. The common denominator is the local name space. Since prototypes are defined once but instantiated several times, node names in the individual instances would clash if they lived in the same name space.

Fortunately for us the X3D specification makes the described commonality explicit through the `ExecutionContext` object, and as a natural consequence, both scenes and prototypes are derivatives of the `ExecutionContext`. As the reader may have guessed, we find that the execution context also supports the concept of a 3D-Console application well. We will explain this matter further in section 5.7.

Scripting

X3D supports embedding scripts in the scene graph. This is done with the `Script` node. It serves primarily as a way of adding autonomous animation into the scene. This relies heavily on the notion of events. Support is required only for the ECMA[3] scripting language.

4.2.4 The choice of X3D

We have expressed before, that we find X3D to be a good foundation for 3D-Console. The benefits of choosing X3D as opposed to inventing our own alternative, is that X3D is standardized, and thus will be known a priori to many of those who gets their hands on 3D-Console. The other benefit is that the X3D specification contains a huge amount of knowledge about which set of primitives are needed to build three dimensional models. That X3D in fact serves our needs in all relevant aspects is not immediately obvious though. Through the rest of this section we will attempt to make this assertion probable.

Let us first take a look at the requirements imposed by the applications (see section 3.8.1):

Modeling with geometric primitives
: Is directly supported.

[3] *ECMA scripting* is an ISO standard "ISO/IEC 16262". It is based on JavaScript as it was in Netscape Navigator 2.0

Display of text
    Is supported by the `Text` node.
Display of image
    Is supported by textures (e.g. `ImageTexture` node).
Shut down event
    Not directly supported.
Input of text
    Is supported by the `StringSensor` node.
Receiving of gesture events
    Not directly supported.
Touch sensitivity on specific surfaces
    Is supported by the `TouchSensor` node.
Movement sensitivity on specific surfaces
    Is supported by various sensor nodes (e.g. the `PlaneSensor` node.
Inter-application communication
    Not directly supported.
Colliding object notification
    Not directly supported.
Privacy, no interference with model from other applications
    Is supported partially by the execution context.
Locality, application models in local coordinates
    Is supported by the `Transform` node.

There are four items from this list that are not directly supported, these will be describe further in the next section.

In section 3.8.2 we mention several requirements imposed by the user. The four standard types of navigation (see section 3.7.3) are supported by the X3D specification, but in X3D it is the scene that specifies which kinds of navigation may be chosen via the `NavigationInfo` node. In 3D-Console we cannot normally allow application to control which type of navigation the user may use. This restriction can be applied through permission control (see section 3.7.7). A sensible user navigation will require detection of collision between the user and the geometry around him (or her). This kind of collision detection is supported by X3D via the `Collision` node in a selective manner.

Another user requirement was the manipulation of application location,

orientation and scale. This is of course not directly supported by X3D, since in X3D, applications do not have a section of the scene graph assigned to them. On the other hand, the X3D specification requires that a hierarchy of oriented bounding boxes (OOB) is maintained withing the scene graph. In section 9.2 we propose a way in which these (normally invisible) boxes may be used to help the user to manipulate the applications coordinate system.

The last user requirement which is mentioned regards the forceful limitation on spatial application extent. This requirement, again, is not directly supported by X3D, but may again utilize the oriented bounding boxes in the scene graph. This aspect is also described further in section 9.2.

The final point we will make to justify the assertion, that X3D is suitable as a foundation of 3D-Console, is that the interface between server and application is very well supported by the SAI specification. A number of extensions will be needed, but the major part of the protocol is given from the SAI.

4.2.5 What is missing

In this section we will sum up all of the aspects of 3D-Console that, at this point, are known not to be supported by X3D. We will focus here on whether we find the addition of those aspects complicated due to specification conflicts, or if we find them to be easily integrated into 3D-Console as an X3D compliant system.

A shut down event from the server to an application is an addition on top of the system, and we regard this as a real easy one. Reception of gesture events will for the same reason be easy to add to the system. The actual recognition of gestures will, of course, not be easy to implement, but in this section we only consider the problems in merging the features with the X3D specification.

Inter-application communication will be an easy feature to add, since it can be build on top of the existing features. A solution is discussed in section 9.1.

Colliding object notification would be hard to deal with if it had to be extensively integrated into the existing features of the specification. On the other hand, if we restrict it to be a feature offered to applications, it will be much easier to manage. A solution is discussed in section 9.4

Two pointing devices (two hands) are not immediately supported. That is, the information of which device that activates a touch sensor is not available. The addition of this feature is somewhat complicated since it requires extensions to the fundamental parts of the system.

The notion of an external application in the SAI specification is quite different from that of a 3D-Console application. In the SAI specification an application is given (because it is assumed to require) full control over the scene graph. The main idea here, is that an external application can use an X3D engine as a kind of companion to render whatever it wishes. It may even be the application, which launches the X3D engine in the first place.

So this obviously stands in sharp contrast to the concept of a 3D-Console application. We require that the 3D-Console acts as a server and the applications act as clients, and not vice versa. Further more, applications should be regarded by the 3D-Console, as processes are regarded by the kernel of an operating system. That is, applications must be assumed to be hostile towards each other and must be dealt with accordingly. For example, it should only be possible for an application to manipulate the part of the scene-graph which belongs to itself. Another example would be the enforcement of limitations on resource allocation to prevent one application in degrading system performance for the other applications.

One the very good things about X3D is that it is extensible. Therefore many of the things which are needed in a full-blown 3D-Console can easily be added to the system. In this section we have focused mostly on fundamental requirements, but if we gaze a little into the future, it is obvious that we will need to build graphical objects to represent the basic entities of an operating system such as files, streams and processes. The objects should be customizable to a certain extent. Such objects could be introduced into X3D via the `EXTERNPROTO` feature (see the X3D specification for more details on `EXTERNPROTO`).

4.3 CORBA

CORBA is an open architecture and infrastructure specification, which applications can utilize when working together over networks. CORBA is an acronym for Common Object Request Broker Architecture and its specification was produced by OMG (Object Management Group). One of the

benefits of using CORBA is that CORBA-based applications can interoperate with other CORBA-based applications, even though they are not on the same computer architecture, operating system or network. This is possible because CORBA separates the interface from the implementation.

What CORBA defines is a way of wrapping objects in code with CORBA interfaces, in such a way that operations on objects can be requested no matter where the object is, as long as you have a CORBA object reference to it. Since CORBA uses its own unique object references, it is also transparent, where the object is actually instantiated.

Described at a very high level, CORBA has an Object Request Broker (ORB) on every server which utilizes CORBA. The ORB handles request for operations on CORBA objects for applications. If the object is not local, then the ORB on the server, where the CORBA object is instantiated, is contacted using the network protocol IIOP. The remote ORB then handles the request, and sends back the result to the local ORB. The local ORB then returns to the application. For the applications, all this is hidden. Seen from the application it is just operating on a local object.

CORBA can do a lot more than described above. It has mechanisms for load balancing, resource control and fault tolerance on the server side. More detailed information about CORBA, can be found on OMG's website. [Groa]

Mainly we use CORBA to handle the communication between client applications and the 3D-Console server. We have gone a great distance to hide the CORBA implementation in the client API, since we feel that CORBA is just an architecture for handling the communication, it is not something the client application developers needs to know about when creating 3-D environments/applications. See section 6.12, for more information on how we use CORBA in the client API.

We use an open source implementation of CORBA called OmniORB. It has been tested and certified to be CORBA 2.1 compliant. It is freely available from SourceForge.[Omn].

Chapter 5

System design

This chapter will step through the various issues which must be dealt with to achieve a sound design of the 3D-Console. This includes aspects of the design of the X3D engine[1] itself, as well as issues regarding the client API, which will be built on top of the SAI. X3D and SAI are introduced in section 4.2.3. We will put our primary focus on the client API, since most of the design of the X3D engine will follow more or less immediately from the X3D specification. Further more, the design of the client API is far from trivial, since here we have to extend the specification to achieve something which at the same time supports the requirements for 3D-Console and is compatible with the X3D/SAI specification.

5.1 Client/server

From the previous chapters it should now be obvious that our system will be build around the concepts of X3D. This means the user interface as a whole will be represented internally in the system as a scene graph which is, at least, compatible with the X3D specification. The important thing to note here, is that all the interfaces of the running applications will be gathered inside one single-rooted scene graph. Any other approach, like having a cluster of smaller scenes, would be hard to merge with the concepts of X3D.

[1]The phrase *X3D engine* is a synonym for a complete X3D system.

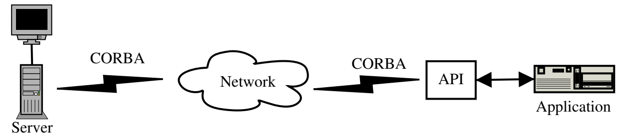

Figure 5.1: Overview of 3D-Console

Having one big "global" scene graph also means we need to have a central server to manage it. Since we need to service multiple independent operating system processes, the server needs to be an independent process itself - a "stand alone" server.

3D-Console must be designed in such a way that when it is started from an operating system shell it loads in a base scene. It then connects to or initializes a viewing system, and starts rendering the scene to it. At the same time it initializes the CORBA ORB, and starts to await connections from client applications. For an illustrative overview, see figure 5.1.

When applications are started, they must know how and where to contact the server. The most obvious way to do this with CORBA, is to utilize the CORBA name service. This requires that a CORBA name server (a standalone program) is running and accessible to both server and applications. The server then registers itself during ORB initialization, with a name which must be known by the applications too. For maximum flexibility, we will support that the name service name of the server can be specified as a parameter when launching the server and when launching the "3D-Console aware" applications.

Altogether we should arrive at a client/server system quite similar in concept to the X Window System available on UNIX/Linux. Here the X server owns the screen and is responsible for rendering the 2-D windows base user interface. X applications connect to this server, and uses a network based protocol to create new windows and build up the contents of them.

5.2 X3D compliance

So, our server will for a large part be an X3D engine, of course. In fact we want this server, to be able to act entirely like a `Browser` as defined by the X3D specification. That is, it should be possible to launch the server with any valid X3D scene file as input, and then have the server presenting it in the way required by the specification. This includes presentation, navigation, behavioral aspects and input features. The reason for this requirement is that it would be unfortunate (and disappointing for us), to have a system which in very many aspects resembles an X3D Browser, but never the less, cannot be used as such. Especially today when so few X3D implementations exists.

On the other hand we wish to make perfectly clear that the process of achieving 100% X3D conformance is not our main focus. Instead we will concentrate (here and in the following) on the functionality that we need to add to X3D to reach 3D-Console, and how it can be added without violating the X3D specification.

5.3 Type system

The type of an X3D node is an important issue with respect to design. Let us define the phrase *node type* for a certain node `N` to mean the specification of its fields (both the inherited ones, and the ones it adds by itself). Each field specification must contain the name of the field, the type of values for the field, and finally its input/output capabilities - see section 4.2.3, Fields.

With this definition of a node type, we can use the node types to model a variety of aspects of the 3D-Console. The point is, the node type only describes the interface to the node. The rest of the semantics of a node need only be known to the node itself. Therefore we can use the node types in those parts of the 3D-Console, which deals only with input and output of field values. Examples of such parts are the entire client API, the XML and VRML parsers and the ECMA scripting facility. The rendering code is an example of a part where the node type is not enough to know. Here we need the entire semantical definition of the node.

To be able to utilize the node types in a convenient and effective manner we need to be able to query an instance of a node for its node type at

run-time. Without this ability we would be forced to handle each of the 160 different node types individually in the client API, in the parsers, and in the scripting interface, and even worse, we would need to make changes to all those parts whenever a new node type is added or modified. We shall thus require that a run-time type system for nodes and fields is build into 3D-Console.

5.4 Multi-threading

The server needs to handle many different jobs in a way that is not obviously sequential. It has to render the scene at regular time intervals (frames). It has to service incoming requests from all the applications, and it has to load content from anywhere on the Internet without influencing the frame rate. To properly meet these requirements we have chosen to employ a multi-threaded[2] design. CORBA supports this directly, in that it can be configured to automatically spawn a new thread for each incoming request. The reader should note here, that this may give rise to a large number of concurrently executing threads, since CORBA does not wait for one request to finish, before it deals with the next.

Loading of content will require some care. The requirement posed by the specification is that when a new URI[3] is assigned to a URI field of some node (e.g. an `ImageTexture`) then the X3D system must initiate a download of the referenced image. The assignment operation must finish immediately, and, as stated, the rendering must not be delayed. As soon as the image is fully downloaded it must appear to the user. Since the old texture is immediately disposed in the visual sense, this means there may be several frames where no texture is displayed on the affected surface.

We believe these requirements are best handled with a *job queue*. A job queue is a general purpose utility, which allows its users to submit new jobs to it. The job queue will then execute the jobs in a fashion that it sees fit. Normally this means that it will allocate a certain number of threads, a thread pool. Whenever there is an available thread in the pool and there

[2]A *thread* is also known as a light weight process, and the phrase *multi-threaded* describes that several threads live in the same address space, and thus are able to access shared variables. Synchronization mechanisms will be required though.

[3]*URI* is an acronym for Universal Resource Identifier. To put it simply, it is just a newer name for the more commonly used *URL* (Uniform Resource Locator).

are pending jobs, it will pick out the thread to execute the job. If there are no available threads, the pending jobs must wait in the queue. Using a thread queue has at least two advantages. It prevents allocation of too many threads, which could otherwise have threatened performance, and it encapsulates the functionality into an intuitive utility.

Apart from the request threads spawned from CORBA and the threads governed by the job queue we will allocate one thread with the sole purpose of rendering frames.

5.5 Application resource accounting

For the purpose that 3D-Console has, it is important to choose a design which handles all system resources in a strict manner. Many applications and systems does not possess a strict resources management. Therefore they fail to release resources completely or in a timely manner, which in turn means they need to be restarted from time to time. Often this is because they do not need to run for extended periods of time, but equally often it is due to poor design and/or implementation.

3D-Console must regard the issue of resource management in the same strict sense as is done by an operating system, since it is supposed to run forever, and provide the same, high quality, service to all applications, no matter for how long our system has been running and no matter how many misbehaving applications it has serviced.

To achieve this we have to go against the X3D specification. One of the aspects of the way external applications are viewed by the X3D specification, is that the changes made to the scene graph remain in effect after the applications disconnect or die. This view is incompatible with our stance on resource management. Therefore we will have to introduce the concept of *per application resource accounting*, and require that the server deallocates each and every resource acquired by the application when the connection is lost. This implies that the part of the scene graph added by the application will be removed.

5.6 Memory management

But there is more to resource management than keeping track of resource allocated by application. The server itself must be designed to manage memory and thread allocations. Memory allocation is a difficult issue in languages where the deallocation must be done manually, but the problem can be lessened through either employing an automatic garbage collector or through the use of smart pointers. In 3D-Console we have chosen to use smart pointers. This is because they address the requirement of timely deallocation best.

