FAVE –
A Framework Architecture for Virtual Environments

Daniel Patel, Inge Kristian Eliassen, Tor Langeland

Christian Michelsen Research
P. O. Box 6031 Postterminalen
N-5892 Bergen, Norway
{daniel,ingekr,tor}@cmr.no

Abstract

We present a highly configurable, rapid prototyping framework for immersive virtual environments. The framework is object oriented, event driven and has been designed with collaboration over networks in mind. It consists of several abstraction layers and can be used as a black box framework. Application behaviour is separated from graphical data and is specified using XML. This enables development by non-programmers and facilitates comparisons of different user interaction strategies. Issues regarding 3D user interaction have been investigated, including an efficient, context-sensitive, hierarchical extension to the command and control cube. The framework has been used to prototype an urban modelling application.

Keywords: Virtual environment, 3D user interaction, rapid prototyping, configurable framework, authoring system, urban modelling

1 Introduction

Immersive Virtual Reality (VR) is an aid to mastering spatial complexities, by facilitating improved understanding of 3D data and offering new modes of interdisciplinary collaboration. Immersive VR is gaining usage within a range of application areas, e.g. design, medicine and oil and gas exploration and production. Several frameworks are available, commercially or as open source code, for facilitating development of VR applications. These frameworks offer support for basic VR functionality such as stereoscopic viewing and device sampling. Some of them facilitate collaboration over networks. Several frameworks come with a scripting environment.

This paper presents a highly configurable, rapid prototyping framework for virtual environments. A central property of this framework is that the user interface and application behaviour are specified using the XML language. This enables implementation of VR applications without programming. The framework utilizes existing tools and frameworks. Although we currently use OpenGL Performer for graphics and CAVELib for VR device interfaces, these components can easily be substituted by e.g. OpenIventor and VR Juggler.

One important motivation for developing a system facilitating easy configuration of user interface and application behaviour is that we are interested in comparing user interaction methodologies for VR in general. Challenges related to developing efficient user interfaces for virtual environments (VE) delay the utilization of the full potential of the VR technology [9], [3]. User interface design is held to be one of the great challenges within computer science [18]. The FAVE framework facilitates easy and rapid comparisons of different user interaction strategies. In the urban modelling prototype we have used these capabilities to experiment with different user interface elements.

Section 2 describes some related work. Sections 3-6 present and discuss the design of the FAVE framework; its kernel, its configurability, interactivity and GUI library. Section 7 describes an example application for urban modelling (Figure 1), which was implemented using the FAVE framework. Section 8 gives some concluding remarks.

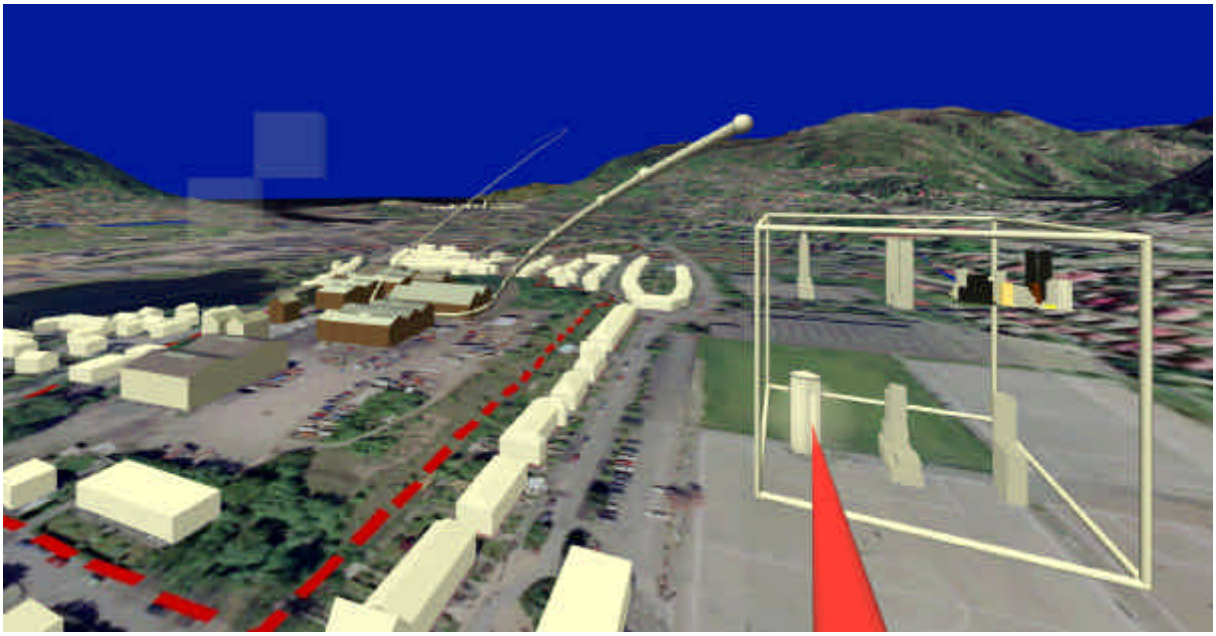


Figure 1: A model of the Bergen area with buildings at Kronstad (near) and downtown (far). To the right is a menu for importing buildings. Also shown is a spline with control points representing an editable flight path.

