Towards a Virtual Environment for Interactive World Building

Dieter Schmalstieg and Michael Gervautz

Institute of Computer Graphics, Vienna University of Technology, Austria
schmalstieg|gervautz@cg.tuwien.ac.at
http://www.cg.tuwien.ac.at/

Abstract

We propose an architecture for virtual environments that is aimed at the construction of a global, continuous, shared, and persistent information space. Such a system must support three-dimensional interaction of a large number of users and allow integration of user-developed applications. Our system is based on a client-server architecture: The server is responsible for management and maintenance of the virtual environment. Users can be present in the environment using software clients. The environmental database is composed of object-oriented units with multiple levels of autonomy. Scalability is achieved by a network of servers that cooperate by taking on responsibility for different regions within the simulated environment. We also report on the status of our ongoing prototype implementation of this architecture.

CR Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems - *Distributed Applications*; I.3.2 [**Computer Graphics**]: Graphics Systems - *Distributed/Network Graphics*; I.3.7 [**Computer Graphics**]: Three-dimensional Graphics and Realism - *Virtual Reality*;

1 Introduction

In this paper, we propose a virtual environment architecture which aims at the construction of a Cyberspace, a true virtual "world" in the sense that like in the real world we inhabit, there exist no limits on content or size other than our imagination and skill. We show what software component are necessary for such a task, and also report on the state of our ongoing prototype implementation.

1.1 System requirements

We list a suite of requirements that we believe are essential for a global virtual environment, and that go beyond capabilities of scene viewers for simple browsing of 3-D scenes (for a more complete analysis, refer to [SG95]):

- The performance should be adequate for interactive working. In particularly, if the environment data is distributed, efficient management of the network is necessary.
- The three-dimensional information space should be continuous. While the connection of virtual environments with portals (3-D hyperlinks) as a metaphor works in principle, we believe that a geometric arrangement of the information spaces is more natural for 3-D interaction.
- The environment should not restrict the support for multiple concurrent users. Sharing of the data contained in the environment should be supported, and multiple users should be able to interact with each other. Such a requirement also means that the system must scale to allow a large number of concurrent users in the same environment.
- Persistence of content should be possible. Not only should the data and applications within the environment continue to exists even if no human participant is logged into the system, also the environmental simulation and execution of applications should continue.
- The system should provide interfaces that allow arbitrary applications to be developed to work within the environment. In particular, users should be able to populate the environment with software agents that provide services to others.

1.2 Related work

Several researchers have reported on software architectures for shared virtual environments that support a rich set of interaction and applications [SW94] [CH93] [Sha93] [BC94] [Sin94] and rapid development [WGS95] [PBC94]. However, these are mostly limited to local networks and do not scale beyond a handful of users. Some work has been published on reducing communication cost between large groups of participants to make larger scale environments possible by use of clever algorithms [Mac94] [Fun95] [BF93] [Sin95] or novel network protocols [Mace95]. Recently there are development

for the integration of virtual environments and the World Wide Web [HMR]. In that context, a spatial subdivision of the universe has been proposed [Bre95]. However, research work on a global virtual environment is just in its beginning.

1.3 System overview

To address the needs of a global virtual environment, we propose the following system design:

The virtual environment is represented as a database in a virtual environment server. The server executes a simulation process that keeps the virtual environment alive. Users connect to the server via a client software over the Internet using standard protocols. They are then free to choose a so-called Avatar that represents them in the virtual environment and mediates interaction. While the Avatar is controlled by the user over the network, the server manages the Avatar and forwards communication and interaction between the Avatar and other user's Avatars and software agents that inhabit the environment. The software agents - that we call actors - are the representation of applications in the virtual world.

The actors' object-oriented architecture allows multiple levels of actors complexity. Actors can either be "dumb" and respond to stimuli in a predictable way, or they can carry their own behavior inside. Behavior is either specified by a scripting language, or it is controlled by external applications over a standard interface. To the environment manager, the user is another external program in control of the Avatar actor. Our actor architecture allows dynamic modifications to the system by creating new actor types while the system is executing. It also allows actors to migrate between multiple virtual environments.

With the simulation being carried out within the server, the task of the user client is to mediate the interaction between the environment and the human user. It supports the dynamic mapping of interaction styles to the available interfaces and devices. It also displays the three-dimensional scene to the user. A sophisticated protocol provides high performance remote rendering of the environment at the client site.

Scalability is achieved by assigning parts of the virtual universe to different servers based on regions. Our initial proposal is a regular 3-D grid. The server is responsible for the actors that live within its region. We assume that the regions are large enough so that the majority of interactions is local. Such coherence allows most interactions to be efficiently carried out within one server, and inter-server communication happens only between adjacent servers.

The rest of this paper discusses various aspects of our design. We present the fundamentals of our approach: the client-server network infrastructure (section 2) and the software architecture of actors (section 3). We then examine some interesting aspects of our design in greater detail: constraint-based behaviors (section 4), group and world management (section 5) and efficient database management by exploiting spatial coherence (section 6). We also report on our efforts for a prototype implementation (section 7).