Smart pointers work by counting the number of references to a memory region, and the memory region is then automatically freed when the last reference is dropped. Smart pointers are normally introduced as a new class that encapsulates the details of reference counting, but works outwardly exactly like a conventional pointer.

5.7 Application sessions

We have just concluded that the server needs to track application resources tightly. We will achieve this by introducing the ideas of an application session and an application scene within the server. An application session is allocated upon each new connection from an application. The session lives until the connection is broken. Sessions are illustrated on fig. 5.2.

5.8 Application scenes

In X3D the notion of an execution context is particularly interesting to our applications. See section 4.2.3 for an explanation. From the concept of an execution context we derive the *application scene*. By doing this, we immediately gain the desired protection between applications and the base scene in the server. In X3D an external application can construct several new scenes, therefore we shall also allow our applications to construct any number of application scenes. Application scenes will be remembered by the session, so they can be removed from the scene graph when the application disconnects from the server.

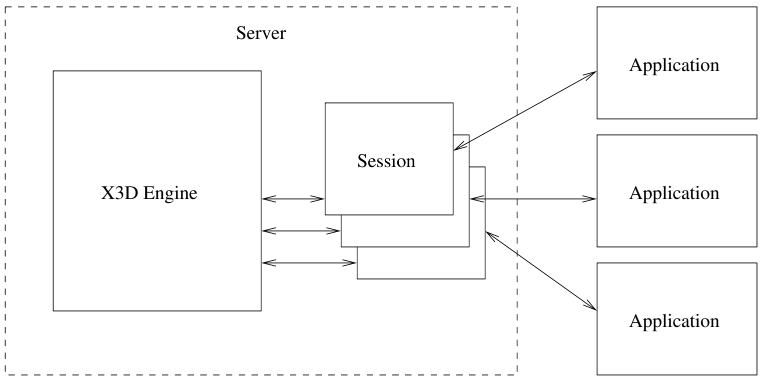

Figure 5.2: Application sessions. Sessions are created for each new application connection, and are destroyed when the connection is broken. The role of the session is to track application resource allocation.

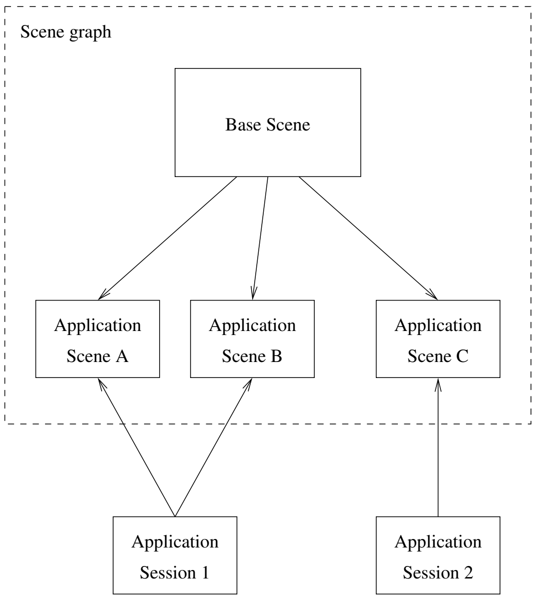

Figure 5.3: Application scenes. An application scene is similar to a `Scene` as defined by X3D. It is based on the `ExecutionContext` which protects the rest of the scene graph from it. One external application may construct one or more application scenes in 3D-Console. When a session dies, all its scenes are removed from the scene graph.

The automatic removal of scenes, is one of the areas where an application scene differs from an ordinary X3D sub-`Scene`. Another difference will be that an application scene is not added to the scene graph in the same way an ordinary X3D `Scene` is. The server will be responsible of where to insert the application scene into the base scene. Therefore we will need to add a special `launch` method to an application scene. When this method is invoked by an application, the server will add the scene to the scene graph in a place it sees fit.

By requiring that the application fills in the scene before launching it, the server will be able to calculate the bounding box of the application scene upon launch. This in turn will allow the server to display the bounding box of the application, and let the user manipulate its position, orientation and size, before finally accepting the application. This "launch" scenario is further described in section 9.2.

The reader may ask himself/herself how application are prevented from using ordinary sub-scenes and inserting them anywhere in the scene graph. We want complete conformance with X3D, so how do we force applications to use our extensions, where restrictions apply, instead of just using the plain X3D alternatives? We imagine this should be achieved through permission control, which would be negotiated at connection time. Thus, when the server is started normally, it will not allow applications to use ordinary sub-scenes. On the other hand, it must be possible to start the server in a "relaxed" mode, where applications are free to use all parts of the SAI. This mode should be completely conforming to the SAI specification.

5.9 Client API

The SAI specification defines a rather large set of operations (entry points) which must be offered to external application. It does not, however, dictate how these operations are to be offered to the applications. The implementor may choose to provide the SAI operations by linking[4] application code directly into the server, such that the application lives in the same address space as the server and is able to call functions in the server directly. Alternatively the SAI operations could be encoded into a network protocol and

[4] In this context the term *Linking* refers to a phase in the process of compiling programs. It is the final step where references to code and data are resolved between separate object files.

thereby be offered to applications anywhere in the world. Since our applications are independent programs, we need to choose the latter proposal, the link independent one.

As stated in section 4.3 we use CORBA to handle the encoding of the SAI operations so they can be passed over the network. CORBA has a number of properties of relevance to 3D-Console:

- Network transparency
- Platform independence
- Language independence

By choosing to base the client API on CORBA, the API inherits these properties.

Network transparency means that the SAI operations are accessed in exactly the same way, whether the application runs on the same computer as the server or they run on distinct computers which are connected by a network. Platform independence refers to the fact that different computer architectures can communicate with each other. Language independence is achieved because CORBA is available for a wide variety of programming languages.

When using CORBA, one writes the interface in the IDL[5] language. The IDL for the SAI, and the extensions to SAI, is included in appendix H. Just like in the SAI specification, the IDL divides the services of the API into four major groups, session services, execution context services, node services and field services. In the specification there are several other groups of services but that functionality is either just barely implemented or not implemented at all in 3D-Console — yet. Also, session services are called browser services.

The idea in the IDL is that an application establishes a connection to the server by obtaining an instance of the `Session` interface. All the implemented services exists as methods in the `Session` interface. Most services correspond to services from the SAI specification, and their names will match closely. The services which we have added will be explained in the next section. In this API, nodes, execution contexts and fields are all represented by identifiers rather than actual object references. The

[5] *IDL* is an acronym for Interface Definition Language. It is a specification language, independent of any particular programming language, and CORBA specifies how such specification are mapped to the particular programming languages, i.e. language mappings.

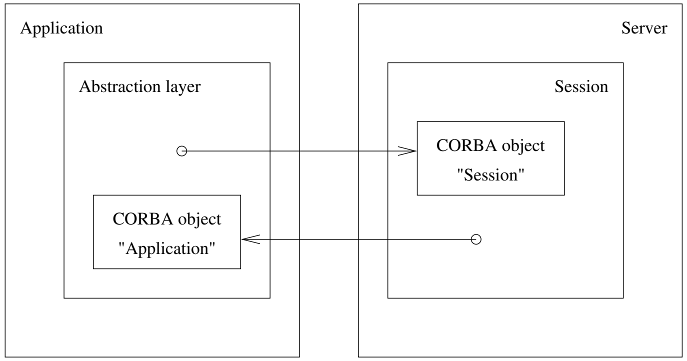

Figure 5.4: Established connection. When a connection is establish between the server and an application, the application has a session within the server. The application requests services on the server via a reference to a CORBA object named `Session`, and the server can likewise request services from the application via a reference to a CORBA object named `Application`.

identifiers are formed by small integers, and this makes the interface simple and efficient.

5.10 Additions to SAI

In the previous section we stated that a connection was established through the acquisition of a `Session` object, but we did not specify how that was done, and that is not defined by the SAI specification either. In 3D-Console the procedure will be, that an application, as the very first thing, acquires a CORBA object reference from a CORBA name server to a `Server` object. With the `createSession` method on the `Server` object, the application can then ask 3D-Console for a new session, and, thus, for a reference to a `Session` object.

When the application calls the `createSession` method, it must pass a reference to an `Application` object that represents itself. This is illustrated in fig. 5.4. Sending the `Application` object reference to the server, allows the server to contact the application at any time it wants, and the server has several important reasons to do that. First of all, the SAI specification requires that an application can register interest in field changes.

The application does that with the `registerFieldInterest` method of the `Session` object, but when the server needs to inform the application of a change, it must contact the application via the `Application` object. The `handleEvent` method in the `Application` object will be called by the server, and the application must implement this method as a general event handler. To minimize the overhead in establishing network connections the server will buffer events and send them in batches to the application.

As we have mentioned earlier, we require that the server frees resources in a timely manner. This means the server must detect the death or unavailability of an application, in case it should happen before the application explicitly chooses to disconnect. When using CORBA, there is no immediate way for the server to know when that happens. That is, an application could crash or hang, or the network connection could be broken, but no one would tell the server about it. Therefore the application scene would remain in the scene graph, inactive and stale. To cope with this problem we have chosen to use the `handleEvent` method. The trick is that the server will call this method on a regular basis regardless of whether there are any events to send or not. When there are no events, an empty list will be passed. We should think of this, as the server "pinging" the application. If the application has not responded within a certain time limit, the server interprets this as an application crash, and takes the same actions as if the application explicitly disconnected.

Although the SAI specification requires that an application gets notified about field changes when it registers interest, the specification does not require that the application will receive a distinct message for each of the values the field attains. The stance is that if a field goes through a series of rapid changes, only the last change needs to be reported to the application. In 3D-Console we need to tighten this up, and require that every change is reported distinctly and uniquely to the application as separate and correctly ordered events. We shall require this because our applications have a higher importance than what is generally assumed by the SAI specification. An example is an application which accepts textual input from the user through a `StringSensor`. We cannot allow any of those pieces of text to disappear, no matter how fast the user types.

In 3D-Console the server and the applications are very dissociated. In particular they may all be running on different computers. This condition imposes a problem for the application when it builds its model. How

should the application refer to local files containing textures and similar kinds of content when all it has to refer with is a URI which is evaluated on the server? One possibility would be to install a web-server on the client computer, and use it to give access from the server computer to the client files. But that is not really an acceptable situation. Instead we have chosen to extend the SAI with a feature which allows the server to request client files directly. It does this through the `requestFile` method of the `Application` object.

But since a URI is still the only means of reference available when specifying which texture to use in a `ImageTexture` node, we are not yet "home free". We will solve this last problem by inventing a new custom URI scheme which in it contains a reference to a specific session on the server. In X3D every execution context has a base URI against which all relative URIs are resolved. If we then give the application the opportunity to use our alternative URI scheme in the base URI for its application scene, then a relative texture URI will be resolved against the base URI and the server will know which application it should request the file from.

In section 3.7.7 we suggested that permissions were negotiated at the time of connection establishment. Even though we have omitted support for this in the current version of the IDL, we wish to note that the `createSession` method of the `Server` object would be a particularly good place to pass credentials to the server and thereby obtain a certain set of permissions. The reason is that the specification does not cover the establishment of the connection, but later on it will be hard to incorporate the notion of permissions since the SAI does not mention them.

If you look carefully through the IDL, you will find a couple of services more, which have not been mentioned, and are not part of the SAI specification. These services are there to support a client side abstraction layer, and their explanation will therefore be postponed to the next section.

5.11 Abstraction layer

In section 2.4 we state that one of our goals is to provide a system which reduces the burden of creating 3-D applications. A system which lifts the level of abstraction to a higher ground. In the light of this, we find that direct use of the SAI is too cumbersome for application developers, and therefore we need an abstraction layer on top of it.

It is the intention of the X3D specification to provide SAI language bindings for several programming languages. A language binding for C or C++ would bring us part of the way, but at the time of writing, such a language binding has not yet been specified. The only language bindings that has been proposed so far is one for the Java programming language and one for the ECMA scripting language.

Even if there was a language binding we would probably still need to add another layer on top of it, to achieve the required level of abstraction and accessibility for application developers. Since the language binding would encapsulate some of the gory details of using CORBA (or some alternative), it would have been fortunate to build our abstraction layer on top of it. Anyway, in the remainder of this section, we will explain which problems our abstraction layer needs to deal with, and how to deal with those problems.

It should be noted that even though we provide an abstraction layer, there is nothing which prevents an application from ignoring it and use the low-level SAI services instead. This is an important aspect, since otherwise we would loose the language independence property which CORBA gave us in the first place. Even though we might choose to provide our abstraction layer for programming languages other than C++, we should expect that CORBA will continue to support more languages. Never the less, if an application developer chooses to use plain SAI services, he will be forced to write a lot of protocol related code, which would not be necessary with the abstraction layer.

This client abstraction layer must be implemented as a stand-alone library with no required bindings to the server source code. This will free application developers from having to acquire and build the complete server source code, when all he/she wants, is to build an 3D-Console application.

Since the X3D specification is highly object oriented in its definition of node types, we find that an abstraction layer should reflect this by being object oriented itself. Therefore it shall provide classes for each individual node type, and arrange them in an inheritance hierarchy identical to that of the specification. Further more each class shall contain a member for each field of the corresponding node type with the same name.

Instances of these classes shall not in any way contain the state of the real node on the server. Instead they must act as proxy objects. Therefore the field members are not ordinary variables of the field value type. Instead

they must be variables of a special field type. The field type needs to be defined as a class which allows some kind of read and write operation on the fields. All that these operations must do, is to encode the arguments for CORBA, then call the corresponding SAI service, and finally decode the response and return it the the calling application code.

To make the API appear as natural as possible to the application developer we shall require that writing to a field is done with the standard assignment operator. Similarly a read operation must be done through the conversion operator which converts the special field type to the field value type. Doing this, will allow the developer to write something like `double r = sphere->radius;` and `sphere->radius = 7;` assuming that `sphere` is a reference to a `Sphere` node.

A slight problem with this, is that the usual semantics of variable assignment will not hold when the buffered mode[6] is in effect. Say, a field contains the value 2, then you assign 3 to it, but after this you will still read out the the old value, 2. Never the less, we will stick to this natural interface, but make sure that the documentation clearly describes the issue above.

Although it is not an easy thing to implement, we shall require that as long as an instance of a proxy object exists in the application, the actual node shall continue to exist on the server. Further more we must guarantee that no matter how many times or in how many ways an application acquires a proxy object for a specific node, it shall always be the same proxy object that is acquired. Or stated differently, there must never exist two or more proxy objects in an application that refers to the same actual node unless they belong to distinct sessions in the server.

Finally we shall require of our abstraction layer, that it makes the process of obtaining a session within the server very easy.

[6] *Buffered mode* means that the field write operations are not applied immediately. Instead they are buffered, and when the application commits the changes by leaving the buffered mode, all the changes will be applied atomically.

Chapter 6

Implementation

3D-Console has evolved into a relatively big project already, when it comes to the amount of source code that we have written, and this is even though only the basics of the system have been implemented at the time of this writing. The system is implemented in C++ and features no less than 232 source files, 60411 lines of code, and 689 classes.