2 Related work

VR frameworks cover many different domains. Frameworks like CAVELib and VR Juggler offer support for basic VR functionality such as stereoscopic viewing and device sampling. The Quanta (previously CavernSoft) framework is an API for network communication, such an API is necessary for extending a single user VR application to a collaborative application. Many higher-level VR frameworks use these or similar building blocks accompanied by specialized graphics libraries or commercial ones like OpenGL Performer. In addition some frameworks come with a scripting environment, making them more flexible and configurable. Frameworks supporting scripting often also support the dataflow approach seen in SGI's Open Inventor and in VRML 2.0. In Inventor and VRML 2.0 the flow of data between virtual world entities is established by creating channels or connections between different entities' attribute values. An outgoing event of one object can be routed to an incoming event of another object.

The GNU/MAVERIK [7] framework was developed in 1995 and has been around for some time. It was built from ground not using any of the above mentioned frameworks. It deals primarily with device sampling and graphical and spatial management and offers different display management structures. It has been extended to the multiuser application DEVA [12]. Some frameworks, e.g. VIRPI [4], focus on interaction with scientific simulations, possibly where simulation and visualization run on different machines. VIRPI has a high-level VR steering system in a scripting language and offers techniques to start, store, browse and replay simulations. Avango [17], Lightning [13] and Ygdrasil [11] are

frameworks with support for scripting and the dataflow approach as described above. All three frameworks build on OpenGL Performer. Avango supports Scheme scripting, Lightning supports Tcl scripting and Ygdrasil uses its own scripting language. In Lightning the dataflow model is implemented with objects having slots that can connect to other objects' slots. Analogously Avango has objects with fields, where a source field can be connected with another object's field by using 'fieldconnections'. Ygdrasil operates with messages and events that can be sent, received and reacted on by objects that are nodes in the scene graph. Avango and Ygdrasil support distributed scene graphs. Avango lets each process have a copy of the entire scene graph by using the reliable multicast mechanisms implemented in Ensemble [6]. In Ygdrasil processes store a copy of the parts of the global scene graph they 'want' by using the CavernSoft network package. The FAVE framework falls roughly into the same category as these three frameworks.

3 System overview

The FAVE framework is designed to be highly configurable and to allow easy prototyping of new applications. It is event driven and adaptable for collaborativity. The framework has a modular design implemented in C++ and it consists of several layers of abstraction.

A simplified schematic layout of component dependence is depicted in Figure 2. The application is built on top of basic libraries for device input and graphics output such as CAVELib and OpenGL Performer. The design makes it easy to substitute CAVELib with for instance VR Juggler and facilitates use of other scene graphs than OpenGL Performer.

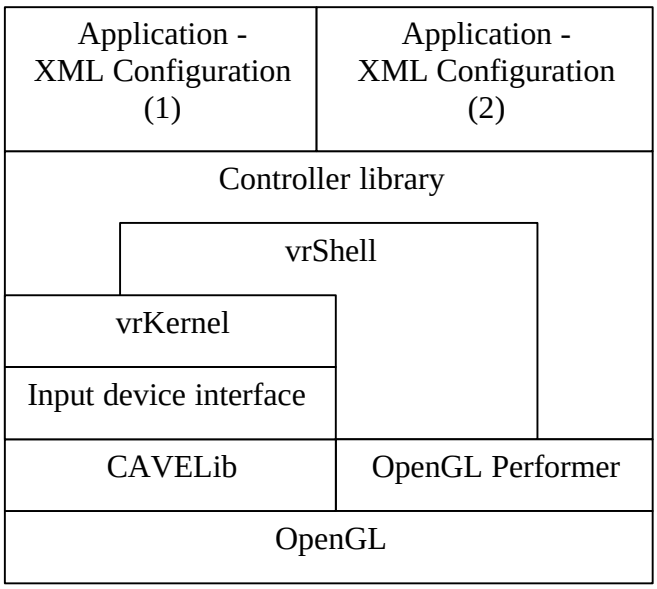


Figure 2: System architecture and dependence of main components

A central part of the system is the vrKernel. Its main tasks are event management and input device handling. To make the vrKernel independent of the libraries it uses, input devices are accessed through an abstract interface layer.

The vrShell is layered on top of the vrKernel, it implements methods for XML configuration, user interaction, navigation, object selection and manipulation. This layer uses some functionality from the scene graph library (OpenGL Performer).

Above the vrShell is a library containing a collection of controller classes. The behaviour of all objects, graphical and non-graphical, is encapsulated in controller objects, they are the only objects capable of receiving events. A controller can be used as part of the MVC (model, view, controller) design pattern [10]. Each controller provides a specific functionality which enables it to be used as a widget or as a tool with complex behaviour (subsequently referred to as tool in this text). The system makes no clear distinction between these types of uses. The collection of controllers constitutes a library which is used to design application functionality and appearance.

The top layer is the behaviour configuration, which specifies controller settings and relationships. This layer, although it consists entirely of XML, constitutes in a sense the actual application in that the vrShell acts as an interpreter of the specifications given. When a new application is being developed most of the work will be done on the XML behaviour configuration, in addition to possibly implementing new application specific controllers that are not part of the controller library.

