A Meta-Authoring Tool for Specifying Interactions in Virtual Reality Environments

Zayd Hendricks, Gary Marsden, Edwin Blake
Collaborative Visual Computing Laboratory
Department of Computer Science
University of Cape Town
{zhendric, gaz, edwin}@cs.uct.ac.za

ABSTRACT

When creating virtual reality environments a large amount of the interaction needs to be programmed. The problem with this is that non-computer expert users lack the programming skills necessary to create useful applications. Specifying interactions remains in the domain of the programmer. Creating a single, generic authoring tool for every different kind of application would be 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 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 found in previous systems.

Categories and Subject Descriptors

H.5.2 [**Information Systems**]: User Intefaces—*interaction styles*; I.3.7 [**Computing Methodologies**]: Computer Graphics—*virtual reality*

General Terms

Human Factors

Keywords

Virtual reality, virtual reality authoring, migrating user support, behavior, interaction, scripting languages

1. INTRODUCTION

Virtual reality (VR) is a concept that has developed rapidly over the past few years. This growth is being stimulated by research into the field, as well as entry-level technology being more readily available. VRML, described as a three-dimensional analogy to HTML [4], is one such technology that has contributed to this growth. Such modelling languages have greatly increased the spread and usage of virtual reality environments since they provide an easy mechanism for describing VR worlds over the web.

The problem for novice users[1] trying to create useful VR applications arise when they need to specify the behavior and interactions of the objects in their environments. This task, to be made flexible, requires the user to program.

What is still lacking is the serious consideration that these novice users, who require these technologies the most, more often than not, lack the skills necessary to create them – VR authoring remains largely in the domain of the programmer. (A parallel problem exists for users wanting to create graphical user interfaces (GUIs), although there is currently more research into GUI authoring techniques than into VR.)

VR authoring systems fall into two general categories: those that support novice users and those that support a more advanced user. Systems that have been created for novice users are normally too simplistic and lack the sophistication necessary to develop the specific VR solution they need. On the other hand, those systems that provide the necessary sophistication are too general to provide the solution; they are difficult to use and take too long to create a useful application.

We describe a VR authoring system that overcomes some of the problems associated with current authoring systems. One way in which we do this is by providing both novice and advanced users a *single* system on which to develop, with the opportunity for novice users to smoothly migrate to creating more complex applications.

Please note that our work does not deal with user interfaces, instead we define a framework and architecture on which such user interfaces may be built. As such, it defines the functionality those interfaces may possess.

2. GENERAL CONCEPTS

2.1 Creating Virtual Worlds

[1]We refer to novice users as having little to no programming skills but are, however, experts in their own fields.

Creating a virtual world generally consists of two distinguishable steps – modelling the environment and specifying interactions and behavior (althrough some research is being done into merging these two steps [21]).

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 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 its interactions or 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 a three-dimensional world is the scene graph [16].

Interaction and behavior specification: Interactions 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 behavior of the environment (gravity, kinematics, etc.).

Specifying interactions typically requires some form of programming for them to be useful. Novice users wishing to create VR applications therefore find two main problems with specifying interactions:

- *they are difficult to program*: Authoring virtual environments still remains largely in the domain of the programmer [13]. Current systems require that developers possess some degree of programming skills before they can create an environment that contains a significant amount of interaction.
- *they take a long time to create*: Tools for creating VR applications exist mainly as a set of interface libraries, for example Sense8's WorldToolKit [6]. 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 environment development are usually specific to a type of environment (for example the automotive industry [7]) 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.

2.2 Interactions and Behavior

Interactions and behavior in VR systems have been implemented in two general ways [3][22]:

- **Behavior-based systems**. In behavior-based systems, the interactions of the objects in the world are implemented as 'attributes' of the objects. The objects run according to the behavior that 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 [3][14]. The objects may also be restricted by rules the environment imposes.
- **Event-based systems**. Interactions are based on sets of events that occur in the environment. These events are generated by user interactions or are generated by a change in the state of objects. (These events are not different from the events used in graphical user interfaces where events are generated, for example, by the user clicking on a button.)

