An Architecture for Constructing Large VRML Worlds

Lee A. Belfore II

Department of Electrical and Computer Engineering, Virginia Modeling, Analysis and Simulation Center (VMASC), Old Dominion University, Norfolk, Virginia 23529, lbelfore@odu.edu

The Virtual Reality Modeling Language (VRML) offers capabilities far beyond the virtual exploration of solid models. The scripting capabilities and the execution mechanism enable the creation of complex applications. Indeed, applications can be implemented as a collection of modules interacting in a decentralized fashion, communicating through the exchange of messages. In this paper, an architecture is presented for creating large VRML worlds. The architecture includes a definition of functional modules that serve as the foundation for the application. In addition, object model architectures are structured into three layers that separate pure behavior from editing controls and also from the interface to the visualization, supporting module diversity and reuse. An urban planning example application is presented that incorporates these ideas and features a graphical user interface (GUI), collision management, a simple simulation manager, and an interface to a web server. Finally, several object examples are described followed by a presentation of an example session.

Keywords: VRML, virtual reality, architecture, live updates

1. Introduction

The growth of the Internet has provided opportunities for technological advances in how people communicate information and ideas. Among the factors driving the advances is the desire to enhance the way internet content is presented to the user. On the surface, displaying a web page appears to be a simple matter of displaying text and images. With the introduction of programming capabilities offered by Java, Javascript, and other programming and scripting languages, the web page is dynamic and can be customized for individual users. Paralleling the advances in web technology, advances in computer technology support approaches that were until recently beyond the capabilities of machines generally available. As evidence and in support of these advances, a 3D interactive visualization language, the Virtual Reality Modeling Language (VRML), was created, has been standardized, and is widely available [1]. VRML offers an opportunity to demonstrate the utility in Internet delivery of interactive 3D web content.

Current web technology provides a variety of architectural options that distribute the responsibilities between client and server in various ways. An excellent overview of virtual world architectural issues can be found in [2,3]. From the literature, several VRML applications have been demonstrated and show promise in delivering three dimensional content over the Internet using the unique capabilities offered by VRML. The types of VRML applications fall into several broad, but non-exclusive, categories. These categories include education and training [4–9], virtual collaboration, [6, 10, 11], information visualization and retrieval [12–14], games and entertainment [11, 15, 16], design tools [3, 17], and modeling and simulation [11, 18–20]. Indeed, the applications include components in several of these categories. The application domains are also broad and include urban planning [19, 21], geographic information systems (GIS) [14, 22], medical [5, 9, 23], engineering [3, 17], business [6–8], and combat simulations [18].

Applications can have many general operational features. These features affect the flexibility of the application and the complexity of the implementation. A virtual world is an interacting collection of components that compose the world, each component manifested by solid models and/or behaviors. The structure of the virtual world is represented by a scene graph defining the interrelationship among components in the world. We define the extent of the virtual world by the way components are allocated. The extent can have a great impact on the capabilities of the world. In a world with static extent, all components are loaded and inserted into the scene graph at startup. Throughout the life of the session, the world scene graph does not change; components are neither added nor deleted from the scene graph. A world with reconfigurable extent allows components to be added or removed from the scene graph at run time, but the number of components is fixed at startup. A world with dynamic reconfigurable extent enables the addition of any number of components at run time, limited only by the resources of the client machine.

The three extent models vary in their capabilities and performance. A static extent world offers high performance that is well suited to many applications. In the event the world scene graph must be changed, changes must be made to source files and the world must be reloaded, resulting in significant latencies when reloading complex worlds. A reconfigurable extent world offers the flexibility to change the scene graph by including a fixed size pool of components that may be inserted.

Received: January 2000; Accepted: May 2001

TRANSACTIONS of The Society for Modeling and Simulation International
ISSN 0740-6797/01

Volume 18, Number 1, pp. 23-39

Since the pool of components is immediately available, the scene graph can be updated quickly. A reconfigurable extent world has two shortcomings. First, the number of components is fixed. If it is not known how users expect to modify the world, many components provided may be unused, resulting in significant startup latencies and greater resource requirements for many sessions. Second, if several different components are desired, additional demand is required at startup because a fixed allocation of each type must be provided at startup. Indeed, if too many different components are available, the reconfigurable extent world becomes impractical. A dynamic reconfigurable extent world offers a solution to the start up resource problems of the reconfigurable extent worlds. In dynamic reconfigurable extent worlds, components are allocated and added at run time, as needed. As a result, startup times are lessened compared with reconfigurable worlds. Dynamic reconfigurable extent worlds incur some implementation complexities not present in the other two extent models. The component architectures may require special features and treatment so that the component can be inserted into the world. Furthermore, inserting the component into the scene graph may require several steps (i.e., allocation, initialization, scene graph insertion, event routing). This paper describes an architecture for dynamic reconfigurable extent worlds.

Looking in more detail at some of the applications suggests the power and also the limitations of VRML technologies. We describe some of the more notable applications. The Nerve Garden architecture [10] describes how algorithms mimicking life processes are used to create and plant artificial plants. Nerve Garden is a collaborative laboratory allowing users to create and germinate plants in an applet window with subsequent insertion into a scene graph common to all users. Since the number of plots is fixed, the application likely uses a reconfigurable extent model along with some fairly significant coupling to the server. The CADETT interactive multimedia business learning environment (CIMBLE) [6–8] describes a virtual training environment allowing participants at distant locations to collaborate. Interaction among users is accomplished using a multiuser domain (MUD). A key contribution of this work is the description of a software architecture. [9] describes the architecture of a web based training system for endoscopic procedures treating abdominal aortic aneurisms. The architecture includes server and client processes, uses the VRML External Authoring Interface (EAI) [24]. Computations on the server model the dynamics of the simulated procedures. [3] describes an architecture for virtual prototyping in the context of web-based development environments. In this architecture, virtual prototypes are built using virtual components, suggesting reconfigurable extent, whose appearance is defined in VRML and behavior defined in Java through the EAI. Component interaction is managed at the highest level in a virtual reality prototyping simulation engine. In the realm of urban and geographic visualizations, several applications have been reported [14, 19, 25, 26]. Such applications require the manipulation and display of large data sets. The general approach followed is for the server to accept queries from client machine and then perform server side processing to satisfy the request. Each modification requires the entire world to be reloaded and thus uses a static extent model. For small data sets, reloading is not a significant issue; however, reloading large worlds can add undesirable latencies.

The primary contribution summarized in this paper is the development of a virtual world architecture for creating VRML based virtual worlds featuring large scale interactive, modular visualizations characterized by dynamic reconfigurable extent. Indeed, few VRML applications support the allocation of dynamic content at run time. The modularity of the application is achieved through the development of generic functional modules for the application infrastructure with a collection of objects composing the visualization. The application demonstrates the integration of a graphical user interface (GUI), dynamically allocated objects, standard architecture for objects, and the various modules that link these together into an application. More specifically, the fixed backdrop defines the work area where all additions and modifications occur. The object architecture is organized into three layers implementing the interface to the visualization, defining editing controls, and specifying the model behavior. The interface layer is sufficiently robust to be automatically generated, simplifying the integration of new models into the application. Furthermore, our architecture supports some simulation capabilities whose extent shall be explored in the future. The demonstration application is a tool developed for a local municipality to complement conventional marketing efforts for marketing an office park. The architecture is constructed so that updates to the world occur dynamically on the client machine, lessening the load on client, server, and network resources when compared with a full reload of the virtual world. For any modification to the world, the server logs for the session are updated. Finally, the server supports save and restore of client sessions by storing all client sessions and supporting user authentication [27].

This paper is organized into eight sections, including an introduction, a brief tutorial on VRML, a presentation of the architecture, a description of the object architecture, a summary of some example objects, a run through of an example session, a discussion of the architecture, and a summary.

2. VRML

A VRML application is typically a hierarchical organization of geometries, sensors, light sources, script methods, and grouping primitives assembled in a meaningful fashion. In VRML, these primitives are termed nodes. Each node can have several fields to define node appearance and also to define inputs to (eventIn fields) and outputs from (eventOut fields) the node. A node can define a solid model, such as a box; a capability, such as tracking the pointing device; or an arbitrary behavior, such as what can be defined in a script. Extensibility and model reuse are accomplished with prototype definitions that specify user defined nodes. The subset of VRML node types (language primitives) used in the architecture is discussed in this section. Several excellent VRML resources exist in the literature [28, 29] that the reader is encouraged to consult. Figure 1 gives an abbreviated overview and relationships of the more significant VRML nodes used in our application.