4 Core functionality: the vrKernel

We refer to the core of the framework as the vrKernel. Parallels can be drawn to what a micro-kernel does for an operating system [7]. The vrKernel is a small C++ framework handling device sampling, event propagation and offering utility functions such as math operations for 3D graphics. Functionality for handling graphics is outside the vrKernel's domain. The vrKernel maintains and administers a registry of all objects that are part of the event system thus assisting the developer with object deletion and management. The vrKernel has somewhat similar responsibilities as MAVERIK [7], AURA [13] and the Application Control Module of Lightning [1].

4.1 Input device handling and event system

The vrKernel defines an input device as an object with buttons (sequence of Booleans), valuators (sequence of floats) and sensors (sequence of 6DOF – position and orientation). This general definition allows the vrKernel to handle a wide range of existing and possibly future input devices. An important aspect of the vrKernel is its input device abstraction layer. By implementing new abstraction layers, other device-sampling libraries and thus input devices can be used without changing any existing parts of the system.

Our system operates with two types of events: device-events and high-level events, both received by controller objects. A device-event is a discrete low-level event, like a button click, created by an input device. A high-level event is an event carrying a command signalling a particular action to be performed possibly along with command parameters.

4.2 Controllers

The vrKernel offers the Controller class as a virtual base class for the programmer to subclass. Subclassed controllers are the objects that encapsulate the behaviour of a VR application. Controllers receive device-events from the vrKernel, refine them into high-level events, and react on high-level events sent by other controllers by performing actions such as updating their data model (the scene graph) and creating new high-level events. Thus when new application specific functionality is needed, it is sufficient to implement new controllers.

The vrKernel has mechanisms for event propagation allowing controllers to send high-level events to other controllers in the system via an event queue. It handles event

subscription allowing controllers to subscribe to input devices and thereby receive device-events. And it takes care of creating 'tick' events allowing controllers to subscribe to regular ticks. If a controller is to perform some continuous action over a period of time it will need to receive control regularly, and it does so by subscribing to a tick event. For instance updating a geometry's transformation based on the position of some input device requires a regular tick. For each tick it reads the state of the input device and updates the geometry's transformation. When the controller is finished updating, it unsubscribes to the tick event.

4.3 Multithreading

The vrKernel operates with two threads; the event creation thread and the event handling thread as shown in Figure 3. The threads communicate with each other through a thread safe event queue. The event creation thread creates high-level events. Such an event is generated after the DeviceManager in the vrKernel has polled its input devices, created device-events from them and given the device-events to the controllers subscribing to them. Some controllers will then refine a device-event into a high-level event, set a recipient controller for the high-level event, and put it in the vrKernel's event queue. Since the device sampling happens in this thread it is important that it cycles quickly, hence the refine action of a controller must be swift. The event handling thread will pop an event from the queue, and give it to the recipient controller allowing the recipient controller to perform the described action.

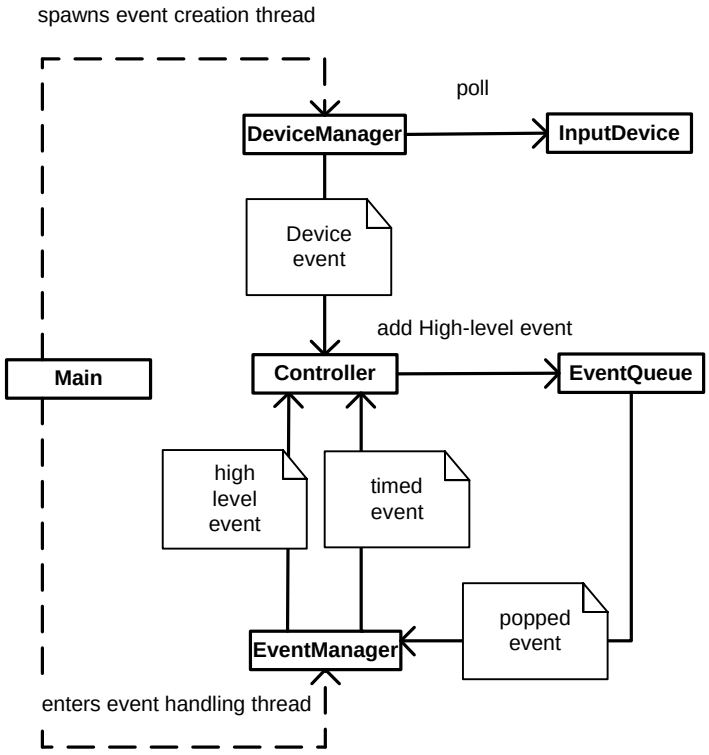


Figure 3: Event system: Main function spawns the event creation thread and then enters the event handling loop. The DeviceManager handles event creation and the EventManager takes care of event handling.

The event creation thread also handles creating and sending tick events. Thus controller objects are being accessed from both threads. The event creation thread will only call controller functions that do not change their state. Consequently, no synchronization or locking of data in the controllers is needed. So if a high-level event requires more time to be

handled than the time of a poll cycle, the system will not freeze. Input devices will still be sampled from the event creation thread, and new high-level events will be created and stored in the queue, waiting for the handle thread to get ready to process them.

