An Architecture for Collaborative Virtual Environments With Enhanced Awareness

by

Dennis Brown

A thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1998

Approved by

Professor Prasun Dewan, Advisor

Professor David Stotts, Reader

Professor Greg Welch, Reader

©1998
Dennis Brown
ALL RIGHTS RESERVED

ABSTRACT

DENNIS BROWN: An Architecture for Collaborative Virtual Environments With Enhanced Awareness
(Under the direction of Dr. Prasun Dewan.)

An architecture for distributed and collaborative virtual environments with support for enhanced awareness is presented. The virtual environment is made up of objects that exist and interact with other objects in the environment. “Distributed” means that the objects can exist on different machines. “Collaborative” means that many user-controlled objects can interact within the same environment. “Enhanced awareness” is a set of services that allow a user to control how objects within the environment appear to him, and vice versa. This architecture draws some characteristics from four existing works, which are described. This architecture was implemented and a sample application for training a user to navigate a maze was developed with it. Usage of the sample application showed that enhanced awareness services were effective in helping the user learn how to navigate the maze, to find targets within the maze, and to see other users as they navigated the maze.

ACKNOWLEDGEMENTS

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported under DARPA grant N 66001-96-C-8507.

CONTENTS

Page

LIST OF FIGURES vi

Chapter

I. INTRODUCTION 1

II. VIRTUAL ENVIRONMENTS 3

A. Virtual Reality vs. Virtual Environments 3

B. Components of a Virtual Environment 3

C. User Interaction 5

D. Distributed and Collaborative Virtual Environments 6

III. ENHANCED AWARENESS 7

IV. RELATED WORK 9

A. Model/View/Controller Architecture 9

B. MASSIVE 10

C. NPSNET 11

D. SOLVEN 13

V. COLLABORATIVE VIRTUAL ENVIRONMENT ARCHITECTURE WITH ENHANCED AWARENESS 16

A. Scalability 17

B. Personalization 20

VI. IMPLEMENTATION 22

A. Language and Platform 22

B. Implementation of the Connection Manager 23

C. Implementation of Virtual Objects 24

VII. SAMPLE APPLICATION 29

A. Application Description 29

B. Results from Running the Application 30

VIII. CONCLUSIONS AND FUTURE WORK 37

WORKS CITED 38

LIST OF FIGURES

Figure 2-1 Sample view of a virtual environment 5

Figure 4-1 Model/View/Controller example 10

Figure 4-2 Focus-nimbus interaction example 11

Figure 4-3 View-matrix example 15

Figure 5-1 Connection Manager example 18

Figure 5-2 Object connection example 20

Figure 6-1 User interface for a UserObject 27

Figure 7-1 User has turned off bad path 32

Figure 7-2 User has turned off maze walls 33

Figure 7-3 User views maze from above 34

Figure 7-4 User sees four identical flags 35

Figure 7-5 User now only sees one flag 36

I. INTRODUCTION

This paper will discuss what constitutes a *virtual environment*, how a virtual environment can be extended for many users, how to distribute the processing required to run the virtual environment on many processors. Next, it defines awareness in a virtual environment and how to enhance awareness by tailoring each person's view of the same virtual environment. In this discussion, several existing systems and architectures will be listed and their relevant features detailed. Then, a new architecture for building general-purpose virtual environments designed by the author will be explained both at a high level and at the implementation level. Finally, a sample application built using the author's system will be described.

The reader is assumed to understand the basic concepts of virtual reality, virtual environments, collaborative systems, and distributed systems. Chapter II gives background information on these four topics as they pertain to this research. Even if the reader is familiar with these topics, he should familiarize himself with the following definitions of terms used throughout the remainder of this paper before skipping to chapter III:

- *object:* an entity with a three-dimensional appearance and a set of behaviors, either autonomous or user-controlled (such as an avatar)
- *virtual environment (VE):* a program using three-dimensional graphics to represent a real or synthetic scenario in which the user, represented by an avatar object, may move around in the environment to see and interact with other objects
- *collaborative virtual environment (CVE):* a virtual environment in which more than one user operates concurrently and which facilitates interaction between those users
- *distributed virtual environment (DVE):* a virtual environment in which the processing required to operate the environment is spread across many processors

- *simulation:* a session in a virtual environment that was built for simulating a realistic situation for training.

In addition, readers may with to skip appropriate parts of chapter IV if they are already familiar with the Model/View/Controller architecture or the MASSIVE, NPSNET, and SOLVEN projects. Chapter V describes the author's architecture for collaborative virtual environments and forms the meat of the paper. Chapter VI describes how the system was implemented, chapter VII details an application built with the system, and chapter VIII sums up the work.

II. VIRTUAL ENVIRONMENTS

A. Virtual Reality vs. Virtual Environments

In this paper, *virtual reality* is considered to be a field of study that has produced innovations mainly in display technology (high speed three dimensional graphics, head mounted displays, spatialized sound, and image capture, for example) and physical human interaction (movement trackers, touch-based input devices, and force feedback devices). A *virtual environment*, on the other hand, is a system of hardware and software based on virtual reality technologies built to give the user the impression he is immersed in a certain situation and the ability to act upon that situation. While even text-based systems such as multi-user dungeons or interactive fiction systems can be considered virtual environments, this paper will concentrate only on those using three dimensional graphical displays and real-time interaction with the environment to simulate realistic situations, for example, those used for training purposes.

B. Components of a Virtual Environment

The ultimate virtual environment would model situations down to the resolution of subatomic particles. Using the laws of physics and raw computational power, any scenario could be modeled exactly as it would happen in the real world, in real-time. Unfortunately, this "ultimate VE" is not possible with current technologies, and so reasonable approximations of realistic scenarios can be built from objects, terrain, and scripts.

An object is an entity in the virtual environment that can act upon the environment and other objects. The object has some physical representation, usually a set of three-dimensional polygons for its appearance and a set of attributes for features such as mass, material, etc. In

addition, the object has a behavior, either because it is controlled by a user of the virtual environment application, or because it follows some algorithm. The object acts on the environment through collisions with other objects and terrain and through predetermined methods of communications with other objects. For example, in a virtual environment for training a tank operator, a tank object may affect a tree object by colliding with the tree, knocking it over. Or, a human avatar may set up a voice channel to communicate with another human avatar within the tank to decide together what to do.

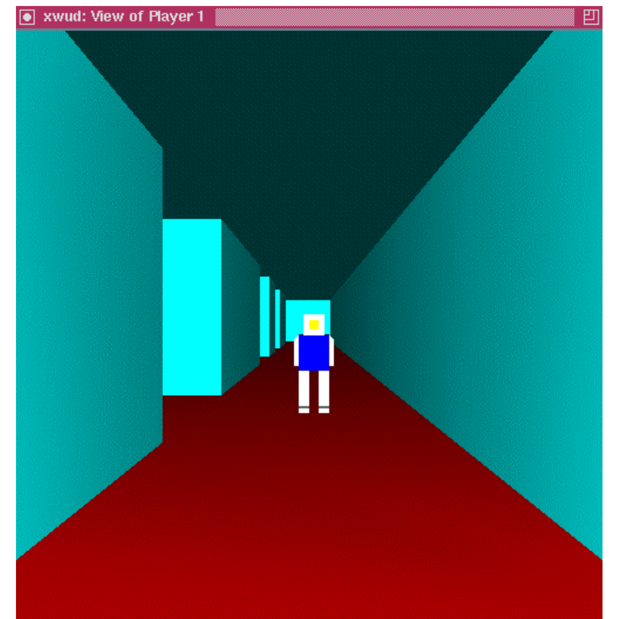
Terrain defines boundaries in the environment but does nothing more than stop movement of the objects in certain directions. Consider the tank training virtual environment again. The terrain in the environment consists of the flat ground, hills, ponds, buildings, and so forth. These elements do not directly act upon the environment except as providing boundaries within it and are generally immutable. If terrain elements need to do more than be stationary and immutable, they could be modeled as objects instead of terrain in the case of monolithic structures. Research on mutable terrain is currently underway [Lisle 1994]. The importance of differentiating terrain from objects is in efficiency. If the terrain is immutable, it requires no processing power (since it's not doing anything) and in the case of distributed collaborative virtual environments, it can be replicated in each process without needing to keep it synchronized across the processes [Macedonia 1994].

Finally, a virtual environment needs scripts as glue to make the environment provide the experience that the author intends. One could build a simple virtual environment where user-controlled objects and terrain were instantiated in the environment, and whatever happens from that point on makes up the simulation. However, at the very minimum, there must be a script to load these objects into the environment. If there are objects not controlled by the user, there must be scripts to define the behavior of those objects (which may exist as behavioral code within the objects themselves). In addition to controlling the behavior of each object, many virtual environments use time or masks (invisible "buttons" within the environment which are "pushed" when an object collides with them) to trigger events in the environment. For example, the amount and quality of lighting may be changed in the environment based on the time of day. An example

of a mask is found in a driving simulator, in which a certain part of the road is the mask; when the car passes over it, a computer-controlled car object on an upcoming cross-street is told to start driving toward the user's car [Hering 1998].