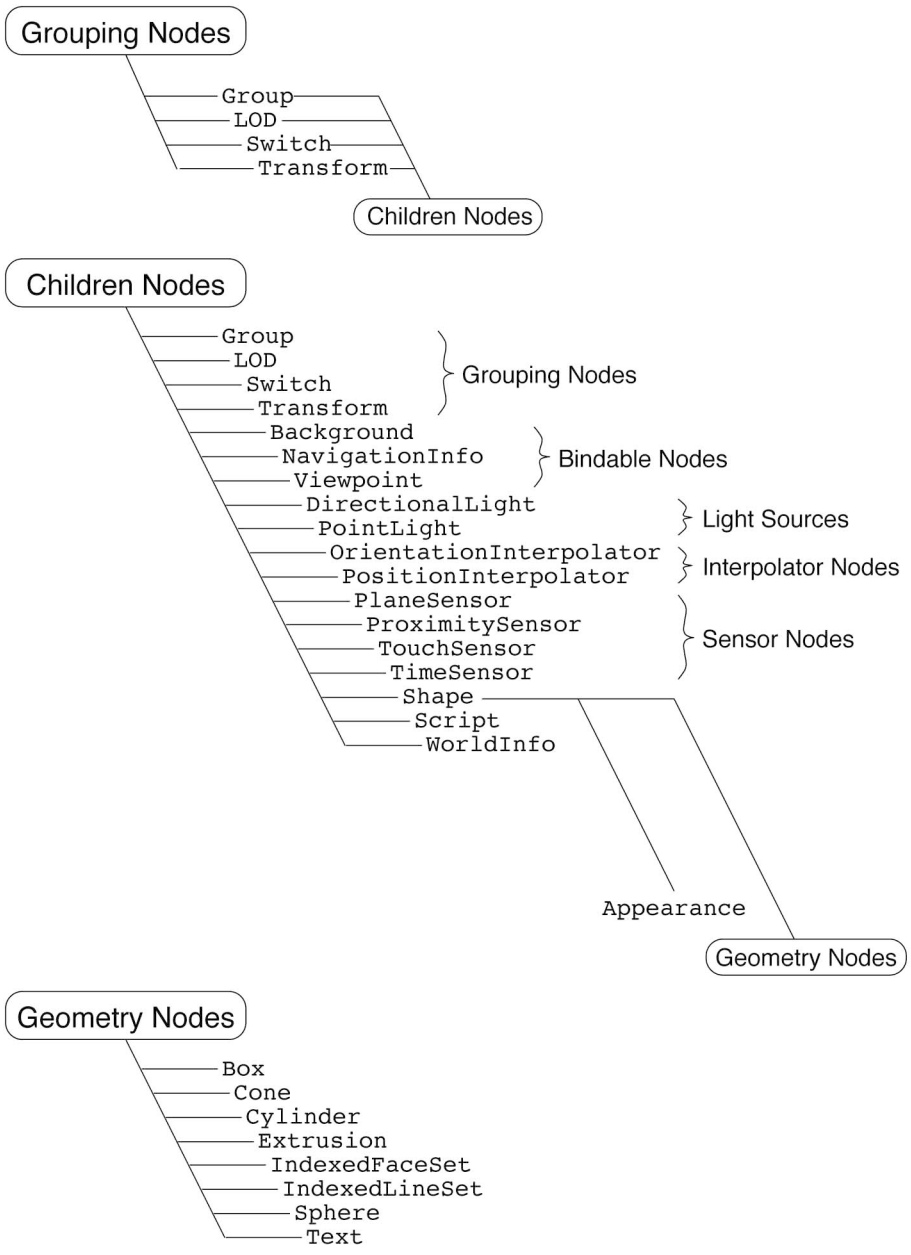


Figure 1. Abbreviated Overview of VRML Nodes Used in the Architecture

Grouping Nodes. The Group, LOD, Switch, and Transform nodes define how a collection of nodes can be grouped and also provide some useful control over their appearance. The Group node is the most basic grouping node that associates groups of children nodes. One effect that the Group node imparts is to associate a sensor with the entire child hierarchy. The Transform node provides the same capabilities, but additionally allows arbitrary scale, translation, and rotations of the child hierarchy. Nodes can be dynamically added and removed from Group and Transform nodes. The LOD node provides some control over the level of detail, or appearance as a function of the distance from the camera. The Switch node allows the selection of one node to be visible from a collection of nodes.

Sensor Nodes. VRML has a total of eight sensor nodes, of which four are used here, that generate events when the conditions occur to which they are sensitive. For example, the TouchSensor node generates an event when the mouse cursor is over an associated geometry and the mouse button is depressed. The PlaneSensor senses mouse drag actions while the mouse button is depressed. Using a TouchSensor in conjunction with a PlaneSensor enables a mouse click to select a geometry (TouchSensor) and a mouse drag to change the position of the geometry (PlaneSensor). The TimeSensor provides a mechanism for inserting delays and synchronizing different animations within the visualization.

Geometry Specifications. Shape geometries can be basic or general. Basic shapes such as boxes, cylinders, cones, and spheres are defined by name with fields to specify the expected dimensions. More general shape geometries include Extrusion and IndexedFaceSet geometries enabling specification of fairly arbitrary geometries. The Extrusion geometry is an analogy with the shape a material, such as modeling clay, makes after being forced through an opening with a varying cross section. The IndexedFaceSet provides a general appearance representing the object with simple faces.

Scripts. The Script node allows the implementation of arbitrary methods and behaviors. The script receives input, does the required processing, and then generates the necessary output events. For example, a script can take events from a PlaneSensor that is tracking a mouse drag and transform this into an output event consisting of a set of vertices that defines another geometry. In addition, Script nodes provide an interface to the browser plug-in to facilitate such things as inserting new nodes into the world or loading uniform resource locators (URLs). Script functionality can be provided using ECMAScript or Java methods. The Script interface can be customized to serve whatever purpose necessary. Each ECMAScript is a collection of methods including a method for each input event. ECMAScript methods provide basic programming structures (looping, conditional) that may call one and other as well as other plug-in browser methods. For dynamic worlds, several browser methods supply essential functionality. For example, the loadURL browser method can be used to request web content. The createVrmlFromURL browser method provides the run time function of allocating content dynamically and then inserting it into the scene graph at a selected point. Additionally, the addRoute and deleteRoute browser methods make and respectively break event routes between nodes. In the proper context, createVrmlFromURL in conjunction with addRoute can be used to allocate and then fully integrate any dynamic content at run time.

Fields and Events. VRML has a variety of built-in data types that can serve as variables or pass information between nodes. The basic types relevant to our application are outlined in Table 1. The fields can be further qualified by field, eventIn, eventOut, and exposedField qualifiers. The field qualifier declares a variable that can be changed only at instantiation. The eventIn and eventOut fields declare variables that are either inputs or outputs respectively. An exposedField variable serves as field, eventIn, and eventOut, simultaneously.

Events can be sent and received by most nodes. For example, a TouchSensor node generates an event resulting from a mouse click on the associated geometry. This event could be received by a script resulting in the appearance of a wire frame bounding the extent of the selected geometry. ROUTE declarations connect output events from one node to input events of another. Routes can either be explicitly declared or be dynamically formed in script nodes. Events can be simple, passing a simple value, or complex, passing a node, enabling many interesting capabilities. These capabilities include simple interactions with objects such as dragging an object through the visualization, transmitting information between models, and dynamically inserting objects. Finally, it is possible to circumvent explicit routing by directly accessing node input and output events within a script method.

Prototype Nodes. PROTO declarations enable the definition of new node types that can be used in much the same way as the standard VRML nodes. The prototype can be defined either in the same file or in a separate file declared using an EXTERNPROTO declaration. PROTO nodes provide the invaluable benefits of model reuse and hiding of implementation details.

The VRML Execution Engine. A VRML world can be viewed schematically as a scene graph where nodes describe functional units and routes define information flow between nodes. The

Table 1. A Description of some of VRML's basic types

Field Type	Description
`SFBool`	Boolean
`SFInt32`	32 bit signed integer
`MFInt32`	Array of `SFInt32`
`SFFloat`	32 bit floating point number (IEEE 754)
`MFFloat`	Array of `SFFloat`
`SFVec3f`	point in 3D space represented as three `SFFloat` values
`MFVec3f`	Array of `SFVec3f`
`SFColor`	RGB color represented as three `SFFloat` values in range [0,1]
`MFColor`	Array of `SFColor`
`SFRotation`	rotation in 3D space represented as four `SFFloat` values
`MFRotation`	Array of `SFRotation`
`SFNode`	VRML node or reference to a node
`MFNode`	Array of `SFNode`

```
. . .
DEF S Script {
      eventIn  SFBool set_create     #some precipitating event
      eventIn  MFNode set_node       #event that receives new node
      eventOut MFNode     addChildren #route this to desired group node
      field    SFNode     S USE S    #node that receives new node
      field    MFString   urls []    #used to help package browser call
      directOutput TRUE
      . . .
      url "javascript:
            . . .
      function set_node(val,ts) {
            // val contains the newly created node.
            // Node fields are initialized, routes are added, and
            //   the node is inserted into a grouping node
            addChildren=val;
      }
      function set_create(val) {
            urls=new MFString('foo.wrl');
            Browser.createVrmlFromURL(urls,S,'set_node');
      }
      . . .
"
}
. . .
```

Figure 2. Example Call to createVrmlFromURL Browser Method

Table 2. Major Events

Signal	Purpose	Source
`command`	user request	menu
`objectType`	send information about selected object, usually occurs after mouse click on object	object
`request`	object request for menu control	object, resource manager
`grant`	grants object menu control	main arbiter

VRML execution engine is an integral part of VRML browser plug-in that "executes" the scene graph [29]. Consistent with the existing routes, the execution engine receives events from the scene graph and then delivers the events to the respective destinations, assigning a time-stamp along the way. Events that cascade from an initiating event during the same time delta receive the same time-stamp.

