A META-AUTHORING TOOL FOR SPECIFYING BEHAVIOUR IN VIRTUAL REALITY ENVIRONMENTS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Zayd Hendricks

March 2004

Supervised by

Gary Marsden and Edwin Blake

Abstract

When creating virtual reality environments, much of the behaviour and interaction needs to be programmed. Non-experts of computers inevitably lack the programming skills necessary to create useful applications. Specifying behaviours and interactions remains in the domain of the programmer. Novice users wanting to create virtual environment solutions require that the creation of their environments be mediated through a programmer's interpretation.

In this dissertation, we explore methods for empowering non-programmers with the ability to develop their own virtual environment applications. We explored some of the existing systems to determine what methodologies have already been successfully (or unsuccessfully) applied in the fields of virtual environment systems, authoring tools, and graphical user interfaces. From these methodologies we describe an ideal virtual environment authoring system with which comparisons may be drawn to evaluate existing systems. This ideal system represents a tool ideal in its ability to allow users of differing levels of skill to rapidly create virtual environment applications of any sophistication.

Creating such a single, generic authoring tool for every different kind of application is, practically, an impossible task – more so if the authors are non-programmers. A more realistic solution to the problem would be to think of every environment as having a particular context such as a virtual museum or gallery. Creating authoring tools specific to these types of environment contexts greatly reduces the problem. We have therefore produced a progressive meta-authoring system that allows both novice and advanced users to create useful virtual reality applications, allowing the smooth migration of novice users to becoming more experienced. We believe that our system overcomes problems in architecture and support for novice users that can be found in many other authoring systems for virtual environments.

Acknowledgements

Firstly, I would like to thank all those people involved with the Caves project for providing the necessary equipment and funding without which this project would not have been possible. It has truly has been a pleasure seeing the project develop from start to near completion. I wish it and all those that I worked with every success.

To Rudy Neeser and Dr. James Gain for providing me with the opportunity to test the API implementation as part of their second year undergraduate project. To Matthew Hampton for the hard work he did on creating the essential tool for converting graphics formats. To all those in the Collaborative Visual Computing Laboratory who lent their invaluable ideas and opinions. For those who have helped me as well as those that have distracted me with our "ritualistic" afternoon sessions. Those times were most definitely memorable.

To Assoc. Prof. Gary Marsden and Prof. Edwin Blake for all their time and valuable insights and experience in developing and focusing this thesis.

To Jack with whom I've always worked very close with and who has remained a good friend throughout these years.

To my parents who have patiently put up with me for my years of being a student and who have never lost faith in me.

Last but not least I would like to thank my Bacsha without whom I would never have been persuaded to do this in the first place. For supporting me for these past three years and making it as an enjoyable time as it was. I will treasure that precious time we spent forever.

Contents

List of Figures

Chapter 1

Introduction

1.1 Virtual Environment Authoring

As new technologies are developed, different types of applications become possible. For example, with the advent of technologies such as accelerated graphics hardware, three-dimensional virtual environment interfaces have begun to be realised. Virtual environments (or virtual reality) are an exciting new technology that holds many promises for many different applications in the future.

According to Winograd [62], all such emerging technologies progress through a process of maturity. In the first phase of any new technology, initially it is difficult to employ. The potential and benefits of it may also not be initially realised – "its appeal is mainly for those who are fascinated with it for its own sake". Virtual environments are a technology that is still in this first phase of maturity – as an exciting, new technology, its benefits as a practical technology have not yet been fully realised. Consequently, its main current use is by researchers and special interest groups. In the second phase of emerging technology development, the economic benefits of using a technology are developed to a point where it can be used within industry; where businesses will use it for practical purposes. The third and final phase of a maturing technology would be when the technology becomes appeal driven – it would be used purely because it satisfies some need or urge.

For some part, virtual environments are driven by improved graphics hardware. This, in itself, is being driven by the computer gaming industry, already in its third phase of maturity. Although virtual environment technologies have been hyped extensively, many of these hyped "beneficial

applications" have failed to materialise. The Cultureware project [60] is one such project, an aspect of which was to provide a "convincing virtual context", that ultimately failed to deliver a fully interactive experience. The Virtual Reality Modelling Language (VRML) is described as a three-dimensional analogy to HTML [11]. Previously hyped as the "next best thing" for the World Wide Web, it has also failed to gain market acceptance [39]. Such failures have unfortunately resulted in these technologies lacking commercial credibility. Commercially, virtual environments *can* provide a method of visualising new products and processes that are either expensive or dangerous to develop. There are many commercial applications for which virtual environments have been shown to be effective, such as in the fields of training (e.g. flight simulators) and manufacturing (e.g. automobile prototyping).

The lack of proliferation of virtual environment applications can be attributed to the challenges involved in creating virtual environment applications [8]. Authoring tools for creating such applications are so complex that currently only experienced programmers can use them [43] – interfaces to virtual environments are considerably more complex and challenging to design compared to interfaces of conventional desktop applications [10]. We believe that this has been one of the key factors in determining the success of virtual environment projects: for virtual environments technologies to become more successful would require that more users of a non-technical nature be able to develop virtual environments. Complex authoring tools exclude users such as artists, architects and other domain experts, from creating their environments as their interaction is mediated through the programmer's interpretation.

The aim of this dissertation is then to look at a method for creating a system that, firstly, empowers the non-programmer by providing non-technical users with the ability to develop their own virtual environments, and secondly, to allow experienced users, or novice users gaining experience, to develop more complex applications. In other words, our system would allow applications to be created by users with a limited amount of experience but still allow users with more experience to create more complex applications.

1.2 The CAVES Project

The CAVES Project (Collaborative African Virtual Environment System Project) [58] is a multi-party project aimed at creating advanced software tools and methods for developing local content for delivery on a cost-effective, interactive, communicating platform.

The CAVES project aims to apply virtual reality research to various applications within the African continent. The process that is to be used by the project involves, firstly, a *methodology* that will allow multi-cultural, collaborative virtual environments (CVEs) to be developed and, secondly, *tools* that can be used by non-technical people to author these virtual environments on a low-cost platform.

1.2.1 Objectives of the CAVES project

Three main objectives have been laid out by the CAVES project:

1. To discover the aspects of collaborative virtual environments that makes them effective. This includes the environment itself and also the representation of the user within that environment.
2. To build authoring tools which aid non-specialist collaborative virtual environment authors in creating engaging and useful environments, which can be populated with virtual objects and virtual copies of real world objects.
3. To define a cheap hardware platform for displaying and interacting with the collaborative virtual environment, for which we can then produce an integrated software system.

Current research into CVEs focuses mainly on technologies that are being used and little about the content that is conveyed in them. The end product for the project is an affordable, low-cost system that will allow novice users to create interesting virtual environments.

1.2.2 Addressing some of these requirements

This research project is one of many undertaken as part of the wider CAVES project. The research contribution focuses on providing a single-user authoring tool that would enable non-specialist users to develop useful virtual environments (as part of fulfilling the second objective described above).

There are two specific requirements that we will be addressing for the CAVES project:

1. We will investigate ways in which users that do not possess the programming skills necessary to create useful virtual environments, may do so. This will involve building a low-level, prototype of an authoring tool that would later serve as an architectural prototype for the CAVES system.

2. Through the process of creating such a tool, we will provide a basic architectural description of the authoring system, looking specifically at overcoming some of the problems associated with current authoring system architectures.

1.3 Methodology

Providing support in three-dimensional toolkits suffers from problems that two-dimensional toolkits had 15 years ago [36]. A parallel problem to the one we have described above then existed for users wanting to create graphical user interfaces (GUIs). Currently there is more research into GUI authoring techniques than into virtual environment authoring. Much can be learnt by looking at this research and the processes GUI development has undergone; by looking at how we can apply some of the concepts prescribed for GUIs.

Programs such as the spreadsheet have been the most successful applications to be used [37], the primary reason being that they allow the end-user to create programs (by writing formulas and macros) without realising they are programming. We believe that for three-dimensional tools to be successful they must build on this type of success from two dimensional applications.

Overview of our approach

The first stage of our approach was to briefly review several of the concepts involved with virtual environments and with authoring. This was looked at from both a low level, by looking into the architectures and designs of virtual environment systems, and on a higher level, by looking at general design concepts.

Common tools that have been used for authoring virtual environments were then investigated. These tools we categorise into different systems according to the different user support types they target. In doing so, we determine the advantages and disadvantages of each of the different categories of tools.

The third stage in our approach was to define an ideal of the system we are trying to create using the information from the previous two stages. From this analysis, we derived features describing an ideal authoring system for virtual environments.

We then proceeded to build a prototype authoring system, embodying the best of the ideas seen in the analysis phase.

Finally, we evaluate our system based on a comparison with the ideal system that we described.

1.4 Dissertation Overview

Chapter 2 - Background

In this chapter, we review the concept of virtual environments and current research fields within the topic. We then describe the processes involved in creating virtual environment applications. On a lower level, we look at virtual environment systems, their architectures and designing them. We end the chapter by describing the concept of a meta-tool.

Chapter 3 - Survey of Tools

Various tools for authoring virtual environments can be split into several categories: tools for experienced users, for intermediate users and for novice user. This chapter surveys and describes several of these applications in each of the categories.

Chapter 4 - Defining an Ideal Virtual Environment Authoring Tool

In this chapter we gather several criteria from different fields involved in building a virtual environment authoring system. With these criteria, we evaluate the systems that are described in Chapter 3, producing a set of features that we use to describe an ideal virtual environment authoring system.

Chapter 5 - Designing a Meta-Authoring Tool

This chapter introduces our meta-authoring tool. We describe how the various goals for solving the system as a tool for different types of users were considered and implemented into the system. We describe the system on both a low level implementation, describing the components that make up the system, and on a higher level, describing how the components work together to provide users with a means to create applications.

Chapter 6 - Event-Actions and Specifying Behaviours Interaction

In this chapter we describe how different users work to provide a single system. A more detailed description is given on the workings of the different components that are used for creating events in system. We provide a simple example on how the system could be used by different users to specify behaviours using events.

Chapter 7 - Conclusions

We conclude this dissertation presenting those features prescribed for an ideal and comparing them to the system that was presented.

Chapter 2

Background

2.1 Introduction

This chapter introduces concepts involved with virtual environments, focusing particularly on those of authoring. The topics involved are only briefly covered and are used to provide a basis for understanding the concepts upon which the rest of this dissertation is built.

In section 2.2 we look at virtual environments: the research involved at looking to improve the quality and effectiveness of virtual environments; how they are created and common application of use. Section 2.3 looks at architectures of virtual environment systems, how they are constructed and criteria for their construction. Section 2.4 looks at authoring of virtual environments and requirements for authoring tools. In section 2.5 we discuss rapid application methodologies and how they can be effectively applied to authoring virtual environments.

2.2 Virtual Environments

There is no real agreement on a definition for virtual environments. The "loosest" of definitions describes *virtual environments* as being merely a computer-generated, simulated environment. Although this describes a variety of environments, the focus of our research concerns those environments that require virtual reality interfaces. Ellis [19] provides the following, more detailed, definition:

> "*A virtual environment consists of content (objects and actors), geometry and dynamics, with an egocentric frame of reference, including perception of objects in depth, and giving rise to the normal ocular, auditory, vestibular, and other sensory cues and consequences.*"

In this context, a more agreed upon concept of virtual environments includes some form of three-dimensional graphics display representing some real-life or artificial structure or place in which a user exists with a position and a view – in this environment, the user may navigate and interact with the environment.

2.2.1 Aims for virtual environments

Achieving realism has been one of the main aims for researchers in virtual reality. Many factors may attribute to the realism of a virtual environment. Four such important factors are described below:

1. *Visual realism*

   The level of realism that is portrayed in a virtual environment aids considerably in making it a believable environment [32]. Techniques such as ray tracing that are used with professional animation systems produce incredibly life-like images. Unfortunately, providing this kind of level of detail and sophistication is computationally very expensive and requires a great deal of rendering time. A large proportion of the research into virtual environments is in methods to decrease the rendering times of these techniques.

2. *Image resolution*

   This is a factor that is closely related to virtual realism. At higher resolutions, the discrete nature of a display becomes less apparent, with the disadvantage being the number pixels in the image becomes greater. With rendering, there is an overhead in generating each pixel. At higher resolutions this puts a heavier load on the rendering system.

3. *Frame rate*

   To give the impression of a dynamic picture, the graphics system needs to update the display at a certain frequency. To produce a flicker free environment, the system takes advantage of the human phenomenon of "persistence of vision" – the ability to integrate a rapid succession

of discrete images into a visual continuum. In order to achieve this, a renderer is required to produce images of an environment about fifty times a second.

4. *Latency*

   Latency is considered one of the more important aspects for virtual environment in terms of immersive environments [63]. Latency, or lag, is the time between when a user performs an action and when the system displays the result of that action. As latency increases, user's senses become increasingly confused as their actions become more delayed.

Producing a virtual environment system requires that the factors above be considered when planning the target application types the system will be made to support. Visual realism may be required in an environment such as an architectural walk-through of an historical environment where the details of the environment play a vital role. Due to the complexity of creating and rendering visually real environments, there is a tradeoff made with the latency and frame rate. It is important that the correct balance of features be established.

2.2.2 Creating virtual environments

The research that has so far been mentioned forms only a small part of research in virtual environments in which the above has been extensively covered. In contrast, we find that very little research has gone into the authoring of virtual environments. Most of the research that covers authoring is given as a process whereby usable environments may be authored. They provide only a process for creating virtual environments similar to the software engineering process.

Even less consideration has been given to the problem of virtual environment authoring for non-programmers, a process usually considered difficult because of the need for authors to program. Even with the lack of research, this is an issue that is gaining importance with the emerging technology.

Virtual environments are being popularised by their increased accessibility to the public. This is aided by technologies such as the Internet, VRML (Virtual Reality Modelling Language) as well as the equipment for using virtual environments becoming readily available. There is also an increase in commercial interest in virtual environments (such as the virtual shopping mall [52]).

One problem is the fact that authoring tools for virtual environments require some form of programming for them to be made useful. More research into authoring tools for virtual environments is

required for the field to become what user interface toolkits have become today[1].

Creating a virtual world generally consists of two distinguishable steps – modelling the environment and specifying the behaviour (although there is already some research into merging these two steps [64]).

Environment modelling

Environment modelling is the process of populating an environment through object creation and placement. It is the creation of the *static* environment.

Three-dimensional modelling packages, such as 3D Studio Max, are normally used for object creation. Although this is the process of creating the *static* environment, object animations are considered part of this modelling process (we refer to this as *static* animation as it describes changes in the object that are not defined by a behaviour or by the system). Objects in the environment are also not restricted to being visible entities – many systems include sound as being objects in a world.

There are many packages and languages that can be used for modelling a virtual world (essentially the placement of modelled objects). A modelling language such as VRML can be used to specify objects and models within a virtual world. A common hierarchical specification for object positioning in such a three-dimensional world is the scene graph [50].

Behaviour specification

Behaviours are used to describe the *dynamic* operations of the user, the objects and the environment. They define how the user interacts with the environment and with the objects in the environment. It also includes how objects interact with each other as well as defining the general behaviour of the environment (gravity, kinematics, etc.).

Specifying behaviours typically requires some form of programming for them to be useful. Novice users wanting to create virtual environment applications therefore find two main problems with specifying behaviours:

[1] There is a large amount of research and techniques for creating user interfaces for non-programmers. Many of these are based on an artifi cial intelligence technique called *programming by example*. It is a technique for teaching a computer a new behaviour by demonstrating actions on examples. The system records user actions and generalises a program that can be used in new examples.

- *they are difficult to program*: Authoring virtual environments still remains largely in the domain of the programmer [43]. Current systems require that developers possess some degree of programming skills before they can create an environment that contains a significant amount of behaviour.

- *they take a long time to create*: Tools for creating virtual environment applications exist mainly as a set of interface libraries, for example Sense8's WorldToolKit [16]. Even expert users find that it takes a long time to build a fully interactive environment. Overall, developing useful virtual environments still takes too long.

  Those systems that provide for a more rapid development environment are usually specific to a type of application (for example the automotive industry [17]) or they do not provide the flexibility novice users need to create their worlds. Extending these environments requires that advanced users program at an API level.

Modelling tools such as VRML usually provide only simple behaviour and interaction mechanisms. When more complex interactions need to be added, these still require some form of programming.

2.3 Virtual Environment Systems

A distinction needs to be made between a virtual environment, a virtual environment application and a virtual environment system. A *virtual environment* refers to a world with which a user may interact. It includes all objects, sounds, etc. as well as a set of interactions between the user and the objects, and between the objects themselves. A virtual environment provides a set of laws that apply logically to all things that exist in it (e.g. gravity). Any other behaviour (that is not from the user) is regarded as due to the *virtual environment application* [49]. A *virtual environment system* is a platform on which the virtual environment application is run. The virtual environment system controls the drawing, updating and is generally responsible for maintaining the application it runs.

This section provides some background information into the lower-level aspects of virtual environment systems.

2.3.1 System architectures

Virtual environment systems have been implemented in a variety of different ways – most popular are those that are designed as a set of modularised parts. The system developed in [29] presents an

example of this. They have extended the idea of objects within a world to have multiple representations e.g. an object could be represented as some model as well as having an audio representation. Each representation is a separate module in the system and each can be 'rendered' separately to their corresponding output device (a graphics object to a screen or to a head-mounted display, a sound object to speakers or headphones, etc.). If a new type of representation were then to be developed (such as a haptic system), it would then just be a matter of plugging it into the system (as a haptic module).

Another example is the Alice [44] system which separates its high-level and low-level functions. Each is built as a separate module: the high-level module runs the simulations and the interpreter (providing the interface to the user) while the low-level module provides the low-level graphics functions, the graphics database and manages the I/O devices.

These systems will be looked at in more detail in Chapter 3

Virtual environment system components

Building a virtual environment system requires knowledge drawn from a broad range of topics within computer science. These topics include areas in graphics, networking and databases [31]:

1. **Graphics**

   The graphics component is responsible for converting a three-dimensional world into the two- dimensional view the user sees. This view is what is displayed in the visual output. (The graphics component might also be responsible for *rendering* the sounds that the user hears.)

   Several suggestions are given as to why separating the graphics component (also called the rendering engine) from the system would be advantageous – by separation meaning that the component is run as a separate process from the rest of the system.

   (a) Splitting the graphics component across several processes frees up the main application to do its job smoothly, making the overall environment more immersive [29].

   (b) The component can be changed depending on the needs of different virtual environments. Also, by creating different platform versions, the environment can be made portable across different platforms.

(c) Associating the input device with the rendering device frees the application from performing this task as well as reducing the turn-around time for changes in the environment [24]. This increases the immersive quality of the system.

2. **Networking**

   Networking virtual environments is a topic that has been extensively researched. There exist numerous different distributed models for supporting collaboration in virtual environments.

   Some of these models look at a distributed systems using VRML since it currently does not support multi-users [13, 6]. Many of these include networking infrastructures for use over the Internet using the Internet Protocol (IP) standard [22, 9, 7] (effectively allowing it to increase in size at a later stage).

3. **Databases**

   In a collaborative virtual environment, objects need to be shared between users. If one user moves an object, this needs to be reflected in all the other users' environments. The database typically contains the geometric representations and attributes of the objects within the environment. It is used to keep track of these objects and manages synchronisations when objects are referenced or altered at the same time by different users (a typical database problem that can be solved using database techniques).

   There are several ways in which the database can be implemented and is closely linked to the way in which the networking of the system has been implemented. Every user may keep a copy of the database and send (via the network) any changes that it makes [22]. A different method is to have a single copy of the database (kept on a server machine), which every user would then refer to. Each method has its own advantages and disadvantages.

   Bangay *et al.* [4] have implemented their database as a *shared memory space*, a data-structure that is publicly accessible to all processes (or users). A virtual shared memory manager has the task of managing access requests of the various processes within the world to the data in the shared memory. The manager would then also proliferate the virtual shared memory data across the application topology.