C. User Interaction

The user of a virtual environment will see a rendering of the objects and terrain in the environment on his monitor or head-mounted display. In most cases the objects and terrain have three-dimensional polygonal representations. Knowing these geometries, and having a viewpoint and view direction (usually corresponding to the location of the user's avatar object and the direction it faces), a view of the environment is rendered on the user's display from that view point, as if the user is actually in the environment with his eyes at that point. If the equipment allows, the scene can be rendered stereoscopically to give the illusion of depth. To change the position of this viewpoint (and of his avatar in the environment), the user manipulates an interface, either through key presses, mouse movement, tracker movement, or the manipulation of some other specialized input device (like mock tank controls for the tank simulator).



The player (not visible to himself) sees another player standing in a maze.

Figure 2-1: Sample view of a virtual environment

D. Distributed and Collaborative Virtual Environments

Using the framework presented in the previous section, it is relatively straightforward to create a collaborative virtual environment: just introduce many user-controlled objects into the simulation, and ensure that each can be controlled through non-interfering input and output devices. For example, one user may use the keyboard, mouse, and conventional CRT display monitor; a second user may use a head-mounted display and data glove hooked up to the same workstation; the virtual environment software will have been written to take input data from either source and send it to the user's avatar object, and to render the scene to either display. However, due to limited processing power and the small number of independent input and output devices one finds on a single workstation, it makes sense to distribute the virtual environment across many workstations. In addition to allowing more users to participate in the simulation, it allows geographically distant users to collaborate.

Distributing a virtual environment requires a method of dividing the environment among different processors and a protocol of communication and synchronization between the processes. There are many ways to do the distribution, which is a major research area for applications in general, not just virtual environments; some methods for virtual environments are covered in part IV (related work). One simple method is to use an object-oriented language and model each virtual environment object using a language object; this encapsulates the virtual object and gives an interface for communicating with the object. Then, objects can be instantiated on many different machines. The objects communicate with each other over a network connection using a protocol based on the interfaces created for the objects. This method, implemented by the author, is explained in more detail in parts V and VI.

III. ENHANCED AWARENESS

Most virtual environment applications rely on a certain set of human perception traits, such as depth perception, color and shape differentiation, and spatial sound perception, to form the user interface. In order to use these traits effectively, the applications must be realistic enough for those traits, developed in the real world, to work in the virtual environment, and vice-versa for training simulations. One example of a training simulation was built by NASA and the Virtual Environment Technology Laboratory at the University of Houston, which uses a virtual environment to teach astronauts from two different continents how to replace parts on the Hubble Space Telescope [Loftin 1995]. In this case, it is important that the simulation accurately depict the set of circumstances the astronauts will face so that the skills they have learned in astronaut training apply in the simulation, and that skills learned in the simulation can be applied in reality. In fact, the use of virtual reality for training has been the subject of much research, and this research shows it to be effective in that skills learned in the simulation transfer to the actual environment [Johnston 1995].

However, while realism is important, it may help to use purposely artificial constructs in a virtual environment (as opposed to aspects of the virtual environment that are accidentally artificial due to limitations of the technology) to make them more useful, taking advantage of findings from the cognitive sciences. For example, in training someone to quickly infiltrate a building about which much is already known by the simulation developers but not the trainees, it would help to model the building and highlight paths to take and targets to hit. These highlights would not exist in reality, but would replace directions, such as "take the third right then the fourth left," with an obvious path that could be followed in the simulation, over and over, until it is learned by the trainee. Or, in an augmented reality system, the building could be modeled in a wearable computer which projects the proper path on glasses, worn by the trainee, over the

trainee's view of the physical building during the actual operation. The mechanism proposed to allow these artificial constructs is *enhanced awareness services.*

According to Greenhalgh, *awareness* is the "quantification of the subjective importance or relevance of an object to the viewer" [1995]. In a collaborative virtual environment, a user needs to be aware of the locations and actions of other users (represented by avatars) and of autonomous objects in the virtual world. For example, awareness of objects in the virtual environment can also be enhanced in such a way as to benefit the users. In the real world, when given an array of almost identical potential targets, only one of which is the true target, to choose the correct one requires close identification of each possible target(this takes time, and time is critical in many training situations. However, if the one target differs significantly from other potential targets (contrasting color, blinking, etc.), it is found almost instantaneously [Goldstein 1996]. This and other enhanced object awareness capabilities can potentially be used to help pupils learn tasks in virtual environments.

As another example, consider the case where one user, the "coach," is attempting to show a particular path through an environment to another user, the "pupil." A user's viewpoint in a virtual world is the user's position and view direction. By varying the level of viewpoint awareness, the pupil's viewpoint in the environment can be shared to some degree with the coach's viewpoint. This situation is considered enhanced awareness since in real life, people have complete control over their positions and viewpoints, and cannot exist in the same place at the same time. Methods to implement enhanced awareness will be described later in the architecture sections of this paper.

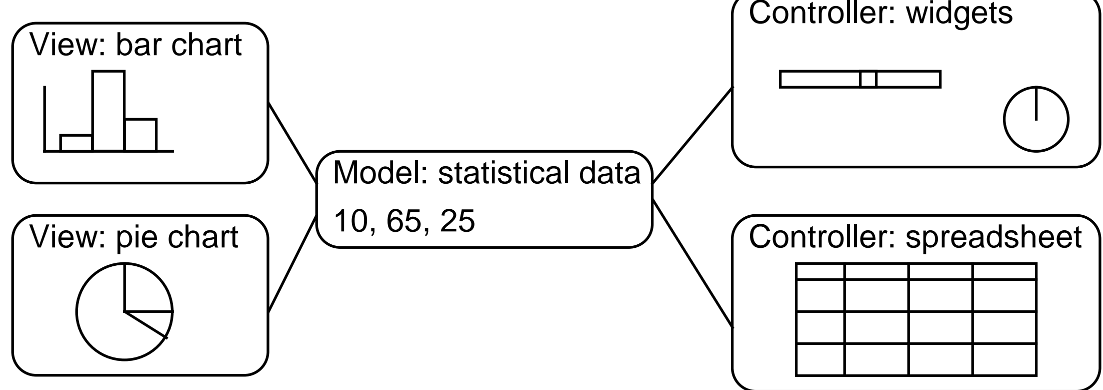
IV. RELATED WORK

Several architectures and systems will be presented in this section. None of them supports general-purpose collaborative virtual environments with enhanced awareness, yet all of them have techniques that can be applied to such a collaborative virtual environment architecture.

A. Model/View/Controller Architecture

The Model/View/Controller architecture, introduced in Smalltalk-80, provides a method of presenting many different synchronized views or presentations of the same set of data. In an application written in this paradigm, the data are stored in the model. There are one or more views of the data in this model. These views all show the same data but may show them in different ways. Finally, there are one or more controllers that allow a single user to change (and view) the data. An example of an application using the Model/View/Controller architecture is given by Sunsted, who describes a graphing program that maintains bar-chart and pie-chart views of the data in the model; the data are changed (controlled) through a user interface similar to that of a spreadsheet or through GUI widgets. Any time the data are changed with one controller, all graphs (views) of the data and other controller are updated automatically [1996].

This architecture was created for use in single-user non-distributed applications. Specifically, the model, views, and controllers exist in the same process space. One single user manipulates the controls and sees the views. The views "register" with the model, and any time the model is changed, all of the registered views are updated with the changed data. Thus for a general purpose collaborative virtual environment system, the M/V/C architecture would need to be expanded to work with many users and over several workstations.



The model's data can be viewed in two ways and controlled in two ways. When the data are manipulated, all views are synchronized automatically.

Figure 4-1: Model/View/Controller example

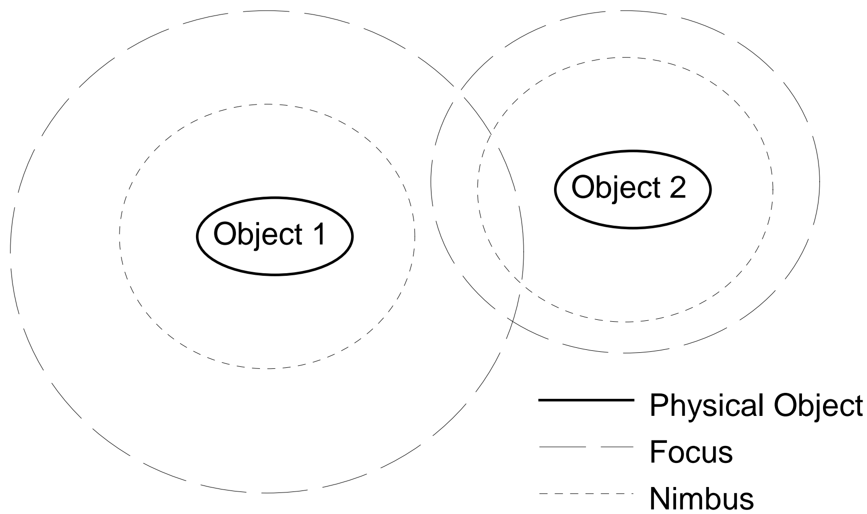
B. MASSIVE

The MASSIVE system [Greenhalgh 1995] is a virtual reality teleconferencing system. Its goals are the support of many participants, heterogeneity of interface equipment, operation over wide area networks, scalability, and spatial mediation instead of traditional floor control. It is not appropriate for a general-purpose collaborative virtual environment because it is a system built specifically for teleconferencing and not for simulations. It uses the spatial model of interaction as a way of maintaining scalability and controlling the amount of interaction between users.