RhoVer [2], and more recently CoRgi [19], are systems for rapidly developing virtual environments and are used primarily as test beds for investigating virtual reality. Applications on the system are implementations in Java [18] and are natively compiled for optimization. Interactions that are created are *behavior-based.* Objects in the environments are created as object instances in Java, each object defining its behaviors in the world. The RhoVer system provides only a framework of a system. Creating a VR application requires that the user learn and understand some of the concepts of VR; the user is required to program the graphics of the environment.

Avango [17] is a framework for building interactive virtual environment applications. It uses an *event-based* interaction system by providing *sensors* in the environment. All high-level Avango objects can be created and manipulated with an interpreted Scheme script [15] (a functional scripting language). Complex and performance critical parts are written in C++ which is then called from the Scheme scripts.

DIVE [10] is a "multi-user, scalable network architecture for distributed virtual environments". The system is used as a platform for collaborative VR experimentation. As such, some of its main aims have been that environments be built quickly. As a collaborative system, they have concentrated their works on it being a transparently networked application.

Interactions within the system are both *behavior-* and *event-based.* Each object in the environment may contain Tcl [9][11] scripts that are executed wherever the object is replicated (defining the *behavior-based* interactions). Tcl scripts can also be triggered by events defined by the system (defining the *event-based* interactions). There are pre-made events for interaction: signals, timers, collisions, etc.

The DIVE system contains no support for novice users. Their focus has been on the intermediate type users (providing these with the scripting facilities), and on advanced users, by providing a low-level API that can be compiled to create a DIVE client system. The DIVE system has tried to improve creation time, and is an advantage of the RhoVer system, in that they have provided scripting routines to, on a high-level, manipulate the graphics.

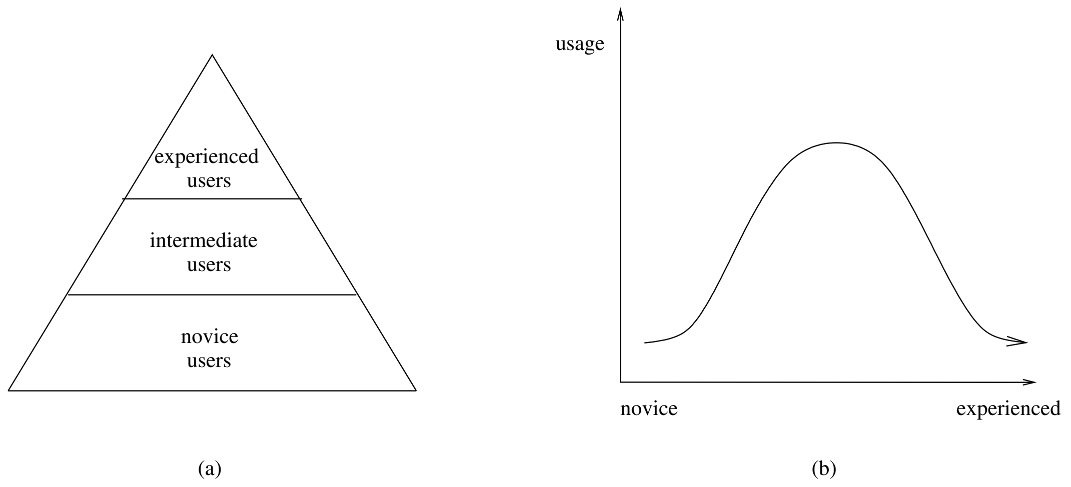

Figure 1: ***(a) A static model describing the types of users of a system, novice users making up the majority. This model shows the usually inaccurate perception about the types of users of a system. (b) A dynamic model that more realistically models the dynamic progression of users. Users of a system spend only a short time as 'novices' of a system.***

Alice [13] is a "rapid prototyping system for virtual reality". It was created to overcome the problem that writing virtual reality software is a difficult process.

They have created a set of pre-made functions to create events that novice users can use to specify their interactions. There are, however, a limited amount of these.

The Alice system provides a set of Python [20] 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 some basic user interaction events which can be used to spawn scripts (it does not allow new events of these types to be created).

For the advanced programmers, Alice allows the user to create Python extensions and modules and add them to the system. These extensions are created with a low-level language.