3. The Application Architecture

In this paper, we present an architecture for representing large VRML applications that support dynamic reconfigurable extent. The architecture consists primarily of several generic functional modules coupled with an application specific GUI and objects.

A high level overview of the resulting software architecture is shown in Figure 3. The software architecture is a decentralized collection of seven interconnected, concurrently operating modules. These modules are (1) the GUI, (2) the resource manager, (3) the arbiter, (4) the fixed landscape (not shown in the figure), (5) the collision manager, (6) a simple simulation manager, and (7) the object models. Object models are self-contained and encapsulate all object capabilities. The significant events that drive the flow of information processing are presented in Table 8. The first six modules are discussed in this section. Because of special requirements and capabilities, objects models are discussed separately in Section 4.

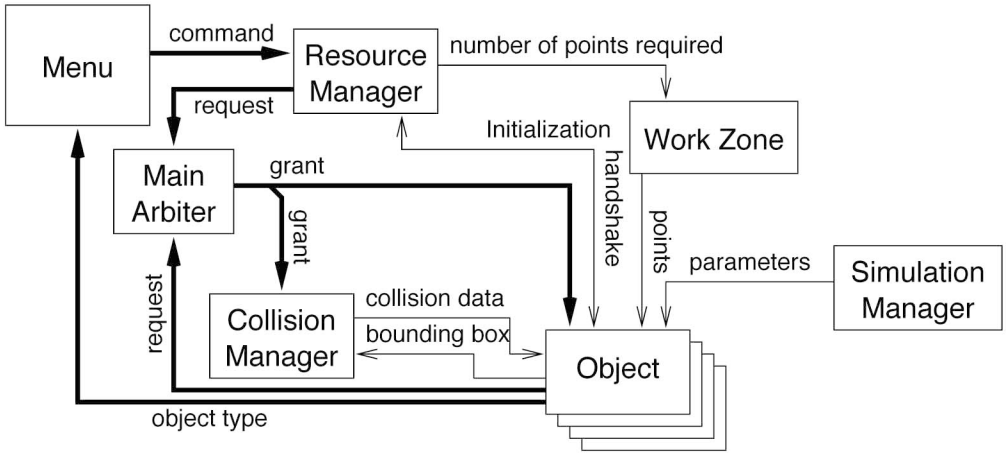


Figure 3. Overview of the Visualization Software Architecture

3.1. The GUI

The GUI enables the user to control the major operational modes and make specific input selections. The GUI consists of the menu and the work zone. The menu is a hierarchy of interconnected geometries as illustrated in Figure 4. The output produced by the menu hierarchy is the command event that initiates object addition and controls object manipulation. Elsewhere in the architecture, the command event is monitored by the resource manager and existing objects. The menu hierarchy receives the objectType input event, sent by an object selected by a mouse click, and results in the relevant menu appearing. Furthermore, the menu hierarchy is enabled or disabled as required during new object processing. Edges in Figure 4 represent the flow of events between menus. Individual menus are constructed from a collection of button nodes with an accompanying script node to encode button events into command events. The button node is defined as a PROTO node having the expected behavior. For example, the button changes color during the mouse-down interval to simulate the appearance of a depressed button. Also, the button function can be disabled and the button itself can be hidden. A typical menu is shown in Figure 5. The work zone defines the working area for adding and modifying objects in the session. In addition, the work zone provides a visual cue to the user during object insertion and detects the appropriate number of mouse click events to define the initial geometry of an object. Figure 6 shows the work zone in both active and inactive states. The work zone prototype is designed so that the work zone is specified parametrically.

3.2. Resource Manager

The resource manager is the key module in making architecture support worlds with dynamic reconfigurable extent. The resource manager manages the sequencing necessary to dynamically allocate and insert objects into the world. Upon receipt of an add command event from the menu, the resource manager creates a new object using the createVrmlFromURL browser method. The createVrmlFromURL browser method enables an application to allocate new VRML content, defined by a URL, and then dynamically insert the new content into a declared location in the scene graph. Figure 2 gives an example call as deployed in this application. The createVrmlFromURL browser method should not be confused with the addChildren events associated with grouping nodes. addChildren events require that the VRML content is already allocated within the application, whereas the createVrmlFromURL browser method allocates entirely new and even dynamic content from anywhere on the Internet. Immediately after allocation, the new object is assigned a unique serial number and is initialized. Once the object signals that initializations are complete, the resource manager connects routes between the object and the relevant managers using the addRoute browser method, fully inserting the object into the scene graph. The resource manager has tables to define the initial properties of objects and to track objects that have been added to or deleted from the current session.

3.3. The Arbiter

The arbiter controls access to a common resource, in this application, the menu, among a collection of objects. An object is selected by a user mouse click on any visible geometry on that object. The arbitration scheme employs centralized and decentralized processes characterized by two events—request and grant. Each is an SFInt32 data type that is used to communicate the serial number of objects involved in the arbitration process. The arbitration process has a centralized arbiter that receives all selection request events from objects and then identifies the object to be selected through the grant event. During quiescent times, the arbiter retains the serial number of the selected object and simultaneously monitors the request event that can come from unselected objects. Each object has a local arbiter module that submits requests for selection and then receives the results determined by the centralized arbiter. The local arbiter will initiate selection and deselection processes for the object. After an object has been selected, the object issues an objectType event. As described above, the objectType event is used by the menu hierarchy to identify and then display the appropriate menu for the selected object. A schematic outlining arbiter operation is presented in Figure 7.