Using the *aura*, MASSIVE can support scalability. Interaction is possible within a certain volume around each object in the virtual world; this volume is the object's aura. Interaction between two objects is only possible when their auras intersect; this is the only time a network connection is made between the two objects. The aura may be a bounding region around the object, or it may be a continuously valued function based on distance between objects, or any of many other possibilities.

Awareness is controlled at a high level through the aura, in that only objects with intersecting auras communicate. However, at a more detailed level, each object has a *focus* and a *nimbus*. An object's focus describes its allocation of attention, and the object's nimbus describes its ability to be observed. As Greenhalgh and Benford put it, "...the more an object is

within your focus the more aware you are of it, and the more an object is within your nimbus, the more aware it is of you" [243]. Like the aura, the focus and nimbus can each be bounding regions or one of many kinds of functions. The level of interaction between two objects is directly related to the awareness between the two objects, which is calculated in an application-specific way based on the focus and nimbus of each of the objects.



In this case, the focus of the first object intersects the nimbus of the second, therefore the first object can see and affect the second object. However, the second object's focus does not intersect that of the first, so the second object has no knowledge of the first object.

Figure 4-2: Focus-nimbus interaction example

C. NPSNET

NPSNET [Macedonia 1994] is a virtual environment built by the Naval Postgraduate School to support large-scale virtual environments (those with thousands of users) over the Internet. It is typically used for battle simulations. The primary concern of those working on NPSNET is developing strategies to support thousands of users given the limitations of current network technologies. NPSNET-IV, the current incarnation of NPSNET, is considered unique by its creators due to its support of the Distributed Interactive Simulation (DIS 2.03) protocol which provides application level communication among independently developed simulators, and IP

Multicast, which is the Internet standard for group communication, and heterogeneous parallelism for system support pipelines (graphics, application updates, and network communication).

Using the DIS protocol, very few data need to be sent across the network because the appearances of all possible types of objects are known to every workstation before the simulation starts. Once these libraries are loaded, an initial set of entities (tanks, soldiers, planes, etc.) and a terrain for the simulation are created on each workstation. New instances of these entities can be created and destroyed as the simulation progresses, but no new types of entities are allowed. Each workstation hosts one "active" entity, controlled by a human player, while the other entities are replicated from entities on other hosts. So, only information about an object's position and whether it is active or inactive is communicated between hosts. This allows the simulation to proceed at a high update rate but at the expense of runtime flexibility. Also, only asynchronous views are supported, because no communication of viewpoints is necessary between users. When the views between users is synchronized, there is too high a communication cost to support the sharing of views.

To synchronize the various replicas of entities around the network, NPSNET-IV uses the *players* and *ghosts* paradigm. Each entity is known on its host workstation as a "player." All of the other entities in the simulation are modeled on that host workstation as "ghosts." Periodically, but not at every frame of the simulation, a player on a host sends an update of its position to every other host. Every other host then updates its ghost of that player. Between these updates, the host updates the ghost using dead reckoning (using previous updates, it predicts where the entity should be moving at each frame. The function to compute this prediction is well known, so the player also knows where each host thinks it should be at each frame. When the players sees that this prediction differs from its actual position by more than a certain error threshold, it sends its actual position to every other host.

D. SOLVEN

Gareth Smith [1996] looks at ways to allow many users to see tailored views of a collaborative virtual environment. He reports that while current CVEs follow a strict What-You-See-Is-What-I-See (WYSIWIS) policy, it is more desirable to allow individual users to tailor their own display of the virtual world, so "relaxed" WYSIWIS is considered. However, user interface tailoring and WYSIWIS don't mix well; for example, consider the case where an application in a typical graphical user interface system has two buttons side-by-side. In this example, the shared data (the WYSIWIS part) are the cursor coordinates, while the button positions are the tailored interface. The first user has configured his left button to be "stop" and the right button to be "go," and the second user does the opposite. Now, in a collaborative version of this application, the second user sees the first's cursor in his display. The first user, over an audio channel to which the second user is listening, says, "press this button," pointing to the "stop" button. However, in the second user's display, the pointer will be on the "go" button, which is not what the first user intended.

The goal of his paper is to apply lessons, like that above, learned from 2D interfaces, to 3D interfaces. In particular, only subtle personalized changes in the 3D world database will be explored; major transformations are just "too hard," according to Smith. The model used by the SOL shared object toolkit was expanded for 3D worlds. In SOL, there is a set of application semantics, on top of which sits an access model, through which users access the application semantics. This access model uses a set of user configurations in order to determine how each user's interface is configured in order to deliver the data properly to that user. The 3D version is called SOLVEN (SOL-Virtual Environment extensioN) and is configured similarly, except that the "application semantics" from SOL is now the shared 3D world definition.

Smith gives four examples in which it is desirable to have different users see the same 3D world model, but with slight differences:

- *Shared virtual city:* Many users share the same city model for various tasks. Some users, however, are taking a virtual tour of the model. In this case, the tour guide can highlight the

structures of interest. These highlights will only appear in the displays of the users taking the tour; other users will not see them.

- *Data visualization:* In the application, selected data are surrounded by a highlighting structure. Instead of having all users see the highlights of all other users, each user sees only his own selections.
- *Multi-lingual shared spaces:* Signs in the virtual world are displayed in the language of the user's choice. This prevents signs from having duplicate messages (in different languages) and allows each user to use whatever language is most comfortable.
- *Interior design:* A model of a bathroom is presented. The full model contains all plumbing, electrical, and data/phone lines. However, the plumber may only want to see the plumbing, so she turns off the electrical and data/phone line displays.

To build environments that allow tailoring as in these examples, Smith proposes the Multi-Perspective Model. In this model, objects in the shared virtual world are described by two independent factors: an appearance and a modifier. The appearance is the geometric definition of the object. For example, in the multi-lingual case, the geometry of the text would change to form words in different languages. The modifier describes the brightness of the object (bright, normal, dim, and "off"--turning an object "off" makes in inaccessible to a user, giving a method of access control in the virtual world). For each object, and for each user, there is a view matrix (appearance x modifier). One entry in this matrix is "on," giving the appearance and modifier for that object for that user. Some of these entries may be blocked so that they cannot be chosen; this allows finer-grained access control.

Appearance

Mod.	Bathroom	Restroom	Toilette
Off			
Dim	Bathroom	Restroom	Toilette
Normal	Bathroom	Restroom	Toilette
Bright	**Bathroom**	**Restroom**	**Toilette**

Here, a view-matrix is shown for the sign for a restroom. User 1 (dotted oval) has chosen to see the sign in English and dim. User 2 (dashed oval) has chosen not to see any sign.

Figure 4-3: View-matrix example

V. COLLABORATIVE VIRTUAL ENVIRONMENT ARCHITECTURE WITH ENHANCED AWARENESS

In designing the mechanisms to implement enhanced awareness, there are two important considerations. The first is *scalability*: the mechanisms must not impede the scalability (in number of users) of the simulation. The second is *personalization*: each user's settings must not affect other users' views of the world model.

To achieve scalability, the number of connections, and data transmitted over each of those connections, need to be controlled. In designing this architecture, a simpler version of the aura will be borrowed from the MASSIVE project. As in MASSIVE, two objects can interact when their auras intersect. Without intersection of auras, there is no awareness between two objects. This will dramatically reduce on the number of connections that need to be made compared to the naive model of having all objects connected to each other. Replication and dead reckoning will be used to reduce traffic over these connections. In NPSNET, replication is done using players and ghosts. A similar method will be used here, implemented through the Model/View/Controller architecture. A "ghost" in this system will be a view and controller of the remote object as well as some parameters to update the view automatically and then synchronize it on the basis of an error threshold.

In support of personalization, the Multi-Perspective Model from SOLVEN can be used within each user's focus and nimbus to change the appearance of objects in the virtual world for that user and to change the appearance of that user to other objects, respectively. To change how other objects appear, as updates to object geometries and other properties are received by an object to be applied to its copies of other objects, they are passed through a filter associated with the user's focus. Similarly, information can be filtered by the object as it leaves the object to be viewed by others. Awareness not based on appearance, such as viewpoint sharing, is not part of the personalization problem since the viewpoint and other non-appearance aspects of an

object are already independent of those aspects of other objects. Both scalability and personalization will now be discussed in more detail.

A. Scalability

Using the Connection Manager to Control Connection Count

Each object maintains an aura which in this case will consist of two spheres, the focus and the nimbus, concentric with the object (the center of the object will be determined by the object's designer). The focus determines the volume of the world that the object can see and affect; the nimbus determines the volume of the world that can see and affect the object. Two objects need to establish communication between one another when one object's focus intersects a second object's nimbus. In this case, the first object can see and affect the second object, thus one two-way line of communication will be created (two-way because the first object can send commands to the second, and then receive the results).