4.4 Distributivity

The vrKernel has been designed with distributivity in mind. In particular this influenced the event system design, resulting in a consistent event system with a rigorous set of rules. When the programmer follows these rules, all data-changes that alter the system state are isolated as high-level events and these are the only data necessary to distribute. This means that for distributive VR applications to be synchronized, it is enough that they start up in the same state and that they receive each other's high-level events. The result is a system allowing remote distributivity with low network traffic.

Our approach differs from other solutions which might create higher network traffic such as using virtual pipes, each with different viewpoints [16], duplicating the shared parts of the scene graph [11] or using distributed shared memory (DSM) [17].

5 The vrShell

The vrShell uses the vrKernel as an abstraction layer for handling common low-level VR functionality. The vrShell is a framework for building applications by using XML to describe application behaviour and scene graph files to describe application geometry. XML is used to interconnect the flow of events between controllers. This resembles VRML [19] and the XP part of Ygdrasil [11] except for the fact that we have separated the geometrical data description from the behavioural description. In cases when the vrShell does not contain all the behavioural elements (controllers) required for a specific application to be fully described in terms of XML, the new behavioural element must be implemented by subclassing the controller class. Implementing controllers such as widgets and tools is simplified by the vrShell through its library, which supports interaction features such as picking, highlighting, context menus and displaying state symbols and tooltip information. Since these features are implemented in the vrShell and not in each application using it, we hope to get consistent and familiar interaction behaviour between all applications built on the vrShell.

5.1 Configurable behaviour

A FAVE application consists of three distinct parts; executable code, graphical data and XML configuration for behaviour. The XML configuration files contain a high-level description of the behaviour of the application, i.e. how graphical objects respond to interaction, in terms of which controllers are attached to graphical objects and how these controllers communicate with each other. The separation of configuration and geometrical data makes it possible to use the same geometrical data with different behaviour configurations, or to apply a behaviour configuration on different geometrical datasets. The separation also reduces the dependencies between the application framework and the graphical library.

The coupling between geometry and behaviour is specified through labelling various parts of the geometrical data. Each label identifies a controller specified in the XML configuration. When a geometry data file is loaded, the resulting scene graph is traversed and when a label is encountered a controller is instantiated according to how the label is specified in XML (label specification) and is attached to the corresponding sub tree in the scene graph. We refer to this sub tree as the controller's geometry, and conversely we refer to the

geometry’s controller. When a user selects this sub tree, the geometry’s controller is selected and receives user input.

An XML controller label specification defines a controller type. An instance of this type is created for each label in the scene graph referring to it. Alternatively, a controller type can be declared global, in which case only one instance is created. That one instance will then be shared by all sub trees with labels referring to it. The label specification specifies the controller class, its initial values and communication settings. Thus one controller implementation can be instantiated several times with different settings.

The configuration specifies how different controllers communicate with each other in terms of events. This communication goes through a communication manager routing events from a source controller to one or more recipient controllers. The XML configuration of a controller type includes a map from other controller types which it wants events from to what event that should result in for the controller, thus linking controllers by defining communication channels between them. An example is shown in Figure 4, where two buttons are defined for allowing the user to switch navigation between walking and flying. The two button definitions result in the instantiation of two buttoncontroller objects. The objects will send an event to the communication manager when selected. The communication manager then routes the event to the ‘NavigationController’ with event code ‘WALK’ or ‘FLY’ depending on which buttoncontroller sent it. Figure 6f shows the geometries of the button controllers as two icons in a menu. The navigation controller is not attached to any geometry. Instead it is specified to be instantiated independently of geometry by using the <create> tag.

```
#Inventor V2.1 ascii

Separator {
   Separator {
      DEF WalkButton Rotor { on FALSE }
      File { name "icons/sphereMan.iv" }
   }

   Separator {
      DEF FlyButton Rotor { on FALSE }
      File { name "icons/bird.iv" }
       }
    }
```

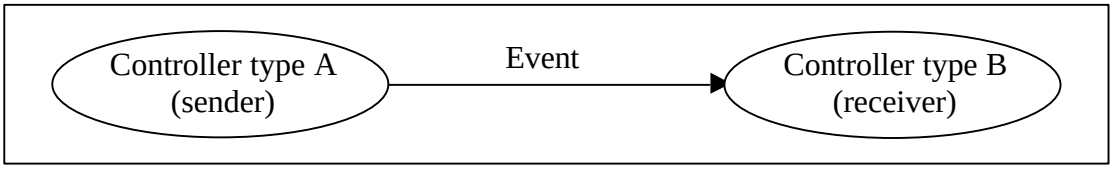
a. The Open Inventor data file. Rotor nodes are used for labelling geometry.

```
<create>NavWandController</create>

<NavigationController name="Navigation">
   <commonArg contextMenu="NavigationMenu"
      toolTip="This is the navigation controller"
      highlightable="true" temporary="true"/>
   <listen source="WalkButton" event="WALK"/>
   <listen source="FlyButton"  event="FLY"/>
</NavigationController>

<BUTTON name="FlyButton"  global="true"/>
<BUTTON name="WalkButton" global="true"/>
```