Each of the above systems caters for a different set of users. In the next section we discuss the concept of migrating user support and how this concept applies to each of these systems.

3. MIGRATING USER SUPPORT

Most systems base their user support on the perceived model for the types of users of a system, depicted in Figure 1(a). In reality, we find that this is inaccurate as it describes a static model for a dynamic system. The graph shown in Figure 1(b) gives a more realistic model – it describes the dynamic progress of novice users to advancing their expertise in a system [1]. We therefore find that system implementations do not take this 'migration' of users into account.

The figure shows that a user 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 experienced users.

With respect to VR systems we find that authoring tools for VR applications provide sophistication support for usually only a certain type of user. The progress of users can be modelled by the graph shown in Figure 2(a) [12]. There is a rapid learning curve attributed to the user's initial working with the system. As they use more of the system, their knowledge and ability to use the system increases. Once the user has reached a 'comfort' phase (the maximum sophistication level they can get out of 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, the RhoVer system gives an example of the kind of graph that can be shown in Figure 2(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.

Systems such as Avango and Dive represent a two-phase learning system. The first is that the user requires learning the scripting language. Learning the API is required to become an expert with the system – normally for advanced programmers only, programming with a low-level language.

Alice 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 [5] 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

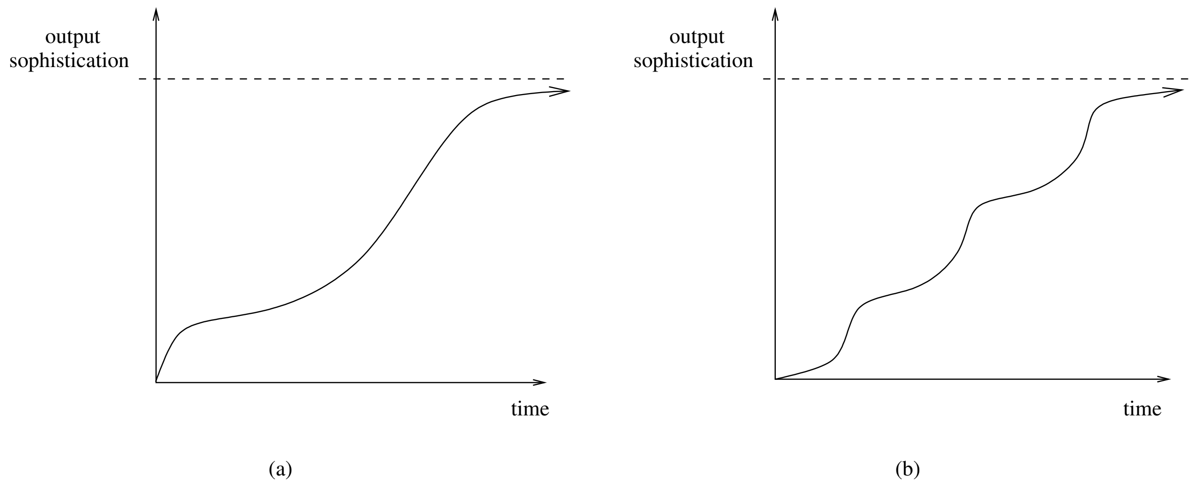

Figure 2: ***(a) The graph shows a user's progress with respect to the sophistication of the types of applications they can create. There is an initial steepness in the curve associated with the user first learning the system. As they become more experienced, they reach the maximum sophistication the 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.***

Figure 2(b).

4. AN IDEAL VR AUTHORING SYSTEM

Through our experience, we have considered what features we would most like in a VR authoring system that are lacking in many systems. These features present an ideal system that is useful, not only to advanced programmers, but to novice users as well.

1. **Smooth migrating user support.**
   It is important that users with differing levels of skills be able to use a system. As such, novice users gaining experience should be able to advance to using the more complex features of the system.

   This implies that novice users should be able to use it without sacrificing the sophistication of the environments it can output. Similarly, advanced users should be able to use the system without it being so simple that it lacks the sophistication necessary to create the environments they need.