2.3.2 System constituents

On a more abstract level, virtual environment systems consist of several main components which have been built using the graphics, networking and database concepts discussed above (see Figure 1). Although each is separate in concept, they form a common set of traits that can be seen in all systems. How these components have been integrated to work as a single system, however, is different for each system implementation.

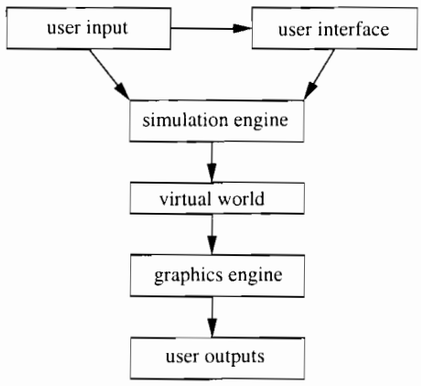

Figure 1: *High-level virtual environment components. Virtual environment systems can be seen as being made up of several distinguishable, high-level components, each of which exist in many different systems. The flow of the diagram shows the flow of information from the* user input *to be reflected as* user outputs.

- **User inputs**

  User inputs in this context refer to the transmitting of information from external hardware into the virtual environment. This information would consist of cues for navigation, for example a set of tracking devices which would be used to replicate the user's actions in real life into the virtual environment, or for interaction, for example a dataglove or pointing device.

- **The user interface**

  The *user interface* provides the mechanism whereby the user can navigate and interact with the environment. This effectively takes information from the *user inputs* and translates it into dynamic actions within the environment. (It refers to the software handling of the user's action, whereas the *user inputs* refer to the hardware side.)

- **The simulation engine**

  The *simulation engine* does the work to maintain the virtual environment. It is responsible for affecting the dynamics of the environment: applying changes within the environment. This includes how the environment responds to interactions by the user.

  The dynamic behaviour of the environment may have been specified as the behaviours and interactions by the *virtual world.*

- **The virtual world**

  In all systems, there exists the concept of a *virtual world.* The virtual world describes the appearance of the environment: all the objects within the world, their colours and textures, sound, lighting, etc. On a lower level, the virtual world is a scene database that contains all the objects and their attributes within the environment. It contains the geometry of all the objects necessary to reconstruct the environment. Depending on the implementation, the virtual world may also define the behaviours and the interactions that exist within the environment.

- **The graphics engine**

  The *graphics engine* is responsible for taking the information provided by the *virtual world* and constructing the three-dimensional environment that it describes. This environment, given the user's position and view, is then transformed into an appropriate output.

- **User outputs**

  The *user outputs* are the outputs generated by the *graphics engine* and sent to the appropriate hardware for viewing. These outputs may be different depending on the type of hardware used. For example, stereoscopic viewing in an HMD requires that two different views be generated: one for the left eye and one for the right eye.

2.3.3 Designing virtual environment systems

There are many implementations of virtual environment systems some of which we will discuss in Chapter 3. Regarding these implementations, many suggestions have been made in identifying and classifying important features for these systems. These suggested features present what the system designers have found, in their experience, to be desirable traits.

The following provides a brief summary of those discussed in [49, 47]:

Multiple application support: The system should not be limited to running a single application, meaning that the system should be able to concurrently run, not only the environment, but any tools, interfaces, etc. that are available in the environment.

Flexibility: Systems should provide support for multiple types of environment applications and should not be designed to support a single environment type.

Separation: When implementing users in a system, it is suggested that these remain separate from the world (as opposed to merely being another object in the world).

Extensibility: The system should be made extensible with regards to the size of the virtual environments it supports.

Distributive: With respects to the architecture of the system, this should be separable where different machines are able to perform different tasks on the same environments.

Configurable: Virtual environment systems should remain *configurable* to allow for newer equipments to be incorporated with relative ease as the hardware becomes available.

Virtual reality support: Virtual environment systems should maintain *virtual reality support*. This feature refers to issues such as the latency and the general performance of the system.

2.3.4 Behaviour and interaction

Interactions and behaviour in virtual environment systems have been implemented in two general ways: as behaviour-based systems and as event-based systems [5, 65].

- **Behaviour-based systems**. In behaviour-based systems, the interactions of the objects in the world are implemented as 'attributes' of the objects. The objects execute their behaviour according to what has been programmed for it. Each object knows what it must do and how it responds to users and other objects. The objects take inputs from the environment, make decisions and act accordingly [5, 45]. The objects may also be restricted by rules the environment imposes.

- **Event-based systems**. In event-based systems, interactions are based on sets of events that occur in the environment. These events are generated by user interactions or changes in the state of objects. (These events are no different from the events used in graphical user interfaces where events are generated, for example, by the user clicking on a button.)

2.4 Authoring Virtual Environments

2.4.1 Designing virtual environments

Fencott [20] presents the following methodology of design stages in virtual environments which form the basis of the process of authoring virtual environments. The list is an adaptation of what was originally presented by Kaur [27]:

1. *Requirements modelling*. This concept parallels closely with the software engineering process. Fencott states that it is here that "*purpose should be clearly established*".
2. *Conceptual modelling*. This refers to the gathering of material, photographs, sketches, sound and video recordings, etc. from the real world that are to make up the virtual world. It also includes techniques such as storyboarding which contribute to the virtual environment builders "getting to know" the environment they need to build.
3. *Perceptual modelling*. This is the process of modelling the intended user's experience of the virtual environment.
4. *Structural modelling*. What is normally produced after structural modelling is a scene graph as well as the behavioural aspects of the environment.
5. *Building*. This is the final "coding" phase. Building refers to the actual authoring of the environment.

Smith *et al.* [48] suggest that the best approach to virtual environment development is to delay the binding between the design and implementation for as long as possible: *"The end-user requirements should be used to drive a pre-implementation design."*

2.4.2 What are authoring tools?

Computers have been one of the greatest automation tools ever developed. Advanced programming techniques and tools have been used to speed up the program development process without compromising performance [2]. Authoring tools are applications that are used specifically for this purpose. They provide an automated means of performing a task that, manually, would normally take much longer. There exist many different types of authoring tools, each one to suited to a different need or stage in a development process.

2.4.3 Toolkits

Applications built today use a windows-icon-menu-pointer (WIMP) model. Traditionally, the word used for application programming interfaces (APIs) that manage graphical user interfaces (GUIs) is *toolkit* [30].

This concept has, however, evolved into authoring not only GUIs but to other fields as well, one of them being virtual environments. The idea of using a toolkit to aid the creation of virtual environments holds the same benefits as it does for GUI application development. The most important of these is that they provide flexibility and an ease of use that allows for rapid prototyping of these environments.

2.4.4 Virtual environment authoring tools

Creating virtual environments is a difficult task. Many toolkits for virtual environments have been created to aid in their design but are still too inadequate for novice users.

There has not been much research into authoring virtual environments and, as such, criteria have not been given on how to even approach such a system. In searching for such criteria, we turn instead to the field of GUI authoring which is matured in terms of the wealth of research it provides. The criteria used in GUI development can be adapted, to create a set of criteria suitable for virtual environments authoring.

Some of these criteria are:

1. Transparency.
2. Instantaneous feedback.
3. Extensibility and flexibility.
4. Scalability.
5. Rapid application design.

Each will be discussed in more detail in section 4.1.2.

2.5 Rapid Application Design

2.5.1 What is rapid application design?

Rapid application development (RAD) is a model that is used in the development cycle of software creation. It was designed to give faster development and higher quality results than traditional life cycle models. No universal definition for RAD exists. It can, however, be characterised [1]:

- as a methodology prescribing certain phases in software development, and,
- as a class of tools that allow for speedy object development, graphical user interfaces and reusable code for client/server applications.

RAD methodologies are used to produce software quickly. The thought bases for RAD tools focuses on using object-orientated and virtual programming techniques. It uses objects and message-passing metaphors emphasising the concepts of re-usability, visual programming and graphical user interfaces.

2.5.2 Evaluating rapid application design applications

In order to assess the usefulness of a RAD application, we need to compare it to an application that users would normally use to accomplish the same task. There are three perceptions of users that are used to measure their *positive usage intentions*. These can be used to predict the actual usage of a RAD application[1]:

- *Relative advantage* assesses the extent to which a developer believes that the technology represents an improvement over prior methods.
- *Ease-of-use* measures the perceived cognitive effort necessary to effectively utilise the new tool.
- *Comparability* measures the perceived congruence of new technology with preferred methods of accomplishing tasks.

With any such model, there are always problems associated with it:

- *Loss of programming power*: Sometimes, greater abstraction and automation comes at the cost of power and flexibility.
- *Performance overhead.*
- *Lack of portability.*

Rapid prototyping systems should use an object-orientated, interpreted language. This would allow interactive changes to be made without having to wait for code to be recompiled [44]. Scripting and visual programming support these ideas.

2.6 Scripting

Scripting languages are high-level programming languages normally used as the glue for plugging components in a GUI together [41]. As such they are not intended for writing full applications from scratch. They are rarely used for complex algorithms and data structures. Their importance, however, has increased with GUI development tools and toolkits.

One important feature of scripting languages is that they are interpreted, as compared to system programming languages, which are compiled. This property makes them useful for rapid program development because the compile time in the development is cut out. In most cases, the fact that they are interpreted makes them easily portable across platforms.

Modern scripting languages have become as powerful as their compiled counterparts. They include all the complex language features that are provided by system programming languages.

Scripting languages

There have been many different scripting languages created, each made to serve a different purpose. Some of the more common ones that have been used in virtual environment development are described below.

Python: Python [59] is an object-orientated scripting language that was initially created to provide scripting for various toolkits. It is a fast growing scripting language that has been used in a variety of applications. Python is the scripting language used in the virtual environment system Alice [14].

Tcl/Tk: Tcl and Tk are two separate packages used together to develop and use GUIs [40]. The scripting language itself is Tcl. Tk extends Tcl by providing a set of commands for building GUIs. Dive [12] uses Tcl/Tk to provide interactions within their collaborative virtual environments [21].

JavaScript: JavaScript was initially a scripting language introduced by Netscape for use over the Internet. It is a scripting language based on the Java programming language. The advantage of using JavaScript is that it is already extensively used on the Internet for creating web pages. VRML [11] uses JavaScript in order to provide some of its more complicated interactions.

2.7 Meta-Tools: Tools for Tools

There exist many different tools for different purposes. Creating tools, however, is difficult and requires considerable effort [34]. An ideal solution is to have some kind of *meta-tool* – a tool for creating tools. One such system, the Garnet toolkit [35], essentially a toolkit for creating GUI toolkits, looks at making high-level graphical tools easier to create. This problem, though, has as yet not been extended to virtual environments.

A virtual environment meta-authoring tool

Using guidelines from research in GUI authoring, we can apply this concept of meta-authoring to a virtual environment meta-authoring tool.

The following is a list of some of the requirements issues that have been raised:

- *Novice users should be able to create useful environments.* The difficulty in creating virtual environments is specifying behaviour in that it requires some form of programming skill. A work-around for this problem requires the authoring tool to provide an interface where the behaviour can be specified in a non-programming fashion. This is the key requirement for the project and a basis for specifying the rest of the project.
- *A virtual environment that allows interactive editing.* There is an architectural requirements that the authoring tool allow developers to create their environments interactively inside the virtual environment itself.

- *The system should be modular.* Modularised systems provide a solution to portability. Concepts such as shared memory spaces and data-flow ports work well with visual programming languages. Each module itself should be a separate entity and should depend on another module.

A generic authoring tool for creating virtual environments would be an impossible task, however. Virtual environments have many different applications. There are various issues that also need to be considered:

- The system should not cater for only one kind of user (an inexperienced user vs. advanced user). It should be able to cater for both types without restricting the other type.
- Novice users should be able to progress to higher levels without having to change the system and without losing the power that can be gained by higher-level users.

These concepts are discussed in more detail when we talk about an ideal authoring tool in Chapter 4.

Chapter 3

Survey of Tools

In this chapter, we perform a survey of different systems looking at different tools and implementations that have been used for virtual environment creation. Our aim is not to provide an exhaustive list of systems that have been created for virtual environment creation (perhaps an impossible task), but rather to get a variety of systems that will provide us with the types of systems and ideas that are common in many such systems.

3.1 Classifying Tools for Virtual Environments

Different types of tools have been created to help users develop virtual environments, each varying in complexity and sophistication. Before we begin our survey, it is necessary that we first categorise these tools. Doing so allows comparisons to be formed more easily between tools that fall within the same category by extracting and looking at similar features and how they have solved common problems.

We classify these tools by looking at the types of users they support (by looking at the level of the tool provided for programming or specifying the behaviour in the environment). That is the level of sophistication they seem to provide their user base with respect to a tool's ability to create virtual environments. Three different types of user support might be classified as follows:

1. *Tools for experienced users.* We define experienced users as having an ability to program in a low-level language and as having low-level concepts of at least one aspect of virtual

environment programming available to them. Examples of these aspects include the ability to program three-dimensional graphics, networks, databases, etc. Experienced users generally work with tools in the form of toolkits and low-level APIs to create their applications. The virtual environments they create are usually specific to a problem they were built to solve, e.g. a visualisation environment.

2. *Tools for intermediate-level users.* We define intermediate-level users as being users that have a fair amount of programming experience but are perhaps not specialised in any particular field. For our purposes, we focus only on those that have little or no experience in virtual environment programming: we refer to these users as non-virtual environment programmers (or non-VE programmers). Non-VE programmers generally would use tools that provide scripted solutions for generating applications; tools that hide the details of programming the graphics or networking of an application even more than an API would.

3. *Tools for novice users.* We refer to novice users as those having little to no programming experience. They form the largest of the three groups of users; usually experts in their own fields that require a solution to problems using virtual environments but not possessing the skills necessary to create it themselves.

3.2 Tools for Experienced Users

3.2.1 Toolkits and APIs

Graphics toolkits were created to increase the speed with which experienced users could create applications. Not in the least ideal for novice users, these toolkits are used to abstract the difficulty of programming graphics applications and the hardware that is usually associated with it. Although the term was initially used when referring to GUI authoring, the concept has extended to including graphics application programming.

There was a need in the graphics community to standardise the drawing of three-dimensional graphics, not only in software, but also in the hardware that was used to support it. One of the first packages to abstract the fast drawing of common primitives and manipulation of three-dimensional graphics were the OpenGL [38] libraries. These libraries form one of the most widely used APIs for developing three-dimensional applications. It is a multi-platform, graphics standard that has gained much favour with experienced graphics programmers. The API provides a set of functions

for performing common, low-level graphics routines that can be freely used to build up 3D environments. As these functions provide routines for specifying graphics primitives, they do not restrict the sophistication of what can be created using the toolkit.

Although having these low-level routines available provides an advantage for some, for most application programmers OpenGL may still be too low-level to work with. The toolkit provides the set of routines for using graphics primitives for which the toolkit has been optimised. With more complex scenes, however, writing good OpenGL applications demands that the users understand the intricacies involved with graphics programming as well as the technical, stack-based model OpenGL implements as its system. To make the task of writing OpenGL applications easier and to provide some of the basic framework required to create OpenGL applications not directly supported by the toolkit (such as its non-support of a windows manager and event handling) tools such as Glut and, on an even higher level, OpenInventor were created.

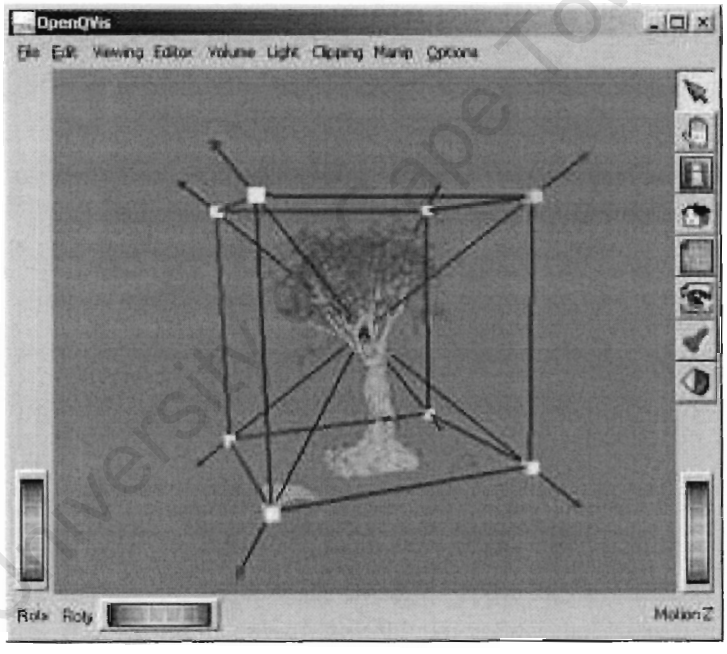

Figure 2: *The OpenInventor system. This image of an OpenInventor application shows an object with a bounding box that can be used to interact with the object. For example, control boxes on the corners of the bounding box allow the user to rotate the object. The widgets on the side bar also allow the view of the model to be controlled.*

OpenInventor was designed as a toolkit for 'rapid prototyping of interactive, 3D graphics applications' [50] and is, simply, an object-orientated interface to OpenGL. It provides a set of higher-level 3D graphics routines that make the creation of complicated models easier by wrapping OpenGL functions and providing a scene graph implementation to represent its environments. (The OpenInventor file format forms the basis for VRML.) The system was designed to be fully extensible by allowing programmers to add new objects and operations through the use of subclassing and callback functions. As with all such toolkit APIs, OpenInventor was built on OpenGL and may provide too high an abstraction level for those users wanting to use control specific OpenGL parameters.

The ideas that were used in OpenInventor have been passed on to many other toolkits that are used to perform similar tasks, such as the use of scene graphs and providing functions that can be used to manipulate them. OpenInventor also provided several novel interaction concepts that were used to interact and manipulate objects within environments (see Figure 2).

With interest turning towards virtual reality solutions and the hype that has surrounded it, development of toolkits specifically made for building virtual environment applications became more popular, with the commercial sector taking an interest. Some of these toolkits are described below.

Commercial Toolkits

Probably one of the oldest commercial toolkits is Sense8's WorldToolkit [16]. The toolkit provides the application developer with a high-level, cross-platform API in which to develop a virtual environment, in a similar way to OpenInventor. The toolkit specifically provides all the supporting features necessary for building an immersive virtual reality: support for a large variety of virtual reality devices, spacialised sound, HMDs, etc. It also supports collaborative environments by supplying functions for networked, distributed, multi-threaded and multi-process applications. It is also one of the few toolkits that has support for CAVE-like, immersive displays.

dVS/dVise [25] are two components by Division Ltd. that are used to create environments. dVS is a low-level toolkit similar to those so far described. dVise is the high-level component that allows users to implement their environments in a text file similar to VRML. The text file contains references to models and describes behaviours and constraints. Audio references can also be used and can be attached to objects. The low-level toolkit provides an API with which the developer has more control in implementing new interaction methods and features not supported by dVise.

Research Toolkits

The research community have for a long time needed to develop their own virtual environment systems to perform their research. Using commercial toolkits for this purpose is expensive and problems can often arise when using them. With commercial toolkits, it is often not possible to obtain the source code, making it impossible to access and modify the internals of the toolkit if some required feature is not supported. For this reason, many researchers have taken to developing their own toolkits instead of having to rely on commercial products. As a result, literally hundreds of such toolkits have been created. We present only a few such toolkits below.

MR Toolkit [47], from the University of Alberta, is one of the earliest non-commercial toolkits that was developed. The toolkit was built to investigate different architectures for support of virtual reality hardware and to reduce the amount of programming time that was necessary for the development of virtual environment applications. To achieve this, the system is conceptually split into four components: a master process, a server process for input, a computation process, and a slave process for output. The advantage of splitting the processes is to allow the computation process, which may be more intensive, from interfering with rendering and the input speeds that make for immersive environments.