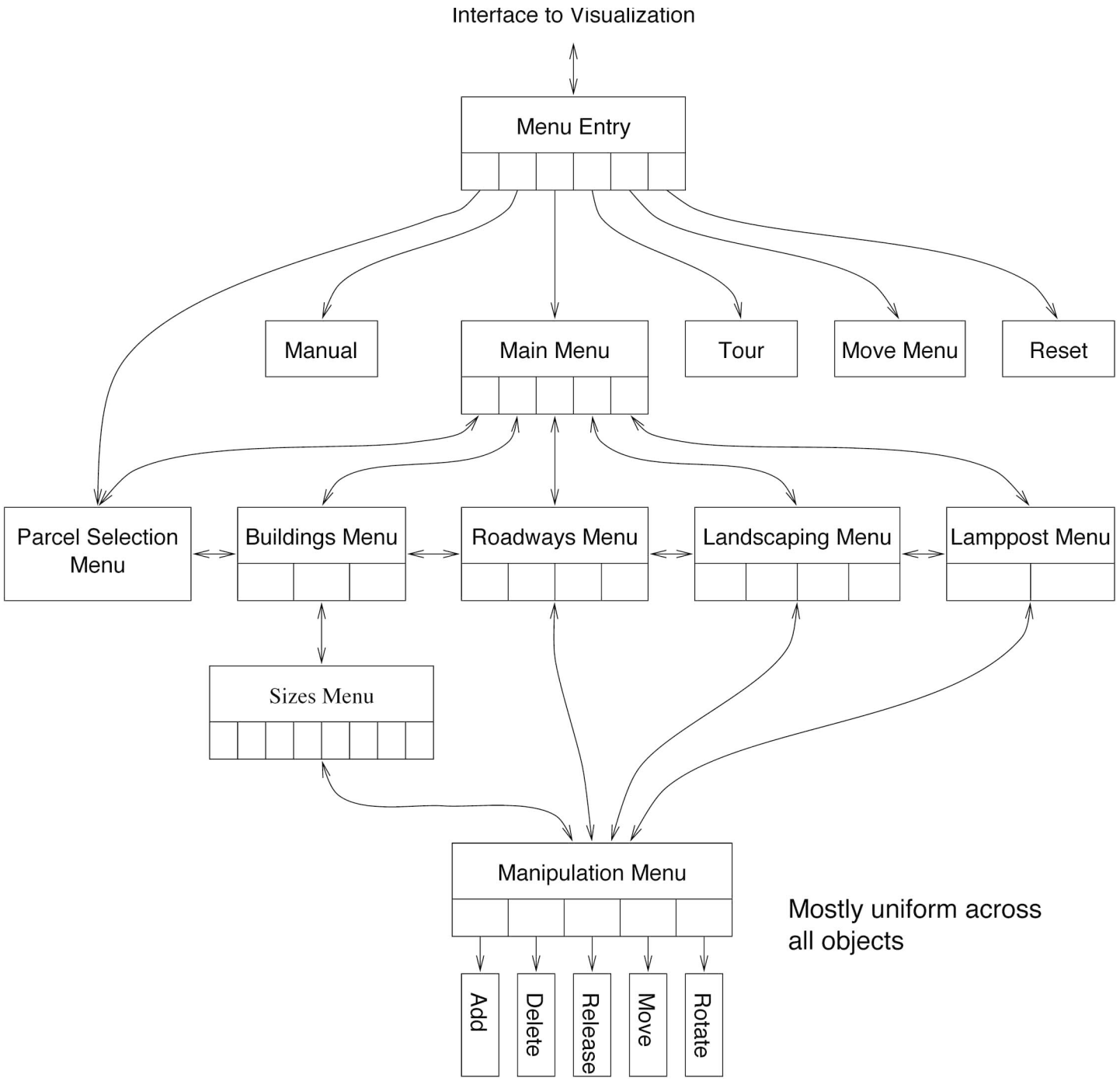


Figure 4. Menu Hierarchy

3.4. Collision Manager

The collision manager monitors spatial extent of all objects and signals when collisions occur between affected objects. The collision manager maintains a sorted list of bounding boxes for all relevant objects in the scene. In our application, only one object at a time moves or can have its geometry modified. When objects move or dimensions change, a bounding box event is sent to the collision manager. The collision manager compares the new bounding box to those of all other objects being monitored. When a collision is detected, a collision event is transmitted to the moving object. In this application, the response is simply to freeze the ability to move or otherwise modify the object. In more sophisticated simulations, the object behavior can use more complex bounding volumes and collision responses. While only simple collision detection is supported, the collision manager can be extended to support more sophisti-

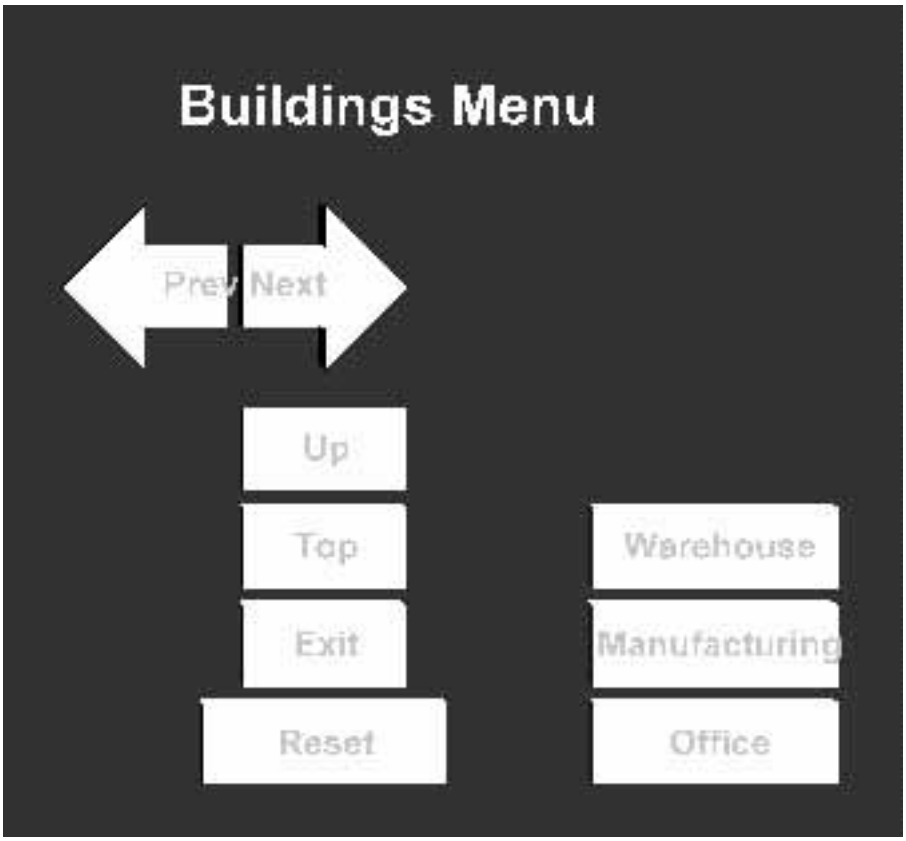
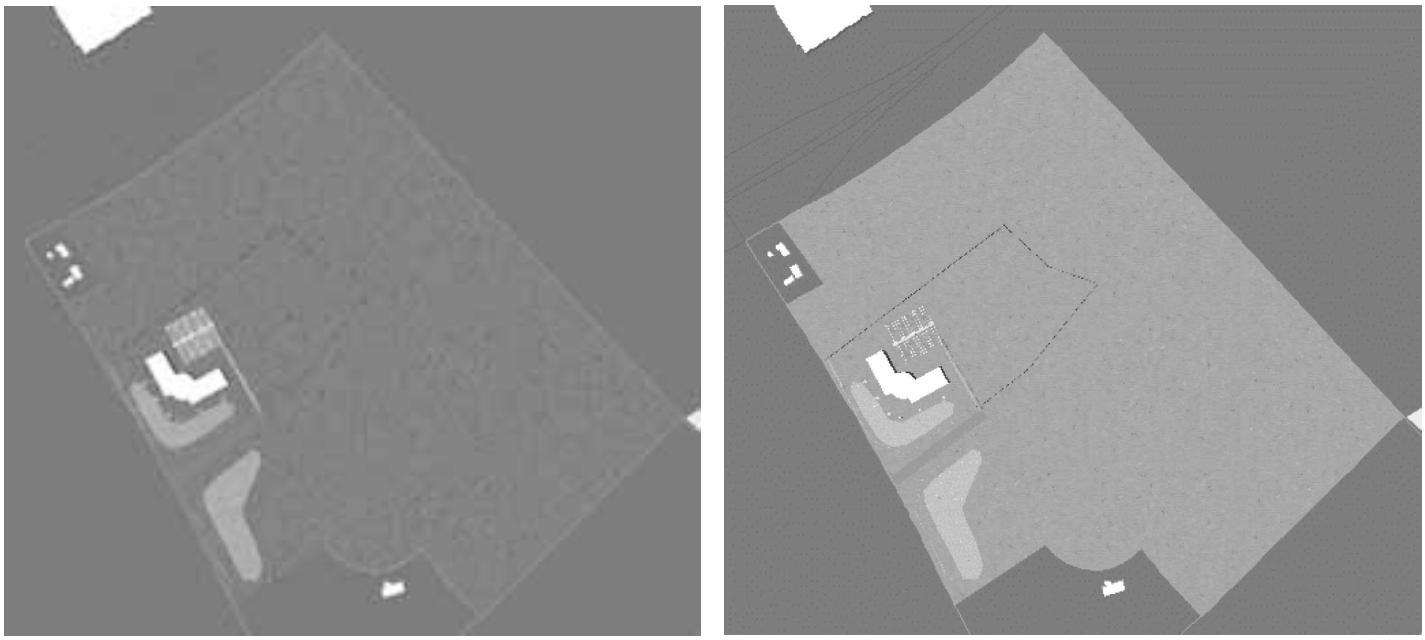


Figure 5. An Example Menu



(a) Normal

(b) Prompting Input

Figure 6. Work Zone

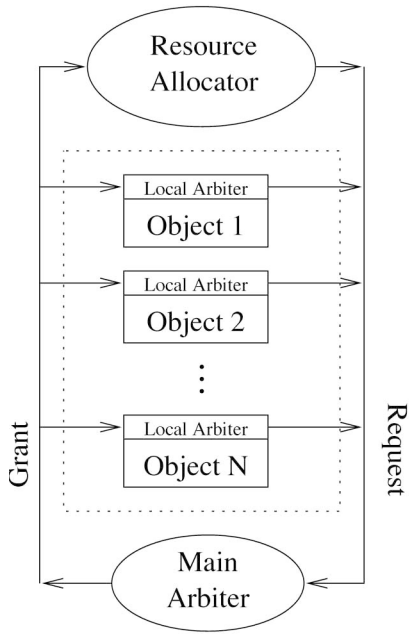


Figure 7. Arbitration

cated collision response scenarios such as keeping a car on a road, restricting an object within a set boundary, or enforcing setbacks.[1]

3.5. Simulation Manager

A simple simulation manager is included that broadcasts simulation data to objects. As a demonstration, the simulation manager has been programmed to generate an oscillating wind and the pine tree model responds by swaying in the wind. The simulation manager sends periodic updates to all objects, so each can respond in its own particular fashion. More advanced capabilities envisioned include modeling the mutual interaction among different objects in the application. For example, the plug and socket interaction model described in [30] is one mechanism that can be used to enable objects to exchange information.

3.6 The Fixed Landscape

The fixed landscape is the solid model for the work zone. This includes information converted from city GIS data, the outlying area, example landscaping, roadways, buildings and any other models that are anticipated to be permanent and unmovable. In addition, contents restored from prior sessions are inserted into the fixed landscape.