   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 a API for experienced users. At each stage, however, the user is required to learn something new. The ideal system should give a smooth migration, as shown in Figure 3.

2. **Support VR applications of any complexity.**
   Irrespective of the types of users, the system should be able to produce VR applications of any complexity. We have found that VR systems tend to restrict the types of environments they can produce to their target user types.

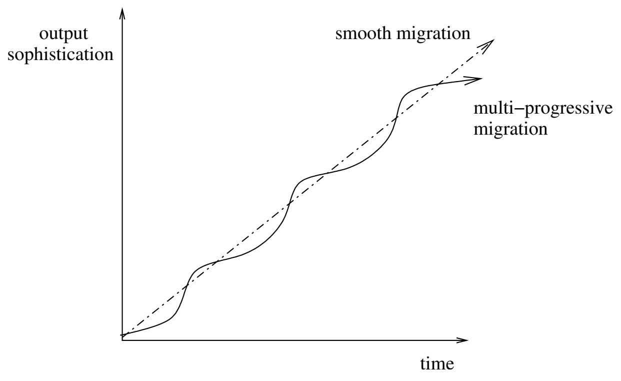

Figure 3: ***The graphs shows two different types of user migrations. The solid line represents the types of systems that provide multiple types of progression. The dotted line represents an ideal system that provides the user with a linear learning curve.***

   Relative to the graphs in Figure 2, this would imply that no asymptote to the complexity of a system exists, and that there is no limit to the complexity it supports, such as is shown by the smooth migration line in Figure 3.

3. **Much shortened development time.**
   Currently in systems, developing a useful VR 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 (currently ranging from several weeks to months).

   The time taken to create a VR application should be comparable to the time it takes to create a GUI application.

4. **Completely configurable.**
   The system should be able to cope with new hardware

and emerging technologies with minimal changes to the architecture of the system. Anything that is new or added on should not change the way that they are used for defining interactions – the system should remain consistent in the way it is used.

5. SYSTEM OVERVIEW

Our system is divided into three main modules communicating with each other in different ways (see Figure 4). Each module presents a simple front-end with which it may communicate with another via an IMC (inter-module communication) module. The front-end is used to hide the more complex implementation details. The method in which they communicate (the IMC module) may be easily changed if it is decided that the method used is inefficient or inflexible. Presenting this front-end provides an easy way for different implementations of a particular module to be rewritten: a newly written module need only present the same front-end.

The Graphics Module

The Graphics Module is comprised of two simple parts: a *World* component and several *Objects* stored in the Object Database. The *World* component is responsible for controlling the rendering of the database (the outputs of the VR system), and for receiving inputs, keyboard, mouse or other, via its *input handler*.

In order to have the *World* as a 'tangible' entity in the environment, we virtually represent it as an object stored in the Object Database. In doing this, we allow all its attributes to be accessible to objects and events.

The Scripting Module

In order to provide a quick turn-around time for specifying the interactions, it was decided that the interactions should be scripted. The biggest advantage to this is that no compilations are required and changes can be made while the environment is running.

The Scripting Module is again just an interface on which the scripting language chosen to be used is hooked. This allows the system to be uniquely scripting language independent which provides the system with several advantages, which will be discussed later.

The Events Module

We have focused on using *event-based* interactions and behaviors in our system; it is easier specifying events for object interactions in a world and can be achieved without the user programming. Even with only a few events available to a user, complex interactions can be created.

The Events Module is responsible for handling the interactions in the environment. It contains the structures for storing attribute variables, conditions and actions discussed below. Executing the scripts for the events is handled by the Event Module. The following sections provide a more in-depth discussion, as this forms the most important module in the system.

6. EVENT-ACTION PAIRS

A virtual environment is composed of several static nodes and models – these 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 the nodes and models; moving or rotating an object is merely a change in a transform node associated with the object in the scene graph.

We define an event-action pair 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.

There are therefore several components that describe a virtual world: the objects and their static animations (describing the modelling process), conditions and actions (describing the dynamic behavior). 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 not only keeps things tidier, it promotes the re-use of each element.

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

Zachmann [22] 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.