We have implemented a lot of the basic functionality of 3D-Console. In particular we have provided functionality to an extent that has allowed us to write and test a number of sample applications against 3D-Console. The implementation of the sample applications will not be discussed in this chapter, but see chapter 7.

The current implementation covers a large part of the X3D specification. Included herein is a large part of the SAI specification. The SAI specification was implemented as a network transparent API through the utilization of CORBA.

To meet the requirements listed in section 3.8 we have implemented a number of extensions to the above. This includes extensions to the SAI that enables per application session management, strict resource accounting, access to client content and an improved performance of the API. We have also implemented an object oriented client side abstraction layer on top of the SAI, which closely resembles the node descriptions from the X3D specification.

There are also a number of elements from the X3D specification which have not been implemented yet. Examples are: Collision detection between viewer and geometry, prototypes, multi-texturing, sound, NURBS surfaces and 2-D geometric primitives. There are also a number of requirements not relating to the X3D specification which has not been implemented yet. Examples are: Linking of applications, multiple pointing devices, and gesture recognition.

In the following we will introduce and describe many different C++ classes, but to avoid cluttering up the text, we will not point out which source file the class is defined in each time it is mentioned. Instead we will conclude the chapter with a table that maps class names to files. So whenever the reader wishes to consult the implementation he/she should first look towards the end of this chapter.

6.1 Use of 3rd party libraries

In our implementation we utilize a number of 3rd party libraries, all licensed under the GNU GPL, LGPL[1] or compatible licenses except Expat which is licenced under the MIT license[2].

- w3c-libwww
- libpng, libjpeg, libungif
- FreeType
- SpiderMonkey (ECMA-script)
- Expat (XML-parser)
- OmniORB (CORBA)
- Simple DirectMedia Layer (SDL)

See appendix C for more details on the individual libraries.

Apart from using these libraries we do in two occasions derive code from other developers code. One is a utility to determining the smallest enclosing ball of a set of points in n-dimensional Euclidean space. It is based on a utility called Miniball [Gar99]. Our derived utility is `Archon::Utilities::MinimalBoundingBall<N>`.

[1] For information about the GNU GPL/LGPL see http://www.gnu.org/copyleft/gpl.html and http://www.gnu.org/copyleft/lgpl.html

[2] For information about the MIT licence see http://www.jclark.com/xml/copying.txt

The other occasion is a quartic equation solver which is taken from Graphic Gems V [Pae95]. Our derived utility is
`Archon::Math::quarticSolve`.
The quartic equation solver is currently used only to compute the intersection between a line and a torus when determining which scene object is indicated by the pointing device.

6.2 Overall structure

We have utilized C++ name spaces to help structure the code. Since we have chosen the name *Archon*[3] for the entire framework around 3D-Console we have confined all the code into a top-level name space called `Archon`. Apart from `Archon` there are a number of lower level name spaces which are important for an understanding of the overall structure of our code, and they are listed in table 6.1. The directory structure of the source tree is laid out to roughly match the name spaces. The precise relationship is listed in table 6.2.

Table 6.1: Name spaces

`Archon::Math`	Linear algebra + geometric formulas
`Archon::Utilities`	Various utilities
`Archon::Display`	Abstraction of rendering targets
`Archon::Render`	Abstraction of multi-pipe rendering
`Archon::X3D`	X3D core implementation
`Archon::X3D::SAI`	SAI service layer implementation
`Archon::X3D::Proxy`	Client side abstraction layer
`Archon::Console3d`	3D-Console implementation
`Archon::SaiTestApps`	Test application for 3D-Console

In appendix I we have included all the source code files residing in the `Archon::X3D` and `Archon::SaiTestApps` name spaces. This includes the name spaces `Archon::X3D::SAI` and `Archon::X3D::Proxy`. We have added a few selected files from the `Archon::Utilities` name space too. We have not included everything, since that alone would have taken up more than 1.000 pages. We believe the included source code covers the most important parts of our implementation.

[3] Hint: *"Warlord — thy progress is astounding"*

Table 6.2: Directory structure

`Archon::Math`	`archon/math`
`Archon::Utilities`	`archon/util`
`Archon::Display`	`archon/display`
`Archon::Render`	`archon/render`
`Archon::X3D`	`archon/x3d/server`
`Archon::X3D::SAI`	`archon/x3d/sai`
`Archon::X3D::Proxy`	`archon/x3d/proxy`
`Archon::Console3d`	`archon/x3d/3dconsole`
`Archon::SaiTestApps`	`archon/x3d/sai_test_apps`

6.3 On-line project resources

Should the reader wish to see the rest of the source code, it is all accessible through the Internet on our SourceForge home page. Our SourceForge home page is located on the following address:
`http://files3d.sourceforge.net`

The source code is browsable via the following address:
`http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/files3d`

A complete distribution package of all the source code (aka. a tar-ball) is available through our SourceForge project page on the following address:
`http://sourceforge.net/projects/files3d`

Since development is destined to continue, it is best if you locate and download the package named `Archon-0.1-src.tar.gz`.

All the embedded source code documentation (comments) is written in such a way that it is understood by Doxygen[4], and the resulting, automatically generated Doxygen documentation, is also available. It may be accessed on the following address:
`http://files3d.sourceforge.net/doxygen`

[4] *Doxygen* is a documentation system for C++, C, etc., that extract structural information and comments from the source code. With the extracted information Doxygen generates a set of web pages which helps you to read the comments and comprehend the structure.

6.4 Libraries

Some of the name spaces listed above have been arranged into individual libraries. This is done partly to divide the code into manageable pieces and partly to support reuse. Some libraries are more fundamental than others, and some libraries depend on other ones. This gives rise to an ordering determined by dependency. In the list below the first one, `Math`, is the most fundamental one, and thus the only one that itself does not depend on other libraries. The second one, `Utilities`, depends on `Math`, and in general a library in the list will depend on all the libraries above it. The libraries are listed in table 6.3.

Table 6.3: Libraries

Archon::Math	libarchonmath.so
Archon::Utilities	libarchonutil.so
Archon::Display	libarchondisplay.so
Archon::Render	libarchonrender.so
Archon::X3D	libarchonx3d.so
Archon::X3D::SAI	libarchonx3dsai.so
Archon::X3D::Proxy	libarchonx3dproxy.so

In section 5.11 we stated that the abstraction layer should be a stand-alone library with no required bindings to the server source code. Therefore the library `libarchonx3dproxy.so` does not depend on `libarchonx3d.so` nor on `libarchonx3dsai.so`. It does however require the more fundamental libraries `libarchonmath.so` and `libarchonutil.so`, but that is alright, since these libraries contain generic code that ought to be reused as much as possible.

6.5 Node and field types

A fundamental part of the X3D specification is the X3D node hierarchy. This inheritance hierarchy consists of the abstract ultimate ancestor node, `Node`, and a large number of descendant nodes such as `Transform`, `Cylinder` and `TouchSensor`. We need to be able to represent all these types of nodes in the server. There are two obvious alternative ways to do

this. One is to write a distinct C++ class for every kind of X3D node, the other alternative is to model all kinds of nodes with a single general C++ class. The latter alternative has the advantage of being less verbose, since a large number of similar class definitions can be omitted. Still, we have chosen the former alternative, because it provides better performance in the rendering code. The reason is that it is a lot faster to read the value of a real class member than to acquire the value of a field in a generalized class where fields would reside in some sort of list or table. It should be noted however, that if we ever dismiss our own rendering code in favor of a 3rd party product such as OpenSceneGraph then the decision would tilt towards the one-class representation.

In the X3D specification all abstract node types have an `X3D` prefix in their name. Since we define all the node classes in an `X3D` name space we have omitted the X3D prefixes from the node type names.

Now, let us take a look at how the `Transform` node is actually implemented. The formal definition from the specification was:

Listing 6.1: Formal definition of `Transform`

```
Transform : X3DGroupingNode, X3DBoundedObject {
  MFNode      [in]     addChildren                   [X3DChildNode]
  MFNode      [in]     removeChildren                [X3DChildNode]
  SFVec3f     [in,out] center           0 0 0        (-∞,∞)
  MFNode      [in,out] children         []           [X3DChildNode]
  SFRotation  [in,out] rotation         0 0 1 0      [-1,1] or (-∞,∞)
  SFVec3f     [in,out] scale            1 1 1        (0,∞)
  SFRotation  [in,out] scaleOrientation 0 0 1 0      [-1,1] or (-∞,∞)
  SFVec3f     [in,out] translation      0 0 0        (-∞,∞)
  SFVec3f     []       bboxCenter       0 0 0        (-∞,∞)
  SFVec3f     []       bboxSize         -1 -1 -1     (0,∞) or -1 -1 -1
}
```

Which states that the `Transform` node derives from the abstract node types `X3DGroupingNode` and `X3DBoundedObject`. We will find this exact derivation repeated in the implementation of the `Transform` node:

Listing 6.2: Implementation of `Transform`

```
struct Transform: GroupingNode, BoundedObject
{
  static Ref<Node> instantiate(BackRef<ExecutionContext> c)
  {
    return new Transform(c);
```

```
  }

  static const NodeType *type;
  const NodeType *getType() const { return type; }

  Transform(BackRef<ExecutionContext> c):
    Node(c),
    center(0, 0, 0),
    rotation(AxisRot3(Vector3(0, 0, 1), 0)),
    scale(1, 1, 1),
    scaleOrientation(AxisRot3(Vector3(0, 0, 1), 0)),
    translation(0, 0, 0) {}

  // ...

private:
  Vector3 center;
  AxisRot3 rotation;
  Vector3 scale;
  AxisRot3 scaleOrientation;
  Vector3 translation;

  EventSource centerChanged;
  EventSource rotationChanged;
  EventSource scaleChanged;
  EventSource scaleOrientationChanged;
  EventSource translationChanged;

  Time centerStamp;
  Time rotationStamp;
  Time scaleStamp;
  Time scaleOrientationStamp;
  Time translationStamp;

  // ...
};
```

Apart from the derivation, we see that five of the "input output" fields have a corresponding variable in the C++ class. All the other fields in the formal definition are not to be found in the C++ class, and this is because they are inherited from the sub-classes. We will take a look at the abstract `GroupingNode` in a minute, but first we note that every implemented "input output" field also has a corresponding `EventSource` and `Time` variable. The `EventSource` variable is a collection of route objects which we will get back to. The `Time` variable is a time stamp that records the time of the last value assignment to the corresponding field. The time stamps are a part of the event cascade implementation, and are a requirement in the X3D specification. Finally we note that the default values from the formal definition are mirrored in the variable initialization of the `Transform`-constructor.

Let us now turn to the `GroupingNode` from which `Transform` derived several fields. The formal definition is:

Listing 6.3: Formal definition of `GroupingNode`

```
X3DGroupingNode : X3DChildNode {
  MFNode [in]     addChildren
  MFNode [in]     removeChildren
  MFNode [in,out] children        [] [X3DChildNode]
}
```

As we expected, we see some of the `Transform` fields repeated. By inspecting the implementing C++ class we see that the three fields are defined within it. Ergo they are not derived from the `ChildNode`. The implementing C++ class follows:

Listing 6.4: Implementation of `GroupingNode`

```
struct GroupingNode: virtual ChildNode
{
  static const NodeType *type;
  static const NodeSequenceFieldBase *childrenField;

  // ...

protected:
  GroupingNode() {}

private:
  vector<Ref<ChildNode> > children;

  EventSource childrenChanged;

  Time childrenStamp;

  void addChildren(const vector<Ref<ChildNode> > &);
  void removeChildren(const vector<Ref<ChildNode> > &);

  // ...
};
```

From this listing we also see that "input only" fields are implemented as class methods rather than variables. This is possible because no one attempts to read the value of an "input only" field. It is fortunate to do it like this, since it allows much more flexibility in how the node reacts to an input event. In the case of `addChildren` and `removeChildren` the flexibility is in fact necessary, since they have to manipulate the value of another field, children. Finally we note that the GroupingNode class derives virtually from ChildNode. It does so, because ChildNode has to be a virtual base class, and this in turn is because some node types (e.g. the Collision node) indirectly derives multiple times from ChildNode. If ChildNode had not been a virtual base class its fields would occur twice in nodes such as Collision.

As the reader may have seen, the process of implementing node types seems relatively straight forward. Of course, there is another aspect of the implementation that we have not talked about. So far we have only talked about how the fields are implemented. Some node types are like "black boxes". You put something in at one end, and you get something out at the other. There are a number of interpolator nodes (e.g. the OrientationInterpolator) that act like this. Other node types have a graphical representation. Those nodes all derive from the abstract GeometryNode. In GeometryNode a virtual method is defined. This method will be called by the main rendering loop whenever the the graphical node needs to be rendered, and it is the duty of the specific descendant to override the virtual method and perform the required rendering.

Let us conclude this section with one last aspect of field implementation that we have not mentioned yet — the choice of field types. The question is how we represent the values of the various field types defined in the X3D specification. We have already seen a number of examples in the Transform and GroupingNode classes, but in table 6.4 they are all listed.

Note that, here we make use of several classes from both the Math and the Utilities libraries. Each of these classes are written such that they have standard copy semantics (i.e. values of these types can be copied through assignment and argument passing, just like the fundamental types). More over, they are implemented as general purpose utility classes with a lot of nifty methods that make them potent for use in many other contexts of the Archon framework. The various vector classes are mathematical vectors with all the expected operators defined. They can be added, multiplied and so on. AxisRot3 is an axis rotation that encapsulates a unit axis vector and an angle. Time and Image are self explaining. Ref<...> is an implementation of a "smart pointer". This "smart pointer" plays a fundamental role in the memory management of 3D-Console and will therefore be explained in more detail in section 6.10.

Table 6.4: Field types

SFBool MFBool	bool std::vector<bool>
SFColor MFColor	Math::Vector3 std::vector<Math::Vector3>
SFColorRGBA MFColorRGBA	Math::Vector4 std::vector<Math::Vector4>
SFDouble MFDouble	double std::vector<double>
SFFloat MFFloat	double std::vector<double>
SFImage MFImage	Utilities::Image std::vector<Utilities::Image>
SFInt32 MFInt32	int std::vector<int>
SFNode MFNode	Utilities::Ref<...> std::vector<Utilities::Ref<...> >
SFRotation MFRotation	Math::AxisRot3 std::vector<Math::AxisRot3>
SFString MFString	std::string std::vector<std::string>
SFTime MFTime	Utilities::Time std::vector<Utilities::Time>
SFVec2d MFVec2d	Math::Vector2 std::vector<Math::Vector2>
SFVec2f MFVec2f	Math::Vector2 std::vector<Math::Vector2>
SFVec3d MFVec3d	Math::Vector3 std::vector<Math::Vector3>
SFVec3f MFVec3f	Math::Vector3 std::vector<Math::Vector3>

6.6 X3D components

Due to the large number of node classes, we have chosen to divide them into a number of files. The X3D specification divides node types into a number of components, it was therefore natural for us to choose the same devision criterion. This gave rise to the files listed in table 6.5. All these files are located in `archon/x3d/server`. Nodes have been rearranged slightly to enable a strict ordering that satisfies both the inheritance and containment dependencies between nodes. A strict ordering is necessary due to the way header inclusion works in C and C++.