It is the job of the connection manager to know the position of each object and the size of its focus and nimbus spheres. It notifies two objects that need to initiate or terminate communications (based on their positions and intersections of auras) so that they can open or close connections. So, each object will have a two-way line of communication with the connection manager. Limiting the shape of the focus and nimbus to concentric spheres was done to make the job of the connection manager easier since collision detection of arbitrary geometric shapes is a relatively slow process.

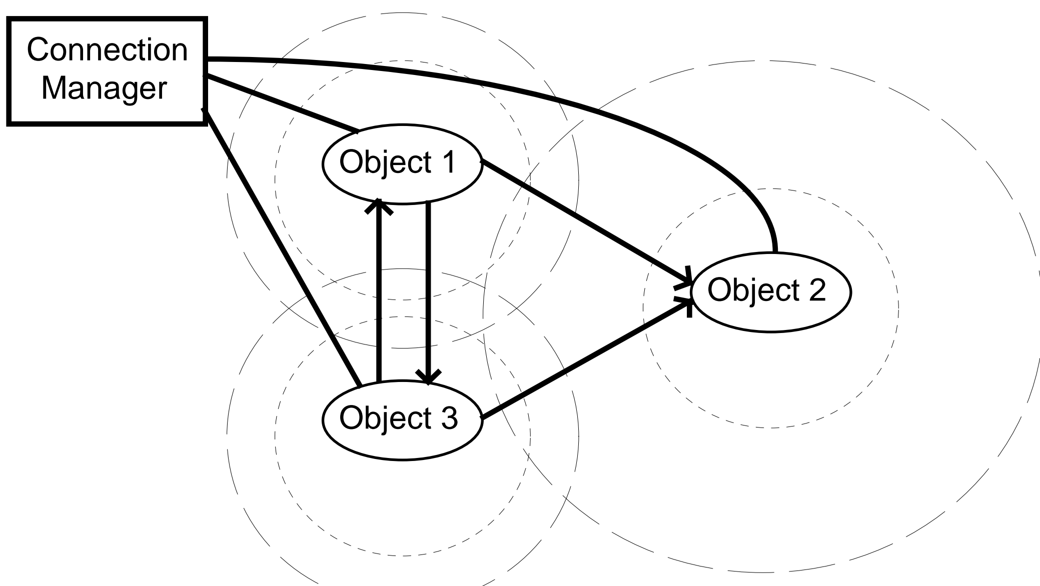
In summary, these are the lines of communication:

- *objects to connection manager:* Each object constantly updates the connection manager with its position and sizes of its focus and nimbus. For n objects, there will be n connections.
- *connection manager to objects:* When the connection manager finds that the sphere of one object (either focus or nimbus) intersect another object's complimentary sphere (nimbus or

focus), both objects are notified of this event. For n objects, there will be another n connections.

- *connected objects:* When the connection manager notifies two objects that they need to communicate, one two-way line of communication is created between them for each focus-nimbus intersection (so at most two two-way lines are created). [Note: If one object is being viewed by many other objects, it would make sense to have that object multicast information about itself to the viewing objects, but the current development platform doesn't support that.]

This communication model, on the average, should require substantially fewer lines of communication than the naive communication model in which there is no manager and every object has a connection to every other object, necessitating $(1/2)n(n-1) = O(n^2)$ two-way connections. Using this model, the maximum number of lines of communication occurs when all objects see and affect each other, giving n(n-1) two-way lines between them plus the n two-way lines from the objects to the connection manager. In the worst case this is more than the naive model but still in the same order (n^2).



Object 1 sees Object 3, and vice versa. Object 2 sees Objects 1 and 3, but they do not see Object 2. In reality these lines are two-way (allowing the object which sees another to affect it as well) but are shown as one way to indicate the "server to client" relationship. Also, all objects are connected to the Connection Manager.

Figure 5-1: Connection Manager example

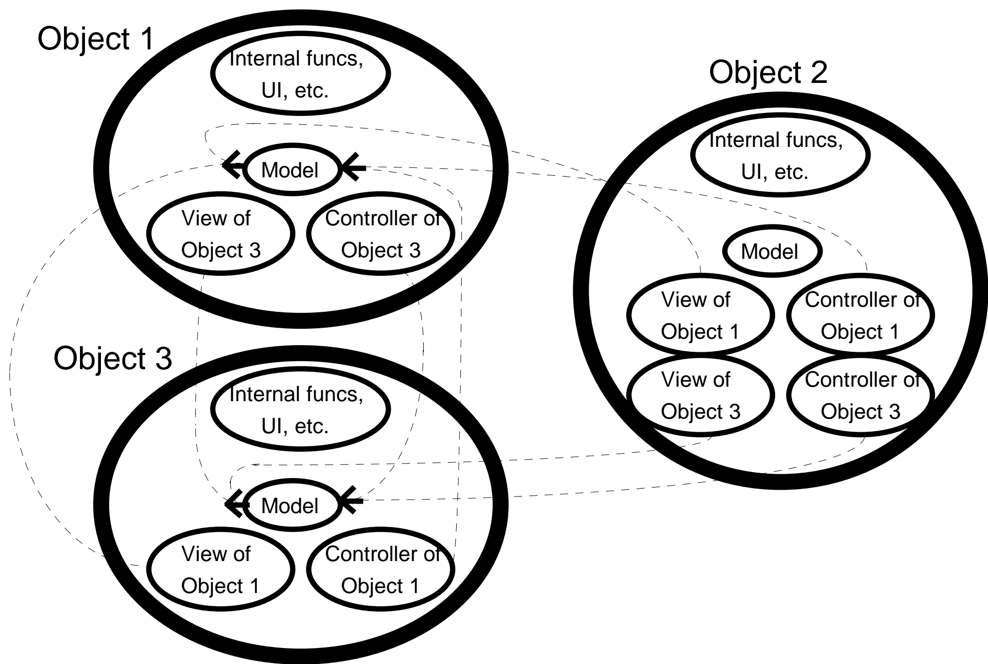
Replication Through Model/View/Controller

Each object follows the Model/View/Controller architecture. The model consists of the physical properties of the object (appearance, mass, material type, etc.) as well as its predictable behaviors (those that can be predicted by dead reckoning) and the sizes of the focus and nimbus. The object can be hierarchical, in that it has parts positioned relative to the main part of the object. These parts can have their own properties, or inherit them from the main object. For example, the tire on a car is made of much different material and has different behavior from the body of the car.

A view of an object is a replication of its model within some other object's process space. This view must be kept synchronized with the model within some error tolerance. Obviously, if the model changes frequently and it is being viewed by many other objects, a naive update protocol would take up a large amount of bandwidth, by sending every update to the model to every view on every frame of the simulation. Dead reckoning, as described in the NPSNET section, is a solution. The view of the model is changed by the object holding the view in a known way; the original object keeps a copy of its model and changes it using this known way. When this copy diverges from its actual model by a certain amount, the object updates the remote views with the data from the correct model. Straightforward dead reckoning would only work for easily predictable aspects of the model such as position of munitions as they fly through the air. However, it is possible to put behaviors in the model that are distributed to the views. This behavior code would be executed by the remote object. When the original model decides that the preprogrammed behavior has deviated too much from the actual state of the model, corrections will be sent to the views.

A controller of an object gives the viewing object a means to change the model of the viewed object. These can be either direct or indirect changes. In either case, the changes are taken as suggestions and the model of the object being changed decides which changes to effect (some changes may be ignored or altered depending on how the object was implemented). Direct changes are alterations to the data in the model, as when two users are building an object in the

virtual environment; they each can change the model of that object. Indirect changes happen when another object tells this one that some event has occurred, like a collision; the object being changed then decides what changes to make to itself based on this information. So in short, the difference between direct and indirect changes is a level of abstraction: the indirect changes are more abstract and need to know nothing of the data model.



These are the same objects as in the last figure, showing how one object contains a view and controller of another. The thick lines represent process space boundaries while the thin lines represent object boundaries. The dotted lines are network connections.

Figure 5-2: Object connection example

B. Personalization

The way a user can personalize his virtual environment is to change how remote objects appear to him and how he appears to remote objects. To change the local view of a remote object, a user attaches filters to the channels that bring in view updates. To change how he appears to others, he can attach a filter to the channel that carries outgoing view updates. The filter looks at the data as they pass through and makes appropriate changes using a lookup table.

For example, if a user wants an object to appear in a color other than the object's inherent color, he can set the filter to change any color data that pass through it to the color he wants instead. In the look-up table, the entry would be "(color, function to modify color)." So to highlight a certain object, the function to modify color would simply increase its brightness. Any other attribute can be changed similarly, for either incoming or outgoing data.

These lookup tables borrow from the SOLVEN view-matrix the idea that personalizations should be categorized along different attributes of the object into a table. For example, color is separated from geometry. The tables differ in that the possible personalizations are not all enumerated into the matrix; instead, a function which modifies the original data is given. This method avoids the limitations of having to choose from only those enumerated choices and of having to store a large matrix holding these enumerations.