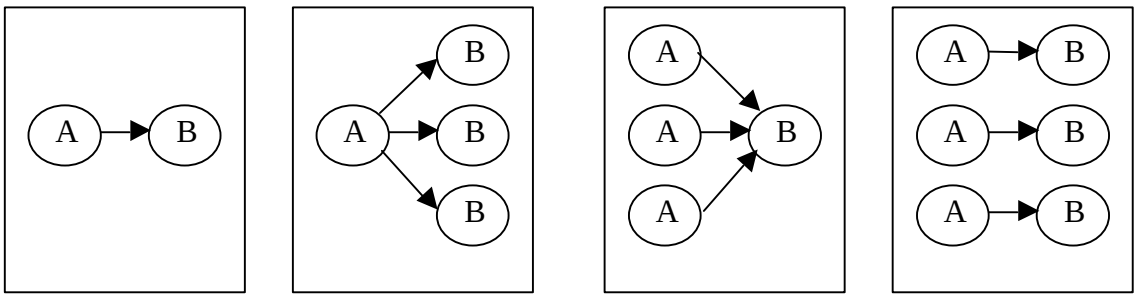
b. The XML configuration file.

Figure 4: Connection between graphical data and XML configuration. The data file contains two labelled geometries. The XML label specification configures them as buttons for switching navigation method between walking and flying. The attributes in the commonArg tag are explained in Section 5.2.

Since the communication between controllers is specified in terms of one controller type listening to another controller type, a special procedure is applied when both types are non-global, i.e. there might exist more than one instance of either controller type. The communication manager keeps track of which instances belong together, and in this case the communication manager only routes events to controllers in the same instance group as the source controller. This makes it possible to correctly manipulate several object of the same type. Figure 5 illustrates the different communication patterns depending on whether sender and receiver types are global or non-global.



a. Specification of controller type A sending events to controller type B.



b. Both A and B are global.

c. A is global and B is non-global.

d. A is non-global and B is global.

e. Both A and B are non-global.

Figure 5: The XML communication specification between two controller types illustrated in a results in different communication patterns between instances of those types depending on whether the types are global or non-global as illustrated in b – e.

The collection of controllers can be looked upon as a toolbox of behavioural elements used in the configuration. As new controllers are programmed, the toolbox grows more powerful. To use and interconnect controllers using XML, it is sufficient to know the controllers' interfaces; which events they send, which events they can receive, and which initial values and parameter settings they take. This creates a black box framework[1], which can be used as an authoring system, allowing non-programmers to work with development of application design and functionality.

Existing tools for manipulating XML has further simplified the configuration process. Such tools allow the user to interact with forms generated from an XML schema describing the structure of a legal specification. We have created XML schemas describing each controller's XML syntax. Our schemas utilize many of the sophisticated features in the schema language resulting in a tight verification of the XML configuration files and eliminating the need for the programmer to create code in the vrShell to perform structural checking of the XML files being parsed. In addition we made a schema transformation converting the XML configuration file to a cross-referenced HTML file readable by people not familiar with the XML syntax.

5.2 Common Interaction features

The vrShell facilitates rapid development and consistent interaction behaviour among the applications. For instance when a geometrical object associated with a controller is 'pointed at', the vrShell will draw a bounding box around it indicating that it is selectable (unless it

[1] "A black-box framework is one where you can reuse components by plugging them together and not worrying about how they accomplish their individual tasks. In contrast, white-box frameworks require an understanding of how the classes work so that correct subclasses can be developed." [8], [14]

states in XML that it is not highlightable). This helps the user distinguish between 'static' objects and the ones that can be interacted with.

The vrShell offers selection by ray intersection as the default selection technique, in this mode a selectable geometry is selected by pointing the ray at it and left-clicking. Then the geometry's controller will receive a 'select' event and will automatically start receiving device-events from the input device that was used for the selection. When another controller is selected, the first will receive an 'unselect' event and stop receiving device-events and the new one will start receiving device-events.

Controllers can be temporary or non-temporary. Temporary controllers will after selection unselect themselves as soon as their task is completed and the vrShell will give back control to the most recently selected non-temporary controller. This mechanism has proved itself useful for quickly switching among different tools and widgets because it removes the need to reselect the tool that was previously used, which typically would require the user to accurately point the ray at the geometry of a tool.

A controller can have a context menu, which is another controller. When a controller with a context menu has been selected and the right button is clicked, the context menu controller will be selected. For instance if the first controller is non-temporary and the context menu controller is temporary and shows a menu, the user can interact with the menu, and when a menu choice is performed, the menu controller unselects itself and control returns to the original controller.

A symbol can be associated with a controller. This symbol will then be visible on the right hand side of the wand as soon as the controller is selected and acts as a visible cue showing what controller is selected. As a default, the symbol shown is the controller's geometry. If a controller can be in different states, it can reflect this by specifying a state symbol for each state. The state symbol will then be shown on top of the wand. In Figure 6e a slider is being turned. It is shown next to the wand with its value on top of the wand.

Controllers can specify a tooltip. If such a controller is pointed at for a specified time it will send a high-level event with a tooltip text string and its position. This event can be targeted at a controller that displays the received text at the specified position.

Figure 4b shows an example of XML specification of these features.