Table 6.5: X3D components

`core.H`	Core component
`time.H`	Time component
`network.H`	Networking component
`group.H`	Aggregation and transformation component
`geoprop.H`	Geometric properties component
`texture.H`	Texturing component
`appear.H`	Appearance component
`geometry.H`	Geometry component
`text.H`	Text component
`light.H`	Lighting component
`interpolation.H`	Interpolation component
`navigate.H`	Navigation component
`mouse.H`	Pointing device sensor component
`script.H`	Scripting component
`aux.H`	Auxiliary component (additions to X3D)

Not all X3D node types have been implemented at the point of writing. Table 6.6 lists the nodes that enjoys at least partial support in 3D-Console. There is a remark on every node type explaining to which extent it is supported. Note that all the abstract node types are not listed in this table.

Table 6.6: List of supported X3D node types, and a remark on the level of conformance with the X3D specification.

Node type	**Remark**
TimeSensor	The “pausing” feature is not supported
Appearance	Full conformance
Background	Is not rendered
Box	Full conformance
Color	Full conformance
ColorInterpolator	Full conformance
ColorRGBA	Full conformance
Cone	Full conformance
Coordinate	Full conformance
CoordinateInterpolator	Full conformance
CoordinateInterpolator2D	Full conformance
Cylinder	Full conformance
DirectionalLight	Full conformance
ElevationGrid	Full conformance
Fog	Is not rendered
FontStyle	Full conformance
Group	Full conformance
ImageTexture	Full conformance
IndexedFaceSet	Full conformance
IndexedLineSet	Full conformance
Inline	Full conformance
Material	Full conformance
MovieTexture	Is not rendered
Normal	Full conformance
NormalInterpolator	Full conformance
OrientationInterpolator	Full conformance
PixelTexture	Is not rendered
PlaneSensor	Works incorrectly
PointLight	Full conformance
PointSet	Full conformance
PositionInterpolator	Full conformance
PositionInterpolator2D	Full conformance
ScalarInterpolator	Full conformance
Script	Partially supported
Shape	Full conformance
Sphere	Full conformance
SpotLight	Full conformance
Switch	Full conformance
Text	Vertical layout is not supported
TextureCoordinate	Full conformance
TextureTransform	Full conformance
TouchSensor	hitNormal and hitTexCoord events are not generated
Transform	Full conformance
Viewpoint	Full conformance

6.7 Run-time type information

The definition of every node type as a separate C++ class gives us compile-time type information, but as described in section 5.3 there must be a way to extract, from a node instance, all the details of the corresponding node type at run-time. It must be implemented in a way which eliminates the need of knowledge about the individual node classes in code that deal only with field input and output. In these parts of the code we will expect that all references to node instances are typed as references to the ultimate ancestor of node types, `Node`, and this is indeed the case. The `Node` class is described further in section **??**.

To sum up, we must be able to fetch the type name from a node instance as a string. The type name will be the exact name of the corresponding class. Next, we must be able to ask which other node types that this one derives from. We must further be able to lookup a certain field by name, and as an alternative we must be able to extract the complete list of fields for the node type. For each field we must be able to ask whether it has the input and/or the output capability, and we must be able to extract the field type.

We found that the concepts of the node type, field and field type were best expressed as classes themselves. Consequently we have implemented a class `NodeType` to represent an arbitrary node type, a class `FieldBase` to represent an arbitrary field, and a class `FieldType` to represent an arbitrary field type. As the name suggests `FieldBase` is just an abstract base class for a variety of different field classes. The complete field inheritance graph is illustrated in figure 6.1. Class names in parenthesis indicate multiple inheritance. This rather large number of field classes is required because the fundamental field operations such as read and write depend on the structure of the field value type.

The various different static fields are used for normal node types. The custom fields are used in special nodes where the set of fields is not known at compile time. Examples of this is the `Script` node as well as prototypes[5]. We have provided a common base class `CustomFieldNode` for these special node types that handles the set of custom fields in a way that makes it

[5] X3D prototypes are not supported by 3D-Console yet, but most of the required functionality is already implemented. This includes the custom field classes and the base class `CustomFieldNode`.

Figure 6.1: Field class inheritance graph

```
FieldBase
|
|-- SequenceFieldBase
|
|-- NodeFieldBase
|   |
|   |-- NodeSequenceFieldBase                         (SequenceFieldBase)
|
|-- StaticField<NodeType, FieldType>
|   |
|   |-- SimpleField<NodeType, FieldType>
|   |
|   |-- SimpleSequenceField<NodeType, FieldType>      (SequenceFieldBase)
|   |
|   |-- NodeField<NodeType, ChildNodeType>            (NodeFieldBase)
|   |
|   |-- NodeSequenceField<NodeType, ChildNodeType>    (NodeSequenceFieldBase)
|
|-- CustomFieldBase
    |
    |-- SimpleCustomField
    |
    |-- SimpleSeqCustomField                          (SequenceFieldBase)
    |
    |-- NodeCustomField                               (NodeFieldBase)
    |
    |-- NodeSequenceCustomField                       (NodeSequenceFieldBase)
```

functionally equivalent to ordinary node types with static fields. This class makes the addition of prototype support relatively straight forward.

All the runtime type information for the normal node types need to be setup during initialization of the system. We have included the snippet of code that initializes the runtime type information for the `Transform` class. These initializations are always placed in the components files with the "`.C`"-suffix, and otherwise corresponding the the files listed in table 6.5. The initialization code follows:

Listing 6.5: Run-time type initialization of `Transform`

```
void initializeGroupComponent()
{
  vector<const FieldBase *> fields;

  // ...

  fields.push_back(newExposedField("center", SFVec3f::type,
                                   &Transform::center,
                                   &Transform::centerChanged,
                                   &Transform::centerStamp));
  fields.push_back(newExposedField("rotation", SFRotation::type,
                                   &Transform::rotation,
                                   &Transform::rotationChanged,
                                   &Transform::rotationStamp));
  fields.push_back(newExposedField("scale", SFVec3f::type,
                                   &Transform::scale,
                                   &Transform::scaleChanged,
                                   &Transform::scaleStamp));
  fields.push_back(newExposedField("scaleOrientation", SFRotation::type,
                                   &Transform::scaleOrientation,
                                   &Transform::scaleOrientationChanged,
                                   &Transform::scaleOrientationStamp));
  fields.push_back(newExposedField("translation", SFVec3f::type,
                                   &Transform::translation,
                                   &Transform::translationChanged,
                                   &Transform::translationStamp));
  Transform::type =
    NodeType::newConcrete("Transform", Transform::instantiate,
                          &fields, GroupingNode::type, BoundedObject::type);

  // ...
}
```

In the function name `newExposedField` of this listing the phrase *exposed field* is synonymous to "input output" field. This name stems from an early release of the X3D specification. What we see then is that a list of fields is assembled. For each new field we specify its name and type. Further more, since this is an "input output" field there must be

a reference to the actual variable member of the `Transform` class that implements the field. In the first case this is `Transform::center` (you may double check with listing 6.2). For all "input output" fields the collection of routes `Transform::centerChanged` and the field time stamp `Transform::centerStamp` must also be accessible through the runtime-type system. After constructing the list of fields, the new node type object can be created. This is done in the last statement above. We notice that a reference to an `instantiate` method is passed as argument, This is done to allow clueless code to instantiate nodes through the runtime type system. We notice further that the node type objects of the `GroupingNode` and the `BoundedObject` are passed, and this is because `Transform` derives from them.

Let us now take a look at the `NodeType` class which represented the runtime information of an arbitrary node type. Many of the methods and other details have been scissored out (indicated by `// ...`), since they are not described here:

Listing 6.6: Implementation of `NodeType`

```
struct NodeType
{
  static unsigned long getTotalNumber();
  static const NodeType *get(unsigned long index);

  static const NodeType *lookup(const string &nodeTypeName);

  Ref<Node> instantiate(BackRef<ExecutionContext> context) const;

  string getName() const { return name; }

  bool isAbstract() const { return !instanceMaker; }
  bool isVirtualBase() const { return virtualBase; }


  bool isDerivedFrom(const NodeType *baseType) const;

  typedef vector<const NodeType *>::const_iterator iterator;
  iterator derivedFromBegin() const;
  iterator derivedFromEnd() const;


  const FieldBase *lookupField(string name) const;

  unsigned long getNumberOfImmediateFields() const;

  const FieldBase *getImmediateField(unsigned long fieldIndex) const;

  const FieldBase *lookupImmediateField(string name) const;
```

```
  // ...

private:
  NodeType(string name, bool virtualBase,
           const vector<const NodeType *> *derivedFrom,
           const vector<const FieldBase *> *fields,
           Ref<Node> (*instanceMaker)(BackRef<ExecutionContext>));

  // ...
};
```

As expected we find the method `getName` which retrieves the name of the type. We find `isDerivedFrom` which determines if this type derives from some other type. We find `lookupField` which looks up a field by name, and finally we find `getNumberOfImmediateFields` and `getImmediateField` which retrieves the list of fields for this type. Note that these last methods only deliver the fields that are defined in this type, not the derived fields. To get all the fields, one must use the methods `derivedFromBegin` and `derivedFromEnd` to recursively iterate through all the super types.

The `instantiate` method can be used in concert with the `lookup` method to get from a type name to a node instance of that type at run-time. This is used in the X3D parsers, since here every node instance is encountered as a textual type name. The `instantiate` method may also be used together with the `get` method to get from the node type index to a node instance of the corresponding type. This is used by the client API (or SAI) when the application requests a new node instance. In the client API we do not use type names for efficiency reasons. Instead type indexes are encoded into the node instance identifiers as defined in the IDL description of the SAI.

The `getTotalNumber` method is used when all the available node types are needed. This is the case in the automatic generation of node classes, in the client side abstraction layer. More on this later.

If we take a look at listing 6.2 again, we see that it has a method `getType` for retrieving the run-time type object. This method overrides a virtual method of the same name in the `Node` class (see listing 6.7). Since every concrete node class overrides this method to return the correct `NodeType` object, it is possible to ask any node instance for its type object.

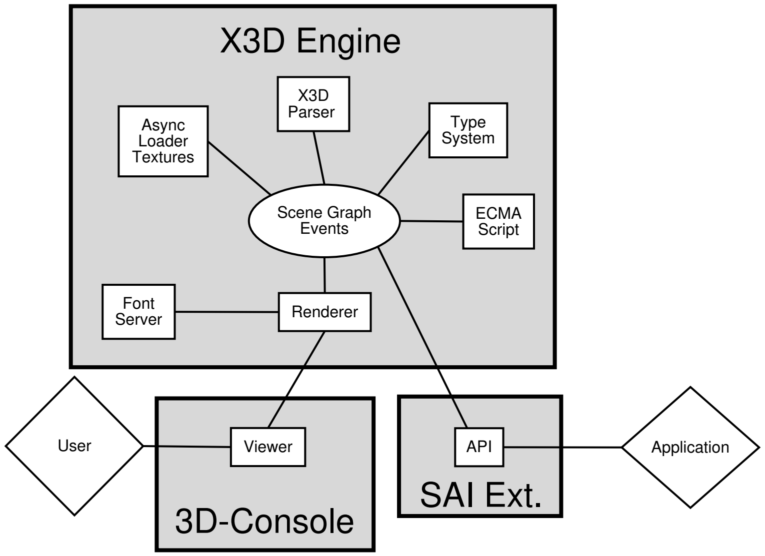

Figure 6.2: Overview of system modules

6.8 Server modules

We will now take a few steps backward, and try to give the reader an overview of the various modules that comprise the server, and how they connect to each other. The central part, of course, is the scene graph. Every other part of the system does in some way or another interact with the scene graph. See fig. 6.2.

6.8.1 The scene graph

The scene graph consists of the root scene (or the base scene if you like) with all of the nodes and routes in it. A number of sub-scenes (probably mostly application scenes) will be linked into the root scene, and are also part of the scene graph. Since the scene graph is a very dynamic data structure, which may be traversed from different corners and modified by numerous parties simultaneously, the coupling between the different objects becomes rather complex. It should be noted that the achievement of the required level of flexibility, whilst keeping good robustness, is very far from trivial, and especially when considering multi threading all along. Here we should give a big kudos to the concept of "smart pointers" which to a huge extent is the reason that it was possible[6]. "smart pointers" will be covered more thoroughly in section 6.10 below.

In the X3D specification the phrase "behavioral graph" is used to describe the graph spanned by the route connections. Therefore the routes are not considered as part of the scene graph in the specification. The two graphs are considered as distinct concepts, even though they are closely related and combined. Never the less, when we speak about the scene graph in this chapter we refer to the routes, and thus to the behavioral graph, too.

6.8.2 Asynchronous content loading

Before we go on and look at some code in section 6.9, we will introduce you to the rest of the modules of the server. The run-time type system has already been covered, so we will skip to the asynchronous content loader. As mentioned in section 5.4 the X3D engine must automatically initiate a download of an indicated resource, when a new URI is assigned to a field such as the `url` field of the `ImageTexture` node. But, as if that was not enough, the specification also requires that a URI field can contain multiple alternative URIs, and in this case the engine must attempt to load them one by one, until successful, or every alternative has been tried.

All this functionality is provided by two general purpose utilities named `Utilities::Web::Client` and `Utilities::Job::Queue`, as well as a custom class `X3D::Loader` that wraps it all up into a neat and encapsulated module of 3D-Console. The triggering of the download is possible because all the URI fields of the various nodes are derived from one common abstract node type, `UrlObject`. The sole purpose of the node type is to provide the URI field and to communicate with the loader module. The implementing class has many methods to handle the communication, which is complex enough to call it a protocol. Finally it should be noted that `Utilities::Web::Client` is just a wrapper around the `w3c-libwww` library from the World Wide Web Consortium, but a wrapper that makes it several order of magnitude simpler to use.

[6] An automated garbage collector with multi threading support would have done the job too.

6.8.3 Parsing X3D files

Parsing of X3D files constitute another module in 3D-Console. At the time of writing, only the XML parser operational. The VRML parser has been partially implemented, but is currently non-functional. The XML parser is accessible through the `X3D::XML::parse` function. This function is automatically invoked for XML encoded X3D files retrieved by the `X3D::Loader` class. This may be the root scene, or it may be any sub-scene inlined into the root scene. Since the `Inline` node is a derivative of `UrlObject`, the download and parsing of the XML files for sub-scenes are carried out asynchronously with respect to the rendering loop. This in turn means that the sub-scenes will be available for rendering in an incremental fashion. This is nice because it allows the user to see the root scene early. It is also a requirement in the X3D specification.