4. Object Architecture

The object architecture is designed to serve several purposes. First and foremost, the object architecture is designed to support independence and autonomy of object behaviors in the visualization, placing few restrictions on object behavior. Second, the objects manage their own user interactions and interface to the visualization. As a result, the visualization architecture is simplified since it is not required to directly handle user-object interactions. Third, the object defines a uniform interface between the object and the architecture.

In order to achieve these purposes, the object architecture is structured into layers that isolate the functions that interface the object to the visualization, from those that change the object geometry, and also from those that define the object behavior. This organization permits modular design of the architecture and provides a framework for adding new capabilities in a regular fashion. Paralleling this structure, the object architecture is organized into three layers—the interaction layer, the edit layer, and the model layer, as shown in Figure 8. The interaction layer manages the interface to the visualization and any user interactions that affect the object as a unit, such as object drag and rotation. The edit layer defines the nature and appearance of

[1] VRML has a limited collision detection capability, but only between the avatar associated with the active viewpoint and objects that are encountered during certain navigation modes.

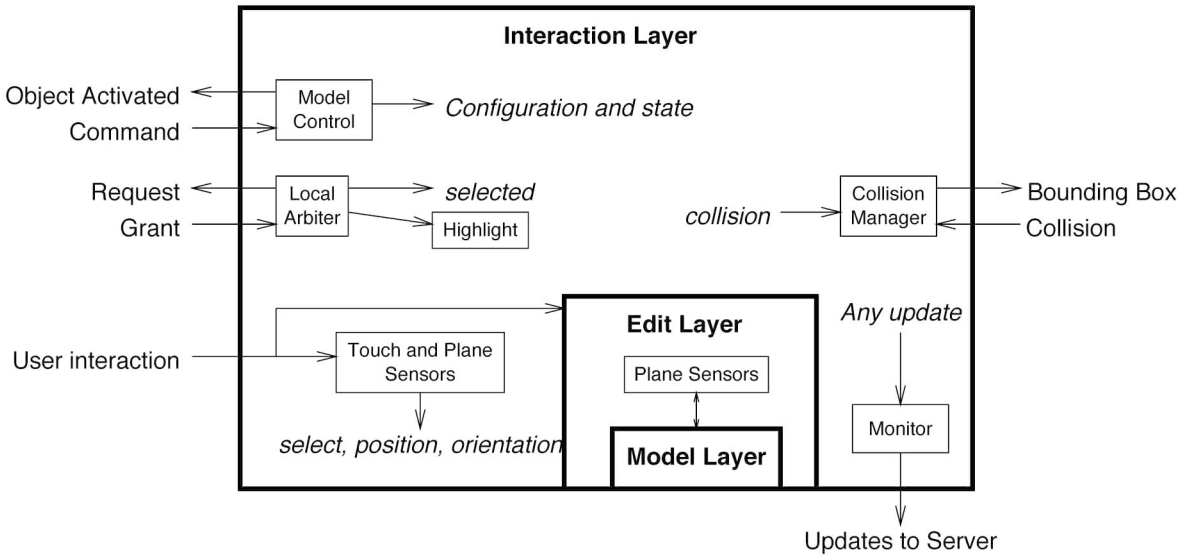


Figure 8. Object Architecture

controls and provides an interface between the model and the interaction layer. Finally, in the model layer, the appearance of the object is defined as is any ability for the object to change appearance or shape. The layer structure provides additional benefits including the support of reuse of layer methods and simpler definition of new objects. Indeed, the interaction layer is sufficiently well defined that it is automatically generated for most objects implemented. The objects could have been defined as flat, i.e., without breaking the architecture into layers. In such an architecture, defining a uniform interface would be more difficult because the details of the interface, editing controls, and model behavior may become tangled. Furthermore, the flattened architecture can make it to difficult expand the variety of new objects.

4.1. The Interaction Layer

The interaction layer encapsulates methods for managing objects, the visualization interface, and the server interface. The interaction layer is independent of the behavior and appearance of the object providing methods that support initialization, object selection, drag and drop, arbitration, object identity, collision response, and the web server interface. In addition, the interaction layer serves as an information conduit from the visualization to the edit and model layers for initialization and simulation events. Figure 8 gives a schematic representation of the interaction layer. The layer functionality and the interface to the visualization are discussed in more detail.

4.1.1. Functionality

The interaction layer functionality consists of a collection of methods and processes that the user can use to control the object through mouse input, serve as the interface to the visualization, and isolate the edit and model layers from the visualization infrastructure. The functions are partitioned into initialization, user requests, object identity, object highlight, object specific views, the monitor, static instantiations, and collision response.

Initialization. Objects are created dynamically and initialization occurs in concert with the resource manager that allocates the object using the createVrmlFromURL method. The complexity arises from the circumstance that no seminal event signals when the object is fully loaded and all data structures are stable. Before the object is loaded, any events sent to the object may not be received.[2] Since the object includes at least three nested levels of EXTERNPROTO instantiations (for each of the object layers), the object signals initialization complete only after all nested nodes report completion of initialization. In addition, using TimeSensor nodes, a delay, whose length is commensurate with the model complexity, is inserted to provide a margin against other nondeterministic effects.

User Interactions. Drag and drop functionality is provided by the coordination of TouchSensor and PlanSensor functionality associated with each object. Upon detecting a mouse click on the object, a request is transmitted to the main arbiter for recognition and to request control of the necessary resources. At times when it is undesirable to activate an object, such as during the initialization sequence of another object, the ability to recognize a mouse click is disabled. When the main arbiter recognizes an object, its interaction layer sends an objectType event to the menu that responds by displaying the appropriate menu. A PlaneSensor tracks the mouse position and is used to drag and drop the object to another position. The PlaneSensor alone can provide the drag and drop behavior, but coordinating PlaneSensor with a TouchSensor provides a way to select an object for drag and drop. In object rotation, the orientation change is computed using the vector formed by the mouse position in the drag plane relative to the initial mouse down point.

Object Identity. Objects are uniquely identified by a serial number assigned when the object is created. When an object is selected, the serial number is sent to the visualization's main arbiter. Furthermore, an object can be activated when its serial number is broadcast on the grant event. The object with the matching serial number is activated while the rest are deactivated. In addition, the serial number appears in all communications with the server.

Object Highlight. Highlighting an object provides a visual cue to the user of the selected object. Two approaches are used to highlight an object. The first method is to surround the object with a wire frame demarking the bounding volume. For example, wire frames are used to highlight a selected tree or street light. The second method requires the object to provide a suitable visual cue. For a roadway object, the cue is the appearance of editing controls.

Object Specific Views. Objects have particular views that are customarily desired to evaluate placement and configuration. The object specific views, including animated view points, are associated with the interaction layer and are added to the browser viewpoint list when the object is selected.

Monitor. The monitor is the object's interface to the web server and consists of loadURL browser method calls to a servlet or common gateway interface (CGI) script that performs the processing necessary to log session information on the server. The interaction layer has direct access to the object serial number and any changes of position and orientation.

Model updates are passed from the edit layer and are then communicated to the server in the interaction layer. The information passed to the server includes the serial number, the kind of manipulation, and specific parameters of the manipulation. For example, an object translation might be logged as

ser. no.	:	type	:	mode	:	x	:	y
12	:	42	:	(translation)	:	1000	:	200

[2] Indeed, CosmoPlayer can malfunction when an event is sent prematurely.

where the different fields are separated by colons. Extra white space has been added here to enhance readability. The translation mode is an example and other modes have different forms.

Static Instantiation. Objects in the static landscape are statically instantiated. In addition, restoring a previous session requires that the previously defined objects be instantiated into the scene at start-up. Static instantiation is achieved by defining fields designed to capture all aspects of the object appearance and location to make the object appear exactly as in a prior session. In the roadway initialization method, the roadway is defined by an array of points defining the course of the roadway, otherwise the roadway object waits until the appropriate number of points have been input.

Collision Response. Collision detection and response is an expected capability when working with solid models that can be moved or modified. With each translation, rotation, or update in geometry, the object sends a bounding volume event to the collision manager. The collision manager determines whether the current geometry overlaps another and if so, sends relevant events to the colliding objects. For the present application, collision detection simply freezes the object's position and adjusts the position in the event that processing latencies result in the collision being detected too late. The freeze is implemented by filtering the mouse position tracked by the plane sensor and upon collision detection, blocking any subsequent position updates. More elaborate collision detection responses can be devised with the response methods implemented in edit and model layers.