One problem that can arise is if a user changes the local replica of the object so much that when he sends control information to the object, these requests make no sense to be applied to the original object. Reverse filters can be applied to these requests to solve this problem. Also, this mechanism doesn't solve the problem of viewpoint sharing; however, as previously stated, the problem of viewpoint sharing does not fall under these personalizations, but can be solved by extracting the viewpoint from the view of the object whose viewpoint the user wishes to share, and then using it to change his own viewpoint.

VI. IMPLEMENTATION

A. Language and Platform

Java 1.1 was chosen as the implementation language for several reasons. First, it is portable across machine architectures, which is desirable for a distributed virtual environment because heterogeneous platforms can be used. Second, it provides a well-designed object model which is easily used for creating representations of virtual environment objects. Third, its support for remote method invocation (RMI) makes the distribution aspect of the implementation very easy, providing almost transparent mechanisms for accessing remote objects, and its introspection features (through Java Beans) can be used by virtual objects to learn about each others' behaviors. Finally, this project is part of the Collaboration Bus project at UNC, which has adopted Java as its implementation language for many of the same reasons listed above (the rest of the Collaboration Bus aims to provide collaborative services for two-dimensional GUI-based programs, and this research is part of an effort to do the same for three-dimensional virtual environments) [Dewan 1996].

In theory, it should not matter which machine is used for development when developing a Java program. However, because this project requires high performance three-dimensional graphics, a Silicon Graphics O^2 was used. Unfortunately, there is as yet no Java package for doing efficient three-dimensional graphics, much less one which exploits the specific hardware of the SGI machine. Therefore, using Java's native method invocation (NMI) mechanism, the OpenGL library was used for rendering the virtual environment [OpenGL 1992]. A public domain stub object by Leo Chan at the University of Waterloo, with a little tweaking by the author, provided a reasonable interface to the OpenGL functions from the Java side. Additionally, since the author didn't want to spend a lot of time building his own slow or ineffective collision detector,

the very efficient and effective V-COLLIDE package developed by the UNC Research Group on Modeling, Physically-Based Simulation, and Applications was used through NMI stubs [UNC 1998]. Thus, the full project is not platform-independent, though objects that do not need to do rendering or collision detection (like terrain objects) can be instantiated on any machine that supports Java, and objects that need collision detection but not graphics (such as computer-controlled objects) can be instantiated on any machine that supports Java and a C++ compiler which can compile V-COLLIDE.

B. Implementation of the Connection Manager

The Connection Manager (`ConMan.java`, `ConManImpl.java` in package `ve.conman`) is a typical remote object. The class `ConMan` defines the remote interface which gives objects methods to call to register themselves with the Connection Manager, unregister themselves, and update their positions, focus radii, and nimbus radii. A "bootstrap" procedure must be followed to start the Connection Manager due to the way RMI works. The Java program `rmiregistry` is run on the host computer and is given a port number to use (it defaults to 1099). It establishes an RMI registry on that machine at that port. Then, the class `ConManImpl` (implementation class for the `ConMan` interface) starts and is given the port number to use. It registers itself with the RMI registry as "ConMan." When a new virtual object enters the simulation, it is told the machine name and port at which the RMI registry resides. It asks the registry for a reference to the "ConMan" object. Once it has this reference, it can access methods listed in the `ConMan` interface just as if the “ConMan” object were instantiated within the process space of that virtual object; Java RMI makes all remote method invocations transparent once the remote references are set up.

After a virtual object registers, the Connection Manager makes a local copy of its pertinent information: its position and the sizes of its focus and radius. Any time an object changes one of these parameters, it calls the appropriate method in the Connection Manager to update the local copy. The Connection Manager uses these copies to decide which objects need

to communicate with each other through a method called `detectAuraCollisions`. This method looks at all pairs of registered objects and for those which need to be connected, which are those whose focus and nimbus spheres intersect, and aren't already connected. By keeping a simple class of information for each object of objects to which it is connected, the Connection Manager knows when to create a new connection, assume one already exists, or break an existing connection. These connections are made and broken with appropriate remote methods within the objects, explained below, which the Connection Manager calls through references to the virtual objects. Note that once the "bootstrap" procedure has been completed in RMI, objects can now pass around references without going through the RMI registry. So each object gives a reference to itself to the Connection Manager, which is used just as a local reference would be.

C. Implementation of Virtual Objects

A hierarchy of virtual object classes was created to support different levels of capabilities. The idea is that the application developer will subclass one of these classes depending on what capabilities his object needs. At the lowest level is the `VObject` interface (this and all classes in this section are in the package `ve.vobject`) which just defines the remote methods that any virtual object must support. At the next level is the `BaseObject`, which implements the methods of `VObject`. `BaseObject` is for implementing simple unaware objects such as terrain; these objects have no awareness of other objects in the environment. The `AwareObject` class adds awareness of other objects to `BaseObject`, as well as collision detection. Finally, the `UserObject` class augments `AwareObject` by adding a user interface that allows the user to move the object using the keyboard and change various enhanced awareness functions.

VObject

The remote methods defined in `VObject` are either for use by other virtual objects or by the Connection Manager. When the Connection Manager decides, for example, that object1's

focus has intersected object2's nimbus, it knows that object1 needs a copy of object2. So the Connection Manager calls object2's `requestRemoteCopy` method, giving it a reference to object1. Through this method, object2 calls object1's `makeLocalCopy` method. The parameters to `makeLocalCopy` are the attributes of object2 that need to be replicated, which are its position, orientation, polygonal geometry, and a name for the virtual object. This method makes a local copy of object2 within object1, then passes a reference to this copy back to object2. Object2 puts this reference in a list of references to its remote copies. Now, whenever object2 changes its position, orientation, or geometry, it walks through this list of remote copies and makes sure each copy makes the same changes.

`VObject` also defines remote methods for changing aspects of the object, such as its position and orientation; these are the remote methods that one object calls when it is updating each of its remote copies and that objects call within themselves to change themselves. Finally, `VObject` defines methods for destroying copies, necessary when two objects no longer need to communicate, or when an object leaves the simulation. When the Connection Manager decides that an object needs to cease communications with other objects, it calls `destroyCopy` in all of the other objects with the first object's id; then those objects destroy that copy. In addition, if the first object is leaving the simulation completely, the Connection Manager calls `forgetRemoteCopiesIn` of all of the other objects with the first object's id so that if the other objects have their own copies held by the first object, they will quit trying to send updates to those copies when the holding object is gone.

BaseObject

`BaseObject` implements the methods listed in the `VObject` remote interface. In addition, it defines a few extra methods for making sure the object's geometry is reset correctly when the object changes position or orientation, removing the object from the Connection Manager, and reporting errors. Since `BaseObject` objects don't have awareness of other objects, any implementation of a `BaseObject` should have a focus of a very large negative

number so that Connection Manager never tries to get it to make local copies of remote objects. In fact, the methods `makeLocalCopy` and `destroyCopy` in `BaseObject` do nothing but remind the user not to call them; they are necessary because they're defined in the `VObject` interface.

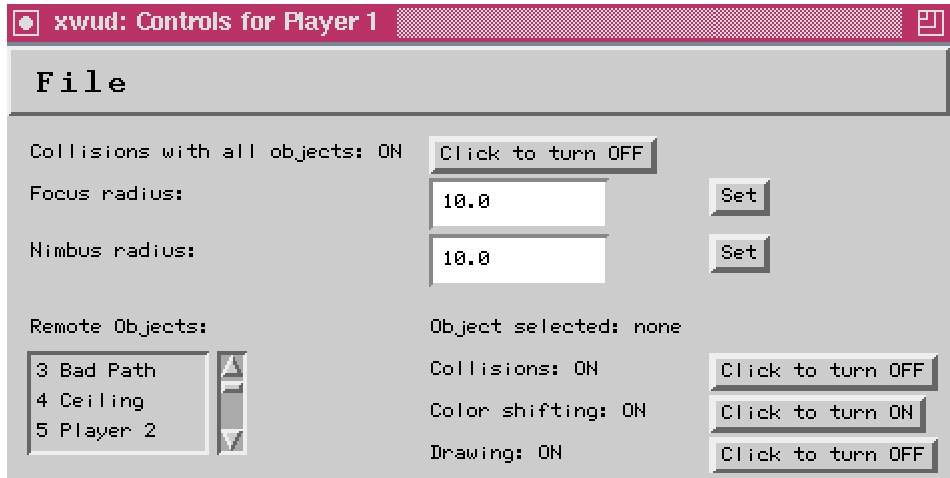
AwareObject

`AwareObject` adds awareness of other objects and collision detection to `BaseObject`. It defines the methods `makeLocalCopy` and `destroyCopy` as explained in the `VObject` section. It also adds a reference to the V-COLLIDE collider and creates two arrays to keep track of which objects this objects is hitting, and for how long (so that new collisions can be differentiated from existing collisions). `AwareObject` starts a new thread that runs a `collide` method that continually checks for collisions and updates the two collision arrays. Finally, `AwareObject` augments many of the methods in `BaseObject` to make collision detection work (for example, the movement functions call the superclass's movement functions, then update the collider based on that movement). Its constructor sets up the collider after the constructor of `BaseObject` is called. Its `makeLocalCopy` and `destroyCopy` add and remove object geometries from the collider, respectively, in addition to implementing the methods (since they weren't truly implemented in `BaseObject`). Finally, its movement methods (which change position and orientation) call the same method in `BaseObject` and then reflect the changes in the collider.