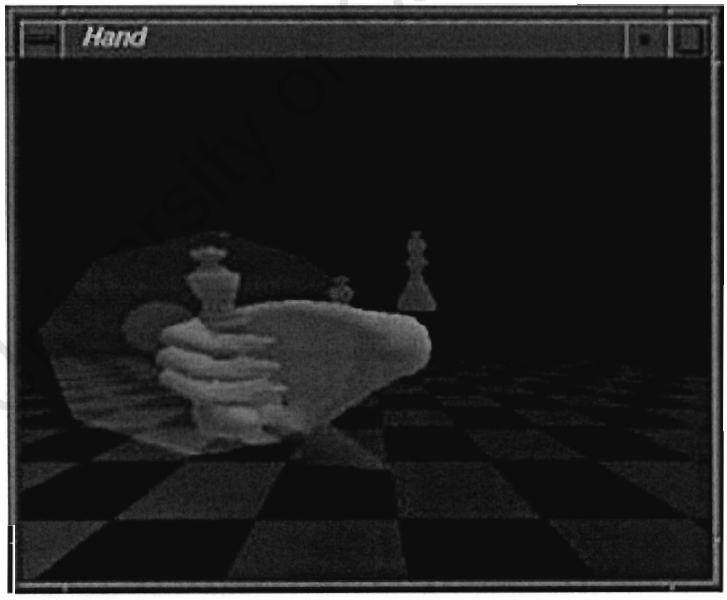

Figure 3: *The CoRgi system. The image here of the CoRgi system shows a representation of an interaction device in a virtual environment which is used to manipulate objects.*

RhoVer [4], and more recently CoRgi [57], are systems for rapidly developing virtual environments and are used primarily as test beds for investigating virtual reality. Applications on the system are implemented in Java [56] and are natively compiled for optimisation. The CoRgi module uses two methods of distributing data amongst its modules: the first is a virtual shared memory mechanism with a streamed method of message passing between its modules; the second method uses a data streaming technique that allows for faster and more direct communication techniques. One of the aims of the CoRgi architecture was to allow for different interaction devices (an example shown in Figure 3) to be quickly added to the system but with minimal changes to the system itself. The system provides several input modules for capturing and processing raw data from the input device and passing a relevant stream of information to those modules that would use the data.

Modular Architectures

Planning for the needs of future requirements such as scalability of environments and the need to cater for new hardware or interaction devices that may not have been invented yet, is of importance if the system is to remain flexible for future use.

AVIARY [49] was one of the first projects that aimed to create a generic virtual environment system architecture based on criteria they had found lacking in many systems. Their motivation for the work was that other interfaces were difficult and unnatural to use. They argued that many systems lacked the ability to provide diverse applications; it is pointless to support features only required by a single application area. Their architecture was designed to support a large-scale environment that would be "demanded by future applications". The focus of their work has been on providing a set of criteria that could be used for developing large-scale environment systems and providing general virtual environment systems. It was important that their system support a broad range of different virtual environment applications [61]. The proposed solution was to implement a modular system that would allow this. Since then, many systems have adopted the idea of modularisation to design flexible systems.

COOL-VR [29] was designed based on the idea of having multiple renderers to support new types of output hardware that could be added to the system. 'Rendering an object' would then be a process in which representation is given to an abstract object. Each of the renderers was created as separate modules. These modularised renderers allowed modules to be switched without impacting on the rest of the system. The different abstract representations of an object then form part of the scene database. For example, an auditory representation was associated with an object in the scene,

allowing for spacialised sound. A change in the position of an object would be reflected by a change in the position of where the sound associated with it would be emitted.

The SVE [28] (the Simple Virtual Environments) toolkit has been targeted as a system for allowing extensibility with regards to adding interaction devices to the system. They describe their system as having an "open" architecture in that it provides a structure to any extensions in order to make them modular, easy to develop and re-usable. Kooper *et al.* claim that some of the drawbacks of the SVE system are that the system does not allow for easy modification of the different renderers or for the addition of any new renderers. The COOL-VR system described above was designed using the SVE toolkit as primary development environment, as a means to overcome the limitations of the SVE toolkit.

Modularised systems that have distributed architectures, such as the MR-Toolkit mentioned earlier, have the advantage of distributing the workload of the system amongst its modules. Generating worlds modelled after physical problems requires intensive amounts of computational power [49], which can be solved by using such a distributed architecture or with an architecture that allows the use of parallel hardware.

Another such system is DIVER [24]. DIVER was one of the earlier virtual environment system platforms that aimed to create an architecture that transparently distributed the rendering and input processes. The system is presented as a set of C-library modules. Each of the modules is used to provide an ease in manipulating objects in an environment with respect to transformations. The DIVER architecture itself provides a graphics database, the low-level graphics functions and manages input and output devices. Of particular interest is that the system is used as a back-end to other virtual environment systems, such as Alice [44], and supports a variety of hardware for virtual reality environments, such as trackers, gloves and stereo outputs for head mounted displays.

3.2.2 3D games engines

One of the greatest feedthroughs from graphics research into the commercial world can be found in the entertainment and gaming industries. Three-dimensional gaming has become a popular market in the gaming industry, so much so that new games not incorporating three-dimensional worlds in some way are rare, and graphics hardware has been created with the sole purpose of supporting and increasing the performance of 3D gaming [23].

The graphics and gaming programming community have created several game engine products

which are used to drive 3D games and allow for faster developments. Some of these have become available, as commercial packages or as OpenSource projects.

Examples of such OpenSource engines are the Genesis3D engine [55] and the Crystal Space engine [54]. Such engines are usually provided as GDKs (game development kits) and support real-time, three-dimensional rendering environments. The engines provide routines for common procedures and effects as can be seen in many current games, although such routines are not trivially implemented nor were they developed with novice or users with an intermediary experience in mind.

Some of these engines, such as the Genesis3D engine, have been used as tools for virtual environment support and research [26]. Because they implement the superb graphics optimisations required for interactive gaming, and do so already using the graphics hardware that is available, they allow for fast, interactive applications to be created.

3.3 Tools for Non-VE Programmers

As was described in Chapter 2, scripting languages have made the task of creating virtual environment applications much easier because they hide many of the details involved in working with low-level graphics, networking, etc. There are several virtual environment systems that employ scripting languages and this makes them ideal for rapid implementation of applications by non-VE programmers.

One such system is DIVE [22], a "multi-user, scalable network architecture for distributed virtual environments" and is used as a platform for collaborative virtual reality experimentation. Some of its main goals have been in allowing environments to be built quickly and as such have concentrated some of their work on creating a scripting interface that could be used to, firstly, hide some of the networking implementations and, secondly, to increase the speed with which behaviours and interactions could be created in the environment.

Interactions within the system are both *behaviour-* and *event-based.* Each object in the environment may contain Tcl [21, 40] scripts that are executed wherever the object is replicated (defining the *behaviour-based* interactions). These Tcl scripts can also be triggered by events defined by the system (defining the *event-based* interactions). DIVE defines several pre-made events for interaction such as signals, timers and collision detection.

Another system which provides a similar type of scripting scheme is Avango [51], a framework for building interactive virtual environment applications. Rather than using the normal procedural type

scripting languages, Avango uses an interpreted Scheme script [18], a functional scripting language. It uses an *event-based* interaction system by providing *sensors* in the environment. All high-level Avango objects can be created and manipulated through the scripting. Complex and performance critical parts are written in C++ which are then called from within the Scheme scripts.

Alice [43], a "rapid prototyping system for virtual reality", was created to overcome the difficulties of writing virtual reality software. The Alice system provides a set of Python [59] classes for manipulating the objects in its environments. These classes include methods for testing object-based events (events generated through objects). The system also provides a few basic user interaction events, shown in Figure 4, which can be used to spawn scripts (it does not allow new events of these types to be created). Unlike the scripts used in DIVE, the scripts that are used and named in Alice do not use the standard graphics notations, such as the use of transformations to move objects around. Instead, Alice has, for example, dropped the standard X-Y-Z axis for referencing positions and instead renamed it using a Logo-like syntax: forward, left and up, as is shown in Figure 5. These 'easy-to-use' functions provide more transparency with regards to dealing with the behaviours of objects than the DIVE script functions.

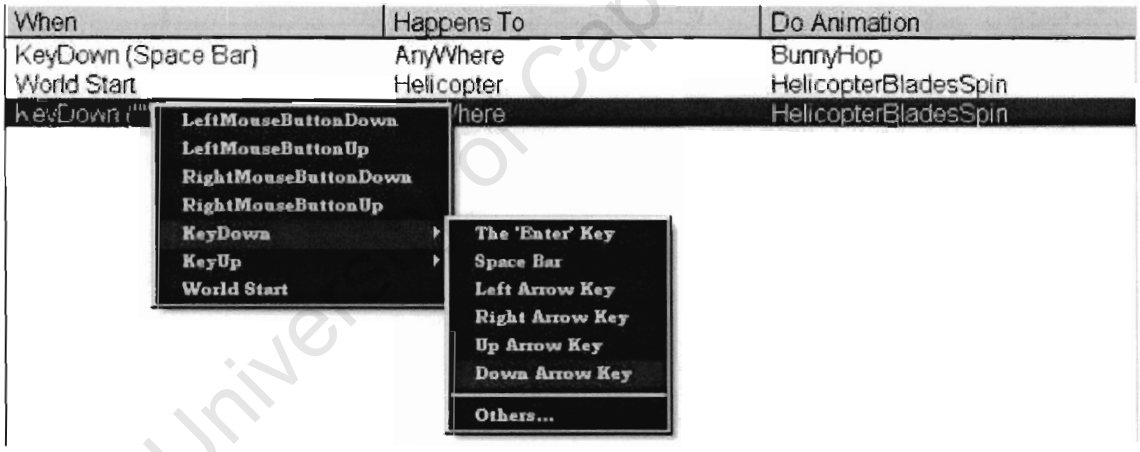

Figure 4: *Alice event table. An example of an event table that is used in the Alice system. The events are selected (as shown in the menus), applied to a specific object (in the "Happens To" column) and the associated animation script is executed (in the "Do Animation" column).*

```
ChickenMove = Chicken.Move( Forward ).Stop()
SnowmanMove = Snowman.Move( Forward ).Stop()
DinosaurMove = PurpleDinosaur.Move( Forward ).Stop()
BunnyMove = DoInOrder( Bunny.Move( Forward ), DinosaurMove ).Stop()

Chicken.RespondToCollisionWith( PurpleDinosaur, ChickenMove )
Snowman.RespondToCollisionWith( Chicken, SnowmanMove )
Bunny.RespondToCollisionWith( Snowman, BunnyMove )

DinosaurMove()

Camera.PointAt( Snowman, EachFrame )


def DinosaurReport( obj1, obj2 ):

        print obj1, "collided with", obj2

PurpleDinosaur.RespondToCollisionWith( Chicken, DinosaurReport )
```

Figure 5: *Alice scripting. An example of some scripting used in Alice. Commands to objects are not made in the usual graphical notation of using X-Y-Z references but instead use commands such as Forward and PointAt.*

3.4 Tools for Novice Users

There are very few tools that are available for novice users with respect to creating useful virtual environments. Tools for novice users tend to be highly simplified and are usually not useful for creating complex virtual environments. We look at several tools in general that have been created with novice users in mind to gather some ideas and principles that can be applied to tools for novice users.

One of the earliest applications aimed at novice users is Logo [42]. Created initially as a tool for teaching children how to program computers, Logo uses a strategy of shifting the programming task from a third person point of view to a first person allowing users to take advantage of knowledge they already possess about spacial movement.

Alice is one of the few systems that support users of differing levels of sophistication. As was mentioned earlier, the whole design philosophy behind the system is that it would be created for novice users. Simple changes such as replacing the traditional axis terms with a more Logo-like reference are examples of how they have tried to achieve this. In terms of providing environments with functionality, the system provides a set of pre-defined functions to create events that novice users

could use to specify their interactions. As these are predefined, there are a limited number of events that are available to novice users. Another interesting change from using graphical terminology to specify directions in 3D worlds to using a more simplified language is with object manipulation. For example, rotating an object 90 degrees around the y-axis would simply become 'turn the object left'. Although the creators of Alice claim that their system is for novice users, they give the assertion that any user needing to create an environment, would learn to program. These would then not be novice users in the sense defined at the beginning of the chapter.

A common means of providing novice users with tools for creation is to provide them with functions that would relate to scenarios they might use. An example of this can be found in AVS/Express [3], a tool used for visualisation applications. Visualisation authoring tools tend to be a specialised type of software that can normally only be created by experienced programmers. AVS/Express has tried to change this in that it is a data visualisation tool "created for both non-programmers and experienced developers". Their software supports both desktop and fully immersive three-dimensional environments. They provide several themes or case study scenarios for common types of visualisations that they have divided into themes such as aerospace, engineering, etc. Novice users would select a theme suitable for the type of visualisation they want to create and be able to apply specific data to form the visualisation they require.

Tools for novice users and the concept of end-user programming have been a focus of several studies in user interface tools.

HyperCard is a tool that is widely used as a graphical user interface authoring tool for non-programmers on the Macintosh system. The system uses a metaphor of cards (a single screen) and stacks (collection of cards). Links can be created between cards and form the interactive component for HyperCard – a form of a graphical programming language. The system was designed in such a way that novice users can create complex interfaces without needing to program.

Garnet is another toolkit for designing and implementing user interfaces. The toolkit comes along with an interface builder that allows new toolkit items to be created and is essentially a tool for creating user interface tools. In the system, they explore different ways in which high-level graphical tools can be easier to create. Most such interface tools provide a means to create user interfaces but without specifying the behaviours of those interfaces. The Garnet interface builder allows most of the user interface to be created graphically and by demonstration without programming.

All these methodologies, although not directly used for the creation of virtual environments, provide us with useful information on how attempts have been made to approach a system that can be used

by novice users.

3.5 Summary

	System	Advantages	Disadvantages
Experienced users	OpenGL	• provides low-level routines for unrestrictive creation of graphics applications	• difficult, takes long to program
	Inventor	• object-orientated graphics programming	• not as flexible
		• abstracts low-level concepts and common routines	
	WorldToolkit	• support for large varieties of hardware	• as commercial toolkits, expensive and not easily modifiable
	dVS/dVise	• one system provides functionality for extending the other high-level component	
	RhoVer/CorGi	• distributed, modular architecture	• concept does not extend to other modules
		• streaming communications between modules	
		• provides an easy method for adding hardware	
	Aviary	• support for a range of virtual environment applications	
	Cool-VR	• concept of an abstract renderer	
Intermediate Users	DIVE	• scripted interfaces	• access to functionality only through the provided scripts
		• rapid authoring of networked environments	
	Alice	• functions providing transparency over complex functionality	• cannot extend for new functionality
Novice Users	Logo	• "first person" programming	
	Alice	• novice-orientated terminology	• still requires some programming
	AVS/Express	• provides common scenario environments	• is not easily extendible to create new scenarios

Chapter 4

Defining an Ideal Virtual Environment Authoring Tool

In the previous chapter we surveyed several tools that have been used for authoring virtual environment applications. The next stage is to provide a methodology for extracting those features that are useful and those that are not, so that they may be collectively analysed to define an ideal system.

The first part of this chapter looks at gathering the criteria that are prescribed by different systems in Chapter 2 for defining virtual environment systems and authoring tools. Essentially, in looking for criteria in the first part of this chapter, we are asking the question, *what* features do we require for an ideal system? In the second part of the chapter, we analyse those systems described previously in order to determine answers to the question, *how* do we go about providing such features and how would we solve some of the problems described by these systems? Finally, we gather all this information into a collection of criteria that can be used to define an ideal virtual environment authoring tool.

4.1 Gathering Criteria

The three major domains that we looked at in Chapter 2 were: virtual environment systems, authoring tools and concepts from user-interface systems. We revisit each here and provide a list of relevant criteria for defining in more detail what was listed in Chapter 2.

4.1.1 Virtual environment systems

Through all the research that has been carried out, there are several criteria that have been formulated for virtual environment systems. In particular, these criteria were developed in research carried out in systems such as AVIARY [49] and in the MR Toolkit [47], and were listed in the designing of virtual environment systems in Chapter 2. These criteria pertain to general properties that good virtual environment systems should adhere to and are described in more detail:

1. **Extensibility and scalability**

   Virtual environment systems should not constrain the size or functionality that can be provided to virtual environment applications. Size refers generally to the complexity of a scene, where various applications require different amounts of object and scene complexity. By functionality we refer to the dynamic operation of the environment; what functions and operations may be used in the environment. Scalability in environments is something that is not often considered in systems (e.g. what is the upper-bound in the complexity of a particular system?).

2. **Separation**

   Virtual environment systems must provide support for distributing an application over several machines or processors. Separating the architecture from the platform allows the use of different machines for separate tasks; for example, one for rendering and one for computations. The advantage of such a system is that it better maintains the 'reality' of the environment, factors of which were described in Chapter 2. A machine which deals only with rendering is unaffected by the machine that deals with user interactions and, in this way, provides a responsive system. By separation, we also talking about architecture that accounts for distributed systems where several users may interact within a single environment but are at physically separate locations. Both these ideas can be provided by the same solution and may be integrated simply into a single application.

3. **Configurability**

   A virtual environment system should not be constrained in size with respect to its ability to accept new types of technologies. There is, as yet, much to learn about interacting with virtual environments and as ongoing research new techniques and technologies are being created. Virtual environment systems need to remain "open" to new technologies and concepts that

may currently not be available. A common example of this is with interaction devices. When a new interaction device needs to be used with a system, the system should be able to cater for it with minimal changes to the system itself. A characteristic of systems that lend themselves to this are those that provide a rich framework that can easily be extended for configuring new devices.

As newer hardware becomes available for virtual environments, the system should allow it to be incorporated with relative ease. Hardware technology for virtual environments has not matured to the point where a standard set of devices can be expected on any particular workstation. Low-level support for devices should be efficient, with minimal lag and the system should allow such devices to be added with minimal changes to the architecture of the system. High-level abstractions for each such device should also be provided.

4. **Multiple application support**

   These refer to any applications that could run concurrently on the system. They range from the tools that are available in the environment to the environment itself. Very often, the environment in which the tools are used to develop an application is different from the environment in which the application is itself run. As a minimum requirement the system should support a method for quickly changing between development environments with as little work required by the user to change between them as possible.

5. **Multiple world type support**

   A virtual environment system should not restrict the range of possible applications it supports. Many virtual environments are application specific and as such, so are their underlying architecture. While some applications may require the use of some feature, that feature may not be appropriate for others. (An example of this is the use of gravity.)

One important criterion which was only mentioned in passing but which we have not explored fully is the concept of environment performance – those concepts mentioned in Section 2.2.1 that lead to immersive and 'real' environments. As was mentioned, many studies have been carried out on how to increase the performance of such systems. Many of the criteria mentioned above lend to these ideas, but are beyond the scope of this study to include. We must keep in mind that these ideas are important but are less critical when we are thinking about providing an authoring system and an architecture to support it.

Many of the concepts described above contain an overlap of ideas and can be condensed into more general criteria. The first three of these refer generally to an architecture that is flexible. The fourth criterion has interesting implications which reach far beyond the scope the authors of these criteria initially intended. This concept has been looked at by several researchers [14, 64]. The fifth criterion refers to systems that are not restrictive in the types of environments they support – many systems are built with a single thing in mind. Examples of these are virtual environments for the automotive industry or for virtual shopping malls.

4.1.2 Authoring tools

Some of the criteria given for authoring tools, previously listed in Section 2.4.4, are:

1. **Transparency**

   To the developer, everything should just be seen as one thread of control: modelling the environment, specifying the interactions, etc. Transparency aids in this, in that it simplifies the development process using some sort of interface that hides the details of the underlying mechanisms and the author need not concern themselves with each detail the interface covers.

2. **Instantaneous feedback**

   When changes are made within an environment, these should be reflected immediately. Some systems require that when any changes are made, viewing these changes requires some sort of 're-compile' or 're-loading' process. Interactive debugging should also be supported where the environment may be debugged within the authoring environment and where changes do not require any 're-compiling' stage.