The parser makes extensive use of the run-time type system described in section 6.7. In fact the type system is particularly well suited to drive an XML parser. The reason is that the coarse structure of the syntax is handled by the XML parser library (see section 6.1) while the more detailed structure is governed by the run-time type information. In short, for each XML element that the parser encounters in the X3D file, an instance of the corresponding node type is created. Then, for each XML attribute and value, the corresponding field is looked up in the node type object. Then the value is assigned to the field by precisely the same means as an event would be applied to that field.

6.8.4 The renderer

The renderer is implemented by the class `X3D::Viewer` and it comprises yet another module in 3D-Console. In this class there is a function that traverses the scene graph every frame to do the rendering. Here the concrete node class must be determined for each child node that is encountered during the traversal. As we explained in section 6.5, we need to access the field variables directly through the C++ class for the node type for sake of efficiency (i.e. via compile time type information). As a result of this the renderer must know about many of the concrete node types of X3D.

For the sake of encapsulation, as mentioned earlier, the actual rendering of geometry nodes are performed by a common method in the node classes. For efficiency reasons the rendering sub-system does not call the `render` method of a geometry node every frame. The method is called on the first frame and then whenever the rendering sub-system detects that a change has occurred to one or more fields in the geometry node. The rendering sub-system uses OpenGl's support for call-lists to achieve this. An obvious improvement to this kind of optimization would be to dynamically detect sub sections of the scene graph that appears to be static, and then use more call-lists to capture larger parts of the scene graph.

Anyway we have been considering seriously to dismiss our rendering in favor of a 3rd party product such as OpenSceneGraph, see section 9.3. This idea was primarily a result of considerations of how to support multiple viewers. Our goal is to separate the viewer from the server, and put a CORBA based network layer in between. When the viewer is separated out, it will be much easier to employ a 3rd party rendering library.

6.8.5 Font server

Display of text is important to 3D-Console as described in section 3.8.1. X3D offers this through the `Text` node, but to implement the text node we need to be able to render text, character by character. In other words we need to be able to access font information. As discussed in lesson 13-15 on the Neon Helium web site (NeHe) [Pro], there are three well-known ways to render text: with bit-mapped fonts, with outline fonts, or with texture mapped fonts. OpenGL directly supports bit-mapped fonts, but these fonts are not really three dimensional, they appear as billboards. Outline fonts are nice since they are inherently 3-D, they appear just like bent neon tubes.

Anyway, the last alternative, texture mapped fonts, seems to fit best into the context of 3D-Console. There are several reasons for this. First of all, text in 3D-Console has to be functional and not necessarily pretty, that is, the text must be easy to read even when there is a lot of it. Next, we want text to be truly three dimensional, since how else could we model a piece of paper with text written on it, see section 7, TextViewer. The last reason for choosing texture mapped fonts is that it seems to be the only alternative that satisfies all the requirements of the X3D specification.

In 3D-Console the `Text` node uses a font server to deliver the raw glyph images and metrics. The font server is implemented as `X3D::FontServer`,

and uses the open FreeType library to load the fonts from font files. It supports a wide variety of font formats, in particular it supports the TrueType format.

The `Text` node builds strings of text by rendering small texture mapped polygons side by side, one for each character. The size of the polygons, the spacing between them, and their vertical adjustment with respect to the base line is determined from the metrics of the individual glyphs. All the glyph images are automatically compiled into a single texture by `X3D::FontServer`. The font server also constructs the table that maps character codes to texture coordinates and glyph metrics.

6.8.6 Scripting

3D-Console features preliminary support for ECMA scripting. That is, it is possible to define a `Script` node with a custom set of fields, and have the internals of the node be handled by a script written in the ECMA scripting language. The scripting feature is implemented in `X3D::ECMA::Engine`.

6.9 Server internals

Every part of the X3D engine in 3D-Console is expressed as a C++ class. An implication of this, is that a program that utilizes our X3D library may instantiate several X3D engines within the same address space. This is a design quality in itself which most modern libraries possess, but the primary reason for doing it, is because the code is inherently multi-threaded. When working under multi-threading conditions, global variables are prohibited. See the next section for more details on multi-threading in 3D-Console.

Figure 6.3 gives a quick overview of which main objects the server consists of. Ignore the dashed arrows for now. As we see there is a top-level class called `Server`. Instances of `Server` represent a complete X3D engine. The server contains a scene graph in the form of a reference to the root scene object, but the server also contains a `Loader` object (see section 6.8.2) and a `Listener` object. The `Listener` object is the entrance to the client API (aka. the SAI), and will be discussed further below in section 6.12.

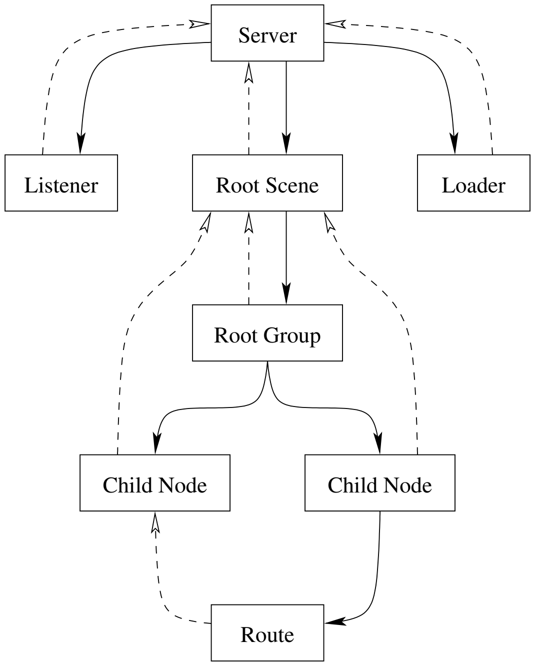

Figure 6.3: Server objects. The server consists of a number of objects. The objects are interconnected in relatively complex way, that is governed by containment relations. When two objects are connected by an arrow, the object appearing above is always a container of the object appearing below. The solid arrows indicate forward references, and always refers to the contained object from the container. The dashed arrows indicate backward references. The distinction between forward and backward references is necessary to prevent cyclic references which will lead to memory leak.

If we step into the scene graph we see that the root scene contains a root group. In fact, any ExecutionContext contains a root group, no matter if it is a root scene, an application scene, or a prototype. The root group is always a `Group` node, and can thus contain any number of `ChildNodes`. In fig. 6.3 we have drawn two of them. Note that the child nodes are in fact some concrete node types derived from the abstract `ChildNode`. The last type of object in the figure is a route. A route object is contained by the node from which the route originates, and not the node where it ends. In listing 6.2 we saw a number of `EventSource` variables in the `Transform` class. It is in these variables that the route references are stored.

6.10 The smart pointer

We have in several previous occasions mentioned that "smart pointers" play an essential role in the memory management of 3D-Console. The cool thing about smart pointers is that they provide implicit disposal of the referenced objects. The disposal is done at the exact time that the last reference to the object is destroyed. For this to work, the referenced object must hold a reference counter. Fortunately the counter variable and the manipulation of it, is all encapsulated into a base class called `RefObjectBase`. The smart pointers themselves are modeled by a C++ class template called `Ref<T>`. Now, any instance of a class `T` derived from `RefObjectBase` may be referenced with smart pointers of the type `Ref<T>`.

To allow multiple threads to handle references to the same object (from now on *references* will be used as a synonym for smart pointers) the counter variable in `RefObjectBase` must be protected from concurrent updates. This is handled smoothly with the `Mutex` class which is explained in the next section.

In fig. 6.3 there was a number of dashed arrows. These arrows do not represent ordinary references (smart pointers), instead they represent a special derivative of a reference, namely a backward reference. The concept of backward references is necessary to prevent cycles in the ordinary (or forward) references. Cycles are a serious problem, since they almost always result in memory leak[7]. Just like ordinary references, the backward references

[7]Memory leak is a consequence of cyclic reference since if A refers to B and B refers to A, then both A and B will stay alive even when all other references to the two objects are dropped.

needs their own counter variable in the referenced object. For this reason we derive a new base class BackRefObjectBase from RefObjectBase that encapsulates the backward reference count too. This gives us a base class for objects which need to be used with both forward and backward references.

In 3D-Console server many objects are derived from BackRefObjectBase. This is because we need forward reference from containers to contained objects as well as backward references from contained objects to containers. The forward references are necessary for the traversal of the scene graph during rendering. The backward references have many important uses. One example is that relative URIs in URI fields need to be resolved against the base URIs of the ExecutionContext that owns the node. Further more the Loader must be contacted, but this requires access to the Server.

6.11 Multi threading and re-entrance

In C++, multi-threading is not immediately supported. We have chosen to use the pthreads library to deliver the native thread support. To be able to work seamlessly with threads in C++ we have implemented an advanced wrapper class Utilities::Thread which encapsulates the low-level thread interface of pthreads. The most important feature of this wrapper class is its ability to terminate treads without memory leak through proper stack unwinding. This is a real hard problem, and is not seen in any other thread wrapper library that the authors know of (and that is more than a few). For an explanation of the difficulty faced, see the pthread man pages on pthread_cancel.

For multi-threading to be useful we need a synchronization facility. Luckily pthreads supplies a mutex (mutual exclusion) and a condition variable. Again we have wrapped these features into C++ classes, and they are called Utilities::Mutex and Utilities::Condition. With the mutex and condition variables we can easily construct full blown monitors out of C++ classes. For an explanation of mutexes, conditions and monitors, see [M.82]. Threads and mutexes are used all over our code. Conditions are a little less frequent. A good example of the use of the Condition class is the job queue, Utilities::Job::Queue.

So how do we employ these synchronization primitives? Let us use the Node class as an example. The Node class was the root of the node type inheritance graph and holds all the methods which form the top-level interface to manipulate field values (the "field services" in SAI terminology). The SAI specification requires, among others, the following methods (see the X3D specification, part 2, 6.7 Field services): get, set, getAt, setAt, add, remove. The getAt and setAt methods sets and gets values at specific indexes in a multi-valued field such as MFDouble. The add and remove methods add and remove values in a multi-valued field.

In the following listing many of the methods has been pruned away since they are irrelevant in this context:

Listing 6.7: Declaration of the Node class

```
struct Node: virtual BackRefObjectBase
{
  static const NodeType *type;
  virtual const NodeType *getType() const = 0;

  void realize();

  bool isRealized() const;

  Ref<ValueBase> get(const FieldBase *) const throw(AccessException);

  unsigned getSize(const SequenceFieldBase *) const
    throw(AccessException);

  Ref<ValueBase> getAt(const SequenceFieldBase *, unsigned index) const
    throw(AccessException, RangeException);

  void handleEvent(const FieldBase *, const Event *)
    throw(AccessException);

  void setAt(const SequenceFieldBase *, Ref<const ValueBase>,
             unsigned index, Time) throw(AccessException, RangeException);

  void add(const SequenceFieldBase *, Ref<const ValueBase>, Time)
    throw(AccessException);

  void remove(const SequenceFieldBase *, Ref<const ValueBase>, Time)
    throw(AccessException);


  const BackRef<ExecutionContext> context;

  // ...

protected:
  Node(BackRef<ExecutionContext> context);

  bool isRealizedNoLock() const { return realized; }

  const Mutex mutex;
```

```
  // ...

private:

  // ...
};
```

We find all the mentioned field service methods except `set` but that is because it is called `handleEvent`. We also see that it declares a `Mutex` variable with the name `mutex`. So let us see how the mutex is used in the implementation of the `add` method (the implementation of the other field services are almost identical):

Listing 6.8: Implementation of the `Node::add` field service

```
void Node::add(const SequenceFieldBase *f, Ref<const ValueBase> v, Time t)
  throw(AccessException)
{
  if(const NodeValue *w = dynamic_cast<const NodeValue *>(v.get()))
  {
    if(!w->value)
      ARCHON_THROW1(ArgumentException, "add: Node was null");
    if(w->value->context != context)
      ARCHON_THROW1(ArgumentException, "add: Node has wrong context");
    w->value->realize();
  }

  {
    Mutex::Lock l(mutex);
    if(!realized)
    {
      if(f->getIsEventTarget() != f->getIsEventSource())
        ARCHON_THROW(AccessException);
      f->add(this, v.get(), t, false);
      return;
    }
  }
  try
  {
    if(!f->getIsEventTarget() || !f->getIsEventSource())
      ARCHON_THROW(AccessException);
    Mutex::Lock l(context->server->scenegraphMutex);
    f->add(this, v.get(), t, true);
  }
  catch(ForwardDestroyedException &)
  {
    ARCHON_THROW(AccessException);
  }
}
```

to understand this code we must first know about the concept of node realization in X3D. A node in X3D has a life cycle consisting of 4 phases. The phase that a node is in, has an impact on whether its fields may be read from and/or written to. See table 6.7. In 3D-Console only the two middle phase have importance, therefore we say that a node gets realized in the transition from the setup phase to the realized phase. Realization happens when the node gets added as a child of another node or when a route to it is created.

Table 6.7: Permitted field access capabilities during the node life cycle. This is an exact copy of Table 4.2 in part 2 of the X3D specification.

Field type	Creation	Setup	Realized	Disposed
initializeOnly field	None	readable/writable	None	None
inputOnly field	None	None	writable	None
outputOnly field	None	None	readable	None
inputOutput field	None	readable/writable	readable/writable	None

In the case of the `add` method in listing 6.8 we are adding a value to a field. If the value is a node reference, then we are adding one node as a child of another, consequently the argument node must be realized during the addition, and that is what happens in the last statement in the first if-construct.

The next thing that happens in listing 6.8 is that we check to see if the node whose field we are writing to is realized, and this is where it gets tricky. If the node is realized, then the rendering engine may access it. The rendering engine will access nodes without locking the individual node mutexes for efficiency reasons. The rendering engine instead locks a "global" scene graph mutex when it is accessing the scene graph. If, on the other hand, the node is not realized yet, then the rendering engine will not attempt to access it (the node cannot be in the scene graph if it is not realized), and in this case it is enough to lock the mutex in the node.

So, we must lock the mutex before accessing any non-constant class members. Then we check to see if the node is realized, if it is not, then we check to see if the field may be updated (according to table 6.7). If it is, then we dispatch the field service to the field object itself while holding the node mutex locked.

If, on the other hand, the node was realized, then we have to lock the "global" scene graph mutex instead of the node mutex, but again we check the access type, and dispatch the service to the field object. Note that the "global" scene graph mutex is accessed through backward references, and therefore this is another good example of why backward references are required (see section 6.10).

Finally note that there are two different methods in the `Node` class for asking whether it is realized: `isRealized` and `isRealizedNoLock`. In 3D-Console we generally use the `NoLock`-suffix on methods that are fast but not thread safe or re-entrant because they access class variables without locking the mutex. In general, methods without the `NoLock`-suffix are thread safe and re-entrant. Note that there is an alternative to using suffixes to distinguish between thread safe and unsafe methods, and that is the notion of *volatile* pointers/references, but the authors were not aware of that until after the bulk of the code was written.