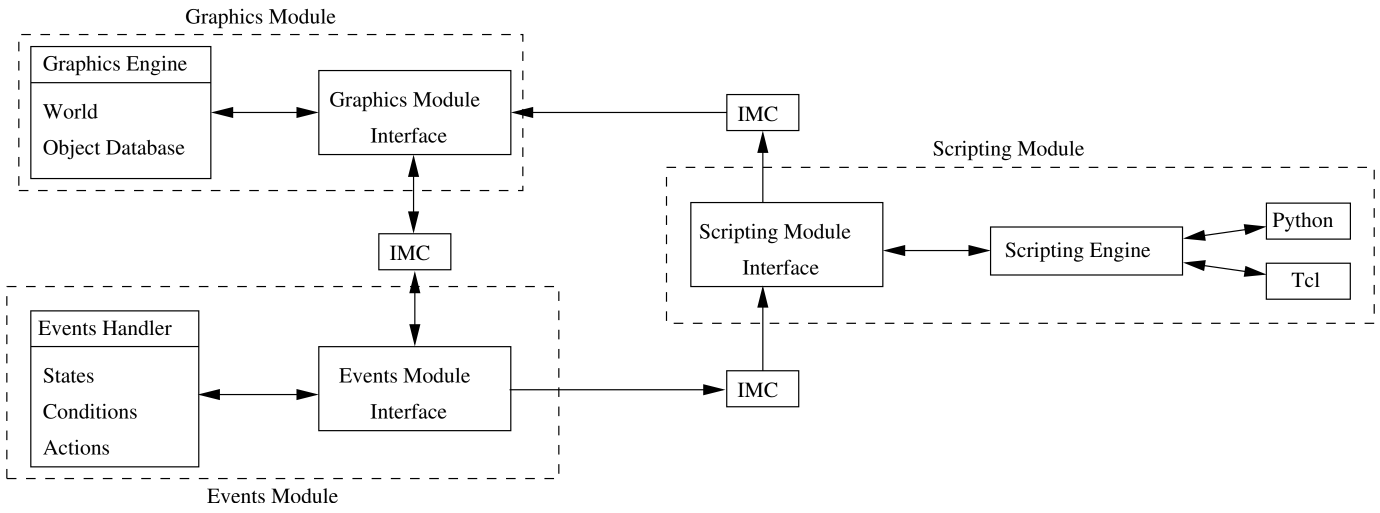

Figure 4: ***The Modular System Overview. Each of the modules is connected via an IMC module. The modules present a front-end with which they may transparently communicate with each other.***

6.1 Attribute Variables

Many of the simple behaviors 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 the environment and are used to describe the properties of an object or the 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, we decided that everything in the environment should be an *attribute variable.* The objects, their properties, the avatar and the environment can all be manipulated through the *attribute variables* that represent them.

6.2 Conditions

Many of the systems that use an *event-based* system for describing their interactions usually provide 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 in the system are scripted; users can then create new conditions or modify existing ones to suite their needs.

Conditions are tested with the use of scripts – the decision made is usually based on the values of *attribute variables.* They work similarly to programming functions – they take in parameters and return 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 ground, a '*condition accepted*' signal may be emitted at anytime during the execution of a condition script.

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

Conditions are also declared similarly to the way functions are created with three main parts:

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

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

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

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.

The *conditionVariables* section is used to define *attribute variables* that may be used as '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 attach certain conditions to certain objects – 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 amount of time. The time that is specified is usually based on a 'wall clock' time and is not dependant on the speed of the system or any external factors.