4.1.2. Object Layer Interface

The interface between the visualization and the object layer is complex and provides a broad range of capabilities. The attributes of major partitions of the interface are summarized in Table 3. Initialization attributes must support both static and dynamic instantiation. For example, in static instantiation, if an initial position is given, the object is placed and made visible. For objects added during a session, the remaining fields control the sequencing necessary to initialize the object and to integrate it into the visualization. The user interacts with objects in a fashion consistent with the function and form of the object. The object level interactions supported in the object layer interface are selection through mouse clicks, and drag and drop. The arbitration process begins with a mouse click on an unselected object. As a result of the mouse click, a request is sent to the main arbiter with an orderly transfer of control synchronized by the clock associated with the main arbiter. The local arbiter monitors the progress of the arbitration process and at the appropriate time, fully activates the selected object. Object control includes receipt of commands from the menu and covers a wide range of capabilities. The set_command input receives commands generated by the resource manager or the menu and the associated method provides the required response. The interface to the web server is implemented using the loadURL browser method. The URL is in reality a CGI-script or servlet that logs the parameters supplied with the URL. After any change in appearance or position, the object sends an event to the collision manager updating the bounding volume. In response, the collision manager determines if a collision has occurred and sends the collision overshoot. The pipeline interface enables information to be transmitted globally to all objects and for an object to pass information back to the visualization. Information is synchronized using a latching signal event to ensure that all parameters are acted upon only after all information has been supplied. The pipeline can be defined so that any desired information can be communicated.

4.2. Edit Layer

The edit layer requirements are uniform across all objects, although this does not necessarily need to be the case. As noted earlier, this facilitates integration of new models into the application. The edit layer instantiates the model layer and contains

Table 3. Object Interface Attributes

Interface	Purpose
Initialization	initialization sequencing for both static and dynamic instantiation
Control	control of object through command events and communication of status information
Interaction	object interactions between the user and the object, such as selection and drag and drop
Arbitration	menu arbitration among all objects
Server interface	enable server communication, mechanism for passing object update information to server
Collision	outputs bounding box when geometry changes, defines object specific collision response
Pipeline	enables information to be passed between the application and through all object layers and provides a mechanism for defining simulation parameters globally

drag controls for controlling the appearance of the model. Changes to the appearance of the objects are reflected in the positions of the drag controls in the edit layer. The changes are communicated both to the child model layer and the parent interaction layer. The edit layer interface is shown in Figure 9. The fields are divided into four groups, representing initialization fields, control fields, geometry change fields, and simulation fields. The initialization fields communicate with the interaction layer when the object is created. The control fields signal the edit layer when an object is selected, should report debugging information, or should reset its state. In the event the object changes its geometry, the specific changes to the geometry are communicated in sufficient detail to recreate the object and also to update the bounding volume. Finally, the simulation parameters can be monitored by the edit layer and can be passed to the model layer.

4.3 Model Layer

The model layer is the most deeply nested layer. In the application presented in this paper, the model layer represents the pure behavior of the model. Requested changes to the geometry are generated by the edit layer and then passed to the model layer. After the geometry of an object has been modified, a new bounding box is calculated. In addition, as a result of internally generated dynamics or the effects of other external processes, the model layer can change its appearance or state.

5. Example Objects and Capabilities

In this section, several objects having a variety of capabilities are described. Generically, the capabilities can be related to the composition and morphology of the object. For example, the path for roadway objects is represented by a sequence of linear segments termed a *spine* whose path is modified by changing the underlying spine. Buildings are composite objects whose final form depends on the architectural definition and desired area. The capabilities are related to the appearance and functions of the objects. In each case, the object is an independent and autonomous entity, managing object relevant user interaction capabilities, modifications, and collision response.

5.1. Trees and Lights: Simple Geometries

The appearance of simple geometries such as trees and external lighting are defined by solid models. The simplicity of the geometries does not reduce the overhead associated with managing and updating these objects. The only aspect of these models that varies is their location. Furthermore, each of these objects achieves full collision detection capabilities between each other and the building solid model.

5.2. Buildings and Parking Lots

The building model is a composite geometry consisting of a one or two story structure integrated with parking lots. Generality in building appearance is achieved through two mechanisms, the first of which is defined in this subsection, and the second is

```
PROTO objectModel [
      # Initialization fields
      eventIn  SFVec3f set_position
      field    MFVec3f     points []
      eventOut SFBool      new_changed
      eventOut SFBool      created_changed
      eventOut SFInt32     workzonePoints_changed
      eventIn  SFBool  set_done  # edit done
      # control fields
      eventIn  SFBool  set_reset
      eventIn  SFBool      isActive
      eventIn  SFBool  set_debug
      field    SFBool      debug FALSE
      # events generated with change in object geometry
      eventOut MFVec3f     points_changed   #specific changes
      eventOut MFVec3f     corners_changed  #bounding volume
      # simulation parameters passed directly to model
      eventIn SFBool   set_latch   # synchronizes simulation parameters
      eventIn  MFInt32 set_intParams
      field    MFInt32     intParams []
      eventIn  MFFloat set_fltParams
      field    MFFloat     fltParams []
      eventIn  MFVec2f set__2dParams
      field    MFVec2f     _2dParams []
      eventIn  MFVec3f set__3dParams
      field    MFVec3f     _3dParams []
]
```

Figure 9. Edit Layer Interface

described in the following subsection. The building solid model is defined using the createVrmlFromURL browser model, allowing any solid model to be inserted. In this manner, any building size can be integrated with the same parking lot infrastructure. The parking lot size is determined from city building codes, or if appropriate, through direct entry by the user. The parking lot area is divided into two pieces that are sized and placed at the sides of the building. The parking lot models dynamically create the appropriate number of trees to satisfy canopy cover building codes. Furthermore, the parking lot aspect ratio can be adjusted and all trees can be individually placed.

5.3. Buildings: One Model, General Appearance

In the visualization, one or two story buildings may appear with one of eight entrance options, selected through the menu. The building solid model is defined parametrically by a footprint and a height. To simplify the application, the building footprint and height are hard coded, but do not need to be in principle. The solid model for each story of the building is the three dimensional shape formed by sweeping the footprint through the height of the building. As a result, only the area and number of stories for the building are selected by the user. The building model scales the inherent foot print to give the appropriate area. If a second story is requested, it is simply a second instantiation of the first floor, translated vertically. In order to render a typical pattern of windows on each floor, two textures have been defined and are shown in Figure 10. The window tile describes a fixed width texture for a single window. The number of windows per side of building is calculated and then is used to define the appropriate texture tiling. Since the building dimensions do not typically require an integral number of window tiles, a variable width corner tile is defined to fill out the building side. Building sides are represented using three IndexedFaceSet geometries. The three geometries specify three faces composed of a center face where an integral number of window tiles are placed, and two edges that use the corner tile. The tiles are translated and scaled to fit the building side and to give a natural appearance. The following equation defines the number of window tiles that must be placed on a side of the building

$$P = \left\lfloor \frac{L}{S} \right\rfloor, \tag{1}$$

where P is the number of window tiles, L is the length of the side of the building, and S is the natural width of the window tile. Next the edges of the corner tiles must be filled in. The length of each corner tile is

$$F = \left\lfloor \frac{L \pm SP}{2} \right\rfloor \tag{2}$$

where F is the size of each corner tile. These parameters are used to calculate the vertices of an indexed face set for the building for all four sides. The faces for one side are illustrated in Figure 11. Finally, the origin of the texture must be translated to give symmetric placement of the windows. The translation of the coordinate system for the window tilings is

$$T = -(L - SP), \tag{3}$$

where T is the desired translation. The translation is negative because the VRML TextureTransform node translates the origin rather than the texture itself.

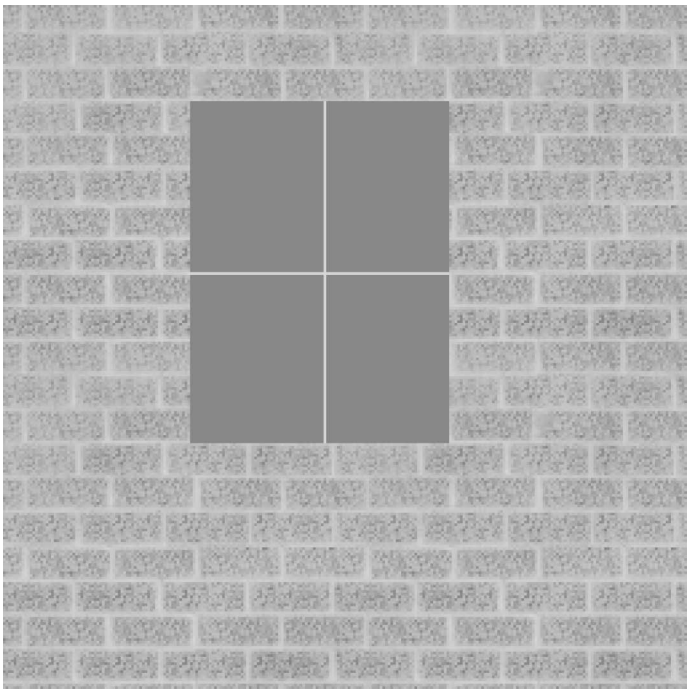


Figure 10a. Window Tile

Figure 10b. Corner Tile

Figure 10. Building Textures

In addition to defining one and two story buildings, a building entrance can also be defined. The visualization allows for three building types, requiring only two structure geometries. An office structure is defined as a rectangular box. Finally, manufacturing and warehouse structures have a loading dock area at the rear of the building.