3. **Extensibility and flexibility**

   If the underlying virtual environment system were to be changed (e.g. an extra component or module were added on), the authoring tool should be able to adapt in the same way: any new features that are made available in the virtual environment system should be reflected in the authoring system and present the author with some method of using it. For example, if a new type of interaction device is supported by the underlying system, the authoring system should allow the author to use it. Authors should be able to design virtual environments flexible enough that, as virtual environment technologies mature, the environments are able to support new technologies.

4. **Scalability**

   Just as the virtual environment system should not restrict the virtual environment in size and in functionality, neither should the authoring tool. Here we are referring to two things: the ability of the authoring tool to support the same scene modelling complexity and more sophisticated scene manipulation tools (such as complex model animations, effects, etc.), and the ability of the authoring tool to support the same set, complexity and sophistication of behaviours that the underlying system supports. The authoring tool should reflect the potential sophistication that the virtual environment system can support, in both a modelling context and in a behaviour-creation context.

5. **Rapid application design**

   Although authoring tools are used to speed up the development process by automating certain tasks, there is a balance to be kept between reducing the speed with which an application can be created and loss of expression with respect to authoring applications of a certain sophistication. The scalability criterion above specifies that the authoring tool should not restrict the potential sophistication of the system, through transparency, hiding the 'apparent' sophistication of the system interface. However, developing environments faster requires that more tasks be automated. Ideally, nothing should be lost through this automation process. Instead, the authoring tool should allow for the ability to prototype applications where the author *intentionally* abandons certain details in order to speed up the initial stages of an iterative development process. However, the tool should allow the iterative process to follow to full completion when a final version of the application is required.

4.1.3 GUI systems

There is currently more research into GUIs and GUI authoring than in virtual environment authoring, and a parallel problems exists with authoring GUIs and virtual environments [36]. We take some of the criteria that have been used for GUI authoring tools and extend them towards virtual environment authoring tools. Adapted from the list in [34], authoring tools should:

1. help design the application given the users' tasks;
2. help implement the application, given a specification of the design;
3. allow the designer to investigate different designs rapidly;

4. allow non-programmers to design and implement applications;

5. allow the end-user to customise the tool.

The first two points refer to allowing the tool to meet the end-user's needs. Point three refers to rapid application design, and has been discussed previously. Points four and five describe an important topic that we will discuss in more detail in the Section 4.2.3.

4.2 Tool Evaluation

In this section, we will use the criteria that were gathered to evaluate those systems described in the previous chapter. In doing so, we aim to achieve an intuitive list of advantages and disadvantages these systems present that we might use to define an ideal virtual environment authoring tool.

4.2.1 Virtual environment systems evaluation

System extensibility

COOL-VR was designed around the fact that other systems did not allow their components to be easily changed or extended. From this they designed a modularised system where each of the components they used was in a portable module that could easily be changed or swapped. Each module was then simply then a different type of renderer. This idea, however, applies only to output devices where the idea can easily be extended to other similar areas.

The WorldToolkit uses the idea of *sensors* for transparently describing different types of input devices and uses this idea as a single method for handling these different devices. Unfortunately, the toolkit being a commercial toolkit, they provide support for only a certain (albeit a large) range of devices, without the ability to create support for new types of devices. The concept of using sensors is another method that can be used in conjunction with that of COOL-VR for providing transparent extensibility.

The CoRgi system, also divided into many different modules, introduces a communication method of transparently communicating between modules. This makes for a flexible model where modules can be added without needing to change the rest of the system. The system was created in order

to experiment quickly with different interaction devices. Their method of using streamed communication between models allows new interaction devices to be easily added by simply creating a new module to process the raw data that would then be sent to those modules requiring the data from the device. This provides a very neat method in terms of defining an architecture of a system for allowing modules to be added to a system. CoRgi have applied this mainly for the purpose of interaction devices but this method is also suitable for extending other parts of a virtual environment system (such as the renderers in COOL-VR).

Separation

Systems such as the MR Toolkit support networked systems, but in such a way that the users are required to know about the details of network programming. In these systems, the authors have to manually set up server and slave processes. Although not suitable for those that do not possess a large amount of networking knowledge, the idea behind the system is to speed up the development process where splitting of processes for optimisation of the environment was concerned, but in such a way as to allow a certain amount of control to still be exercised. DIVER, on the other hand, aimed to completely make the distributing of the rendering and input process transparent, while at the same time providing an environment that was optimised enough to maintain an immersive environment for users. To do so, they provide a set of interface routines for transparently decoupling the application computations from the rendering and input. To the author everything appears as a single thread of control and there is no need for concern about how the system splits the processes.

The DIVE system has concentrated on creating collaborative virtual environments. Their distributed model, a broadcasting system, is made transparent through scripting commands instead of providing lower-level functions to perform this. Creating collaborative environments in DIVE is tidy as the entire focus of the system was built around implementing multi-user applications. DIVE merely provides several functions for configuring an environment and the system takes care of the rest.

Multiple application support

Snowdon *et al.* initially described multiple application support as being applications in the sense that they range from small-scale tools available for use in the environment, to the large-scale activities that encompasses much of the perceived environment. Later systems, however, have incorporated much more into this. All aspects of virtual environment authoring have been brought together

into single systems. Systems such as Blender [53] allow modelling to be mixed with environment creation. The Alice system also allows the environment modelling and specifying interactions to be specified in a single authoring environment.

Multiple world type support

What differentiates environment applications is defined by the behaviour of the objects in the environment and how they respond to different interactions. Gravity, for example, is a behaviour defined by the environment on all objects.

The AVIARY system, as with many visualisation systems, proposes a series of "standard" environment scenarios. The AVIARY system arranges these scenarios in an object-orientated class hierarchy – an environment may then be built using an instance of the scenario. Scenarios may then also be changed and added to the hierarchy. The limitation of this is, of course: when have we provided enough scenarios? For novice users, this would prove to be a difficult task as changing a scenario to suit their need would inevitably require them to add or change the behaviours and interactions of the environment.

4.2.2 Authoring tool evaluation

Lastly we look at some of the systems that were designed for authoring virtual environment applications, in particular looking at ideas that can be gained from systems supporting intermediate and novice users. We look at these in particular, as we are looking for those features that do not require the low-level experience that be expected by tools for experienced users.

Transparency

The Alice system was created with the intention of hiding the low-level details for novice users. For Conway [14], one of the most difficult tasks for novice users not experienced in manipulating three-dimensional graphics when creating virtual environments was for them to manipulate their world based upon the X-Y-Z coordinate system. To move something in a particular direction was referred to as "moving it along a positive or negative axis". In an environment where spacial navigation already poses such a problem, this method of referencing direction makes the task more difficult. Manipulating objects within a three-dimensional environment should be intuitive for a user. Alice

uses the same intuitive method adopted by Logo: the Forward-Left-Up concept. This use of local coordinates hides the complexity behind manipulating three-dimensional graphics in practice.

That their system also provides the entire authoring process in a single system hides the fact that the authoring process is made up of different stages: the object creation stage and the behaviour specification stage. In other systems, where each of the stages in authoring an environment are implemented using different applications, this is made more apparent and authors are required to swap and convert between the different applications and their supported formats. With Alice, these different processes are almost seamlessly put together into a single application, where each component or phase 'understands' the other and there is no need for converting between any types of formats. This is shown in Figure 6 where the graphical environment and the script authoring are used together.

Scripting languages used in systems such as DIVE also provide a means of transparency in that they hide much of the detail required to implement applications in low-level languages, an important feature for intermediate users. They allow various components to be glued together when needed and provide a simple interface whereby the user may configure them for use. An example from DIVE is the interface to the networking component. The complex networking functionality provided by DIVE is made simple through the use of the scripts provided.

Toolkits such as DIVER present the same transparency concepts for experienced users where more refined control is given than is provided through scripting.

The modularised system of COOL-VR also implements transparent authoring to some degree by hiding the details of each of the rendering modules.

Instantaneous feedback

As was mentioned above, one of the advantages of providing transparency within an authoring tool is that the authoring process appears as a single thread of control. The authoring process is not divided into several stages, where changes made in one stage are made immediately apparent in the other stages. Providing scripted behaviour is one way in which this idea can be introduced. As with most systems that do not provide scripted behaviour, making changes to a system results in having to recompile and restart the system every time any changes are needed. Systems, such as Alice, that support scripted behaviour where the users can program and view the environments in the same system provide the advantage of quick turn-around times for changes to an environment.

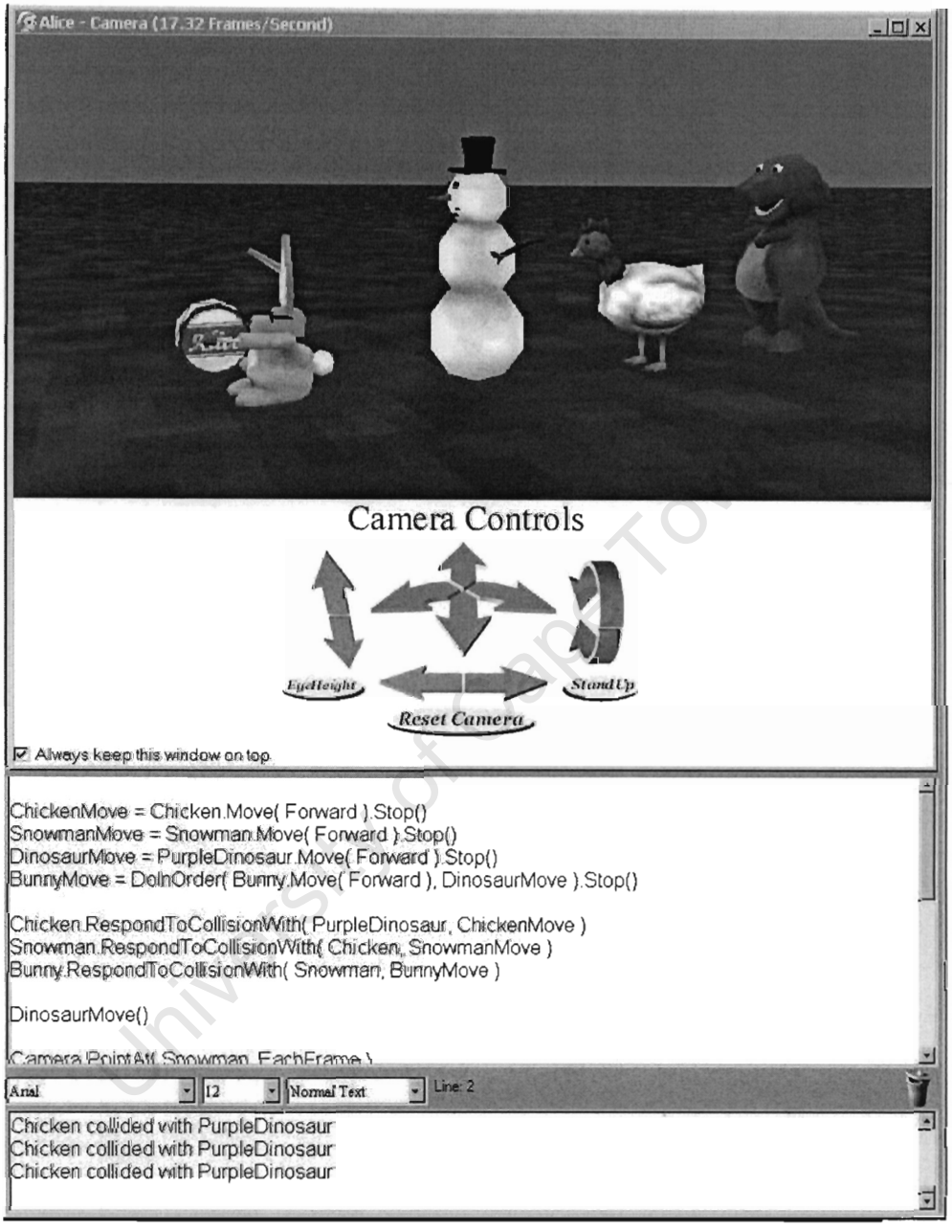

Figure 6: *The Alice authoring system. This screen shot from the Alice system shows the graphical environment and the scripting systems used together. The two are linked and any changes made in the scripting window can immediately be checked in the graphical environment.*

In terms of creating three-dimensional applications, interactive debugging is not very common. For event-driven systems, this can be especially difficult. Tracking what conditions led to an event being generated can be a difficult task and is a great disadvantage for event-driven systems.

Extensibility and flexibility

Modularised systems tend to better support the concept of being easily extendable with minimal changes needed to implement an environment. COOL-VR and CoRgi use a system of communication between modules that allows new modules to easily be inserted into the system and be immediately available to users.

Having such a feature available in an integrated environment such as Alice would be more difficult, however. Such features are normally built into the system and adding new features requires that the entire system be changed. Designing a system in which modules can easily be added onto the system *and* be available in the authoring environment poses a challenge.

Having a system which easily allows new technologies to be incorporated and that allows the authoring tool to maintain and support the system may seem to double the work. This might be avoided by using an extremely modular system where the modules communicate transparently. It would allow the inclusion of addition features, such as the support for new interaction devices, without sacrificing the structure of the system.

Rapid application design

To incorporate rapid application design, it is necessary for the authoring system to reduce the complexity of the underlying system. For most authoring tools, the challenge is to balance versatility and sophistication in such a way that the tool is not overly complex to use (thus lacking in its ability to perform as a rapid application development tool), nor overly simplified (reducing the scalability of the system, as mentioned above). Alice and DIVE use scripting to introduce rapid application design into their systems, providing scripted routines to glue and provide an interface for complex, low-level code common enough to be automated through the scripting. As scripting languages in these systems are key to providing transparency to difficult code routines, they provide a means whereby applications may be created faster. However, the disadvantage to many of these systems is that due to the higher-level scripting, they lose the advantages of the more powerful lower-level programming. We look at this balance in more detail in the next section.

4.2.3 Migrating user support

When we look at what has been learnt with GUI authoring and the trends in GUI authoring, we can apply similar ideas and concepts to the process involved in virtual environment authoring. It is agreed that the added dimension makes this process more complex and suggests that different methodologies be used in approaching a solution to the problem. We aim not to provide the same methodologies, but to rather follow the same trends and goals that the development of GUI authoring has followed and to develop new sets of methodologies for reaching those goals.

Several tools have been created to aid users in designing and creating GUIs. Initially, creating GUIs could only be accomplished through using low-level programming languages. Through the years, more tools were developed to aid users in the design and creation process, the focus moving from experienced users towards allowing novice users to create their own GUI applications. Tools such as interface builders have allowed novice users to construct a GUI interface to almost any application. However, creating the functionality of an application still required that the user learn to program, a task that suddenly increases the learning curve for the novice user required to develop an application to full completion.

Many of the tools that we have looked at follow a similar trend to the one depicted in Figure 7, focusing their target user group to include only a single type of user, i.e. experienced, intermediate or novice users only. Novice users find that application designed specifically for novices allow them to progress only so far before they are required to learn to program to add any more complex functionality to their applications.

In order to increase the user-base of a system (such as Alice), or as a side effect from producing rapid-application systems (such as DIVE, where by using scripting languages to increase the speed at which applications are created they have included intermediate type users in creating environments), different types of users may be supported on a single system.

Moving back to an analogy with GUI systems we find that systems supporting multiple user types are usually based on a perceived model for the types of users of a system, depicted in Figure 7(a), where the majority of the users of a system are novice users and very few users are experienced users. In reality, we find that this is inaccurate as it describes a static model for a dynamic system. The graph shown in Figure 7(b) gives a more realistic model: it describes the dynamic progress of novice users advancing their expertise in a system [15]. We find that system implementations do not take this 'migration' of users into account when they are designed. The figure shows that a user

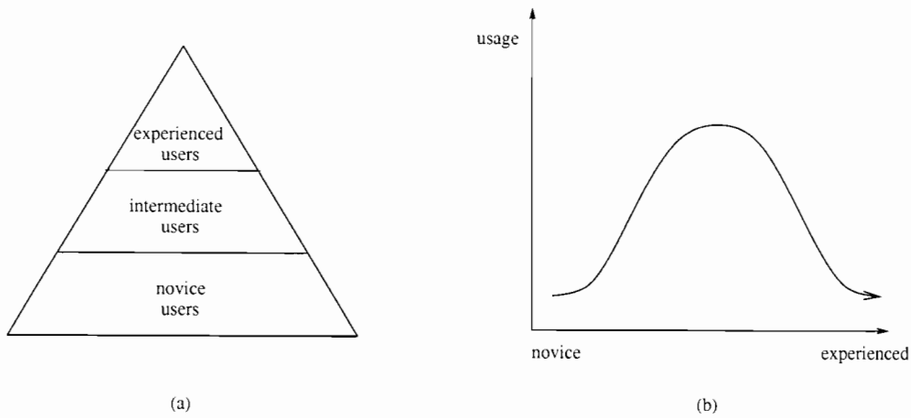

Figure 7: *(a) A static model describing the types of users of a system, with novice users making up the majority. This model shows the usually inaccurate perception about the types of users of a system. (b) A dynamic model that more realistically models the dynamic progression of users. Users of a system spend only a short time as 'novices' of a system.*

on a system spends a short time as a novice of that system as compared to the amount of time they will spend using the system. This implies that applications should be made to support novice users, but more importantly, be made for optimal use by experienced users, *at the same time*. Sacrificing the complexity of a system for the sake of novice users only serves to frustrate the immensely larger audience of more experienced users.

With respect to virtual environment systems, authoring tools for virtual environment applications provide sophisticated support for usually only a certain type of user. The progress of users can be modelled by the graph shown in Figure 8(a) [33]. There is a rapid learning curve attributed to a user's initial working with the system. As they use the system more, their knowledge and ability to use it increases. Once they have reached a 'comfort' phase (the maximum sophistication level they can reach with the system), the curve straightens. This occurs as they reach the sophistication limit of either themselves or the system – the system allows them to progress no further.

In our experience, low-level toolkits such as dVise, Corgi and the MR Toolkit provide an example of the kind of graph that can be shown in Figure 8(a). It is a single rapid learning curve where the user is required to learn everything: the system, programming, low-level graphics concepts, etc. before they can create an application.

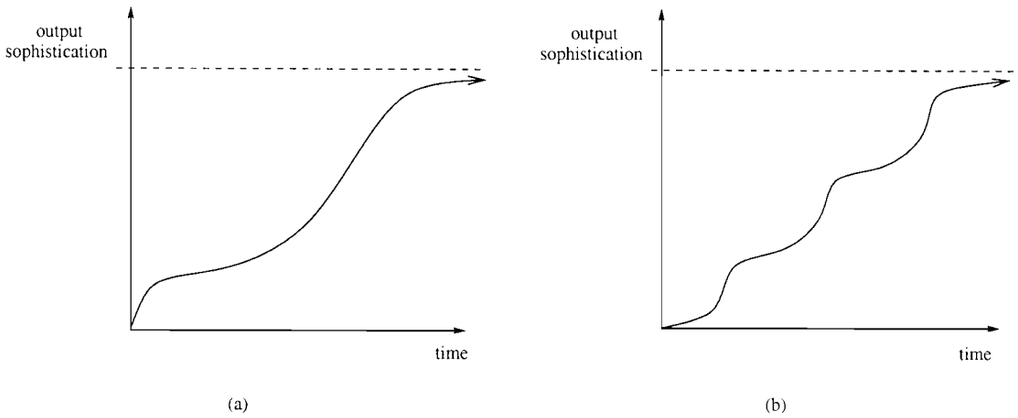

Figure 8: *(a) The graph shows a user's progress with respect to the sophistication of the types of applications they can create. There is an initial steepness in the curve associated with the user first learning the system. As they become more experienced, they reach the maximum sophistication the system can support. (b) Some systems provide multiple stages of development for a user. The graph reflects this by showing multiple stages of learning, each stage effecting a change in the learning curve.*

Systems that have an intermediate user support level, such as Avango and DIVE, represent a two-phase learning system. The first phase for the user is learning the scripting language developed to interface with the system. Learning the underlying API that the scripting interfaces is required to become an expert with the system and is the second phase. Although much can be achieved without it, to progress to using more complex functionality requires that it be learnt, a task normally only suited for advanced programmers as it involves programming with a low-level language.