An example of this is a simulated animation that would need to run for an exact amount of time. This would need to be independent on 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-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-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 timeSnapshot
Script
      snapshot = World.getAttrVariable ("time")
      wt = parameter1 * 60 + parameter2
      dt = snapshot - timeSnapshot
      if (dt >= wt)
            timeSnapshot = timeSnapshot + wt
            return "Condition_Accepted"
```

Figure 5: ***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 5. 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 and *sentence function*. Each condition may define *attribute variables* that can be used. In this example, we define an *attribute variable* called `timeSnapshot` that represents the 'wall-clock' time at the creation of the event.

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

6.3 Actions
Actions are executed by the environment once it 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 in a similar way 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. An action to rotate an object could be defined as:

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

Action scripts influence the behavior of an environment by changing the environment's *static* aspects. Actions can be used to start and stop *static* object animations or change *attribute variable* values that could lead to event conditions being accepted.

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 – they may add, change parameters of, or remove pairs from the environment.

7. SPECIFYING BEHAVIOR AND INTERACTION
Interactions in our system are specified through event-action pairs. The conditions and actions above describe functions that can be used to create these pairs. Specifying a pair is therefore an instantiation of both a set of one or more conditions and an action.

7.1 Specifying Action-Event Pairs
On a lower level, event-action pairs are specified by instantiating their condition and action functions with the required parameters. From the example of the timer condition, an instantiation could be made as follows:

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

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

For the more novice user, the *sentence functions* may be used as guidelines as to the type of values that can be used as parameters. An example is given in Figure 6 where the *sentence function* is presented to the user. Text boxes are used where the user may enter the values for the parameters.

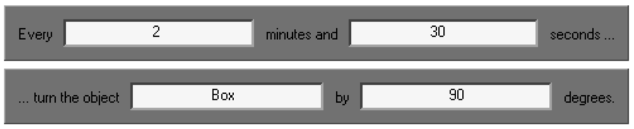

Figure 6: *The* sentence functions *presented in a graphical user interface (GUI). Text boxes are used for the user to enter the parameters for the functions. The top box represents the* sentence function *for the condition, the bottom for the action. This GUI dialog would represent the user statically creating or editing a condition.*

The same process can be used for actions. The function

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

would describe an action that rotates an object name "Box" by 90 degrees (around the y-axis). In our system, all objects are named as strings. This ensures a compatibility with the scripting language being used.

To emphasise that a condition or an action is an incomplete part, we use an elipses ('...') at the end of a condition's and the beginning of an action's *sentence function*. We can then represent an event-action pair 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:

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

The technique above of instantiating event-action pairs is the static method that the user specifies (this is usually the initializing pairs created at the start of an environment). As mentioned earlier, pairs may also be dynamically created or changed through the action scripts.

7.2 Grouping Conditions

In a library of conditions and actions, users should easily be able to find the condition and action they require to specify an event-action pair. We have created the ability for the author to categories and group (and subgroup, etc., if necessary) conditions.

Creating a new event-action pair starts with specifying the condition functions and then specifying the action function that together make up the 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 the condition groups (and subgroups, etc.) that is associated may be selected. Finally, the condition is selected.

In this way, the task of searching for a particular condition in a large library is made easier through the grouping of the conditions. Those conditions not object-related are grouped and connected as part of the World object.

As an example, specifying a condition to check for a mouse click would follow the path shown in Figure 7.

7.3 User Interaction and Object Selection

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

In the case of object selection, behaviors need to be defined for when a user selects an object. In order to keep everything consistent, all input data that are received through the devices are stored in *attribute variables.* Interactions can then be created through changes in these *attribute variables.*

For the example of a mouse, the World defines *attribute variables* for the pressing and the release of the mouse buttons. For each object in the world, an associated *attribute variable* is created to indicate whether, given a set of screen coordingtes, it is currently being selected. This may include many objects at once as in an example of hierarchies of objects.

These associated *attribute variables* may be defined for any type of input device. The values of these *attribute variables* can then be used to generate events.

7.4 Non-Programmers Specifying Event-Action Pairs

An event-action pair is instantiated through first specifying a set of one or more conditions and an action (through a 'browsing' process shown in Figure 7), and then inserting the require parameters for the functions (as in Figure 6).

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 powerful enough to create a diverse set of applications – all FSM-based environments can be created without programming in our system (for example a game of tic-tac-toe with a single player versus a computer).