6.12 The client API

The SAI specifies a fair number of service calls that can be issued against the server by an application. The functionality which the service calls represent, is in general not directly available in the server. Consequently we need to attach a layer to the server, that can handle the SAI service calls. We call this "the SAI service layer" and sometimes just "the SAI". The implementation of the SAI service layer is confined to the `Archon::X3D::SAI` name space and is a library by itself.

We cannot assume that applications are well behaved, since they are external entities which at least in principle can connect to the server from all over the world. Instead we have to expect that applications are harmful and destructive. This mandates that the implementation of the SAI service layer must act as a "fire wall" between the server and the applications. By acting as a fire wall we mean that every service call is checked for having valid arguments and being valid with respect to the state of the server. Any service call that fails this check must be rejected.

Finally the SAI service layer is responsible for establishing a session for each application. The session serves both as a resource account and as a way to identify which applications that issue which service calls.

6.12.1 Value conversion

A large part of the job in the SAI service layer relates to the conversion of values between internal and external (CORBA) representation. To handle this as generically as possible, we have implemented a so-called `Porter` class. The name is chosen because it handles both import of external values and export to external values. Since both the internal and the external value representation is polymorphic we can handle the conversion with two simple methods, `im` and `ex`. The declaration of the `Porter` class follows:

Listing 6.9: Implementation of value import/export in SAI

```
struct Porter
{
  virtual unsigned long node2id(Ref<Node>) = 0;

  virtual Ref<Node> id2node(unsigned long)
    throw(IllegalNodeIdException) = 0;


  Ref<ValueBase> im(const x3d::sai::Value &v)
    throw(TypeMismatchException, IllegalNodeIdException);

  void ex(const ValueBase *source, x3d::sai::Value &target);

protected:
  Porter();
  virtual ~Porter() {}

private:
  // ...
};
```

Note that `x3d::sai::Value` is in fact a class generated from the IDL (see appendix H). `ValueBase` is an abstract base class for a variety of value wrappers used internally in the server whenever polymorphism is required.

The conversion between the internal and external value representation is simple (though cumbersome) for all types except node references. We see that the `Porter` class declares two pure virtual methods, `id2node` and `node2id`, which means that it is an abstract class. This is because the conversion of node references cannot be encapsulated like the conversion of the rest of the value types. The conversion of node references must be implemented by a deriving class. It is in fact the `Session` class that derives from `Porter` and implements the two virtual conversion methods. The conversion of node references is discussed further in the next section.

6.12.2 Node references

Handling of node references in the SAI is indeed a non-trivial subject. One implication of acting like a fire wall, is that when a node reference is passed as an argument in some service call, it must be possible to verify that the reference is valid and that the client is in fact allowed to access that particular node. For example an application may generally not access nodes outside its own application sub-scene.

A natural choice for the external representation of node references may be to use a plain pointer or address of the node instance in the server. This would no doubt be the representation offering the fastest access to nodes. But the choice has at least one grave flaw. The problem is that the only way to verify that the pointer is in fact a pointer to a node instance, is to search for it in a list of all node pointers. This is clearly unacceptable since there may easily be thousands of nodes in the scene graph.

So we must find an alternative external representation that at the same time makes it both easy and fast to verify the validity of the reference as well as the permission to access the node.

Indexes is another often used kind of reference. In fact using indexes as the external representation of node references solves the problem at hand. The idea is that the session maintains a table of node instances that are currently known to the application. We use indexes into this table as the external representation of node references. These indexes are called node identifiers in the rest of this report (particularly in the SAI IDL, see appendix H).

So, the validation of the reference is now trivial, we just test if the index is less than the size of the table. By putting a smart pointer into the table, we also have a fast way of mapping the index to the real node reference. But what about the permission to access the node? That is easy. We just ensure that we only ever add nodes to the table which the application is allowed to access. The trick is, that efficiency is much more important for the services which access known nodes, than for those which establish knowledge of new nodes. This is true as long as nodes (at an average) is accessed more than once.

Of course we cannot let this table grow forever, and a natural way to prevent it, is to make the `dispose` node service of the SAI remove the corresponding entry in the table. This will leave holes in the table, but this is not a problem for efficiency, we just check for a non-null reference in the mapping from index to reference. We then use the holes when adding new node entries.

We decided that this table and the mappings between indexes and references was a generic functionality. We therefore implemented it as a general purpose utility, and named it, `RefMap<T>`. As the reader can see it is a class template. When the template argument is `T` the map holds references of type `Ref<T>`.

In the `Session` class the mapping between internal and external node references is integrated into value conversion feature. This way, the use of the node map becomes completely encapsulated in the value conversion.

Finally we note that references to `ExecutionContexts` are handled just like references to nodes.

6.13 Class to file mapping

In the text above we did not point the reader to the files where each of the classes are defined/implemented. We think this would have been too disturbing for the reader. Instead we have provided a mapping from class names to file names. This mapping is found in table 6.8.

As the reader may have noticed, there are many files and classes which have not been mentioned at all. We are sorry for this, but we would have to chop down several rain forests if we wanted to introduce you to every one of them. The easiest way to get an overview of the contents of all the rest of the files is to use the Doxygen documentation. Here you will, among other things, be presented with a subset of the "global" class hierarchy, and this hierarchy can be navigated both up and downwards. For an example, go to the front page of the Doxygen documentation. Then select "Compound List" in the top menu. Then select `Archon::X3D::AppearanceChildNode` from the list of compound names (3rd item). Note that the items are sorted by the last part of the compound name.

Finally note that the source code has been commented in many places. So, if the reader has a question which is not sufficiently answered in the text above, even for the classes that were treated in this chapter, it would be worth studying the Doxygen documentation.

Table 6.8: Class to file mapping. Here we list all the classes which are mentioned in this chapter. All the class names implicitly refer to the `Archon` top-level name space. All file names implicitly refer to the `src/archon` directory in project directory structure.

Math::AxisRot3	math/geometry.H
Utilities::BackRefObjectBase	util/ref.H
Utilities::Condition	util/condition.H
Utilities::Image	util/image.H
Utilities::Mutex	util/mutex.H
Utilities::Ref<...>	util/ref.H
Utilities::RefObjectBase	util/ref.H
Utilities::Thread	util/thread.H
Utilities::Time	util/time.H
Utilities::Web::Client	util/web_client.H
X3D::BoundedObject	x3d/server/group.H
X3D::ChildNode	x3d/server/core.H
X3D::CustomFieldBase	x3d/server/custom_field.H
X3D::CustomFieldNode	x3d/server/custom_field.H
X3D::Cylinder	x3d/server/geometry.H
X3D::EventSource	x3d/server/event.H
X3D::ExecutionContext	x3d/server/scene.H
X3D::FieldBase	x3d/server/type.H
X3D::FieldType	x3d/server/type.H
X3D::FontServer	x3d/server/font.H
X3D::GeometryNode	x3d/server/geometry.H
X3D::Group	x3d/server/group.H
X3D::GroupingNode	x3d/server/group.H
X3D::ImageTexture	x3d/server/texture.H
X3D::Inline	x3d/server/network.H
X3D::Loader	x3d/server/load.H
X3D::Node	x3d/server/node.H
X3D::NodeCustomField	x3d/server/custom_field.H
X3D::NodeField<...>	x3d/server/field.H
X3D::NodeFieldBase	x3d/server/type.H
X3D::NodeSequenceCustomField	x3d/server/custom_field.H
X3D::NodeSequenceField<...>	x3d/server/field.H
X3D::NodeSequenceFieldBase	x3d/server/type.H
X3D::NodeType	x3d/server/type.H
X3D::SAI::Listener	x3d/sai/listener.H
X3D::SAI::Porter	x3d/sai/value.H
X3D::SAI::Session	x3d/sai/session.H
X3D::Script	x3d/server/script.H
X3D::SequenceFieldBase	x3d/server/type.H
X3D::Server	x3d/server/server.H
X3D::SimpleCustomField	x3d/server/custom_field.H
X3D::SimpleField<...>	x3d/server/field.H
X3D::SimpleSeqCustomField	x3d/server/custom_field.H
X3D::SimpleSequenceField<...>	x3d/server/field.H
X3D::StaticField<...>	x3d/server/field.H
X3D::Text	x3d/server/text.H
X3D::Transform	x3d/server/group.H
X3D::ValueBase	x3d/server/event.H
X3D::Viewer	x3d/server/view.H

Chapter 7

Client Applications

The client applications described in this chapter are small examples, showing the functionality of the 3D-Console. This chapter also gives some guideline for how to use the client API. We have put much effort into designing an API, which is a high level abstraction layer, very close to the X3D specification, which makes it easy to use, as well as easy to maintain.

This chapter is not a developers guide to our client API for two reasons: A developers guide will make up quite a large document, which is not the goal of this thesis, and the second reason is, that the application developer can see the structure of an application in these examples and extend on the applications by consulting the online Doxygen documentation (see section 6.3) and the X3D specification. A good acquaintance with X3D format should get the application developer far, because the level of abstraction in the client API makes it look very much like the formal notation found in the specification. A description on how this client API is designed and implemented can be seen chapters 5 and 6.

7.1 Client Application examples

We have provided four examples of applications for 3D-Console:

Name: `archon_clock`
Model: A simple model of a analogue clock. It has hands to show hours, minutes and seconds.
Description: Analogue clock shows time. The actual displayed time reflects the system time of the client machine. The red button stops the clock application. With the green button the time can be made to pass much faster (fast forward).

Name: `archon_filebrowser`
Model: The model is a blue semi-transparent cylindrical wall of text plates.
Description: The file browser shows files and sub-directories in any directory. Touching a sub-directory makes the file-browser enter that directory. Touching an executable file will launch it (works best for 3D-Console enabled applications - of course). If the file is an X3D file, then the `archon_x3d_viewer` (see below) will be launched with the indicated file as argument. Any other regular file will be displayed with the `archon_textviewer` application (see below) when touched.

Name: `archon_x3d_viewer`
Model: Depends on the file given as argument.
Description: Reads the file given as command-line argument and streams it to the 3D-Console server. This model is then inserted into the scene graph as an application (sub-scene) - and can thus be viewed together with other applications.

Name: `archon_textviewer`
Model: The model for this application is a white rectangular sheet of paper. Centered at the top of the paper is a red box, and to the right of it, is a green sphere. The box and the sphere are semi-transparent. The presence of the green sphere is conditional.
Description: This application interprets the first command-line argument as the name of a text-file to display. Initially the first page of the text-file is displayed on the paper. If one page is not enough to show the entire text-file, the green sphere appears in the top right corner. Touching the green sphere will turn the page. The sphere disappears when there are no more pages to turn. The Red box stops the application.

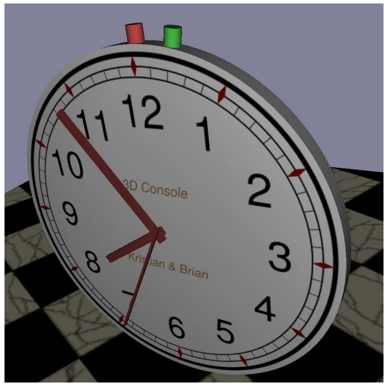

Figure 7.1: Screen-shot of the `archon_clock` application launched in 3D-Console. There are more screen-shots included in appendix G.

7.2 Dissecting a client application

To understand the internals of client applications, we will take a look at the `archon_clock` main function. The full listing of this client application, and all the others, can be found in Appendix I.

The client application starts by instantiating a `Proxy::Session`. A `Proxy::Application` object is then instantiated with a reference to the Session object. The Application object corresponds to an X3D subscene (or in-lined scene) and holds the reference to the root group. The root group is the top-most `GroupingNode` in the client application's scene graph.

After setting up the Application object, the client application normally sets up its main transformation. This is the `rootTransform` in the listing below. In this way the whole application can be translated, rotated and scaled by changing this Transform node. The rootTransform node is added as a child to the root group.

Listing 7.1: Setting up client application session and initial Transform

```
Ref<Session> session =
  Session::create(opt_serverName, opt_corbaEndpointPort);

Ref<Application> application =
```



```
  Application::create(session, Uri());
Ref<Group> rootGroup = Group::create(application);
application->setRootGroup(rootGroup);

Ref<Transform> rootTransform = Transform::create(application);
rootTransform->rotation = AxisRot3(Vector3(1, 0, 0), M_PI/2);
rootTransform->scale = Vector3(25, 25, 25);
rootTransform->translation = Vector3(0, 100, 0);
rootGroup->children.add(rootTransform);
```

After setting up the Session, Application and rootTransform - this clock application begins defining its own geometry. First it sets up the dial plate. This dial plate is defined as the top of a cylinder. Its appearance is the default material with default color, but a texture is added. After setting up the shape and underlying geometry, this is added to the rootTransform.

Listing 7.2: Setting up clock dial plate

```
{
  Ref<Cylinder> cyl = Cylinder::create(application);
  cyl->side = false;
  cyl->top = true;
  cyl->bottom = false;
  cyl->height = 0.1;
  Ref<Shape> shape = Shape::create(application);
  shape->geometry = cyl;
  Ref<Appearance> apperance = Appearance::create(application);
  apperance->material = Material::create(application);
  Ref<ImageTexture> texture = ImageTexture::create(application);
  texture->url.add("clock.png");
  apperance->texture = texture;
  shape->appearance = apperance;
  rootTransform->children.add(shape);
}
```

The clock casing is created in almost the same way. In this case the cylinder sides, and bottom is used instead. No texture in used in the appearance node.

This shape is added to the rootTransform as well.

Listing 7.3: Setting up clock casing

```
{
  Ref<Cylinder> cyl = Cylinder::create(application);
  cyl->side = true;
  cyl->top = false;
  cyl->bottom = true;
  cyl->height = 0.1;
  Ref<Shape> shape = Shape::create(application);
  shape->geometry = cyl;
```

```
  Ref<Appearance> apperance = Appearance::create(application);
  apperance->material = Material::create(application);
  shape->appearance = apperance;
  rootTransform->children.add(shape);
}
```

A close button is created by defining a Shape in which a Cylinder is setup as geometry. This cylinder becomes red by changing the Material's diffuse color.

This button must react if pointed on by a pointing device, thus a TouchSensor is created and a Transform node is created to group the Shape and the TouchSensor into the same level in the hierarchy. The TouchSensor sets up its event handling, by registering an event handler for each field it wants the application to receive events from. The event handler is a function inside this application. Whenever an event reaches this field, it is sent to the function in this application.

The close button is moved into its position by changing the Transform node mentioned above.