Alice is one of the few systems that represents a three-phase learning system. Novice users can use the event system to create a particular set of interactions. If they wish to create more complex interactions, they are required to learn the Python scripting language. Conway [14] justifies this by claiming that users wishing to accomplish anything useful will learn to program. Finally, for more advanced users, they provide the low-level Python extensions. This multi-phase learning can be depicted in Figure 8(b).

4.3 Criteria for an Ideal System

Throughout this chapter we have gathered a list of criteria and features of various systems. Many of the ideas discussed provide common concepts, although they describe different kinds and types of systems. In this section we assimilate the gathered criteria, and how these criteria have been used to analyse various types of systems, into a single set of features specific to virtual environment authoring. We use the ideas that we have discussed on virtual environment systems, authoring tools and GUI systems, and generalise them into a set of features that can more easily be used to define the ideal system we are searching for.

4.3.1 Complete flexibility

The concepts of extensibility, separation and configurability all refer to the single ideal idea of being *completely flexible*.

The system would be completely flexible in the sense that, on a device level, it would be able to cope with new hardware and emerging technologies with no changes needing to be made to the architecture of the system. The new technology would be incorporated merely with a change in some configuration settings of the system. It has been suggested that modularised systems work well for this. In this way, the architecture would be flexible in that different modules may run distributively.

On a software level, an ideal system would allow for seamless integration with new technologies. The way in which the system works is not changed and all abstractions that are used within the system to cater for the new technology would remain the same. Any new abstracations that are added would not involve the user having to learn any new concepts. This would ensure that, even though the technology is new, the user would already know how it would be used (implementation-wise).

On an interaction level, anything that is new or added should not change the way the system is used for defining interactions (remaining consistent with the idea above). The system would remain consistent in the way it is used. For example, if a new interaction device were to be added to the system, how the device is used within an environment remains consistent with the way other interaction devices are handled by the system.

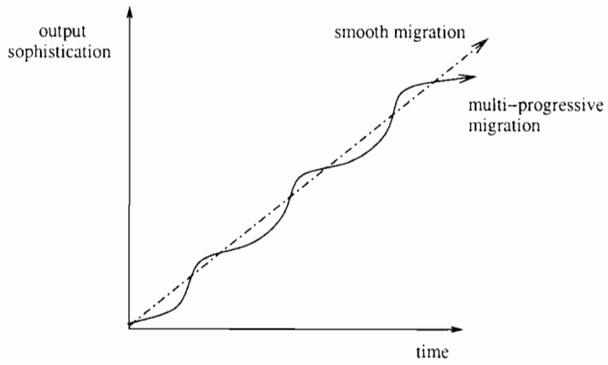

Figure 9: *The graphs shows two different types of user migrations. The solid line represents the types of systems that provide multiple types of progression. The dotted line represents an ideal system that provides the user with a linear learning curve.*

4.3.2 Support virtual environment applications of any complexity

As was pointed out earlier, virtual environment systems tend to limit the types of environments that can be created, depending on the skill of the user. This behaviour was characterised by the graph presented in Figure 8 where an asymptote exists for the maximum sophistication an application created by a user can possess.

For an ideal system, regardless of the experience a user might possess, the system would not place any restriction on the sophistication of an environment that can be created: virtual environment applications of any complexity would be supported. Relative to the graphs in Figure 8, this would imply that no asymptote to the complexity of a system exists, and that there is no limit to the complexity it supports. This is shown by the smooth migration line in Figure 9.

Practically, achieving this ideal requires inevitably reaching the balance between versatility and sophistication. Reducing the difficulty of using a tool requires that the complexity of the tool be reduced through methods such as abstraction and simplification. These methods, however, also result in loss of expression with respect to the sophistication that may be obtained.

4.3.3 Much shortened development time

Currently in authoring systems, developing a useful virtual environment application stills takes a long time, even for an experienced user. The time taken to develop an application should be reduced

to a fraction of what it takes today (in our experience, currently ranging from several weeks to several months from start to finish).

Specifying an ideal for application creation time can only be achieved through some form of comparison. The length of time it takes to create an application varies with the complexity of the application, with an undefined relationship existing between creating more complex applications and the length of time it would take to create it.

We base our ideal on the relationship with other types of research: that of GUIs. So, for an ideal system, the time taken to create a virtual environment application would be comparable to the time it takes to create a GUI application.

4.3.4 Smoothly migrating user support

In the discussion of migrating users, it was pointed out that users do not remain in a single class and that they instead migrate to becoming more experienced the more they use a system. From this, systems should be made to support, not only *just* novice users, but be made for optimal use by more experienced users *at the same time*. Not only is it important that users with differing levels of skills be able to use a system, more importantly novice users gaining experience should be able to advance to using the more complex features of the system. When looking at the graph in Figure 8, between the different stages of learning there is an increased learning curve where, for a user to progress from one to another, the users are challenged to learn new concepts.

It is because of these phases of learning that many users do not progress on to the next phase, regardless of whether or not the system may support more advanced features. So it is not enough, as mentioned in the above criteria, that systems support applications of any sophistication. For an ideal system, this implies that novice users would be able to use it without sacrificing the sophistication of the environments it can output. Similarly, advanced users would be able to use the system without it being so simple that it lacks the sophistication necessary to create the environments they need.

With respect to the difficulty in progressing from one phase to another, there is the added constraint that this migration be smooth. In some systems, provision might be made for multiple stages of progression. An example of this is a system that provides a user interface for use by novice users, a scripting language for more intermediate users, and an API for experienced users. At each stage, however, the user is required to learn something new. The ideal system would provide a smooth migration from one phase to the next, as shown in Figure 9.

Chapter 5

Designing a Meta-Authoring Tool

In the previous chapter we described an ideal virtual environment authoring tool based on criteria and features given by previous systems. We proceed now, in this chapter, to describe how many of those features can be practically applied into a single system. Having defined the three different types of users in Chapter 3, we define firstly what the needs and requirements of each of these types of users are and how these might be provided for.

5.1 A Tool for Advanced Programmers

5.1.1 User requirements

From the perspective of the experienced user, there are few requirements that are needed. As was discussed previously, two of the main requirements that would apply are that environments of any complexity be available for them to create. The other is that development time be reduced.

5.1.2 Modularising architecture

Modularisation has provided many advantages to systems by allowing them to be made flexible and configurable. Many systems have used this modularisation as a standard way of linking different types of hardware and network configurations. Modularisation can, however, be made more useful in a number of different ways.

When looking at different graphics systems packages that are currently available, there is no longer any real advantage to using any one over the other in terms of performance. Most graphics packages rely on an underlying hardware to support them and in essence it becomes the hardware that is important and not the software package being used. A more novel use for modularity with respect to rapid application development is to take advantage of the concept of familiarity to enhance the speed of development: when programmers use systems that they are already accustomed to using, it makes the authoring task easier. With this in mind, allowing authors to use modules they are already accustomed to using, would greatly influence their ability to use something, if they are already familiar with its functionality.

Modularity is not restricted to extending just the graphics system but can also be used for extending other modules in a system as well. This is achieved by creating an interface to a module whereby all other modules communicate with it. The back-end of the interface communicating with the module consists of the module's implemented functionality, for example functionality described by some API.

An overview of the modularised architecture used is shown in Figure 10. Each component in the diagram will be described in more detail through the rest of this chapter.

5.1.3 Communicating with modules

CoRgi uses the idea of streaming data between different modules to make the task of adding modules smoother. Since modularisation allows a system to be more flexible to work with, this concept of modularity was applied to a communications module. This module is used to provide the communications between the modules themselves. Similar to all the other modules used, it provides an interface such as was described with the graphics module. In this way the same modularised concepts are used consistently throughout the system.

To provide an independent method of communicating, we used the idea of having a protocol system. All communications would be processed by a messaging module with the use of a simple protocol. This messaging module, an IMC (inter-module communicater), is responsible for this work.

The IMC module was built as a separate module to increase the flexibility of the system. The module's sole purpose, given a set of interfaces each module presents to the IMC, is to transparently communicate messages to other parts of the system. It forms the backbone to the inter-module communication that takes place between all the components of the system. As a module, how the

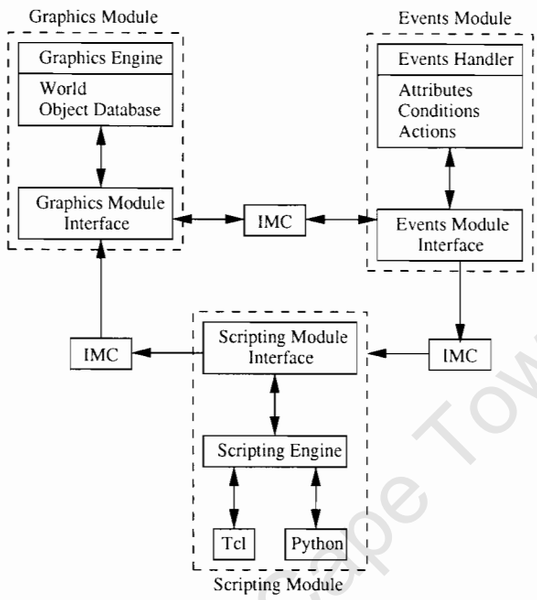

Figure 10: *Modular system overview. An overview of the modular system where each of the modules is connected via an IMC module. The modules present a front-end with which they may transparently communicate with each other.*

communication methods are implemented may be varied, changed or entirely swapped out without it becoming necessary that other modules be aware of any change in the communication module. In this way, the IMC module can be made to communicate on a single computer, between several computers (distributed computing), or on several systems on a network (collaborative computing).

5.2 Graphics

The graphics interface was created as an API through which the graphics of the system could be manipulated. The functions themselves consisted of a minimal set of routines to, on a low-level, initialize, run and maintain the graphics environment.

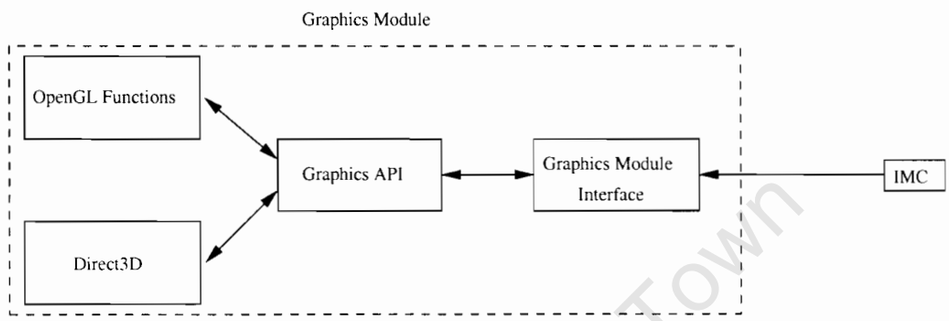

Figure 11: *The graphics module. The graphics module is comprised of several "parts" which are used to translate any requests from the system to the desired graphics package.*

As can be seen in Figure 11, the interface is not only a single layer but consists of two parts to perform the "translation" to the desired graphics package. The module interface itself contains routines to manipulate and retrieve information from a database of objects within the environment. The communication process will be described at a later stage.

5.2.1 Describing three-dimensional worlds

A decision that needed to be made upfront was the format of data storage of objects within the system that would describes the "physical" make up of the world: what the user sees, hears and can interact with. At first a VRML like structured format was decided on. Although the VRML format is useful, it is predominantly used for physically describing environments and provides only a few methods for specifying behaviours and interactions of objects in an environment. It became apparent

that a different format would be required due to the limitations of VRML in terms of how data is stored and coupled with behaviours and interactions. However, as VRML forms a widely used standard for describing graphical scenes, and in keeping with the idea that formats commonly used should be provided for reducing development times, staying compatible with the VRML format proved advantageous. Many graphical authoring tools provide a means to export worlds that are created through them into VRML worlds, and by providing access to these formats allows users to create their worlds in whatever package would suite them. As VRML forms a subset of the graphics format described in the following section, VRML worlds can be easily translated into the system.

5.2.2 Children and siblings

One of the factors that influenced the decision to move to not using a standard format was the opportunity to improve on some of the designs used in these formats. With some graphics APIs such as with Open Inventor, when building a scene graph the order in which you insert nodes into the graph is important. The reasons for this are initially not intuitive and when manually describing worlds, users may experience problems from the side effects of not inserting objects in the correct order into a scene graph. To make the graphics format to be used as simple as possible, we borrow from the standard scenegraph implementation but remove the restriction of having to insert objects into the scenegraph in a specific order: the API implemented has been made 'insert order' independent.

This is achieved by including the concept of sibling and child lists into nodes. Nodes are used to describe some physical aspect of the environment. Siblings are groups of nodes that apply their physical properties to each other, but do not describe some duplicate property already in the group. All nodes in a sibling group are weighted automatically and ordered to ensure that their properties get applied in the correct order. Therefore, attributes of lighter nodes get applied to nodes further down the list. A node may contain any number of children. Each node may only be part of one sibling group and only one type of each node may exist in a sibling group.

The lightest nodes are those that apply to positioning within the environment. The `WORLD` node is used for containing locales of nodes and forms the lightest node. A `TRANSFORM` node is used to position objects and orientate objects within these locales. The next heaviest in this group is a `CHILD` node, described later.

The next set of nodes heavier than those above describe the physical properties of objects within the environment. A `LIGHT` node to provide illumination. A `TEXTURE` and `MATERIAL` node that are

used to describe surface properties of heavier nodes. MODEL nodes describe the geometry of objects to be rendered in the scene. It forms the heaviest of the nodes in a sibling list and is therefore subject to all the properties of the nodes in the same sibling group. Figure 12 provides an example of how the nodes would be added in an environment and placed in the correct positioning in the tree.

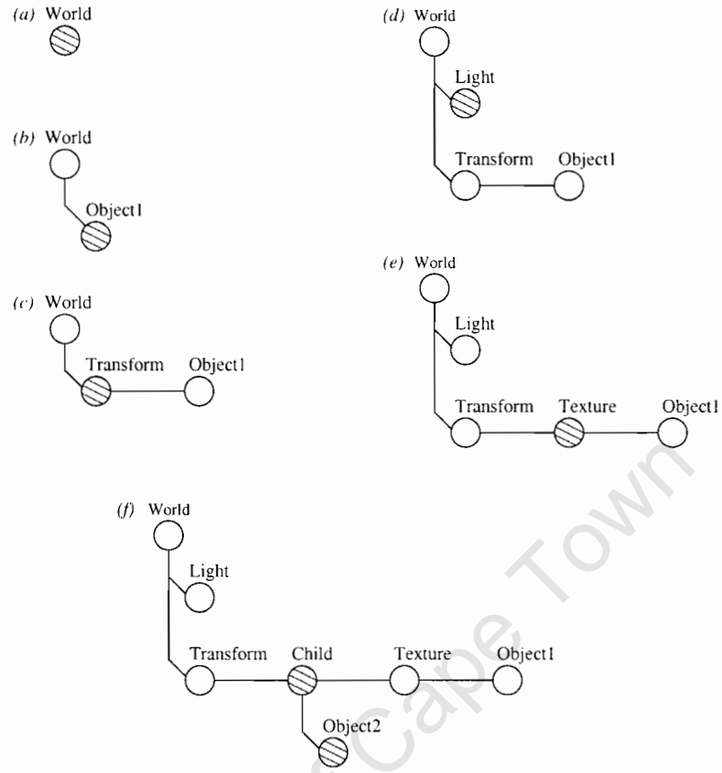

Figure 12: *Inserting nodes into a graph. (a) The environment initially consists of just a World node. (b) An object Object1 is added to the World node as a Model node. (c) A transform to be applied to Object1 as a Transform node is added as a sibling to Object1. As it is a lighter node, it is inserted before Object1. (d) A Light node is added to the World node. (e) A Texture node is added as a sibling to Object1. Lighter than the Model node and heavier then the Transform node, it gets inserted between the two. (f) An object Object2 is added as a child to a Child node added as a sibling to Object1. In this way, it receives the same transformation as Object1. If, for example, the same textures need to be applied to Object2, it could be added as a child of the Texture node.*

As was explained earlier, the idea of creating a scene graph that contained both sets of children and sibling groups was to remove the need to have to insert nodes in a particular order. Although in

some systems having the nodes inserted in a particular order is necessary (inserting in a different order produces a different scene), this ambiguity can be removed by 'rolling out' the nodes – adding children to sibling groups.

5.2.3 Graphics nodes

The following is a brief description of each of the main nodes, many of which are standard in any scene graph:

The World Node

The World node forms the first node in a scene file. The reason for having a World node is to support the idea of locales where many worlds can exist within a single world. Such locales allow optimisations to be performed in their rendering.

It is assigned a weight in sibling group of being the 'lightest', as World nodes provide a starting place for drawing a scene. In terms of defining locales, many World nodes would be children of a single world object (which might be better described as a world universe).

Any node that is added onto a World node would become a child of that node. In formats such as VRML this occurs implicitly where any objects just added become part of a world object. In this implementation, a World node can never have a sibling. For the World node to contain any siblings does not make much sense because, in this implementation, siblings represent a means of providing properties to heavier siblings and children. The World node as a unique type of node provides other means for specifying properties to it, and itself does not provide any properties for its siblings. As the lightest node, it would add no value as part of a sibling group.

The Transform Node

Transform nodes perform the standard functions of any scene graph. The transformation functions from the node apply to all its children and siblings. A single Transform node can perform any or a combination of three functions applied in this order: a translation, a rotation and a scaling.

As is standard with transform functions, the order of the functions to be performed is important. Most modelling packages and formats support this ordering of transformation functions

where similar types of scene graphs are defined. As an example, translating and then a rotating on an object can be achieved in one Transform node by setting the translation and rotation parameters to the appropriate values. However, to perform the functions in reverse order requires two Transform nodes. The first would have the rotation values set, and the second, as a Child node of the first transform, would have the translation values set.

The Child Node

Child nodes serve a purpose similar to standard 'separator nodes' in scene graphs. They exist only as an anchoring point for children to be added. As was stated above, any node may have children. In order to provide some structure (and compatibility), the Child node is formally just used to contain the children of a particular sibling group. Structurally, they are used to group parts of an environment together where various behaviours are to be applied to a single group.

The Light Node

Light nodes serve as illumination sources in the environment. The fact that they are weighted 'heavier' than Transform nodes implies that they are subject to the transformations described by the Transform node. Normally, Light nodes would be made 'static' (unmoving) children of the World node. Sometimes, however, light objects form a source of illumination from a moving object (for example the headlights of a moving car).

Texture and Material Nodes

The Texture and Material nodes serve to describe the appearance of any models that are contained within the same sibling group, or within children that are connected to these or any 'heavier' nodes.

The Model Node

The model node is used to describe all information about the model: the geometry, vertex normals and texture coordinates.

5.2.4 A database of objects

All objects defined in an environment are stored within an object database. The API provides methods for allowing information to be retrieved and changes to be made to the database.

The structure of the graph is maintained by the database (for example, removing the head of a sibling group or inserting a 'lighter' node into a sibling group requires rearranging the graph structure). Interfacing the database allows us to present the database as another module. Different types of data storage strategies were discussed previously, each providing their own advantages. As a system evolves, so might it be necessary to change the type and structure of the database. It also provides an easy mechanism for when the system is to be migrated to using a distributed architecture, by providing a single source where the database may be manipulated.

5.2.5 Naming objects

To be referenced in the database, all objects within the scene graph need to be named. Those objects not referenced by the author do not need to be named (although an internal name is automatically generated for them).