UserObject

`UserObject` adds a user interface to `AwareObject`. To do this, it creates an OpenGL widget to render the object's view of the virtual world, then starts a new thread which continually draws the local copies of remote objects that this object contains. It also creates a new window with a frame of Java components that allow the user to move the object in the world through key

presses (e.g., press UP to move forward, LEFT to rotate counter-clockwise, etc.) and to change aspects of this object and the enhanced awareness functions applied to other objects.



The top half contains controls for this user.
The bottom half contains controls for other objects.

Figure 6-1: User interface for a UserObject.

For each copy of a remote object, `UserObject` keeps some enhancement information. In this version of the software, the enhancements allowed for each copy are limited to making an object blink, turning off collisions with that object, and turning off the rendering of that object. These three functions capture the basic semantics provided by enhanced awareness services: making an object change its color makes it stand out, just as would making it blink or making other drastic changes to its appearance. Turning off collisions with an object allows a user to walk through walls (for example) and thus obtain view points not typically allowed in a simulation. Not drawing an object, when combined with turning off collisions with that object, make it seem as if the object isn't there at all. Because this information is kept locally within the `UserObject` instance, each user's view of the virtual environment is independent--turning off an object in one user's view doesn't turn it off in another's view. To make effect these enhancements, the user chooses an object (listed by name and id in the list box on the left) then presses buttons on the right side of the frame corresponding to what the user wants to do. Note that if the user chooses

`<this obj>`, it is the same as choosing no object, since these controls are not used for changing the user's object.

The interface also allows two changes to be made to the `UserObject` instance itself. First, the user can turn off collisions with ALL other objects. Second, the user can change his focus and nimbus radii. Turning off all collisions really is just a shortcut for turning off collisions with each individual object as described above. However, by manipulating his focus and nimbus radii, a user can become a "stealth" object, a "blind" object, or any level in between. To become a stealth object, the user sets his focus radius to a very large positive number and his nimbus radius to a very large negative number. Then he can see everything, but nothing can see him. Doing the opposite makes the user a "blind" object. By choosing radii between these two extremes, various conditions can be simulated, such as limited visibility, augmented vision, and so on. To effect these changes, the user presses the collisions button to turn collisions on and off, or changes the focus or nimbus radii in the text box then presses the set button to put that value into effect.

VII. A SAMPLE APPLICATION

A. Application Description

The system was used to implement a simple capture-the-flag game (package `capflag`). In this game, there are two flags in the middle of a maze. Two or more players start at different entrances to the maze and the first player to find the right flag wins; if the wrong flag is chosen, the player blows up. This game requires a few basic objects: a player, a flag, and a maze.

Player

The `Player` class is a subclass of `UserObject`. The main method loads in a geometry (as a set of triangles) from a file generated by a three-dimensional modeling program, chooses a name, position, and orientation, then instantiates a `Player` object, which then registers with the Connection Manager. Through command-line arguments the above-mentioned aspects can be changed, so for a simulation with many players, the same class can be instantiated many times with different parameters for each player. The name and port number of host running the Connection Manager is also a command-line option, so that the object can connect to any Connection Manager on the Internet.

The `Player` class augments `AwareObject`'s `collide` method by checking which objects were just hit. If a solid object like a maze wall was just hit, the user is bounced back off of that object. This is not accomplished through physical simulation but rather through a simple mechanism that just undoes the move that made the user hit the solid object. Modified movement methods track the last move made so that the object can backtrack one step in the case of such a

collision. Also, the augmented collide function decides that the player wins when he hits the right flag, and loses when he hits the wrong flag.

Flag

The `Flag` class is a subclass of `BaseObject`. Using command-line parameters, it loads in a flag model and positions it in the virtual environment after connecting to the Connection Manager. In the capture-the-flag simulation, two identical flags are instantiated in the environment. The player knows one flag is called "Right Flag" and one is "Wrong Flag," but he won't know which is which without the use of his enhanced awareness functions.

Maze

The `Maze` class is also a subclass of `BaseObject`. Like the `Player` and `Flag` classes, it reads in its position and other information from the command line. In addition it gets the name of a maze file. Using this file, it creates geometries representing the maze walls, the ceiling to the maze, a good path to the flags, and a bad path to the flags. The two paths are identical, but with enhanced awareness functions, the player can differentiate the two and follow the correct path to the flags. He can also turn off collisions with the ceiling and stop it from being drawn, then fly above the maze and see a top-down view. He can also stop the maze walls from being drawn, increase his focus radius, and see the flags and other players in the simulation with a view unobstructed by the maze walls.

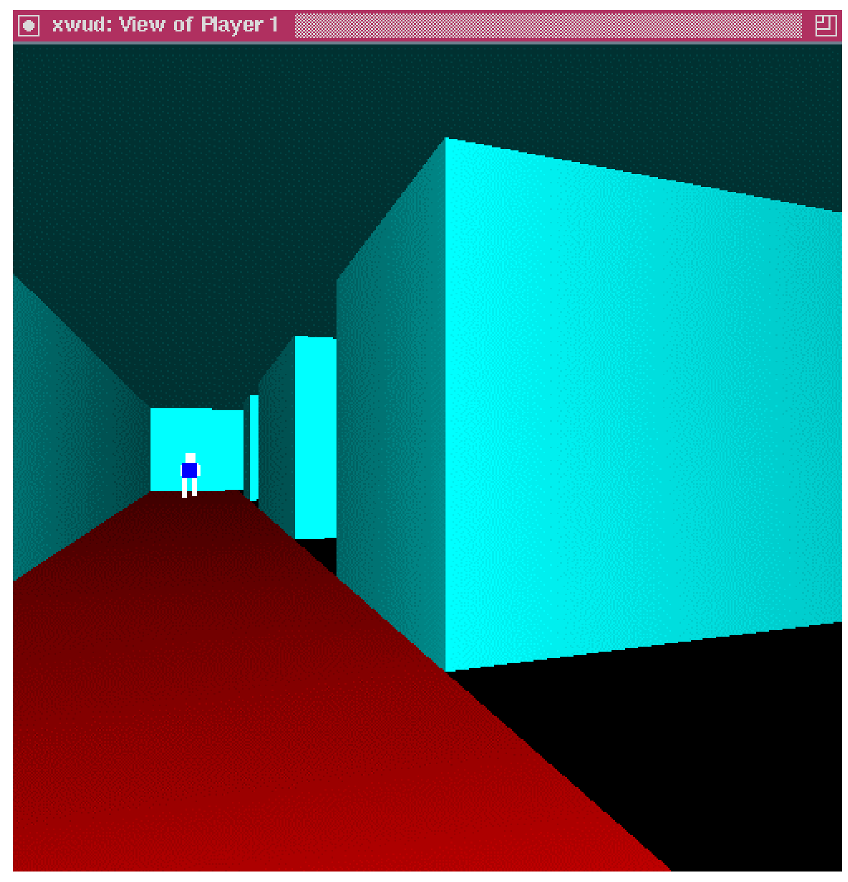
B. Results from Running the Application

The game is started by choosing a server and port for the Connection Manager and starting the RMI registry on that server with that port as its argument. Then the Connection Manager (`ConManImpl.class`) is started on the same host, given that port as its argument.

Then on any servers which support whatever native libraries the objects may need (V-COLLIDE and OpenGL in this case), the objects are started with their necessary parameters including the Connection Manager server name and port.

As the Connection Manager sees that the auras of objects intersect, it directs some objects to make copies of others, depending on the exact focus/nimbus intersections. Then the copies are slowly transferred through remote method invocation. Remote method invocation is really slow for highly interactive programs like virtual environments. There is very noticeable lag in even simple position updates between objects: the more data, the slower the transfer. It has been shown that for some implementations of RMI, any type of value other than `Byte` will be mired in data marshalling and unmarshalling routines, and that it is desirable to convert data to `Byte`s before allowing them to be sent by RMI. This optimization was not implemented however.

Aside from the slow transfers of data, the application showed that enhanced awareness functions can be useful in ways indicated previously. It seems useful, when learning how to navigate a maze, to be able to see the correct path flashing, which clearly differentiates it from the wrong path. In the following two images, one can see in the player's view and in his user interface settings that the wrong path has been turned off completely, leaving only the correct path through the maze showing.