5.4. Roadways and Landscaping Borders

The Extrusion node is used to define the appearance of roadways and landscaping borders. The appearances of each differ in the extrusion cross section and surface texture. The path of the extrusion is defined by a spine that defines the course of the object. Editing controls rather than a wire frame appear for selected roadway or landscaping border objects. The controls are located in a manner consistent with the geometry type. For example, the spines for a straight road, berm, and hedge are defined by the two end points. The spine of the ring road geometry is defined by the center point and one point on the perimeter. Internally, sufficient points are sampled along the circle circumference to give the appearance of a smooth curve. The spine of a curved road is the arc from the circle intersecting the three points input by the user. The appearance of each of these objects is calculated and modified as the controls associated with each are dragged.

6. Example Session

An example session is presented here, beginning with the opening scene of the visualization, and finishing with several views of the scene. The opening of the visualization shown in Figure 12 is a view from about 2,000 feet (about 600 meters) above the work zone. In addition, the menu appears in the right part of the opening view. A site can be successively built up beginning with the definition of the site boundary, addition of the building, placement of the roadways, arrangement of the landscaping, and placement of lighting. Once the site has been created, top and front views of the site are shown in Figure 13. Note the menu has been moved to the upper left hand corner of the view port for screen captures in this figure. Such a scene can be created quickly, in about a half of an hour. A client based version of the application can be found at *http://www.lions.odu.edu/~lbelfore/urbanVisualization* and currently only operates under CosmoPlayer.

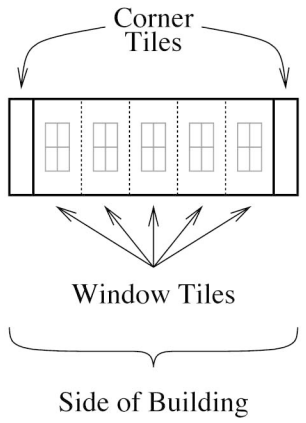


Figure 11. Panel Placement. Calculated Parameters Indicated

7. Discussion

The architecture that is presented in this paper provides many functional capabilities. In this section, we discuss some advantages and disadvantages of the proposed architecture from different perspectives. First, the efforts necessary to develop virtual worlds are discussed. Second, the technological aspects of employing VRML technology are reviewed. Third, the specific issues relevant to the proposed architecture are discussed.

No formal study was conducted to measure the effort required to create applications for different sites, but some anecdotal information is provided. The general goal during development was to construct an architecture that could be easily adapted to different situations and sites. In support of this, the different modules described in Section 3 have been constructed so that custom information is defined either in parameters during instantiation (work zone), in tables associated with each (resource manager), or without change at all (arbiters). Construction of new menu hierarchies requires some additional work, but each object menu is defined parametrically using a generic menu model. The structure and protocols for communications between different layers in the menu are defined so that automating the construction of the menu hierarchy should be straightforward. Depending on the detail required for background models, a more significant effort may be necessary in their creation. Furthermore, different applications may require a different collection of objects. For simple objects with features similar to the objects described in object-examples, new objects can be quickly developed and integrated. For example, using the model for roadways, complementary railway models were created as well as the accompanying menus. The time needed to accomplish this addition required about four hours of development time including menu modifications. On a larger scale, a demonstration application was created to visualize a hazardous waste cleanup site requiring a total effort of about 200 hours. It should be noted that the majority of this effort was devoted to adding capabilities beyond what are described here and that approximately 40 hours was devoted to issues directly related to adapting the visualization architecture.

Strictly from a technological perspective, VRML offers three important benefits. First, in addition to the capability of representing virtual worlds, VRML is an internationally standardized language. Tools that generate VRML content in conformance with the standard are expected to produce content that can be reviewed on browsers that are also compliant. Second, in principle, developers are not restricted to one vendor's software for either development or rendering. Third, VRML is designed to be delivered over the Internet. Anyone with a browser anywhere on the Internet need only download and install a plug-in to browse VRML content. On the negative side, complex virtual worlds can require significant download times. In addition, because the configuration of the client machines can vary, the realism of the experience can be directly affected.

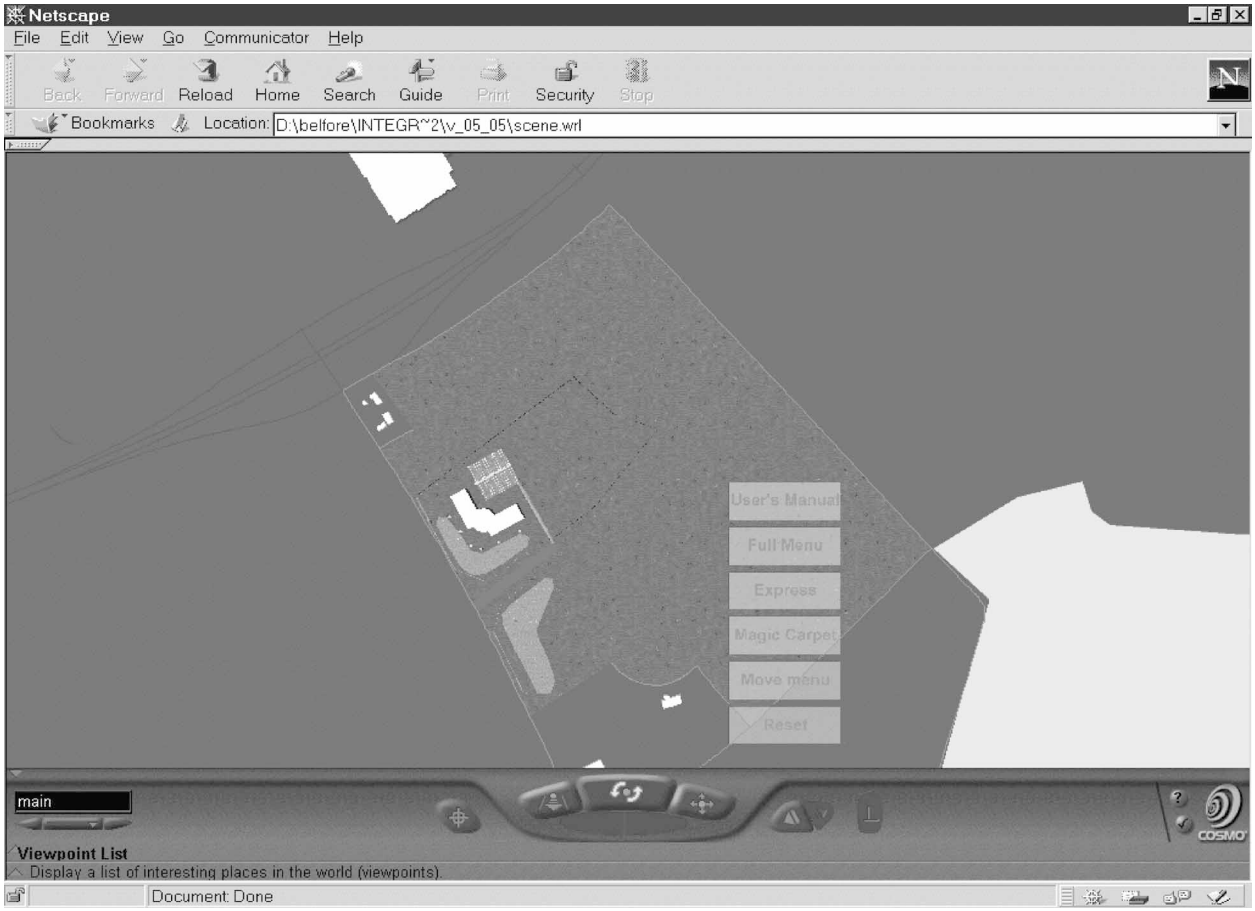
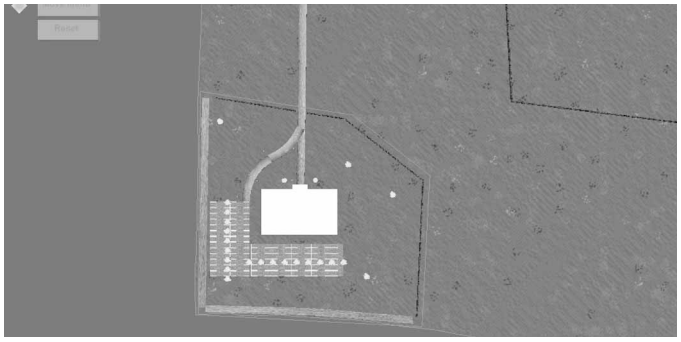


Figure 12. Opening View



(a) Top View

(b) Front View

Figure 13. View Scene Created

The proposed architecture offers solutions that compensate for some of the technological challenges. First, objects can be allocated and added dynamically at the request of the user. This important capability reduces download times because at startup only the initial configuration of the world is required. In addition, the dependence on the client machine configuration is reduced since start-up resources are minimized and additional resources are necessary. Indeed, the world does not have to be reloaded in response to additions or modifications to the world. Second, the architecture is designed to be modular, enabling replacement of any user visible components without modification of the underlying architecture. For example, the complement of objects can be changed and new work zone configurations can be specified. Among the disadvantages in developing the architecture include the added complexity necessary to allocate and insert content dynamically. An additional disadvantage is that the CosmoPlayer [16] plug-in, used to render our application, is neither currently in development nor is new development expected in the future. Similar to ours, some application areas still depend on CosmoPlayer 31, but clearly this dependence cannot continue indefinitely. We have begun exploring porting this work to other VRML plug-ins.