Names that are referenced in the graph need to be unique. These names we refer to as the first-name of an object. The system itself gives the object another name, called the full-name. The full-name is used to partially find the object within the scene graph and to set up properties for sibling groups. The object's second-name is named after the 'heaviest' sibling in the list, unless it is this object, in which case it has no second name.

This technique for naming objects is used for the process of a user selecting an object in some way to manipulate it. Behaviours and interaction can then be applied to groups of objects. This provides an easy method for the author to reference objects in the database.

5.2.6 Authoring applications

For experienced users to create environments, the API and database manipulation provide methods for allowing the objects to be manipulated directly. The API provides an interface for wrapping objects and, in this way, has a similar functionality to that of the Open Inventor system.

5.3 A Tool for Non-VE Programmers

As the section described above is based on APIs and complex programming, the programmer is required to know about the system and the API and how they are integrated together. Typically, this

programmer would also need to have a certain amount of knowledge about graphics programming. The system described above would only be useful to a small percentage of users wanting to create virtual environments.

5.3.1 Scripted graphics

Many systems have used scripting to simplify the authoring process for users. The advantages of using a scripting language over an API are many. As was discussed previously, they fit in well with the concept of providing rapid development cycles. Scripted functions do not require compiling and changes can be made with the results of the change being made instantly available. This ability to make changes to an executing environment is invaluable in the authoring process. The turn-around time for making changes are greatly reduced and ideas can be tested rapidly. Debugging an environment takes less time. APIs no longer provide any advantage over current scripting languages with respects to comparing language features. Programming language concepts such as packages, modules, generality, namespaces and object-orientation are all available as part of scripting languages.

The first choice of scripting language was the Python language. It seemed to be the most popular choice in that it provided plenty of functionality. An important feature of the language is that it provides many routines for integrating its classes with C code. After reviewing the different scripted virtual environment systems [14, 40, 51], it became apparent that no particular scripting language was preferred. It also became apparent that in order for a user to use other graphics systems, they would need to learn new scripting languages and how those particular scripting languages were tied in with those systems. Most virtual environment systems integrate their environment with the scripting language they use. The advantage of this being that they fully utilise the scripting language by binding the functionality of the scripting language into the behaviour of the system. However, this normally also means that there are some quirks about how the language is used: restrictions within the language limit the way in which the scripting language can be used to create behaviours. An example of this is demonstrated in an example piece in Alice. On the one hand they demonstrate the power of the embedded Python language by using Alice functions in lists. On the other hand, they pointedly show that because of the way Alice is bound to Python, using it does not always provide the results expected. As powerful as it is, it does not execute as intuitively as one would think.

5.3.2 Scripting language independence

When deciding on the scripting language that should be chosen, ideally, a user should be able to use a scripting language of their choice. The advantage to non-VE programmers would be to allow them to continue using languages they are accustomed to and not learn new concepts related to the language, and how the language has been bound to a particular system, as was described above.

There are, however, some problems associated with realising such an ideal. Not all scripting languages are suited to performing virtual environment functions. Normally, for any scripting language to become useful to another system, the scripting language needs some way of being embedded within that system. Not all scripting languages are suited for this purpose.

The ideal solution to this problem would be to use an authoring tool that could use and take advantage of any scripting language. A proposed solution on how this ideal could be realised is defined in the next section.

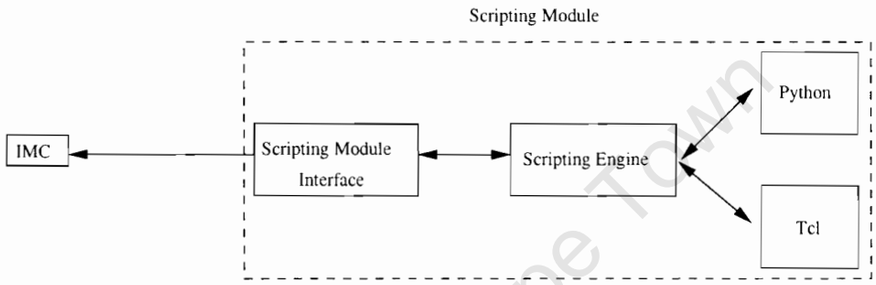

Figure 13: *The scripting module. The scripting module is set up using an interface that allows the module to be scripting language independent. Here, the scripting engine provides an interface for interpreting commands into commands used by a particular scripting language such as Python or Tcl.*

5.3.3 Module architecture

As in shown in Figure 14, each module within the system contains an interface using a protocol to encode and decode information it sends or receives. The IMC module is used to connect these interfaces to transport the data to the destination module.

As the interfacing holds for all modules, the IMC module itself is a layer of interface and implementation. Although in the diagram we refer to a *Graphics IMC module* and a *Scripting IMC module*,

they are essentially the same module, merely different instantiations of them servicing different modules. In other words, a change in the back-end of the IMC module would mean a change in all the IMC modules as all IMC modules are the same and neither the interfaces nor the modules need to be modified. IMC modules differ only in what they communicate or what is communicated to them, which is determined by the module that has instantiated them for its use.

As was stated earlier, each interface describes a protocol that other modules wishing to communicate with it need to adhere to. For example, the graphics module provides services for manipulating the data it possesses (similar to the data accessible using the API). A protocol is established for anyone wishing to access these services and it provides an IMC module. Anyone wishing to use the graphics module need only, using another IMC module, connect to the IMC module the graphics module has provided as a service. For example the graphics module provides service in two basic categories:

- changes to the graphics database, and
- information querying from the database.

Similar services are common to all modules: they provide a means for requesting information and a means of enabling some change in the module.

In this way, some form of scripting language independence can be achieved similar to the way in which graphics implementations may be changed and swapped. Since all communication with the rest of the system is interfaced through the IMC module, the scripting module can be used to translate various scripting languages into a form that can be understood by the system. This scripting language module is shown in Figure 13.

An example of how the scripting language module may be used to retrieve various objects from the environment is demonstrated as follows. The behaviour of something in the environment can be defined by some piece of scripting language. The script is sent to the scripting module for it to be executed. The appropriate scripting parser and engine is selected by the scripting engine and the script is executed. Within the script itself, reference is made to objects within the virtual environment which the script requests. There are a set number of such requests that are provided as a service by the scripting engine (defining the protocol of the system). The specific translation of each type of request for a certain scripting language is provided in the scripting engine. The request is sent back through the IMC to the appropriate module where the information can be obtained. The requested information is then returned, translated and sent back to the scripting language. In this

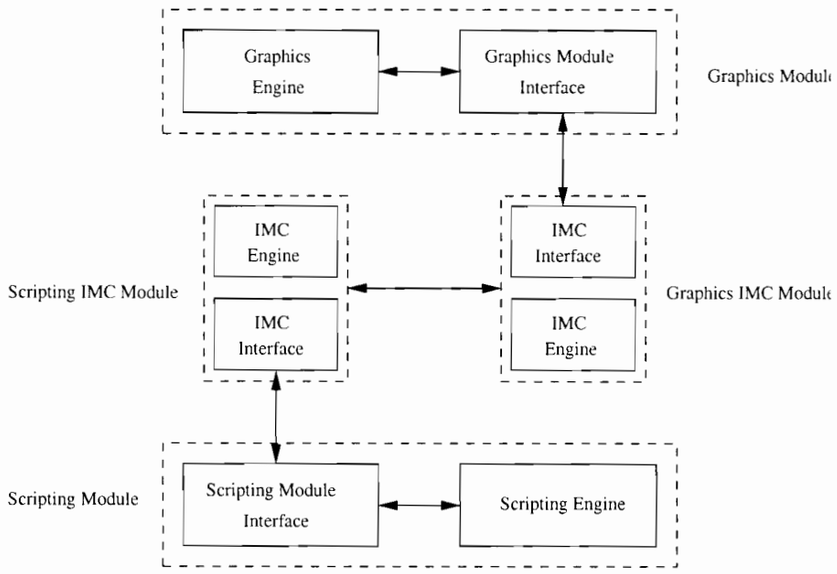

Figure 14: *Interfaced modules. Each of the modules has a front-end interface with which it presents itself which allows modules transparently communicate with each other.*

way, the scripting engine acts as the interpreter for any scripting language to interface with the rest of the system.

5.3.4 Creating environments

The three modules described so far: the graphics module, scripting module and IMC module, provide a system that allows non-VE programmers to create virtual environment applications. As with many other systems, the scripting language is used as a wrapper for the API. This provides an ideal system for non-VE programmers, programmers that we described as not possessing a great deal of knowledge of graphics programming. Using scripting languages, they can focus on programming the behaviours in the environment and not have to worry about the details of the virtual environment.

An analogy to this kind of authoring would be a GUI programmer and someone who would use a GUI editor to add an interface to their program. They would not need to posses any knowledge of GUI programming and they would only need to know enough to program the behaviour of their user interface.

5.4 Implementing Behaviour

In the previous chapter we described how a programmer, not used to programming virtual environments from scratch, would be able to use the system to produce a virtual environment. All interactions and behaviours would be defined by the scripts the programmer would use; these would all be manually inserted into the environment.

Although this defines a *behavioural-based* approach to specifying behaviours and interactions (see section 2.3.4), this approach does not provide an easy way to define exact event-type behaviours and interactions that would make up a significant part of the behaviours and interactions defined in the environment.

In the next chapter we will look at how we use events for defining an event-based behavioural system, one that can particularly be used by novice users.

5.4.1 Events for behaviour

A virtual environment is composed of several static nodes and models which are typically placed in some type of scene graph. Animations and scripts transform the environment into a dynamic world by changing the static configurations of these nodes and models: moving or rotating an object is merely a change in a transform node associated with the object in the scene graph.

An event-action pair can be described as something that happens at a certain time in the environment. Events are composed of one or more conditions that determine whether or not an event has occurred. Consequently, an event-action pair can be decomposed into two distinct parts: an *action* ('something that happens') and a set of one or more *conditions*, which determine the circumstances under which the action should be executed.

A condition can be satisfied by user interactions, or by a change in the property of an object. User events are typically in the form of the user clicking on some object or triggered by the position of the user. An example would be either the user clicking on some virtual light switch or just walking into a room to activate the light.

Zachmann [65] lists a set of requirements for event conditions and actions that, through their experience, allows them to be 'most flexible':

1. Any action can be triggered by any condition.

2. Several conditions can trigger the same action. Actions can be triggered simultaneously.
3. Conditions can be combined by boolean expressions.
4. Conditions can be configured such that they start or stop an action when a certain condition holds for its input.
5. The status of an action can be the input of another condition.

5.4.2 Storing the environment

There are several components that describe a virtual world: the objects and their static animations (describing the modelling process) and conditions and actions (describing the dynamic behaviour). We have found that when building environments, it is cumbersome to have all these elements described in a single file, as many systems do. For this reason, we have split some of these elements and have stored them separately. This ensures that all the information stored for an environment is not cluttered in a single file. It allows the scene to be logically split into its various elements and also promotes the re-use of each element.

The separation of files also suitably describes the separation of objects, conditions and actions. Each one is defined independently of the other, although associations may exist. If some object has a set of *conditions* that are associated with it, they are simply linked by named reference to the file they are contained in. For example, a set of 'light conditions' would be defined that applies to any light object: they all maintain a state defining whether they are *on* or *off*, behaviours describing what to do if switched on, etc. All objects representing a light would then use the same set of 'light conditions'.

5.4.3 Animated behaviour

Animated behaviours are those behaviours which are not defined as part of the model[1] but are instead animations described through a script. An example of this would be an animation to smoothly rotate some object through some angle around an axis over a particular amount of time. Actions that are defined as part of events are typically executed over a single frame-rendering period.

With animations, this becomes slightly more complex due to many scripting languages not supporting multi-threaded scripting where many scripts can safely be executed at the same time. To get

[1] In Chapter 2 we classified object animations (those animations created through a modelling application) as part of the environment modelling process and not as part of the behaviour and interaction specification process.

around this problem, animations are seen as single actions executed over several frames but which hold a certain remembered state between each executed action.

Animations, like normal actions, are also triggered by the event described above and are, to a novice user, indistinguishable from normal actions. The next chapter looks at how these types of events may be used for defining an event-based behavioural system, one that can particularly be used by novice users.

Chapter 6

Event-Actions and Specifying Behaviours and Interaction

6.1 Defining Event-Actions

6.1.1 Attribute variables

Many of the simple behaviours that are created in virtual environments are based on finite state machines (FSM). Of these, many of them require only two states: a light is *on* or *off*; an object is *selected* or *not selected*.

Attribute variables are defined in an environment and are used to describe the properties of an object or the properties of an environment. Their values (or changes in them) are used in generating events.

Since *attribute variables* represent an interface whereby a user authoring an environment may cause change in an environment, all quantifiable values in the system are assigned an *attribute variable* to represent them: objects, their properties, the avatar and the environment can all be manipulated through the *attribute variables* that represent them.

6.1.2 Conditions

Many of the systems that use an *event-based* system for describing interactions usually provide novice and non-VE users only a limited set of conditions on which to test events. What these systems

do not provide is a means to create new or custom made conditions. In our system, conditions are scripted; users can then create new conditions or modify existing ones to suit their needs.

Conditions are tested[1] with the use of scripts and the decision made on whether a condition is satisfied is based on the values of *attribute variables*. The inner workings of conditions are not unlike programming functions in that they accept parameters and return a value signalling whether or not the condition has been satisfied. The condition script signals a '*condition accepted*' to the system to indicate an affirmative; no signal received would be considered a condition failed. Since a condition may sometimes be met on several grounds, a '*condition accepted*' signal may be emitted at anytime during the execution of a condition script.

Since conditions return a boolean value indicating whether or not that condition has been satisfied, conditions may be logically combined into more complex conditions when defining events.

Conditions in the authoring system are declared similar to the way some programming functions are created with three main parts:

```
Header    : conditionName (conditionParameters)
Variables: conditionVariables
Script    : conditionScript
```

The condition name in the header section of the condition is used to reference the condition. The condition parameters are created as *sentence functions*: they are written functions that use redundancy to describe their parameters. An example of this would be:

objectClick: "*When the object (object) has been clicked on (number) times ...*"

(Here, we talk about an object being *clicked* on as an object which has been selected using some pointing device.)

The type of the parameter is described in the parenthesis of the *sentence function*. There are two main advantages to using *sentence functions* that we have found useful: they give a meaningful description of what the function does and they provide a context for the types of the parameters that the function accepts. *Sentence functions* have provided us with a novel method for describing function parameters that allow novice users to easily specify functions.

[1] As events are made up of several conditions, checking whether or not an event occurs requires checking whether or not each condition is met. The phrase 'conditions are tested' implies this check.

The *conditionVariables* section is used to define *attribute variables* that may be used as static 'local variables' within the condition.

Conditions are associated with objects in a many-to-many relationship. A particular object that has a condition associated with it defines the types of events that can be generated with that object. These objects can be seen as offering 'services' in the form of conditions. We associate only certain conditions to certain objects, as not all objects would offer the same 'services'.

An example using time conditions

Frequently when creating a virtual environment, we need to create an event that occurs at a particular time or that would perform some action for a specified duration. The time specified is usually based on a 'wall clock' time and is not dependent on the speed of the system or any external factors.

An example of this is a simulated animation that is required to execute for an exact amount of time. This time period would need to be independent of the speed of the environment, processor, etc. Time in an environment is represented as a single global *attribute variable* that is maintained by the World object. The value of the *time attribute variable* reflects the number of seconds since the start of the environment. Objects wishing to use time in some way take snapshots of the *time attribute variable* and use it to calculate their progress relative to the global time. In this way, a virtual world can be orchestrated using time-orientated events.

```
Condition
        conditionEvery
        "Every (number) minutes and (number) seconds ..."
Attribute Variables
        int time_snapshot
Script
        snapshot = World.getAttrVariable ("time")
        wt = parameter1 * 60 + parameter2
        dt = snapshot - time_snapshot
        if (dt >= wt)
                time_snapshot = time_snapshot + wt
                return "Condition_Accepted"
```

Figure 15: *An example of a timer condition. The structure for defining a new condition is divided into three main sections: the declaration header information,* attribute variables *and a script. In this example a pseudo-script is used to show how the condition is programmed.*

An example of using the *time-variable* to create a condition is shown in Figure 15. The figure shows that conditions are made of three main parts: declaration header information, *attribute variables*, and a script. The declaration information contains the name of the condition and *sentence function*. Each condition may define local *attribute variables* that can be used. In this example, we define an *attribute variable* called `time_snapshot` that represents the 'wall-clock' time at the creation of the event.

Any object that would need to use a timer to define its behaviour in a world would simply reference this condition. An instance of the condition would be created for it.

6.1.3 Actions

Actions are executed by the environment when the system has received a '*condition accepted*' by the associated event. As was mentioned earlier, conditions are associated with objects. Actions on the other hand are completely separate from the objects and their conditions. In other words, the action that is executed on a condition is not related to that condition.

Events form a many-to-many relationship with actions: the same event may trigger many actions and different events may trigger the same action. The same event-action pairs may be defined multiple times with or without the same parameters.

Actions are defined similarly to conditions. They are comprised of an action name and the parameters to the action (given again as a *sentence function*), the local definitions of any *attribute variables* and an action script.

`Header:` *actionName (actionParameters)*
`Variables:` *actionVariables*
`Script:` *actionScript*

Some simple examples of actions are those that manipulate the transformations of objects in the world. A sentence function of an action to rotate an object could be defined as:

turnObject: "*... turn the object (object) by (number) degrees.*"

Action scripts influence the behaviour of an environment by changing the environment's *static* aspects (in Chapter 2 we referred to the *static* aspects of an environment as part of the result of

populating an environment through object creation and placement). Actions can be used to start and stop *static* object animations or change *attribute variable* values that could lead to event conditions being triggered.

Since both conditions and actions use scripts to define their functionality, the entire functionality of an event-action pair is said to be scripted. This allows the run-time changing of event-action pairs to occur. So within action scripts, changes may be made to the event-action pairs in that they may add, change parameters of, or remove pairs from the environment.

6.1.4 Event-action pairs

An event in the system is defined as consisting of one or many conditions and occurs when the logic combining the conditions that make up the event is satisfied. Exactly how the conditions are put together to form an event will be described later.

Event-action pairs are created as sets of conditions and actions and are specified in the following way:

$event_1$ (parameters), $action_1$ (parameters)
$event_2$ (parameters), $action_2$ (parameters)

The same event-action pairs may be defined several times. The actions and the conditions that make up the event remain as separate entities: the action that is executed on the conditions is not related to the conditions, nor are the individual conditions in an event related to each other. Event-action pairs are a many-to-many relationship between the events and the actions: the same event may trigger many actions and many different events may trigger the same action.

6.2 Specifying Event-Action Pairs

As was mentioned, interactions and behaviours in the system are specified through the event-action pairs. The conditions and actions that were described define functions that are used to create these pairs. Specifying a pair is therefore an instantiation of one or several conditions and an action.

On a low level, specifying an event instantiation can be created by providing the condition functions with the required parameters. From the example of the timer condition, the instantiation of a condition could be made as follows:

```
conditionEvery(2, 30)
```

This then describes a condition that would occur every two and half minutes.

For novice users, the *sentence functions* may be used as guidelines as to the type of values that should be used as parameters. An example is given in Figure 16 where the *sentence function* is presented to the user in a simple GUI of text boxes where the user may enter the values for the parameters.

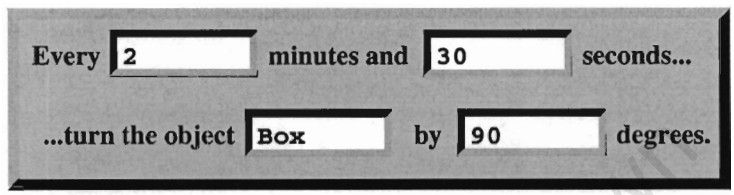

Figure 16: *The* sentence functions *presented in a graphical user interface (GUI). Text boxes allow users to enter the parameters for the functions. The top sentence in the GUI box represents the* sentence function *for the condition, the bottom for the action. This GUI dialog would represent the user statically creating or editing a condition.*

The same process can be used for specifying actions. The function

```
turnObject("Box", 90)
```

would describe an action that rotates an object named "Box" by 90 degrees (around the y-axis). All the objects that are referenced are named as strings. This ensures easy compatibility with the scripting language that is being used to implement the behaviour.