Listing 7.4: Setting up client application close button

```
{
  Ref<Cylinder> cyl = Cylinder::create(application);
  cyl->side = true;
  cyl->top = true;
  cyl->bottom = false;
  cyl->height = 0.1;
  cyl->radius = 0.04;
  Ref<Shape> shape = Shape::create(application);
  shape->geometry = cyl;
  Ref<Appearance> apperance = Appearance::create(application);
  quitMaterial = Material::create(application);
  quitMaterial->diffuseColor = Vector3(0.8, 0.3, 0.3);
  apperance->material = quitMaterial;
  shape->appearance = apperance;
  Ref<TouchSensor> touch = TouchSensor::create(application);
  touch->isOver.registerInterest<SFBool>(new FuncCaller<SFBool>(quitIsOver));
  touch->touchTime.registerInterest<SFTime>(new FuncCaller<SFTime>(quitTouch));
  Ref<Transform> transform = Transform::create(application);
  transform->children.add(shape);
  transform->children.add(touch);
  transform->translation = Vector3(0, 0, -1.03);
  transform->rotation = AxisRot3(Vector3(-1, 0, 0), M_PI/2);
  rootTransform->children.add(transform);
}
```

Next, the application defines a generic clock pointer. This pointer is a red Box defined with the appropriate dimensions, so it only needs scaling in one of the axis for it to be used as pointer for hours, minutes and seconds.

A Transform node is created and the pointer Shape is added as child node. This transform translates the pointer out a bit from the clock dial plate.

Listing 7.5: Defining generic clock pointer

```
Ref<ChildNode> pointer;
{
  Ref<Box> box = Box::create(application);
  box->size = Vector3(0.05, 0.02, 0.9);
  Ref<Shape> shape = Shape::create(application);
  shape->geometry = box;
  Ref<Appearance> apperance = Appearance::create(application);
  Ref<Material> material = Material::create(application);
  material->diffuseColor = Vector3(0.5, 0.2, 0.2);
  apperance->material = material;
  shape->appearance = apperance;
  Ref<Transform> transform = Transform::create(application);
  transform->children.add(shape);
  transform->translation = Vector3(0, 0, -0.4);
  pointer = transform;
}
```

Clock dials for hours, minutes and seconds are created by instantiating a Transform for each dial and then, using the generic dial above and scaling it, they are added as children to the rootTransform.

Listing 7.6: Setting up clock hour/minute/second pointers

```
Ref<Transform> seconds = Transform::create(application);
seconds->children.add(pointer);
seconds->translation = Vector3(0, 0.1, 0);
seconds->scale = Vector3(0.3, 1, 1);
rootTransform->children.add(seconds);

Ref<Transform> minutes = Transform::create(application);
minutes->children.add(pointer);
minutes->translation = Vector3(0, 0.08, 0);
rootTransform->children.add(minutes);

Ref<Transform> hours = Transform::create(application);
hours->children.add(pointer);
hours->translation = Vector3(0, 0.06, 0);
hours->scale = Vector3(1, 1, 0.6);
rootTransform->children.add(hours);
```

Then the application is ready to be launched. By launching the application, the scene graph built by the application - from the rootGroup and all its descendant - is added to scene graph of the 3D-Console. It is now possible to see the client applications model in the 3D-Console viewer.

Listing 7.7: Launching the client application

```
application->launch();
```

After inserting the application scene graph into the 3D-Console scene graph, the main loop of the client application starts.

This loop gets the current time from the client applications Operating System, it then calculates the right angles for the clock pointers and updates the rotation fields in the according pointers Transform nodes.

The loop then waits 0.1 seconds and then loops around again.

Listing 7.8: Client application main loop

```
Archon::Utilities::Time delay(0, 100000000l);
while(!quit)
{
  long sec, nsec;
  Time t = Time::now();
  if(fastForward) addition += 60000;
  t.addMilliSeconds(addition);
  t.get(sec, nsec);
  time_t t1 = sec;
  tm t2;
  localtime_r(&t1, &t2);

  double n = nsec/1E9;
  double s = (t2.tm_sec + n)/60;
  double m = (t2.tm_min + s)/60;
  double h = (t2.tm_hour + m)/12;

  session->beginUpdate();

  seconds->rotation = AxisRot3(Vector3(0, -1, 0), s*2*M_PI);
  minutes->rotation = AxisRot3(Vector3(0, -1, 0), m*2*M_PI);
  hours->rotation = AxisRot3(Vector3(0, -1, 0), h*2*M_PI);

  session->endUpdate();
  Archon::Utilities::Thread::sleep(delay);
}
```

There are two event handling functions for the quit button declared outside the main function:

Listing 7.9: Functions handling external events

```
void quitIsOver(const bool v)
{
  quitMaterial->diffuseColor = (v ? Vector3(1, 0, 0) : Vector3(0.8, 0.3, 0.3));
}
```



```
void quitTouch(const Time)
{
  quit = true;
}
```

Chapter 8

Results

By creating client applications, and testing them, we in a sense have proved, that the 3D-Console server and client API works as expected.

Using the X3D specification as a guideline, to the design of our 3D-Console and client API, has been a good choice, as the design and implementation of the system has progressed without exposing any significant flaws. Inventing a well formed specification like X3D, is not easy, and we have had only few issues regarding obscurities in the X3D specification, see appendix E.

The concept of having external client applications connecting to the 3D-Console server and creating X3D scenes is working fine. The 3D-Console server is able to administer all the resources used by the client applications, and no memory leaks occur (we check this before terminating the server). We have tested stability of the 3D-Console for quite some time, launching and stopping several client applications, and no crashes has been provoked yet. So the 3D-Console is very stable.

The resulting code in the client application is simple to read and comprehend. The most important reason for this, is that the 3D-Console client API's syntax is very simple, and thus it is easy to get a nice structure on the client application source code.

Using the X3D as a specification of the scene graph nodes, has shown to be beneficial, as the X3D node definitions are comprehensive and precise. For the client application developers who knows the formal X3D node hierarchy,

the close correspondence with the client API, makes it easy to learn the capabilities of the client API, and thus the learning curve for learning the 3D-Console client API is almost flat.

The performance while building the client application scene is not as fast, as it could have been, mostly because the client application will have to wait to get exclusive access to the 3D-Console scene graph. After the scene is built and the application launched, the performance of the rendered client application is fine.

As described in the introduction, see section 1, it was never our goal to go deep into the area of performance optimization, still the performance is not bad, since the 3D-Console is well designed and implemented. But, by implementing the enhancements described in section 9, we believe the performance will be optimized considerably in many parts of 3D-Console.

Chapter 9

Future Enhancements

Even though we have implemented quite a large system, there are still many enhancements we would like to implement in the future. Some of them will be described in this chapter.

9.1 Linking applications

An important aspect of any multi-application user interface is the ability for the user to request transfers of data between applications. This requirement has been further described in section 3.8.1. In 3-D there is no canonical way to approach this, simply because we have seen no examples of it before. Yet in this section we will discuss one way to deal with it in 3D-Console.

From the 2-D arena we have been accustomed to the "copy and past" paradigm. In this scheme the system acts as a temporary stash for the data in transit. The source and target applications need not even be running at the same time. An alternative scheme, somewhat similar to that of the UNIX shell, is the "rendezvous" where two applications are linked for a period of time while they exchange data. In this case the system acts as a mediator of the rendezvous. Although both schemes may be relevant in 3D-Console, we will in the remainder of this section only deal with the latter one, the "rendezvous". We expect that the "copy and past" scheme would go hand in hand with the introduction of a file concept, where a file

would be an application independent visual data entity supported directly by the server.

In the case of rendezvous we need a way for the applications to announce their willingness to either emit or receive data. We also need to provide a way for the user to choose and establish the link. One might consider whether this scenario could be modeled with the building blocks which are supported by 3D-Console already. The answer is, that it probably would be possible if we allowed applications to access the same nodes in the scene graph, but as we learned in section 3.8.1 this is prohibited, and this is only to protect the applications from each other.

It is important, of course, that we choose a very general solution, since we do not know what the contents and the meaning of the transfer is. We also want to choose a scheme which is as compatible as possible with the existing system. We already have the notion of a stream in the SAI through which the application can communicate with the server. So, we suggest we base the solution on this stream concept.

The idea is that, one application registers one of its objects to represent a data outlet. Similarly another application registers a data inlet. A stream end-point must be passed as an argument when registering the outlet and in the registration of the inlet, the application must pass a handler routine. The inlet and outlet objects must appear to the user in a way which identifies their function. A standardized appearance could be achieved by defining two new node types specifically for this purpose. Anyway, the connection will be made by the user when he clicks on the outlet object and then drags it to the inlet. In the target application the handler routine will be called with the stream end-point as an argument, and the two application are free to communicate. When the stream end-point is closed, the link is broken.

Since our applications are ordinary programs running as operating system processes, one might ask why we do not let the applications handle the establishment of communication themselves. The applications are free to use any conventional means of inter process communication, so why introduce yet another concept for this? The problem is that one application must know which other application to contact. This information cannot be acquired without some sort of support by the server. Of course we could choose to pass only an application identifier to the target application and let the application establish the communication channel. For this purpose such a thing as the process identifier would not do the job, since the two applications may be located on two different computers connected by a network.

Instead of passing a stream end-point object between the applications, one could consider using a CORBA object reference or maybe just a plain UNIX file descriptor.

9.2 Application boundaries

Enforcement of a bounding box which cuts off any geometry which extends beyond its sides is one way to deal with the problem of preventing overlap between applications. This could be implemented with OpenGL custom clipping planes.

Another user requirement was the manipulation of application location, orientation and scale. This is of course not directly supported by X3D, since in X3D, applications do not have a section of the scene graph assigned to them. On the other hand, the X3D specification requires that a hierarchy of oriented bounding boxes (OOB) is maintained withing the scene graph. The primary use of this is to help detect user/object collision, but the bounding hierarchy has other uses, an example is for producing projective textures on `IndexedFaceSet` nodes. In 3D-Console we could utilize these bounding boxes as a wire frame model, when displaying an application's initial location, orientation and scale.

9.3 Independent viewer

We would like to make the viewer an independent application. For this to be possible, we must find a way to send a representation of the scene graph to the viewer. One way to send this representation is by streaming. After the viewer has built its own representation of the whole scene, only changes in the scene, will have to be propagated to and from the viewer. Since viewers are actually interactive parties, we may allow the viewer to push changes to every other party, instead of first telling the server about the change, and then letting the server push it to the rest of the viewers.

By having separated the viewer from the server, we will have the opportunity of having multiple viewers as well, since the mechanism above will

still work for more than one viewer. By having more than one viewer, we enable more users of course. We have to consider the network traffic load though. Streaming via multi cast network protocols could be the solution.

Using multiple viewers for enabling multiple users, introduces another issue: representation of users by avatars. If users should be able to see each other in the 3-D environment, then a graphical representation of the user needs to be combined with the viewer.

When we need to redesign the viewer anyway, we would like to utilize 3rd party rendering systems, instead of building our own renderer. If we use a 3rd party renderer, we need not implement optimizations to our own renderers, since most 3rd party renderers are more advanced. Otherwise we would look into implementing view frustum culling and shadows. View frustum culling will speed up the rendering of complex scenes. Implementing shadows improves the sense of depth, and thereby enhances the immersiveness.

We have thought about using OpenSceneGraph, which is an open source high performance 3-D graphics toolkit. It was actually an article from OpenSceneGraph on parallelizing the rendering process (i.e. splitting the rendering pipeline into multiple phases that can overlap in time), see [Bur01], which fueled our interest in OpenSceneGraph most. By utilizing the multiphase principal, we could get rid of a problem, which had bugged us for some time. The problem is that while the rendering process is traversing the scene graph, no other process can update it, and since the rendering process does this the most of the time, there is too little time for other threads to access the scene graph. With the pipe-lining strategy the first phase of the pipe-line will snapshot the scene graph once per frame, but the snapshot will take much less time to perform, than when rendering while traversing. The pipe-lining strategy of course only makes sense on a multi processor or otherwise distributed system.

More information about OpenSceneGraph see: [Ope].

9.4 Collision detection

Collision detection between objects in the scene is a major issue, and one we have chosen not to embark on in this project. It is a major issue because it generally is too costly in terms of computation time. Therefore one needs to consider carefully how to reduce the complexity of the problem. In inherently dynamic scenes such as our 3D-Console, the problem is even worse due to the fact that the fastest algorithms relies on the scenes being mostly static. An example is the BSP (Binary Space Partitioning) algorithm.

The good news is that there exists several off-the-shelf libraries which does a real good job on inter-object collision detection. The most promising one right now seems to be SWIFT [Ehm01] (judged from the number of references) which is also open source. Other libraries are MathEngine [Mat] and I-Collide [URGoMA95]. If we were to implement collision detection in 3D-Console, we would take a close look at these libraries. Using a inter-object collision detection library e.g. SWIFT, together with a broad-phase collision detection routine using the bounding volume hierarchy in X3D, could be a promising approach.

But as long as there is some kind of inter-object collision detection, there is also the problem of how to react when a collision occurs. Assuming the server detects all object-to-object collisions and all moving objects are controlled by applications, a possible strategy would be for the server to reject any new object position requested by an application which would lead to a collision. From the direction of the movements the server could then compute the direction of reflection, and pass it back to the application. It would then be the responsibility of the application to determine a suitable new location for the object, but the server has acquired its goal - to prevent overlapping geometry.

More information about collision detection can be found in section 3.7.6.

9.5 X Support

As a future enhancement, we would also like to implement X Support. By that, we mean support for X Windows applications. As it is right now, applications has to be re-written to use our client API. We have hereby cut ourselves off from displaying standard X windows applications.

In the future we aim at implementing a sort of window in 3-D. In such a window an X application will be drawn. This window will be sensitive for touch and could be chosen as target for input e.g. keyboard events.

This kind of enhancements could be implemented in many ways. We have thought of using VNC (like 3Dwm does). The power of VNC is that you need only implement a very simple viewer and then you can present a fully functional X desktop. The problem with VNC is that it is desktop based, which is not what we really need here. The ideal solution would be that the 3D-Console would act as an X server on behalf of each viewer and instead of drawing windows on top of each other into a frame buffer it should allocate a polygon and a texture for each open window and then render the window contents into these textures. Keyboard and pointing device events could maybe be captured and sent as X events into the existing X server.

9.6 Connecting several 3-D environments together

Several applications can connect to the 3-D environment we have created i.e. 3D-Console. But connecting several 3D-Consoles will open new possibilities! It gives the possibility of a nearly infinite 3-D environment - imagine the Internet, with each of the computers running a 3-D environment, which are all connected.

Many problems exists in handling multiple connected 3-D environments. A major issue in rendering the environments are the wast amount of polygons or geometry as a whole. This issue can be remedied by restricting the visual view of the users/viewers - thereby reducing parts of the scene. 3-D portals could be the technology which remedies this [Bik]. 3-D portals are used, when we need to restrict the rendering of the 3-D environments, to only render what can be seen from a specific viewpoint (normally the eye-point of the viewer).

Connecting several 3-D environments will give the opportunity of collaboration. Collaboration between 3-D environments is something which needs to be experimented with, as the possibilities seems limitless.

9.7 Fine-grained scene graph locking

Due to the fact that the scene graph must stay constant during rendering of a single frame, we need a way to lock out scene graph updates in these periods. On the other hand there is no reason to disallow reading individual values from the scene graph while rendering. So this problem is best solved with a read/write-lock. That is, a lock which can be acquired by either one writer or several readers. In fact this could be relaxed to allowing several writers if those writers request access to different execution contexts (ExecutionContext). Also a simple read operation could be allowed to occur concurrently with a write operation in another context.