8. Summary and Future Work

In this paper, we have presented an architecture for creating large virtual worlds. The architecture features dynamic reconfigurable extent where nodes are allocated and inserted into the world at run time. The application architecture includes modules comprising the infrastructure of the application that manages the application and interactions with the user. Included in the infrastructure are resource, collision, and simulation managers that suggest a wide range of behaviors are possible. Furthermore, the object architectures have been designed to support modularity and object independence. The object architectures are organized into three layers that separate the object interface to the application, from editing controls, and also from the behavior. In addition, we have demonstrated a significant application that applies the architecture described in this paper.

We intend to pursue two avenues of future work. First, we plan to study the types of simulations that can be performed on the architecture we have presented. In particular, we would like to experiment more with model dynamics and the collective interaction of objects within a visualization. Second, we would like to provide stronger links between the application and a web server. The web server can provide many valuable capabilities including access to data bases and generation of models dynamically.

9. Acknowledgments

We would like to acknowledge the City of Portsmouth, Virginia for funding this work. In addition, we would like to acknowledge the Virginia Modeling, Analysis and Simulation Center (VMASC) for providing facilities and other support necessary to complete this project.

10. References

[1] The Web3D Consortium, Incorporated, "The Web3D consortium," http://www.web3d.org.

[2] L. H. Coglianese, "Towards a visual integration architecture baseline," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 58–63, January 1998.

[3] M. Salmela and T. Tuikka, "Smart virtual prototypes for web-based development environments," in Proceedings of the 1999 International Conference on Web-Based Modeling & Simulation, (San Francisco, CA), pp 127–133, January 1999.

[4] R. Pichumani, D. Walker, L. Heinrichs, C. Karadi, W. A. Lorie, and P. Dev, "The design of frog island: A VRML world for biology," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 31–36, January 1998.

[5] P. Dev, D. Engberg, R. Mather, D. Hodge, and M. Dutta, "The collaborative curriculum web for medicine: A virtual representation of medical school resources," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 29–30, January 1998.

[6] D. P. McKay, P. Matuszek, D. Mateszek, T. Smith, and S. Testani, "An architecture for a training virtual world environment," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 125–130, January 1998.

[7] S. Testani, J. Tumasz, and S. Gnanamgari, "CIMBLE: Experiential training in a VRML environment," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp. 67–71, January 1998.

[8] S. Testani, E. Wagner, and K. Wehden, "CIMBLE: The CADETT interactive multi-user business learning environment," in Proceedings of the 1999 Virtual Worlds and Simulation Conference (VWSIM'99), (San Francisco, CA), January 1999.

[9] N. H. El-Khalili and K. W. Brodlie, "Architectural design issues for web-based virtual reality training systems," in Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation, (San Diego, CA), pp 153–158, January 1998.

[10] B. Damer, K. Marcelo, and F. Revi, "Nerve garden: A virtual terrarium in cyberspace," in Proceedings of the 1999 Virtual Worlds and Simulation Conference (VWSIM'99), (San Francisco, CA), pp 131–135, January 1999.

[11] J. R. Ensor and G. U. Carraro, "Peloton: A distributed simulation for the world wide web," in Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation, (San Diego, CA), pp 159–164, January 1998.

[12] N. Jacobstein, W. Murray, M. Sams, and E. Sincoff, "A multiagent associate system guide for a virtual collaboration center," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 215–220, January 1998.

[13] W. M. Darling and M. J. Welch, "Embodied agent for a virtual library," in Proceedings of the 1998 Virtual Worlds and Simulation Conference (VWSIM'98), (San Diego, CA), pp 225–230, January 1998.

[14] M. Reddy, Y. Leclerc, L. Iverson, and N. Bletter, "TerraVision II: Visualizing massive terrain databases in VRML," IEEE Computer Graphics & Applications, vol. 19, pp 30–38, March/April 1999.

[15] S. N. Matsuba and B. Roehl, ""bottom, thou art translated": The making of a VRML dream," IEEE Computer Graphics & Applications, vol. 19, pp 45–41, March/April 1999.

[16] CosmoSoftware, Incorporated, "CosmoSoftware home page," http://cosmosoftware.com, 1998.

[17] L. S. Indrusiak and R. A. da Luz Reis, "3D integrated circuit layout visualization using VRML," in Proceedings of the 1999 International Conference on Web-Based Modeling & Simulation, (San Francisco, CA), pp 177–181, January 1999.

[18] J. J. Wynn, J. D. Roberts, and M. O. Kinkead, "Visualizing the wargame—web-based applications for viewing a course of action (COA)," in Proceedings of the 1999 International Conference on Web-Based Modeling & Simulation, (San Francisco, CA), pp 227–232, January 1999.

[19] A. G. Bruzzone and G. Berrino, "Modelling of urban services by VRML & Java," in Proceedings of the 1999 International Conference on Web-Based Modeling & Simulation, (San Francisco, CA), pp 34–38, January 1999.

[20] P. Fishwick, "A hybrid visualization environment for models and objects," in 1999 Winter Simulation Conference Proceedings, (Phoenix, Arizona), pp 1417–1424, December 1999.

[21] L. A. Belfore II and R. Vennam, "VRML for urban visualization," in 1999 Winter Simulation Conference Proceedings, (Phoenix, Arizona), pp 1454–1459, December 1999.

[22] M. Reddy, L. Iverson, and Y. G. Leclerc, "Under the hood of GeoVRML 1.0," in Proceedings of the Fifth Symposium on the Virtual Reality Modeling Language VRML2000, (Monterey, CA), pp 23–38, February 2000.

[23] H. Holten-Lund, M. Hvidtfeldt, J. Madsen, and S. Pedersen, "VRML visualization in a surgery planning and diagnostics application," in Proceedings of the Fifth Symposium on the Virtual Reality Modeling Language VRML2000, (Monterey, CA), pp 111–118, February 2000.

[24] The Web3D Consortium, Incorporated, "External authoring interface working group," http://www.web3d.org/WorkingGroups/vrml-eai, 1999.

[25] A. G. Bruzzone, E. Manetti, and P. Perletto, "Object-oriented documentation from simulation for civil protection & industrial plant accidents shared directly on the web," in Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation, (San Diego, CA), pp 168–173, January 1998.

[26] A. G. Bruzzone and R. Signorile, "Crowd control simulation in Java based environment," in Proceedings of the 1999 International Conference on Web-Based Modeling & Simulation, (San Francisco, CA), pp 23–28, January 1999.

Lee A. Belfore II received the BS degree in electrical engineering from Virginia Tech in 1982, the MSE degree in electrical engineering and computer science from Princeton University in 1983, and the PhD degree in electrical engineering from the University of Virginia in 1990. From 1982 to 1985, he was a member of technical staff at AT&T Information Systems. From 1987 to 1988, he was a research scientist in the Department of Electrical Engineering, Center for Semicustom Integrated Systems at the University of Virginia. From 1990 to 1997, he was with the Department of Electrical and Computer Engineering at Marquette University, Milwaukee, Wisconsin. Since 1997, he has been with the Department of Electrical and Computer Engineering at Old Dominion University, Norfolk, Virginia. His research interests include neural networks, data compression, and Internet-based virtual reality applications.

[27] L. A. Belfore II and S. Chitithoti, "An interactive land use VRML application (ILUVA) with servlet assist," in 2000 Winter Simulation Conference Proceedings, (Orlando, Florida), pp 1823–1830, December 2000.

[28] D. R. Nadeau, "Tutorial: Building virtual worlds with VRML," IEEE Computer Graphics & Applications, vol. 19, pp 18–29, March/April 1999.

[29] The Web3D Consortium, Incorporated, "The virtual reality modeling language," http://www.web3d.org/Specifications/VRML97/index.html, 1998.

[30] Y. Araki, "An interactive model for *avatar-habilis,* avatars with capabilities to use tools in 3D-MUVES," in Proceedings of the 1999 Virtual Worlds and Simulation Conference (VWSIM'99), (San Francisco, CA), pp 3–8, January 1999.

[31] The Web3D Consortium, Incorporated, "The distributed interactive simulation DIS-Java-VRML working group," http://www.web3d.org/WorkingGroups/vrtp/dis-java vrml, 2000.