6 Interaction techniques

The toolbox of controllers on top of the vrShell implements a wide range of interaction techniques. Interaction techniques are typically divided into four groups: navigation, selection, manipulation and system control [2]. We have grouped our implemented techniques accordingly in this section.

6.1 Navigation techniques

Different techniques for navigation in the virtual world has been implemented. For all techniques collision detection can be turned on or off.

- The default technique is flying in the direction the wand points. Flying speed is a linear function of the wand's forward/backward joystick deflection, and increases linearly in the distance from the ground. Having maximum speed slower close to ground than for high altitudes has proven to be intuitive even for inexperienced users.
- With gravity introduced one can navigate by walking on ground and inside buildings on floors and up stairs.

- To quickly move from one place to another one can teleport to the point the wand ray intersects. The user can also set a flag so viewpoint smoothly turns 180 degrees under the flight resulting in a new viewpoint looking back at the previous location.
- To get an overview of an area or geometry one can circulate around a chosen point. The user can adjust the viewpoint while circulating.
- By recording and saving the path one is flying one can replay it later. This path can then be represented as a spline with control points, as seen in Figure 1, which can be manipulated by deleting, adding or moving the control points.

6.2 Selection techniques

We have implemented three different ways of performing a selection, for all three techniques a bounding box is drawn around the object in focus if it is selectable.

- The most used selection technique is ray casting where a ray is drawn from the tip of the wand to the intersection of some geometry. The geometry is then selected (if it is selectable) by left clicking.
- The second technique is 'tabbing'. In this mode the ray disappears and by moving the joystick left and right, tabbing will take place between all selectable objects in the same menu level. By left-clicking, the object with tab focus will be selected. If the object has a submenu, the submenu will be shown and the tab focus will move to the objects in the submenu. Moving the joystick up will move the tab focus back up to the previous level.
- Finally we have the technique of touching. In this selection mode the ray is replaced with a semi-transparent ball at the wand tip (Figure 6f). An object is selected by placing the ball over the object and left-clicking.

6.3 Manipulation techniques

Different ways of manipulating the VE include:

- Moving, scaling and rotating objects; Some objects can be moved, scaled and rotated by first selecting them and then selecting a transformation type. How the input device maps to the translation factors is configurable in XML.
- Importing and deleting geometry; geometry can be deleted, and inserted into the scene by selecting from a menu of importable geometries. Imported objects need not be static geometry, they can have behaviour associated with them. Geometry can be imported from a wide range of data formats.
- Using a switch tool to turn on and off the visibility of a collection of objects. This is done by interacting with a geometry representing the switching tool.
- Creating and manipulating a spline by grabbing, moving, deleting or adding new control points, thereby manipulating the shape of the spline. The thickness and degree of a spline and the radius of the control points can also be modified.

6.4 System control

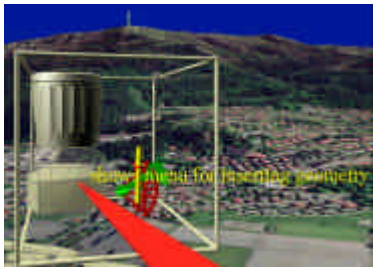
System control is defined in [2] as the task of changing the interaction mode or the state of the system by issuing commands. These commands are issued in our system by means of interacting with controllers.

The Controller class is used to implement simple widgets, such as buttons, sliders and menus, as well as more complicated tools, e.g. for managing editable flight paths. The

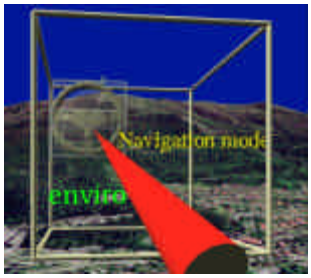
distinction between widgets and tools is not always as well defined in virtual environments as in traditional desktop WIMP environments. Some controllers in the scene can be multi-functional and act as both widgets and tools. One difference, though, is that the controllers acting as widgets tend to provide more elementary behaviour and will probably be reused across different application domains, whereas the more advanced tools are generally more bound to specific application domains.

We have implemented controllers that can be used as a widget library. Figure 6 shows example usages of these widgets. Each instantiated widget will have a tag in the XML file specifying its behaviour as described in section 5. The central widgets in this library are buttons and toggle-buttons for performing discrete commands (the icons in Figure 6 a-d are buttons), sliders for specifying continuous values (Figure 6e), text windows for showing lines of text and menubars (Figure 6d) and C3 cuboids (Figure 6a-c) as two different ways of presenting a hierarchy of commands to select among.

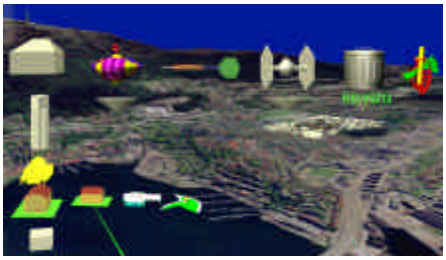
a) Main menu as C3 - front vertical section.



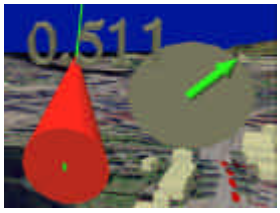
b) Main menu as C3 - middle vertical section.



c) Main menu as C3 - back vertical section.



d) Main menu as a menubar.



e) A slider's value is displayed in two ways.

f) Two button representing walk and fly modes.