With read/write locks starvation becomes an issue. That is, even though each read lock is only held for a short time, they can still occur in an interleaved manner such that there is always one or more readers preventing the writer to ever get through. Preventing starvation requires us to consider the relative priorities of the relevant parties.

To prevent too much disturbance in the frame rate it is important that the rendering thread has a high priority when it wants to acquire the lock. Furthermore these rendering requests happen in a predictive and periodic manner and thus will generally not be the reason to starvation of others. This suggests that when the renderer arrives and finds the lock acquired by a writer, then no new writer should be allowed to enter until the renderer has been through.

Note that read operations are generally much faster than write operations since write operations generally involve event cascading. this suggests that plain read operations should have a higher priority than write operations, but probably not as high as the rendering thread.

Note also, if we were to support more than one renderer in the server (e.g. stereo view) we should allow them to process concurrently as two readers. This means, on an MPU we will be able to get almost the same frame rate with two rendering threads as with one.

9.8 Concurrent Event Cascades

The X3D specification seems to be open to this. First of all we must define a certain property of an X3D field. We will call it "memoryless assignment". The memoryless assignment fields are those fields where, first assigning A then assigning B, has exactly the same global effect on the scene graph as if we only assigned B to it. That is, The scene graph forgets that we assigned A first.

As long as we consider only memoryless assignment fields, we can approach the issue of having concurrent event cascades within one context. We should simply let two or more threads pick up distinct root events with distinct time stamps, and whenever a thread sees a target field with a time stamp higher than its own then it should skip that branch of the cascade.

There is one thing which spoils the idea above, and that is route changes. If a script node uses dynamic route handling, then it could potentially remove a route which is already followed by an event which really has not occurred yet according to the time stamp of the script evaluation. A solution could be to define route changes to be buffered and applied after all event cascades. This strategy would eliminate non-determinism in the restricted scenario above. The X3D specification may actually specify that the route changes should be carried out after event cascades.

Now, what shall we do about the fields which do not have the "memoryless assignment" property? One idea would be to simply pause the thread which is about to assign to such a field, until all threads executing earlier timestamps have completed. The good thing about this is that the worst case scenario is no worse than if we had only one thread executing all event cascades. One challenge is that we must be able to determine the memoryless assignment property of the fields. Instead of pausing a thread we could just add the event back the to set of root events, and get on with another root event.

9.9 Non-conventional text

Traditionally text is the fundamental form of human perceivable information on a computer. This fits well with the 2-dimensional user interface (probably because text in a book is 2-dimensional). On UNIX systems "text" has a particularly important role, since here almost anything we work with (program input/program output) is text. It is the common language of all UNIX programs.

Text is good because it is an effective and compact way to present information to the user, and for the user to instruct the system. Also, we, as humans, are very well suited to extract the information from the text. Text has two other qualities when used in a systematic way: It is precise, and it is easy to enumerate thing in a (close to) unlimited way: file0, file1, file2...

On Microsoft Windows text plays a somewhat smaller role, since interaction is generally done through graphical means. Also, the internals of Microsoft Windows is not revealed to the user as text, like it is on UNIX systems.

In 3-dimensional user interfaces there is an urgent need for alternatives to text. That is, to really get a benefit of 3-dimensional interaction we need to abandon text as the primary form of information/identification. It is really hard to come up with good alternatives. How can we enumerate an unbounded number of different file types with small geometric objects? We could use a box for the first one, a sphere for the next one and so on, but it is hard to automate this procedure while maintaining enough visual uniqueness on every singe type of object. Another problem is in the association from a geometrical form to the concept which it stands for (file type). The problem, compared to text, is even if we could find shapes with good associative quality, it is still not nearly as precise as text. Different people may get different associations.

The really good question is, can we develop a way to use geometric shapes in combination with images to provide the same kind of information as is possible with text, and in an equally or even more precise way than with text. If we could, we would have a strong case for advancing to 3-D. What we need is an alternative kind of an alphabet, which is not 2D limited. This would allow us to combine things the way we need to be able to enumerate things, and present precise information. Now, the real challenge is to create this alphabet such that it possesses a lot more expressional power than what we have with ordinary text. We should expect that this is possible since we have an extra dimension to utilize.

You might ask, how will we ever learn to use such an alternative alphabet? - we are made to read old fashioned text! Well, we think it is going to happen, and rather fast too. When we look at the way we communicate with computers today, it is obvious that we have already developed a common knowledge of how to "read" the contents of a computer screen - we think in windows, buttons, menus and cursors. Not because these concepts match the real world but because they are simple, and we are used to it. But it is still all 2-dimensional.

9.10 Files / Tools

What is the fundamental kinds of interaction we need to do, to get some work done? What is work?

Probably one thing we need is a concept of files as a graphical entity/object. But are files then all of the same form/size? If files always have roughly the same size (a small thing we can grab with our virtual hands), then it would make sense to apply files to tools to get something done, such as printing a file, or mailing a letter. On the other hand we may look at files as something which can vary greatly in size, they could be as big as houses or even worlds to us. In this case we cannot grab them of course. Instead we could then model tools as something of a fixed size which we could grab, and then we could apply tools to files. In any case "scale" is something that needs to be dealt with. It needs to be restricted in some way to be useful to us.

Chapter 10

Conclusion

This project has primarily focused on the fundamental framework for a "full blown" 3D-Console. We think, though, that it was right of us not to strive for a complete end-user interface in our implementation. This would not only have been too ambitious for our project, but it would also very likely have become a failure. That is, the field of interaction in a virtual 3-D environment is too unexplored and still so alien to all of us. As we see it, there is only one way to approach these new possibilities, and that is to create new software which puts everyone in a better position to experiment with the 3-D environment.

An example we found by experimenting, is that client applications was good for generating and manipulating geometry, like meshing surfaces, and the combination of first reading a scene from a file and then start manipulating it, is fertile and very in-verbose/compact. So our job was to create a solid, well designed and complete foundation, on top of which only simple programs are needed to experiment with interaction. This we have achieved.

The new idea in this project was to combine two well known concepts: User interface frameworks for independent applications and 3-D environments. Designing and implementing that idea went quite well.

The use of the X3D specification as a guideline for the design of the 3D-Console and the client API was a good decision, as we had only few X3D related problem issues while implementing the system. Our own extensions

to X3D, for it to be able to support our idea, has been implemented with success as well. We are able to launch client applications, and the server has control of client application's resources, thus all resources is being deallocated and the 3D-Console server remains stable, even after prolonged use i.e. no memory leaks occur.

Our design and implementation of the client API was very successful. We showed in section 7 that client applications was very easy to write using the client API, and we are able to have a nice structure of the resulting source code, as the client API's syntax is very simple. Furthermore we showed that the communication between the client application and the server worked well and was very stable.

We will now relate our results to the original project description, see appendix A.

One objective was, that the 3-D environment must provide spatial navigation. This we have implemented by using navigation types Fly and Examine, which are innate parts of X3D. We have in fact implemented a system that is prepared for future Human Interface Devices (HID), as we expected.

We promised to implement the launch of applications in the 3-D environment. This we have done, although users are not able to adjust the location of the applications when launched. We will let the future designers of 3-D user interfaces (our users) experiment with this feature. See section 3.7.1.
Furthermore, since we are able to launch applications, it follows that we are able to represent applications in the 3-D environment by showing the 3-D models they have created, and applications are able to change their representation dynamically while running. See section 7.
Finally, for enabling client applications to interact with the 3-D environment, an API has been designed and implemented. We call this the 3D-Console client API, for more information on how this is implemented see section 6.12.

So the three features for the environment, as listed in the original project description, has been implemented with the minor shortcomings mentioned above.

As we further promised in the project description, we have discussed in chapter 3, how to make the 3-D environment comfortable and immersive, while remaining efficient for the advanced user. In this discussion we also give our view on how to increase the efficiency, comfort and immersiveness, by using different Human Interface Devices and graphical effects. See chapter 9.

All in all, we have completed our objectives from the original project description.

Work must continue on the 3D-Console before anyone will find it useful for its purpose. Therefore we have strived to provide appealing arguments for the idea and fought to provide a good and appealing design in what is now implemented. One of the features of the design is that it makes the implementation easily extensible, and we have in this report included many considerations for further extensions to the system, which is described in chapter 9. It was important for us that we did not introduce unnecessary burdens (or even barriers) by trying to second-guess the desires of the users, but instead strived to provide a generic and intuitive set of features. In our opinion, we have succeeded in providing this.

It is our hope that we can attract new developers from the free software community, who in a collaborative effort will bring the 3D-Console onwards and hopefully towards our vision of it. But to ever arrive at a well-functioning, effective and general purpose 3-D user-interface we need a team of developers to build upon the foundations which we have created. To be successful with respect to this goal, we believe it was necessary for us to confine the project to the public domain and license it under the GNU General Public License. In fact we have already been contacted by several people, expressing interest in or project and a desire to join forces with us.

Bibliography

[Ben03] Stephen A. Benton. *MIT Media Lab: Research at the MIT Media Lab.* http://www.media.mit.edu/research/ResearchPubWeb.pl?ID=43, 2003.

[Bik] Jacco Bikker. *flipCode - Portal Engine Column.* http://flipcode.com/portal/issue09.shtml.

[Bor01] Karsten Bormann. *Usability Concerns for Virtual Reality Aided Architectual Design.* IMM, April 2001.

[Bri90] Meredith Bricken. *VIRTUAL WORLDS: NO INTERFACE TO DESIGN.* http://www.hitl.washington.edu/publications/papers/interface.html, 1990.

[Bur01] Don Burns. *A New Processing Model for Multithreaded, Multidisplay Scene Graphs.* http://www.openscenegraph.org/download/bazaar/OsgMultipipe, 2001.

[Ehm01] Stephen Ehmann. *SWIFT - Speedy Walking via Improved Feature Testing.* http://www.cs.unc.edu/~geom/SWIFT, 2001.

[Gar99] Bernd Gartner. *Smallest Enclosing Balls - Fast and Robust in C++.* http://www.inf.ethz.ch/personal/gaertner/miniball.html, 1999.

[Groa] Object Management Group. *Welcome To The OMG's CORBA Website.* http://www.corba.org.

[Grob] The X3D Graphics Working Group. *X3D Specification ISO/IEC FCD 19775:200x, December 26, 2002.*

http://www.web3d.org/technicalinfo/specifications/ISO_IEC_19775.

[M.82] Ben-Ari M. *Principles of Concurrent Programming.* Printice Hall, August 1982.

[Mat] *MathEngine.* http://www.mathengine.com.

[Med] Chalmers Medialab. *3Dwm.* http://www.3dwm.org.

[Omn] *OmniORB, Free High Performance ORB.* http://omniorb.sourceforge.net.

[Ope] *OpenSceneGraph - the high peformance open source graphics toolkit.* http://openscenegraph.sourceforge.net.

[Pae95] Alan W. Paeth. *Graphics Gems V.* Morgan Kaufmann Publishers, 1995.

[pd] Dive platform development/SICS. *DIVE.* http://www.sics.se/dive/docs/description.html.

[Pro] NeHe Productions. *Bitmap Fonts, Outline Fonts, Texture Mapped Outline Fonts (Lesson 13 - 15).* http://nehe.gamedev.net.

[SG] Inc. Silicon Graphics. *OpenGL - High Performance 2D/3D Graphics.* http://www.opengl.org.

[URGoMA95] Physically-Based Simulation UNC Research Group on Modeling and Applications. *I-Collide.* http://www.cs.unc.edu/~geom/I_COLLIDE, 1995.

Index

3-D environments, 3
3-D environments, existing, 25
3-D user interfaces, 3
3D-Console, 1
3Dwm, 26
3Dwm, space, 26

application, 1, 15
application scene, 50
application session, 50
application, launching, 16, 25
application, sizing, 16
Archon, 61
avatar, 106

boundary, 25
bounding volume, 19, 25
broad phase, 20

client/server, 45
collision, 23
collision detection, 106
combining applications, 23
compound objects, 30
copy and paste, 103
CORBA, 43

DAG, Directed Acyclic Graph, 30
data inlet, 104
data outlet, 104
discomfort, 8
DIVE, 27
download, 48

ECMA script, 40
event, 38
event cascade, 38
ExecutionContext, 39
experiments, 9

feedback, acoustic, 12
feedback, haptic, 12
feedback, olfactory, 12
feedback, temperature, 12
field, 36
field of view, 7
field, initialize only, 37
field, input only, 37
field, input output, 37
field, interest, 39
field, output only, 37
file browser, 8
files, 112
font server, 79
font, bit-mapped, 79
font, outline, 79
font, texture mapped, 79
formal notation, 38

gesture, 23
gloves, 23
gloves, tracked, 7

graphical effects, 13
gravity, 25

hands, virtual, 7
head mounted display, 7
horizon, 13
horizon, virtual, 8
human capabilities, 12

I-Collide, 107
ideal work situation, 7
illusion, 11
image, 22
immersiveness, 12
indirection, 24
inheritance, 36
Inline, 39
input, 22
interaction resources, 12
interaction, two handed, 8

job queue, 48

keyboard, tracked, 7
keyboard, virtual, 7

language independence, 53
locality, 24

manipulate applications, 25
MathEngine, 107
memory management, 50
memoryless assignment fields, 109
model, 22
movement indication, 23
moving forms, 12
multi-threaded design, 48

name spaces, 61
narrow phase, 20
navigation, 16, 25
navigation, Examine, 17
navigation, Fly, 17
navigation, LookAt, 17
navigation, Walk, 17
network transparency, 53
Nobel API, 26
node type, 47
normal work, 5

Object Request Broker, 44
OmniORB, 44
open source, 10
OpenGL, 29
output, 22

panel example, 30
permissions, 21
platform independence, 53
pointing device, 23
polygon soup, 29
polygons, 29
portal, 108
primitives, geometric, 22
privacy, 23

reaction, autonomous, 14
reality, augmented, 7
reality, virtual, 11
rendezvous, 103
requirements, application, 22
requirements, system, 21
requirements, user, 24
resource management, 49
resources, abusing, 18

SAI, 38
SAI, additions, 54
SAI, external application, 43
scene, 29
scene graph, 30

scene graph, distributed, 26
scene, dynamic, 38
screen, semi-transparent, 9
senses, 12
sensor nodes, 36
server, stand alone, 46
shadows, 13
SourceForge, 10
spatial structure, 29
speech recognition, 15
SWIFT, 107

text, 22
text, conventional, 110
text, non-conventional, 111
thread pool, 48
touch, 12
touch sensitive, 23
tracking, 23
transparency, 13
travel, 25
type system, 47

unistroke pen movement, 15

versioning system, 10
viewer, independent, 105
virtual keyboard, 15
virtual reality, 11
virtualization, 24
VNC, 108
VRML, 33

X Windows, 24
X3D, 33
X3D engine, 47
X3D, custom code, 38
X3D, field, 36
X3D, prototypes, 40
X3D, sensor nodes, 36