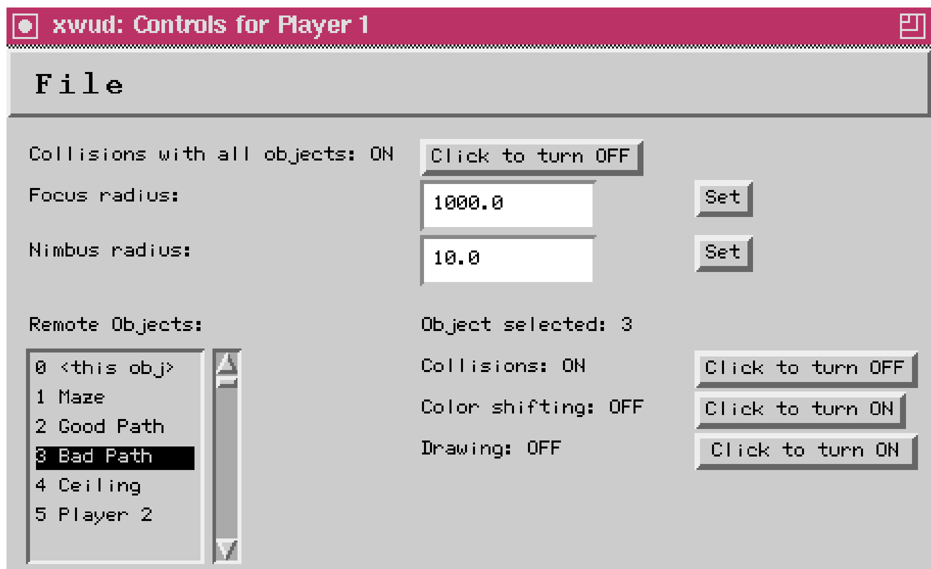
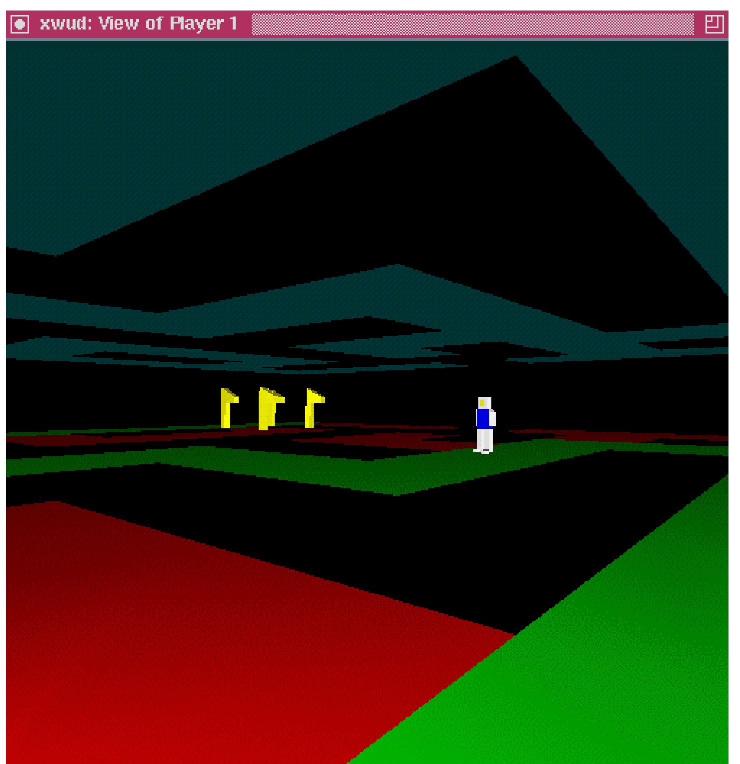


Figure 7-1: User has turned off bad path.

In the following case, the player has turned off the maze walls, so that he can see where the other player who otherwise would be hidden behind a wall (the remaining objects are the ceiling and good and bad paths through the maze).



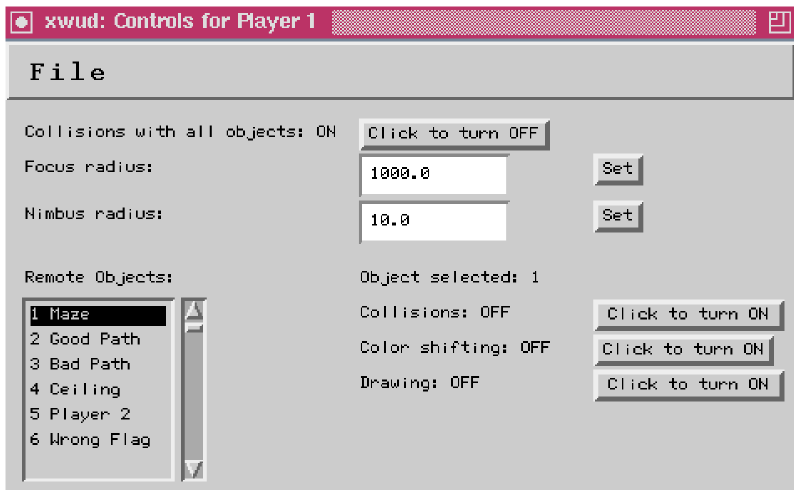


Figure 7-2: User has turned off maze walls.

It also seems useful to be able to exit the maze, fly above it, and see a top-down view of the maze and its contents. In the following view, the player sees the entire maze, including the positions of the flags and the other player. Notice in the user interface that collisions with and drawing of the ceiling have been turned off.

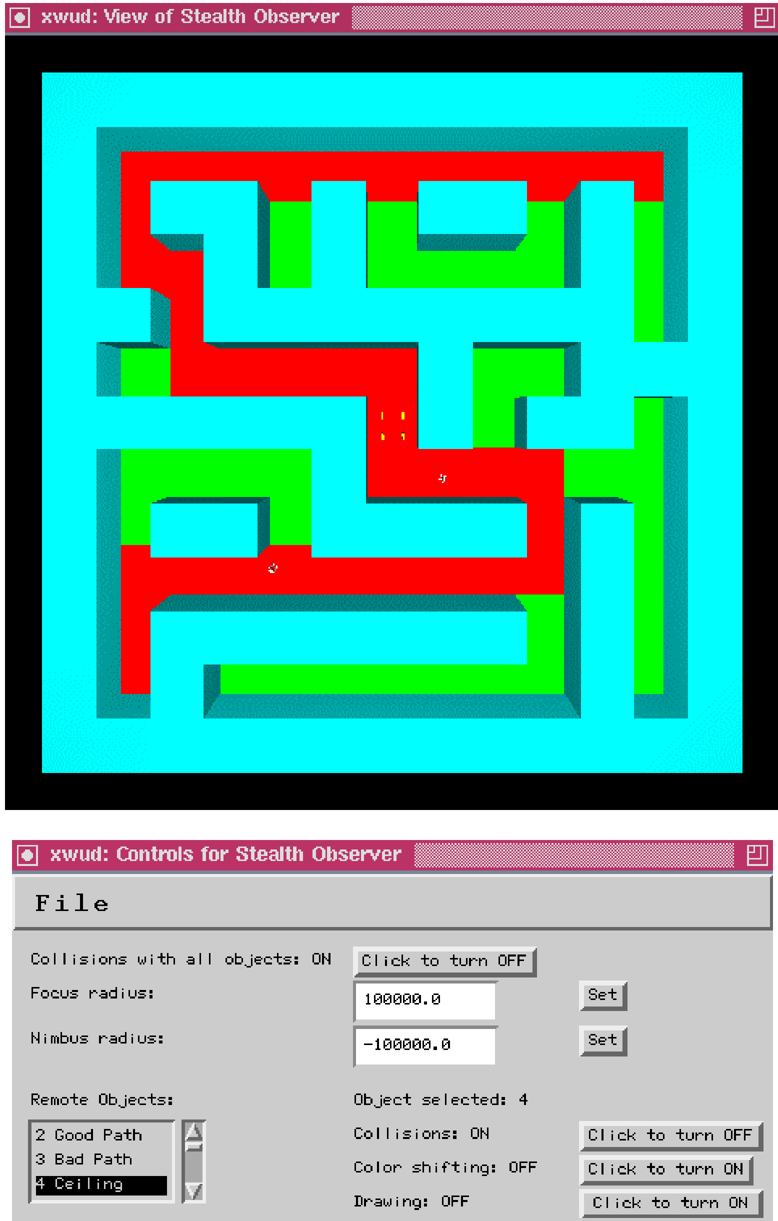


Figure 7-3: User views maze from above.

Finally, it is useful to know which flag is the correct one to pick when there are many identical flags and only one is the correct one. In this case, there are only two flags. At first the user sees the two flags and knows from looking at his object list that one is the right one and one is the wrong one.