Figure 6: Various controller widgets, such as menus, sliders and buttons.

The C3 cuboid was inspired by ideas from the Command and Control Cube in [5]. It is a powerful metaphor for structuring and presenting collections of commands, possibly in hierarchies, using very little screen area and having quick and precise navigability. We have called the widget a C3 cuboid due to its cuboid (3D rectangle) shape, its edges are drawn as lines and represent the boundaries of the C3 cuboid. The cuboid is subdivided into $x \times y \times z$ equally sized smaller cubes, each containing a geometry representing a controller. When opening a C3 cuboid, touch mode for selection is activated and the cuboid is placed at the wand tip, then only the commands (controllers) in the plane extending from the ball and facing the viewer are shown. The dimensions of the cuboid, position, type and geometry of each command are specified in XML. Cuboids can be nested since controllers inside cuboids can be cuboids themselves, in addition controllers can specify cuboids as their context menus.

7 Example application: Urban modelling

For the purpose of evaluation and gaining experience the FAVE framework has been used to prototype an urban modelling application. The application is developed for an SGI Onyx

graphics computer, but could also run on a pc or any other platform with support for the libraries used; CAVELib and OpenGL Performer. The application makes use of head tracking and a 3D mouse with three buttons and a joystick.

The data for this application consist of terrain of an approximately 10×10 km^2 area and contains buildings from downtown Bergen and Kronstad 3 km from downtown Bergen. The terrain for Kronstad is also modelled in increased resolution, this is displayed as a higher level of detail when the user navigates sufficiently close. The buildings in the two areas have different origins. Buildings at Kronstad are recreated manually from aerial photos while the downtown buildings are generated automatically from data in the SOSI format [15].

The application was developed with two goals in mind. It should be both a test case for the framework architecture, and an extensible prototype of an urban modelling application. Focus has been on user interaction techniques for navigation, selection, manipulation and system control.

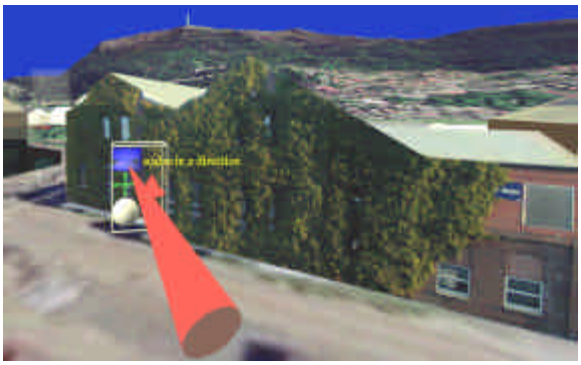
The application is configured with one global main menu and several context menus specific for various tools and objects. All menus are configured as C3 cuboids. The main menu has tools for navigation, for importing graphical objects, such as buildings, and for changing system settings. The context menus provide functionality for changing settings for tools and for manipulation of objects.

Objects in the scene, such as buildings, may have menus attached to them. Such a menu is accessible by selecting the corresponding object. Thus, a building is both a geometric object part of the scene and a button for accessing functionality specific for that building. This multi-functional role prevent a clear distinction between widgets and other objects.

The main menu is configured in two different versions for comparison reasons. One version is a C3 cuboid, the other is a menubar. Both versions are accessible by selecting semi-transparent icons located in the upper left part of the screen. These icons are fixed in device space coordinates making them always close to the user, and by being semi-transparent and not in the central view area they don't disturb the sense of being immersed in the virtual environment. The C3 main menu is also a context menu attached to the ground geometry. This means that the main menu also can be reached by selecting the ground and right-clicking.

The context sensitive hierarchical C3 menu system has proven to be useful and effective. It is conveniently placed right in front of the user when activated, and is otherwise invisible and not cluttering up the view. The context sensitivity provides direct access to local functionality without having to navigate from the top of a menu hierarchy. The C3 selection system is easy to use since it does not require high precision pointing. Selection can even be performed without watching once the positions of the icons within the cuboid have been memorized. The hierarchical configuration of C3, i.e. icons in a C3 can lead to a new C3, provides a convenient way of grouping similar functions, and allows for an unlimited amount of commands. Extending the C3 to span more than 3 icons in one or more directions also allows for more commands. Although this impedes quick selection [5] it can be useful for listing many choices.

This application has proven to be a powerful tool for presentations and explorations. It can be used to investigate the visual impact on new buildings or changes to existing ones. Figure 7 demonstrates how the user can examine the changes in the view from a window in one building when a neighbouring building is extended vertically.



a) The original view. C3 cube for the house is shown.

b) The view during scaling. State symbol for vertical scaling is shown.

Figure 7: The view from a window before and during scaling the height of a neighbouring building. The view to Mt. Ulriken becomes barely visible after scaling.

8 Conclusions

This paper describes FAVE – Framework Architecture for Virtual Environments. It is a highly configurable rapid prototyping system. The main part of developing an application in this framework consists of specifying behaviour configuration in XML. XML related technologies, such as XML schema and XSL stylesheet, allow automatical and manual inspection of the XML configuration. Furthermore it is possible to use general-purpose XML editors that are able to generate graphical user interfaces based on XML schemas. This facilitates application development by non-programmers.