To emphasise the fact that the conditions and the actions are incomplete parts, we use ellipses ('...') at the end of a condition's and the beginning of an action's *sentence function*. We can then represent an event in a readable format to the user as the *sentence functions* with the appropriate parameter values inserted:

> Condition: *"Every* **2** *minutes and* **30** *seconds ..."* Action: *"... turn the object* **Box** *by* **90** *degrees."*

or simply as a single sentence as was represented in the GUI in Figure 16:

> Event: *"Every* **2** *minutes and* **30** *seconds, turn the object* **Box** *by* **90** *degrees."*

The technique above of instantiating conditions and actions is a static method that the author specifies (this is usually the initialising events at the start of running an environment); they may also be dynamically created or changed through the condition and action scripts.

6.2.1 Combining conditions to form complex events

Events are composed of one or many conditions and may be combined in a boolean fashion where a *condition accepted* can be represented by the boolean value `true` and the no *condition accepted* response as `false`. In this way, all combinations of events may be built using a boolean logic.

An example of combined conditions to form an event could be made as follows:

objectClick: "*When the object (object) has been clicked on (number) times ...*"
AND
[

objectColour: "*When the colour of the object (object) is (colour)...*"
OR
objectColour: "*When the colour of the object (object) is (colour)...*"

]

The event could be instantiated into one which occurs whenever some **object** has been **clicked** on by the users, and the object is either **red** or **blue**.

6.3 Virtual Environment Behaviour and Interaction Creation Tool

In the previous sections we described the workings of events within the system. In this section we will briefly discuss how these concepts may be used more practically by authors for specifying interactions and behaviours.

6.3.1 Grouping conditions

As *sentence functions* describe conditions and actions used to specify behaviours and interactions, in a tool-set of such scripts, there may exist many of these conditions and actions required to build the

event-action pairs in a complete system. In a library of conditions and actions, users should easily be able to find the condition and action they require to specify an event. In the system, provision is made for the author of the scripts to categorise and group (and subgroup, if necessary) conditions and actions.

Creating a new event starts with specifying the condition function(s) and then specifying the action function that together make up the event-action pair. Since conditions are associated with the objects they are used with, the process of creating a new event starts with specifying the object. From there, a selection of condition groups (and subgroups, etc.) that is associated with that object may be selected. Finally, the required condition is selected.

In this way, the task of searching for a particular condition in a potentially large library is made easier through the grouping of conditions. Those conditions not object-related are grouped and connected as part of the World object (representing a condition which is associated with the state of the environment). An obvious concern in large libraries is that the complexity these menu hierachies may grow to a point where it effects usability. Various techniques, such as hiding those groups less commonly used, may be applied to alleviate such problems.

As an example, specifying a condition to check for a mouse click would follow the path shown in Figure 17.

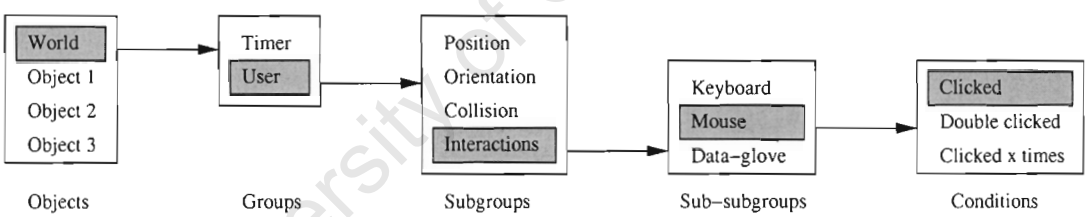

Figure 17: *Specifying a new condition. The first step involved is to select the object associated with the condition. The selection is refined through the groups and the subgroups, etc. to which the condition belongs. The last step is to select the condition from the list of conditions in the group.*

6.3.2 User interaction and object selection

A user interacts with an environment through some form of interaction device. These devices can range from a mouse to a data-glove to a set of trackers, as described in Chapter 2. The author of a world would need to define how the environment reacts to input from these devices.

In the case of object selection, behaviours need to be defined for when a user selects an object. To maintain consistency with the design of the system, all input data received through the devices are stored in *attribute variables*. Interaction events can then be created through changes in these *attribute variables*. The modular approach to handling devices in the system, described in Chapter 5, makes it easy for new devices to be added and for the condition scripts to handles these new devices as all interactions with interaction devices are maintained through the *attribute variables* by the module interfaces.

For the example of a pointing device such as the mouse, the World defines *attribute variables* for pressing and releasing the mouse buttons for selecting objects, as well as for keeping track of the movement and positioning of the pointing device. For each object in the world, an associated *attribute variable* is created to indicate whether, given a set of screen coordinates, it is currently being selected. This may include many objects at once, as in the case of hierarchies of objects. Which hierarchy is then to be manipulated is defined by how the hierarchy of objects was labelled for reference.

These associated *attribute variables* may then be defined for any type of input device. The values of these *attribute variables* can then be used to generate events. Action scripts may also be used to manipulate these values for defining behaviours with them.

6.3.3 Non-programmers specifying events and actions

The simple method of instantiating events and actions is achieved through first specifying a condition and action pair (as demonstrated in the example in Figure 17), and then inserting the required parameters for the functions (as shown in Figure 16). Through the use of this instantiation method, interactions can be created that do not require programming and is thus ideal for novice users.

The condition and actions that have been provided are diverse enough to create a wide range of applications varying in sophistication from simple simulations to complex prototyping environments. All FSM-based environments can be created without programming in the system (for example a game of tic-tac-toe with a single player versus a computer).

6.4 A Tool for Multiple User Types

In this section, we show how the meta-tool can be used as a tool through the various stages in the development of a virtual environment. We will present a simple example to demonstrate this and how different users would use different phases of the meta-authoring system.

6.4.1 Novice user authoring

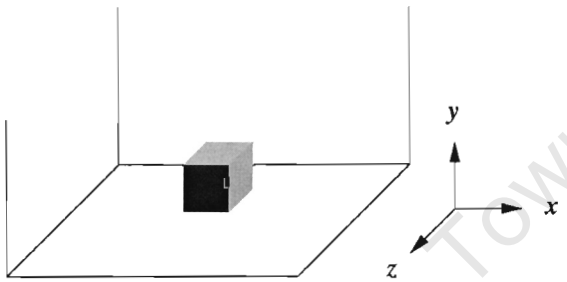

Figure 18: *The example virtual environment presented to the novice users.*

In this example, the virtual environment consists of a single box in a room in a world. A novice user is presented with the environment already created as depicted in Figure 18. They are given the following goal to implement as a desired behaviour for a user of the environment and for the box:

> *"When the user clicks on the box, the box will move 10 units in the x-direction."*

This goal shows a typically structured event-action pair commonly used for creating behaviours and interactions. For the novice user, the first step is to create a new event-action pair, since defining behaviours for anything in the environment are specified through the event-action pair. As was mentioned, event-action pairs are created by first specifying each of its individual components: the conditions that would make up the event, and the action for the behaviour to follow.

Specifying the conditions follows the type of path given in Figure 17. First the object needs to be selected. In this instance (as depicted in Figure 19) there are only two objects available in the environment. Groups that are available for that object are then displayed, and the user selects the appropriate options. Finally, conditions showing the *sentence functions* that are available for the selected groups are displayed, as in Figure 20.

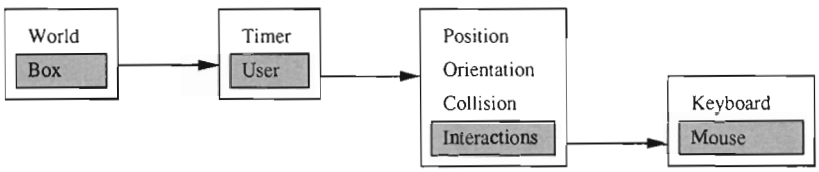

Figure 19: *Selecting the condition for the new event: first the object is selected, followed by various condition-groupings for that object.*

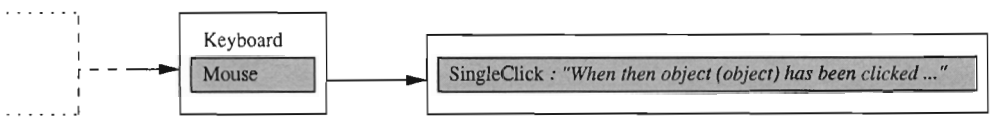

Figure 20: *Completing the condition selection: the final condition based on its name and* sentence function *is selected.*

Once the event condition has been specified, the action that is to be attached to the event needs to be selected. As with the conditions, these are also grouped and a hierarchical path must be followed, shown in Figure 21.

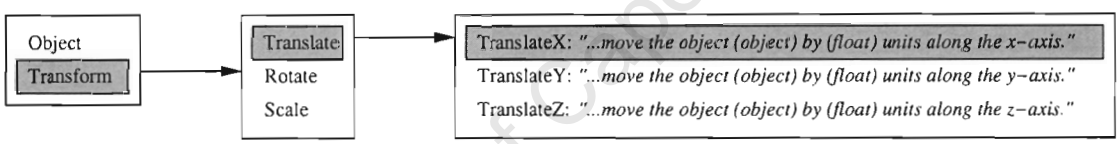

Figure 21: *As with conditions, selecting an action follows an hierarchical path where the final action to be performed is selected based on its name and* sentence function.

Once the event-action pair has been selected, the user may then fill in the values for each of the *sentence functions* that were selected to make up the event-action pair. Again, following guidelines provided by the *sentence functions*, these values can be entered from a simple GUI, as shown in Figure 22:

The event-action pair has been created and the new behaviour now exists: for the novice user, the goal is complete. In the executed environment, when the box is selected with the pointing device, it moves 10 units in the positive x direction.

On a slightly lower-level level, from the view of the non-VE programmer, we can look at how the condition might be implemented in more detail. The `SingleClick` condition for when an object

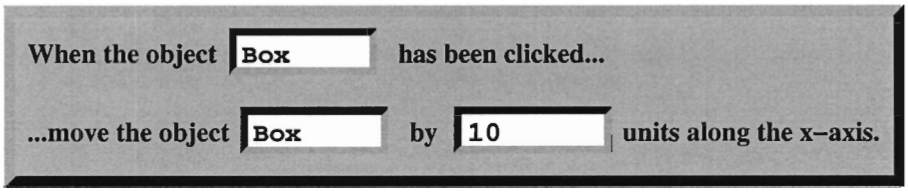

Figure 22: *The GUI representing the* sentence functions *with the values for their parameters entered.*

has been selected with the pointing device could[2] look as follows:

```
Condition
        SingleClick
        "When the object (object) has been clicked..."
Attribute Variables
        (none)
Script
        clicked = World.getAttrVariable("parameter1.clicked")
        if (clicked == "true"):
                World.setAttrVariable("parameter1.clicked", "false")
                return "Condition_Accepted"
```

Figure 23: *Condition script for a single click function using a mouse for a pointing device.*

As was mentioned in the previous chapter, all interactions from the mouse, etc., are stored as *attribute variables* of the objects they apply to. The script simply extracts the *attribute variable* for checking whether or not an object has been clicked: the `"parameter1.clicked"` refers to retrieving the *attribute variable* set of the first (and only) parameter for the *sentence function*, and then checking whether the interaction device's *attribute variable* (i.e. `clicked`), has been set to `true`. This would imply the mouse had been clicked while the pointer was over the object described in the parameter. If it has been set to `true`, the script resets the object's `clicked` *attribute variable* and the condition returns with a *condition accepted* signifying that the condition has been met.

When the event condition has been met, in this case by the user selecting the object, the action script is executed (which would be a simple object database script of changing *attribute variables* defined for the `TRANSFORM` node of the object).

[2]We emphasise that the script *could* look as follows as the system is scripting language independent and a pseudo-scripting language is depicted merely to emphasise from a programming point how some of the features would be implemented.

6.4.2 Non-VE programmer user authoring

If, now, the novice user's goal had to change to:

> *"When the user* double *clicks on the box, the box will move 10 units in the x direction."*

Looking at the current conditions available in Figure 20, the option is not available for the mouse and no appropriate function exists: a new condition is required. As the conditions are all scripted, this implies that a new script is required. For the non-VE programmer, the following goal would then be created:

> *"Add a double-click function to the set of mouse conditions."*

This is a simple matter of them adding the new function and changing the code to correspond to the required behaviour (see Figure 24). The new function would be appropriately inserted into the menu structures and the non-programmer can complete their goal by specifying the behaviour based on the new function.

```
Condition
        DoubleClick
        "When the object (object) has been clicked twice..."
Attribute Variables
        int click_count
Script
        clicked = World.getAttrVariable("parameter1.clicked")
        click_count = click_count + 1
        World.setAttrVariable("parameter1.clicked", "false")
        if (clicked == "true" AND click_count == 2):
                click_count = 0
                return "Condition_Accepted"
```

Figure 24: *Condition script for a double-clicked function using the mouse as a pointing device.*

6.5 Behaviour-Based System Methodology

The decision for taking the approach described in this chapter for specifying behaviours and interactions was made because of the advantages provided by *event-based* systems with regards to the

way in which behaviour could be specified. The main advantage of this method, we have found, is that it made it ideal for combining the scripting language with the use of *sentence functions* used for defining conditions and actions. This ultimately provided us with a method to allow novice users to specify behaviours.

For a behavioural-system, the method of specifying behaviours requires that for each object within the environment, the behaviours be programmed and this behaviour forms part of the object itself. How it behaves within the environment depends on external stimulus which means that each object is responsible for keeping track of the world around it and determining its behaviour based on that. Solving this problem such that novice users would be able to specify behaviours is not easy and is not as trivial a solution as is the method for specifying events in an event-based system.

Chapter 7

Conclusions

We have ventured to look at providing an architecture of a system that allows novice users to develop virtual environment applications. At the start of this dissertation we spoke about the lack of proliferation of virtual environment applications being attributed to the difficulties that were associated with having to author the environments. We have sought to address this lack in research by looking at methodologies for providing novice users with an authoring system that would allow them to create useful virtual environments. The main purpose of this dissertation was to investigate how it is possible to empower users not possessing the programming skills necessary to create such environments. Along with these methodologies, we also designed an architectural platform that would allow such an authoring system to be built.

7.1 Paving Methodologies

After having described the types of problems we were facing, we started by reviewing, in the second chapter, work that had already looked at means of improving the general quality and effectiveness of virtual environments. In particular it focused on how they could be created and where virtual environments have already found a niche. What we were more interested in, however, were various studies that had looked at methods for evaluating virtual environment systems and authoring tools. The list below describes desirable features for virtual environment systems:

- Multiple application support

- Flexibility
- Separation
- Extensibility
- Distributive
- Configurable
- Virtual reality support

A similar process was followed for studies on authoring tools, and the following desired features were extracted:

- Transparency
- Instantaneous feedback
- Extensibility and flexibility
- Scalability
- Rapid application design

7.2 Researching Improvements

Before looking at methods that could be used for creating the required system, we needed to look at what was wrong with what other systems had provided, as well as looking at success or failure in achieving the criteria that were listed as being desirable features. To do so various systems were categorised, in the third chapter, in a way that allowed them to be more easily compared. Categorising the systems made it easier to extract common features and perform comparisons of their good and bad qualities. The three classifications for categorising the systems were divided principally into tools for experienced users, intermediate-level users and novice users.

Tools for experienced users included toolkits and APIs such as OpenGL, the WordToolKit, CoRgi and Genesis3D were discussed. Tools for intermediate-level users included DIVE, Alice and Avango,

as they provide a scripting language for specifying behaviours and interactions. Among tools for novice users we looked at Alice and AVS/Express.

In the fourth chapter, the criteria that were listed above were used to evaluate these systems in order to extract a list of advantages and disadvantages of features those tools contained that could be used to create an ideal virtual environment authoring tool.

In particular, features that we found to be most useful in terms of providing a flexible architecture were those containing a modularised design. The advantage of this was that it allows some systems to be able to customise some of their components. Those that provided a static (or non-modular) design were found to be limiting as tools with respect to the types of environments they could support or produce.

As authoring tools, features that were most useful were those that allowed the complexity of authoring to be hidden using useful metaphors for transparency, effectively reducing the apparent complexity of a given task. Also very useful were those tasks that allowed the different processes of authoring an environment to be combined into single applications, reducing turn-around times in the development process.

One of the greatest things learned was from the field of GUI development, where research exists into looking at tools that can be used to produce GUI applications. Each of the systems we looked at targeted only certain types of users. Those targeting more than one type of user did not allow for more novice users to easily progress between different stages of complexity.

7.3 Applying New Solutions

Modularisation has proved to be the key to solving many problems that are encountered with virtual environment systems. The most useful advantage for us is that it can be used to aid in a user's familiarity with a certain systems when migrating to using different tools: the approach to modularisation that we took in our system allows different tools – scripting languages and graphics packages for example – to be adapted into the system.

To aid in this idea, IMC modules were used as communication bridges between each of the modules and provided a module-independent way of communicating.

One of the problems we found with many of the low level toolkits was the way in which they enforced the building of scene nodes in an environment. Typically, the toolkits require that the

nodes be added to a scene in a specific order for it to work correctly. By using the concept of having children that have ordered lists of siblings, this problem can be worked around. Being ordered, it no longer matters in which order nodes are added, the advantage being that this reduces the complexity of understanding the scene graph and makes them more intuitive for novice users.

Another method that is commonly used for reducing the complexity of authoring is by providing a mechanism for scripting behaviours within the environment: this provides many advantages such as reducing programming time and compiling cycles. One of the disadvantages of this, however, is that systems bind their scripting languages to the system itself. Any limitations of the scripting language would therefore be propagated to the system. The solution is then to not cater for any particular scripting language but rather to provide an interface whereby any scripting language can be used to control the virtual environment. The modular architecture that was implemented allowed for scripting language independence.

Using events for creating behaviour is a technique that has been used for a long time in GUI development and is a technique that many virtual environment systems have implemented for creating their behaviours. An advantage to using this technique is that it provides a means for allowing novice users to specify behaviours, as they do with GUI toolkits. To do this, we introduced the concept of *sentence functions* for creating the behaviours which allow novice users to specify them.

7.4 Evaluation

In this section, we summarise the evaluation process that was used in chapter four to analyse the strengths and weaknesses of the system we have developed.

7.4.1 Virtual environment systems evaluation

System extensibility

We have used the power of modularity to provide a system which does not solely rely on any single component: every component within the system may be changed and swapped out. It therefore does not suffer the same limitations as COOL-VR or the WorldToolkit which only cater for changing certain modules.

The disadvantage of the modularised approach is that there is a defined protocol for communicating between modules. Perhaps as some future work, a solution to this problem would be to provide a flexible interface for interpreting protocols.

Separation

Here again, separation refers merely to the modularisation of the communicating and networking components of the system. A networked system and a distributed system differ only in the communication module which is used.

Multiple application support

This refers to the application built around the system which would be presented to the users. As this deals with user interfaces, it does not form part of the scope of this project. The architecture of the system does, however, completely provide for all aspects of virtual environment authoring to be put into a single system as the architecture encompasses the modelling aspects as well as the behaviour specification aspects and in fact encourages such authoring.

Multiple world type support

The concept of meta-authoring tools directly refers to this and will be discussed in the next section.

7.4.2 Authoring tool evaluation

Transparency

The *sentence functions* that are used allow many of the problems to be abstracted into something novice users can easily understand. As they are more descriptive than simple function labelling, they allow for better descriptions of the functionality they represent. For example, it is a trivial exercise to abstract from the functions for manipulating objects in the X-Y-Z coordinate system and into using the Logo-style convention of Forward-Left-Up.