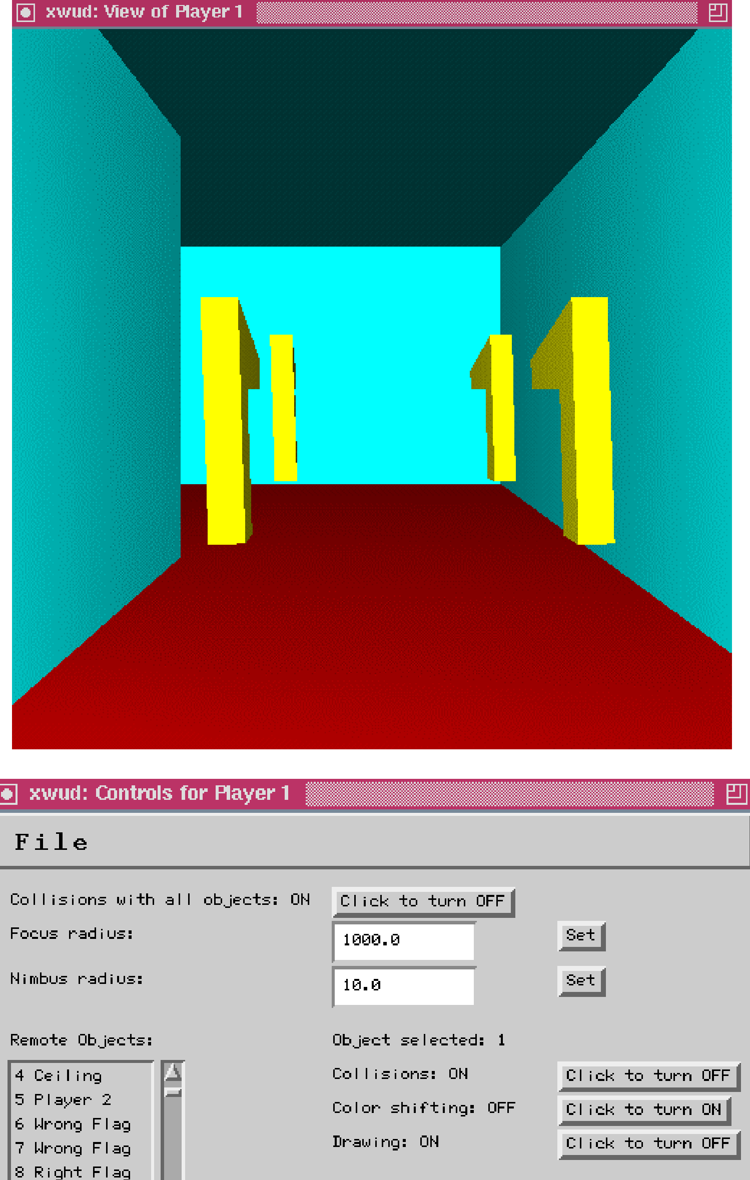


Figure 7-4: User sees four identical flags.

However, the user is able to turn off the drawing of and collisions with the wrong flag, leaving only the right one to see and collide with. If this situation modelled a realistic situation, the user training with this software would easliy learn which flag is the correct one and apply that knowledge in the field.

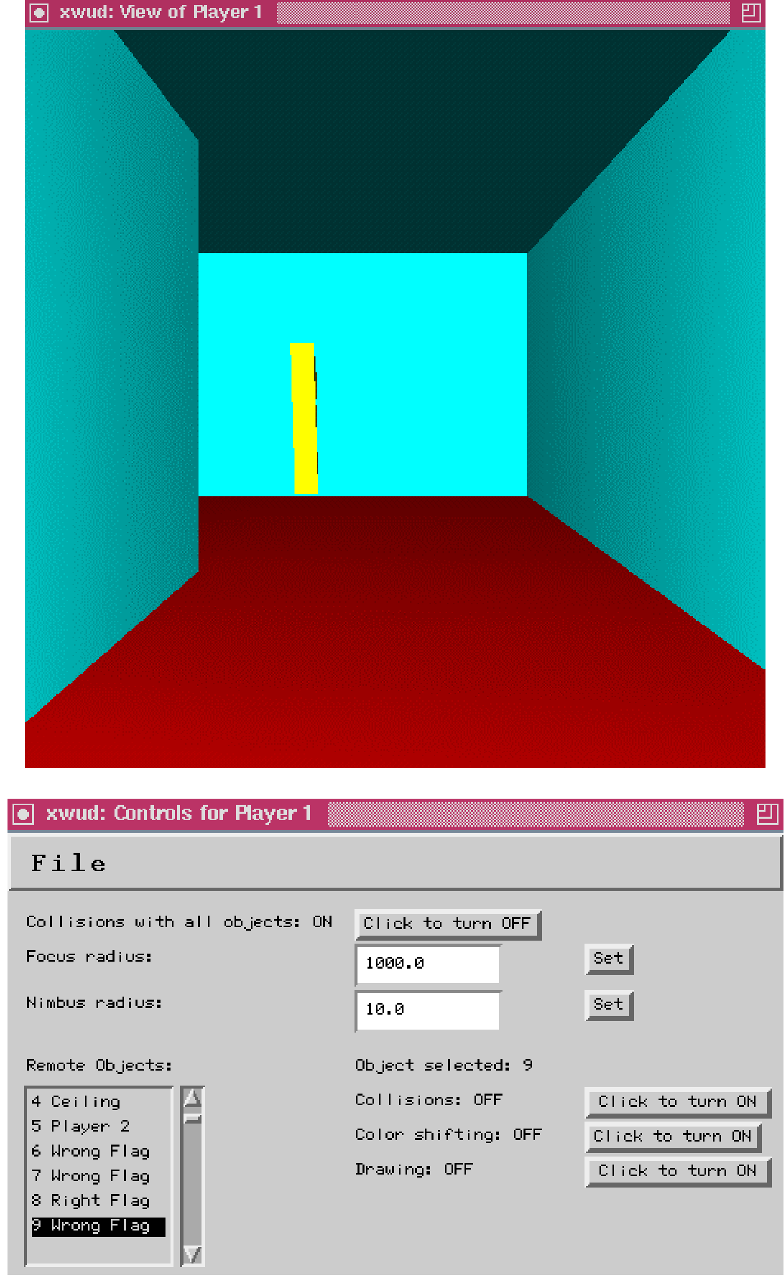


Figure 7-5: User now only sees one flag.

VIII. CONCLUSIONS AND FUTURE WORK

A general purpose architecture for collaborative virtual environment simulations that supports enhanced awareness has been presented. It is based on techniques used in existing special-purpose virtual environments for overcoming scalability and personalization problems, but was designed with general-purpose use as another primary concern. Reusable enhanced awareness services were added to the implementation. The system was used to develop a sample application that showed how the enhanced awareness services can be useful in at least that application. In future work, usability studies should be run to demonstrate the effectiveness of this design in more realistic training simulation.

To make this system support realistic simulations, it needs several improvements. First, it needs to be faster. Currently the main bottleneck is using remote method invocation, which has a high overhead cost per invocation, to transfer data. A simple, faster mechanism like sockets would help this problem immensely. Another slowdown comes from updating every copy of an object every time the original makes a move. Using a dead-reckoning scheme like that of NPSNET would cut down the number of updates sent, but one was not implemented in this system due to time constraints. Second, it needs a better object model. Currently, the objects are very simple. There are no provisions for one object learning another object's specific behaviors, beyond the known methods of the VObject interface. Finally, more enhanced awareness services need to be added. The small set implemented in the application was enough for the capture-the-flag game, but it does not give all of functionality described in earlier sections of this paper.

WORKS CITED

Dewan, Prasun, et al. 1996. *Collaboration Bus: An Infrastructure for Supporting Interoperating Collaborative Systems.* (online at http://www.cs.unc.edu/~dewan/cb.html).

Goldstein, E. Bruce. 1996. *Sensation and Perception.* Brooks/Cole Publishing Company.

Greenhalgh, Chris, and Steven Benford. 1995. MASSIVE: A Collaborative Virtual Environment for Teleconferencing. *ACM Transactions on Computer-Human Interaction* (September).

Hering, Dean and Richard Spencer. March 25, 1998. Presentation of Michelin Simulator System by the Research Triangle Institute at the University of North Carolina-Chapel Hill.

Johnston, Rob. 1995. The Effectiveness of Instructional Technology: A Review of the Research. *Proceedings of the Virtual Reality in Medicine and Developers' Exposition* (June).

Lisle, Curtis, Marty Altman, Mark Kilby, and Michelle Sartor. "Architectures for Dynamic Terrain and Dynamic Environments in Distributed Interactive Simulation." 10th DIS Workshop, Orlando, Florida, March 1994.

Loftin, R. Bowen. 1995. *Hands Across the Atlantic*. On-line demonstration report (http://www.vetl.uh.edu/sharedvir/handatl.html).

Macedonia, Michael R., Michael J. Zyda, et. al. 1994. NPSNET: A Network Software Architecture for Large Scale Virtual Environments. *Presence* 3(4).

OpenGL Architecture Review Board. 1992. *OpenGL Reference Manual.* Addison-Wesley.

Smith, Gareth. 1996. Cooperative Virtual Environments: Lessons from 2D Multi User Interfaces. *Computer Supported Cooperative Work Proceedings* (November).

Sundsted, Todd. 1995. How-To Java: Observer and Observable. *JavaWorld* (online periodical; article at at http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto.html).

UNC Research Group on Modeling, Physically-Based Simulation, and Applications.1998. *V-COLLIDE: Collision Detection for Arbitrary Polygonal Objects.* (online at http://www.cs.unc.edu/~geom/V_COLLIDE/).