The user interface is intuitive and effective. We have developed menus (C3) having a natural spatial position relative to the user, thus enabling quick selection. We have intuitive mechanisms for structuring commands. Commands can be structured around tools due to context menus. Large collections of commands can be structured in a hierarchical fashion through hierarchies of C3 cuboids. Temporary controllers automatically reselecting the previously used tool support a natural flow of interaction.

The VE delivers to the user appropriate feedback with information regarding system mode. The user can see what tool is selected and what state the tool is in by means of state and substate symbols on the wand. The representation of the wand also signals what selection mode the system is in. Such feedback prevents mode errors; errors where users perform actions while believing the system is in another mode than what is the case. The VE also prevents the user from executing wrong commands by displaying command tooltips.

Acknowledgements

We would like to thank the City County of Bergen for providing (SOSI) data of Bergen used in the urban modelling application. We would also like to thank our colleagues at CMR; Tyge Løvset, Jarle Strand, Gro Grov Sørdal and Dag Magne Ulvang for assisting with the design and giving valuable feedback during the process of creating FAVE.

References

[1] Blach, R., Landauer, J., Rösch, A., Simon, A. (1998). "A Highly Flexible Virtual Reality System" *Future Generation Computer Systems* 1998 vol 14 nr 3-4 pg 167-178.

[2] Bowman, D., Kruijff, E., Joseph, J., LaViola., & Poupyrev, I. (2001). "An introduction to 3-D User Interface Design" *Presence*, Vol. 10,No. 1, Feb 2001,96-108. © MIT.

[3] Tanriverdi, V., & Jacob, R. (2001). "VRID: A Design Model and Methodology for Developing Virtual Reality Interfaces", *VRST'01*, November 15 – 17, Banff, Alberta, Canada, ACM press.

[4] Germans, D., Spoelder, H., Renambot, L., & Bal, H. (2001) "VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality" In *Proc. Immersive Projection Technology/Eurographics Virtual Environments Workshop*, Stuttgart, Germany.

[5] Grosjean, J., Burkhardt, J., Coquillart, S., & Richard, P. (2002). "Command & Control Cube: a Shortcut Paradigm for Virtual Environments" ICMI'2002, Pittsburgh, USA, 14-16.10.2002.

[6] Hayden, M. (1998 jan)."The Ensemble System" *Technical Report* TR98-1662, Cornell University.

[7] Hubbold, R., Cook, J., Keates, M., Gibson, S., Howard, T., Murta, A., West, A., & Pettifer, S. (2001 Feb.). **GNU/MAVERIK: A micro-kernel for large-scale virtual environments**. *Presence* 10(1):22–34, MIT Press.

[8] Johnson, R., & Foote, B. (1998 June/July) "Designing Reusable classes". *Journal of Object-Oriented Programming*, 1(2):22-35.

[9] Kessler, D. (1999). "A Framework for Interactors in Immersive Virtual Environments", Virtual Reality'99 Conference, March 13 – 17, Houston, Texas, 1999 IEEE Computer Society.

[10] Krasner., & Pope "A cookbook approach to using MVC", *JOOP* 1(3):26–49 Also: John Deacon "Model-View-Controller (MVC) Architecture", http://www.jdl.co.uk/briefings/MVC.pdf

[11] Pape, D., Anstey, J., Dolinsky, M., & Dambik, E. "Ygdrasil - a framework for composing shared virtual worlds", to be published in *Future Generation Computing Systems*, Elsevier Press. Homepage: http://www.evl.uic.edu/yg/

[12] Pettifer, S., Cool, J., Marsh, J. & West, A. (2000). DEVA3: Architecture for a Large-Scale Distributed Virtual Reality System, *Proceedings of the ACM Symposium in Virtual Reality Software and Technology 2000*, VRST '00, ACM Press, Seoul, Korea, pp. 33-40, ISBN 1-58103160-2.

[13] Renambot, L., Bal, H., Germans D., & Spoelder, H. (2001). "Lightweight Programming for VR: Towards a Persistent Virtual Laboratory" *Workshop on Advanced Collaborative Environments. August 6,* IEEE International Symposium on High Performance Distributed Computing (HPDC-10), San Francisco, California.

[14] Roberts, D., & Johnson, R. (1997). "Evolving Frameworks" *Pattern Languages of Program Design 3* Addison Wesley.

[15] SOSI – Systematic Organization of Spatial Information. http://www.statkart.no/standard/sosi/html/div/sosi_eng.pdf

[16] Staadt, O., Näf, M., Lamboray, E., & Würmlin S. (2001). "JAPE: A Prototyping System for Collaborative Virtual Environments" *EUROPGRAPHICS* 2001 Volume 20 Nr 3 pp C8-C16.

[17] Tramberend, H. (2001). "Avango: A Distributed Virtual Reality Framework", *Proceedings of Afrigraph '01*, ACM.

[18] Brooks, F. (2003 Jan.) "Three Great Challenges for Half-Century-Old Computer Science", *Journal of the ACM*, Vol. 50, No. 1, pp. 25-26.

[19] International Standard ISO/IEC 14772-1:1997. The Virtual Reality Modeling Language.