8. DISCUSSION

Having looked at the good and the bad features of many different systems, we have extracted those features that were useful and combined them into a system that overcomes their limitations as compared to an 'ideal VR authoring system'. We give a brief discussion analyzing our system against the criteria that has been laid down for the ideal.

8.1 Why a Meta-Authoring Tool?

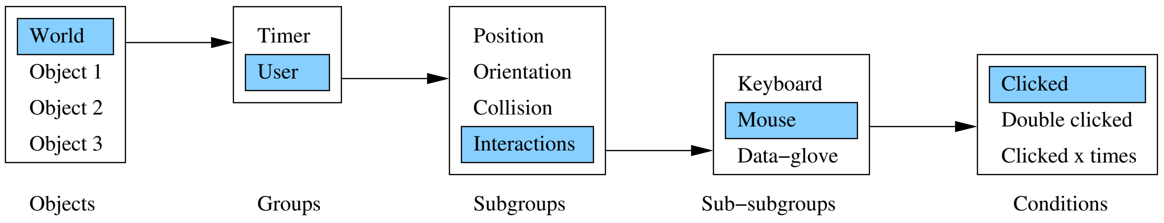

Figure 7: ***Specifying a new condition. The first step involved is to select the object associated with the condition. The selection is refined through the groups and the subgroups, etc. in which the condition belongs. The last step is to select the condition from the list of conditions in the group.***

Creating a single generic authoring tool for every different kind of VR application is an impossible task – more so if non-programmers are to do the authoring. A more realistic solution is to think of every application as having a context or theme, such as a shopping mall of museum. Creating an authoring tool specific to these applications reduces this problem.

The authoring tool in this sense refers to the tool that novice and non-programming users would use to create interactions by *specifying* event-action pairs in the system. A criterion for creating these authoring tools would be 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 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 VR authoring tools.

8.2 Migrating User Support

Most system target at a particular type of user. They provide either support for novice users only, or for those users that can program. Our system 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 gradually do so. 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 – with the help of the *sentence functions* provided with each condition and action, these may be specified without their needing to program. More advanced users may attempt to combine conditions into more complex events. Since the system is scripting language independent, an inexperienced user may start with, for example, a VR scripting language developed for non-programmers [22]. 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 conditions and actions are given, the user may advance by learning through the examples of scripts that are provided. This method provides a more smooth migration of users – the way in which users use the system always remains the same and all types of users with differing levels of experience remain within the same system.

8.3 VR Complexity Support

Dive and Avango provide a scripting language to minimally manipulate objects in the environment. They also provide a low-level API 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 VR 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 short-coming to this approach is that they provide the pre-created and automated functions, sacrificing the ability to extend and expand the system: to allow for the creation of new functions. If systems do provide the feature, to do so is usually difficult and goes against overcoming the concept of "VR software is difficult to create".

In our system, since everything in the system is represented as an *attribute variable* accessible through the scripts, and the entire environment behavior 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 go past using the scripts.

8.4 Shortened Development Time

Shortening the time for creating VR applications has been achieved in different ways. The most perceivable of these is the move from compiling and from API code towards scripted behavior. Even still, systems such as Dive and Alice bind the scripting language being used to the environment. In doing so, they provide a set 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 API. Dive, for example, gives a reference manual for their more than 300 Tcl/C interface functions [8].

Our system provides a minimal set of language independent functions for manipulating the *attribute variables* in the environment. In doing so, a user does not need to learn, on top of the scripting language, the API provided with it in order to create interactions. For non-programmers, specifying interactions can be simulated with a point-and-click interface system.

8.5 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 modularized system for some components that allow them to, for example, add new hardware devices.

Our modularized 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 4, or use different graphics libraries, such as OpenGL or Direct3D.

9. CONCLUSION

VR authoring systems have grown, although in the past, few systems have been targeted to novice users. With its 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 some of the problems present in 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 VR application would be impossible, we have proposed a more progressive system – a meta-authoring tool that would rapidly generate a VR authoring system for a particular VR solution.

The system we have created allows for smoother progressive migration of 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.

10. REFERENCES

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

[2] Bangay S., Gain J., Watknis 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] Frecon E. The Dive/Tcl Reference Manual.

[9] Frecon E., Hagsand O. The Dive/Tcl behaviour interface reference document.

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

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

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

[13] 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.

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

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

[16] Strauss P., Carey R. An Object-Orientated 3D Graphics Toolkit. *Computer Graphics*, 26:341–349, July 1992.

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

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

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

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

[21] 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.

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