The architecture of the system also allows for interactively authoring the various stages of an environment: the modelling process and the behaviour specifying process. Both can be actively worked

on at the same time without the need for swapping back and forth between different authoring applications. The splitting of files for the different tasks greatly helps with this process as well.

On a lower level, the modularised approach to the system transparently allows users to interact with the various modules. As such, the modules can be changed without affecting the user who deals only with the interface of the module.

Instantaneous feedback

As was mentioned above, because the system allows the various authoring processes to be acted on at the same time, changes are immediately seen. Affecting the models in any way does not require that anything be restarted and changes can be seen as they are made. The scripting for the behaviours also means that nothing needs to be recompiled when changes are made to the behaviours.

Extensibility and flexibility

The modularised approach which is used, allows for the system to be easily changed and modified. Providing a system allowing modules to be added, changed or extended and have that same functionality reflected by the authoring tool poses more of a challenge. Users can interact with modules through the *attribute variables* the modules make available. In this way modules, which could for example be used for allowing different interaction devices, can be accessed and used within the environment.

Rapid application design

The use of *sentence functions* combined with a form of GUI provides a powerful tool for quickly and easily specifying behaviours. This allows prototyping of applications to be done where in fact with a minimal set of *sentence functions*, it is a simple matter to orchestrate simple scenarios and storyboard ideas.

7.4.3 Evaluating the Graphics API

A group of around twenty second year computer science students having no previous graphics programming skills were required to create a graphical three-dimensional output for an algorithm to be implemented as part of an undergraduate programming project.

The project itself was to implement a flocking algorithm [46] and to visually demonstrate the algorithm at work with various environmental factors. The students were required to create animated models and show their movements and flocking behaviour in a virtual world. They were provided with only the graphics API (described in section 5.2) to work with and a basic lesson on how the API worked in terms of loading models and manipulating the scene graph structure used and how such an algorithm might be tied into the API.

All the projects were successfully implemented with none of the students experiencing any problems related to the API itself. The biggest problem for the students was with the concepts involved with graphics programming.

Other problems with the API the students reported included the performance of scenes which involved large amounts of objects needing to be displayed (such as with a school of fish). Also, the API itself is in control of the entire environment. Students initially found the concept of using callback methods and implementing the flocking algorithm around the API to allow for time-sliced frames a conceptual problem.

Although these form valid problems in many graphics APIs, the students were considered to be intermediate users as they were not proficient in any graphics programming. The projects themselves were a useful tool for determining any conceptual problems or flaws with the API. Making the API into something that can be used more easily by non-novice users is for future work.

The CAVES Project

At the time of completing this dissertation, the CAVES project had been well underway with many of the concepts introduced in the dissertation being put into use by the system being developed. The authoring system being produced, CAVEAT (Figure 25), is the commercial outcome of the project of which the research project described in this dissertation forms a part.

7.5 End-user Tool Discussion

Having looked at the various features of many different systems, those features considered useful were extracted and combined into a system that would overcome their limitations as compared to an 'ideal virtual environment authoring system'. Below follows a brief discussion analysing the system against the criterion that was described for the ideal.

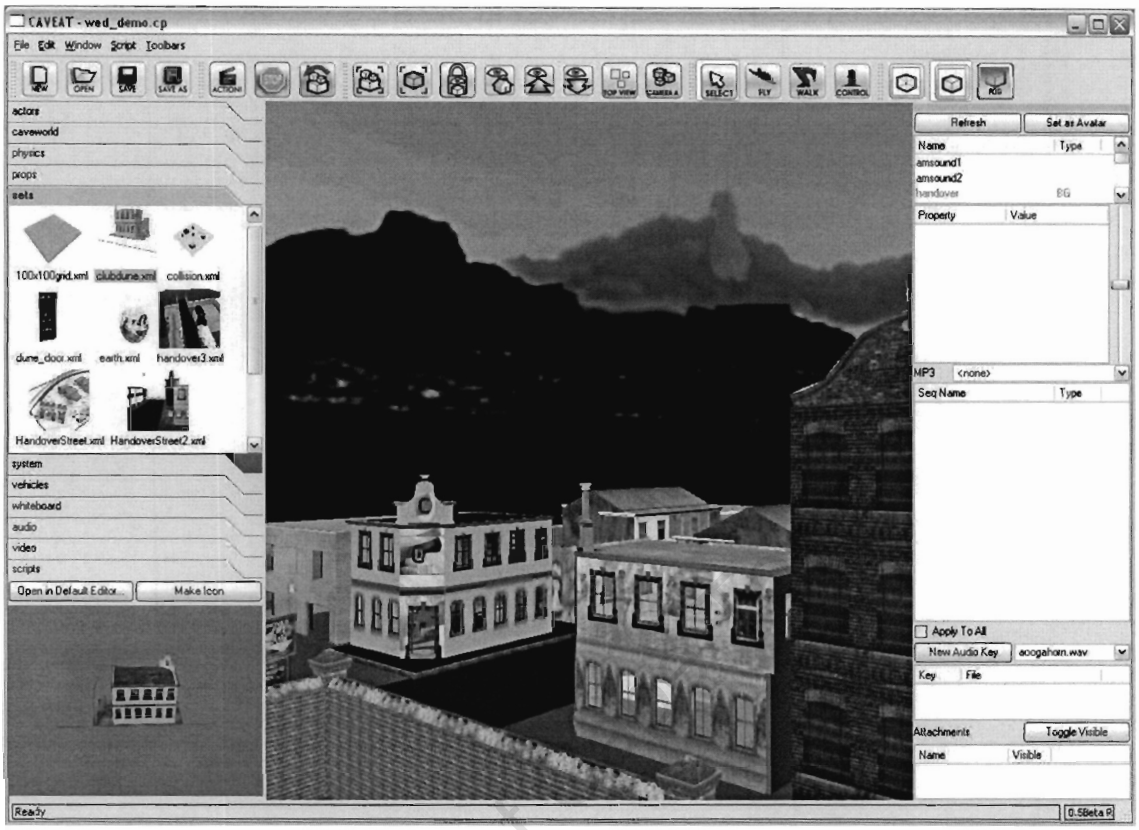

Figure 25: *The CAVEAT authoring system.*

7.5.1 Why a meta-authoring tool?

Creating a single generic authoring tool for every different kind of virtual environment application is an impossible task. This even more so if the author is a non-programmer. A more realistic solution is to think of every application as having a context or theme, such as a shopping mall or museum. Creating an authoring tool specific to such an application greatly reduces this problem.

The authoring tool in this sense refers to the tool that novice and non-programming users would use to create interactions and behaviours by *specifying* event-action pairs in the system. A criterion for creating these authoring tools is that they be quick to develop. With this in mind, the meta-authoring tool would be used to create (through script programming) the necessary conditions and actions for behaviours the novice users would use in the *context specific authoring tool*.

The meta-authoring tool is then, in this sense, a tool for designing and creating context specific

virtual environment authoring tools.

7.5.2 Shortened development time

Shortening the development time for creating virtual environment applications has in the past been achieved in various ways, the most prominent of these is the move from compiling low-level API code towards scripted behaviour. Even so, systems such as Dive and Alice bind the scripting language being used to the environment. In doing so, they provide a set of functions and methods accessible through the scripting language to manipulate the environments. These functions and methods are similar in nature to providing a set of APIs.

The meta-authoring tool provides a minimal set of language independent functions for manipulating the *attribute variables* in the environment. In doing so, the user is not required to learn, on top of the scripting language, the API provided in order to create interactions. For non-programmers, specifying behaviours and interactions can be simulated with a simple point-and-click interface system.

7.5.3 Virtual environment complexity support

Dive and Avocado provide a scripting language to minimally manipulate objects in the environment. They also provide an API written in some low-level language for creating more complex interactions. These are then linked into the system and can be called through the scripts.

In a system such as Alice, the focus has been on overcoming the problem that creating virtual environment software is a difficult process. With such systems, the solution is to provide a set of automated 'functions' that will allow the user to easily specify the interactions of their world.

A shortcoming to this approach is that they provide the useful, pre-created and automated functions while at the same time sacrificing the ability to extend and expand the system by allowing for the creation of new functions. If systems do provide enough flexibility, the process to do so is usually difficult and goes against overcoming the concept of "virtual environment software is difficult to create".

In the meta-authoring tool, since everything is represented as an *attribute variable* accessible through the scripts, and the entire environment behaviour functionality scripted, it allows for extending the use of functions for the creation of new events. Only at a low-level, such as adding a new type of

user interface hardware, or changing the Graphics Module for a better rendering engine, would it be necessary to use the API and not the scripts.

7.5.4 Migrating user support

Most authoring systems target particular types of users. They provide either support for novice users only, or for those users that can program.

The meta-authoring tool provides the support for novice users, without losing the support for more advanced users. In this way, novice user may migrate to using the more advanced features of the system. Although the gap that exists between the authoring tool and meta-authoring tool can only be bridged by a user learning to program, the users may do so gradually. What this system uniquely allows for is the progressive development of a user.

Novice users may use the provided event libraries that allow them to create interactions. More advanced users may then attempt to combine events. Since the system is scripting language independent, an inexperienced user may start with, for example, a virtual environment scripting language developed for non-programmers [65]. The more experience they gain, the more complex scripting languages they can use to develop complex interactions in their worlds. Finally, since the scripts for all the events are given, the user may advance by learning through the examples of scripts that are provided.

7.5.5 Configurability

Most systems do not provide the ability to easily change any of their components. Users are usually stuck with the implementation provided by the creators of the system. To overcome this problem, some have implemented a modularised system for some components that allow them to, for example, add new hardware devices.

Our modularised system allows for different implementations of a module to be used. It is possible to have two of the modules with different implementations running simultaneously. For example, both Python and Tcl at the same time as depicted in Figure 10, or use different graphics libraries, such as OpenGL or Direct3D.

7.6 Conclusion

Virtual environment authoring systems have grown, although in the past, few systems have been targeted to novice users. With increased use, this is becoming a more important requirement.

We have presented a meta-authoring system that, both on an architectural level and on a user-end level, tries to overcome many of the problems present in most systems today. These problems we presented as a list of ideals we found lacking in many systems.

While producing a single authoring system that would be able to create any type of virtual environment application would be impossible, we have proposed a more progressive system – a meta-authoring tool that rapidly generates a virtual environment authoring system for a particular virtual environment solution.

The system also allows for the progressive migration from novice users to more advanced users: the meta-authoring tool being used by more advanced users to produce the authoring tools that can be driven by non-programming, novice users.

References

[1] Agarwal R., Prasad J., Tanniru M., Lynch M. Risks of rapid application development. *Communications of the ACM*, pages 177–188, 2000.

[2] Arlievsky A., Bekkerman R., Medved M. Rapid prototyping.

[3] AVS. AVS/Express. [http://www.avs.com/software/soft_t/avsxps.html] Last accessed 21/02/2003.

[4] Bangay S., Gain J., Watkins G., Watkins K. RhoVeR: Building the second generation of parallel/distributed virtual reality systems. In A. G. Chalmers and F. W. Jansen, editors, *First Eurographics Workshop of Parallel Graphics and Visualization*, pages 277–289, 1996.

[5] Blumberg B.M., Gaylean T.A. Multi-level direction of autonomous creatures for real-time virtual environments. In R. Cook, editor, *SIGGRAPH 1995*, pages 47–54, August 1995.

[6] Broll W. Populating the Internet: Supporting multiple users and shared applications with VRML. In *Proceedings of the VRML'97 Symposium, Monterey, CA, ACM SIGGRAPH*, pages 87–94, February 1997.

[7] Broll W. DWTP – an internet protocol for shared virtual environments. In Don Brutzman, Maureen Stone, and Mike Macedonia, editors, *VRML 98: Third Symposium on the Virtual Reality Modeling Language*, New York City, NY, 1998. ACM Press.

[8] Brooks F.P. What's real about virtual reality? *IEEE Computers Graphics and Applications*, 19(6):16–27, 1999.

[9] Brutzman D.P., Macedonia M.R., Zyda M.J. Internetwork infrastructure requirements for virtual environments. In *VRML'95 Symposium, ACM SIGGRAPH*, pages 95–104, 1995.

[10] Bryson S. Virtual reality in scientific visualization. *Communications of the ACM*, 39(5):62–71, 1996.

[11] Carey R., Bell G. *The Annotated VRML Reference Manual*. Addison-Wesley Pub Co., 1997.

[12] Carlsson C., Hagsand O. DIVE – a multi-user virtual reality system. *IEEE Virtual Reality Annual Symposium (VRAIS '93), Seattle, OR*, September 1995.

[13] Carson J.A., Clark A.F. Multicast shared virtual worlds using VRML97. In *Proceedings of the 4 th Symposium on the Virtual Reality Modeling Language (VRML'99), Paderborn, Germany*, pages 133–140, 1999.

[14] Conway M.J. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, Faculty of the School of Engineering and Applied Science at the University of Virginia, December 1997.

[15] Cooper A. *The Inmates Are Running the Asylum: Why High Tech Products Drive Us Crazy and How To Restore The Sanity*. Sams, 1999.

[16] Sense8 Corporation. Worldtoolkit: Virtual reality support software. Bridgeway, Suite 101, Sausalito, CA 94965, telephone : (415)331-6318.

[17] de Sa A., Zachmann G. Virtual reality as a tool for verification of assembly and maintenance processes. *Computers and Graphics*, 23(3):389–403, 1999.

[18] Dybvig R.K. *The Scheme Programming Language: ANSI Scheme*. P T R Prentice-Hall, 1996.

[19] Ellis S.R., Kaiser M.K., Grunwald A.J. (eds). *Pictorial Communication in Virtual and Real Environments*. London: Taylor and Francis, 1991.

[20] Fencott C. Towards a design methodology for virtual environments. In *The International Workshop on User Friendly Design of Virtual Environments, York, England*, 1999.

[21] Frecon E., Hagsand O. The Dive/Tcl behaviour interface reference document. [http://www.sics.se/dive/manual/tcl-behaviour.html] Last accessed 05/07/2002.

[22] Frecon E., Stenius M. DIVE: A scalable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments)*, 5(3):91–100, February 1998.

[23] Fuhrmann A., Purgathofer W. Studierstube: An application environment for multi-user games in virtual reality. In *GI Jahrestagung (2)*, pages 1185–1190, 2001.

[24] Gossweiler R., Long C., Koga S., Pausch R. DIVER: A distributed virtual environment research platform. *IEEE Symposium on Research Frontiers in Virtual Reality, San Jose, CA*, pages 10–15, October 1993.

[25] Grimsdale C. dVS – distributed virtual environment system, Division, Ltd.

[26] Johns C. Spatial learning: cognitive mapping in abstract virtual environments. In *Afrigraph '03*, pages 7–16, 2003.

[27] Kaur K. *Designing Virtual Environments for Usability*. PhD thesis, Centre for HCI Design, City University, London, June 1998.

[28] Kessler G., Bowman D., Hodges L. The simple virtual environment library: An extensible framework for building VE applications. *Presence: Teleoperators and Virtual Environments*, 9(2):187–208, 2000.

[29] Kooper R., Brian W., Kevin H., Allison D., Hodges L.F. COOL-VR: a virtual environments toolkit.

[30] Laird C., Soraiz K. Gui toolkits: What are your options? *SunWorld*, March 1998.

[31] Leigh J., Johnson A.E., DeFanti T.A. Issues in the design of a flexible distributed architecture for supporting persistence and interoperability in collaborative virtual environments. In *Proceedings: SC97*, 1997.

[32] Madden B., Farid H. *Active Vision and Virtual Reality*. Springer, New York, NY, 1995.

[33] Marsden G. *Designing Graphical Interface Programming Languages for the End User*. PhD thesis, Department of Computer Science, Stirling University, January 1998.

[34] Myers B.A. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1), March 1995.

[35] Myers B.A., Giuse D., Dannenberg R.B., Vander Zanden B., Kosbie D., Pervin E., Mickish A., Marchal P. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11), November 1990.

[36] Myers B.A., Hudson, S.E., Pausch R. Past, present and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7(1), March 2000.

[37] Nardi B., Miller J.R. The spreadsheet interface: a basis for end user programming. In *IFIP INTERACT'90*, pages 977–983, 1990.

[38] Neider J., Davis T., Woo M. *OpenGL Programming Guide*. Addison-Wesley, 1993.

[39] Olszewski R. REVIEW OF: Andrea L. Ames, David R. Nadeau, and John L. Moreland. VRML 2.0 Sourcebook, 2nd Edition. New York: Wiley, 1997. *Telecommunication Electronic Reviews*, 6(2), March 1999.

[40] Ousterhout J.K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[41] Ousterhout J.K. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[42] Papert S. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, New York, 1980.

[43] Pausch R., Burnette T., Capehart A.C., Conway M., Cosgrove D., DeLine R., Durbin J., Gossweiler R., Koga S., White J. A brief architectural overview of alice, a rapid prototyping system for virtual reality. *IEEE Computer Graphics*, May 1995.

[44] Pausch R., Conway C., DeLine R., Gossweiler R., Miale S. Alice & DIVER: A software architecture for the rapid prototyping of virtual environments, 1994.

[45] Perlin K., Goldberg A. Improv: A system for scripting interactive actors in virtual worlds. *Computer Graphics*, 30(Annual Conference Series):205–216, 1996.

[46] Reynolds C. Flocks, herds, and schools: A distributive behavioral model. In *ACM SIGGRAPH*, pages 25–34, 1987.

[47] Shaw C., Green M., Liang J., Sun Y. Decoupled simulation in virtual reality with the mr toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, 1993.

[48] Smith S.P., Duke, D.J. Binding virtual environments to toolkit capabilities. *Computer Graphics Forum*, 19(3):81–89, 2000.

[49] Snowdon D.N., West A.J. The AVIARY VR-system. A prototype implementation. *Presence*, 3(4):288–308, 1994.

[50] Strauss P., Carey R. An object-orientated 3D graphics toolkit. *Computer Graphics*, 26:341–349, July 1992.

[51] Tramberend H. Avocado: A distributed virtual reality framework. In *IEEE Virtual Reality*, 1998.

[52] Website. ActiveWorlds.com, Inc. [http://www.activeworlds.com] Last accessed 06/08/2001.

[53] Website. blender3d.org. [http://www.blender3d.com/] Last accessed 04/02/2004.

[54] Website. Crystal Space 3D. [http://crystal.sourceforge.net/] Last accessed 03/04/2003.

[55] Website. Genesis3D open source engine. [http://www.genesis3d.com/] Last accessed 03/04/2003.

[56] Website. Java. [http://java.sun.com] Last accessed 22/07/2002.

[57] Website. Rhodes University VRSIG. [http://www.cs.ru.ac.za/vrsig/] Last accessed 23/08/2002.

[58] Website. The CAVES Project. [http://www.caves.co.za] Last accessed 06/08/2001.

[59] Website. The Python Home Page. [http://www.python.org] Last accessed 22/07/2002.

[60] Website. The South African Cultureware Project. [http://www.cultureware.net/] Last accessed 03/03/2003.

[61] West A., Howard T., Hubbold R., Murta A., Snowdon D., Butler D. AVIARY – A generic virtual reality interface for real applications. *Proceedings of Virtual Reality Systems, London*, pages 213–236, 1993.

[62] Winograd T. From programming environments to environments for designing. *Communications of the ACM*, 38(6):65–75, June 1995.

[63] Wloka M. *Interacting with Virtual Reality*. In Virtual Environments and Product Development Processes, Chapman and Hall, 1995.

[64] Yang S., Marsden G. Using programming tools in virtual environments. Technical Report CS02-04-00, Department Of Computer Science, University of Cape Town, 2002. [http://www.cs.uct.ac.za/Research/CVC/Techrep/ CS02-04-00.pdf] Last accessed 06/11/2002.

[65] Zachmann G. A language for describing behavior of and interaction with virtual worlds. In *VRST '96*, pages 143–150, 1996.