2 Network topology

2.1 Servers

We run the virtual environment on a distributed network of servers. To achieve load balancing, the simulated universe is divided into regions. Each server manages the actors contained in its assigned volume. The server provides the medium for the actors to communicate among with other, thereby keeping the simulation alive. The server also communicates with its neighboring servers, and routes messages to actors that live on remote servers. Every actor has a name unique in the environment, so it may be addressed transparently, without caring about server connections. The only difference between local and remote communication is that routing the message through the server network will take longer than local communication.

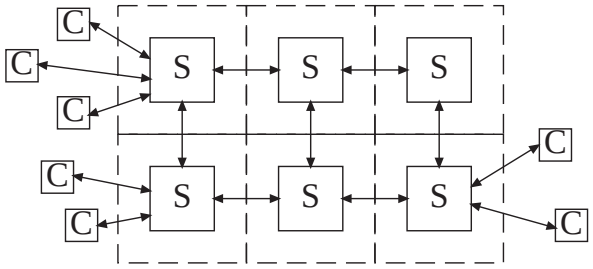

Fig. 1: Client-server architecture. The world is geometrically divided over a network of servers

To make this scheme efficient, geometric coherence is exploited. Communication over the network is costly, whereas communication within a server is not so critical. Most applications take place in a bounded region of space. As long as all actors involved with a particular application are held on a single server, all communication concerning the application will be between actors within the server. Communication to actors on servers far away is possible, but it will

relatively infrequently happen, so that the performance penalty is minimal. The only remaining network communication is to the user's client (fig.1).

This architecture allows to hide the division of the environment and the presence of multiple servers from the user, so that the illusion of a continuous three-dimensional space populated by actors is preserved. When a user leaves the domain of one server and enters another, a new connection is built to the destination server, and the user's Avatar is "gracefully" handed on. Ideally, these changes should be completely transparent to the user.

2.2 Clients

Formally, a client is a process independent of the server, that has some interest in the virtual world. The client may just observe the progress inside the VE or take influence. Observation can be done by sending a request for information to the server or to an actor that is managed by the server. By placing a camera in the VE, the client will receive a constant stream of visual information that reflects the changes in the scenery as seen by the camera. This information stream is then visualized (i.e. rendered) by the client.

Influence on the VE can be achieved by controlling an actor by sending command messages. Special commands involve the creation of new actors, the creation of new actor types, and the deletion of actors. We distinguish two types of clients:

- **Avatar client**: The client creates an Avatar for the user in the VE. It is also responsible for reading the user's input and presenting the VE system's output. The client allows various forms of interaction, determined by what physical devices (3D-mouse, glove, HMD...) are available at the user's site. If a device for a particular form of interaction is missing, the client provides a software substitute (e.g. if an application requires turn-knobs but they are not available, a dialog window with a turn-knob widget is presented instead).
  The user's commands are send from the client to control the Avatar, who in turn interacts with the environment on the server. Updates resulting from the simulation going on at the server are sent back to the client, which presents the modified environment to the user. In particular, the client creates visual output by generating images of the VE.
- **Animation client**: An application may require the computation of complex control for a potentially large number of actors. Such a task cannot be carried out in the server process to avoid computational overload. Instead a client process running on a separate, powerful machine, can perform the calculations, then send the results in the form of update commands to the server. For example, a physically based simulation for deformable objects may compute the deformation of an actor's geometry, then update the polygonal representation according to the deformation. Usually a client of this kind will be responsible for animation, but arbitrary simulation can be carried out.

3 Actors

3.1 Actor architecture

Our actors are object-oriented units composed of properties and behavior. They can be categorized by the way their behavior is implemented.

- **Standard actor**: uses standard (built-in) behavior and does not define any form of additional behavior. It is composed of passive data and does only respond to the standard set of operations that is defined for every actor, such as positioning and picking. Inanimate objects such as furniture are best modeled in that way. This type of actor corresponds to the "artifacts" in Aviary [SW94].
- **Standard actor with static animation**: Like above, but attributed with a static time-driven (i.e. key-framed) animation. This allows to build more interesting "decoration" actors, such as a wall-clock with moving hands.
- **Actors with scripted behavior**: To describe active behavior, static animation is not sufficient. Instead, we use a scripting language to allow the creator of an actor to specify the actors reaction to messages. Two kinds of messages can be distinguished: synchronous messages are sent in regular time-intervals by the server to drive built-in behavior that progresses over time. Asynchronous messages are sent by other actors to provoke a reaction to some event that is happening in the simulation. Built-in animation is mostly predetermined, but it can be influenced by asynchronous messages. E.g., running water may be switched on and off by an asynchronous message, but the water drops are animated by synchronous messages.
- **Actor with external client-control**: If it is known that behavior computation is complex, it should not be carried out on the server for two reasons: (1) scripting is interpreted, so it is flexible but not efficient, (2) an external client may run any kind of software specifically designed to fulfill the task. Both hardware and implementation may be chosen for the particular task at hand. The interface is well-defined by the message protocol between actor and client. The client can for example re-use existing code for FEM, and run on

a high-performance computing server. This sort of diversity cannot be achieved on a single server platform alone.

3.2 Geometry and Animation

Virtual reality applications require interactive graphics display. Current software and hardware rendering engines support polygonal data structures. Animation, such as we desire for visualization of dynamic changes in the virtual environment, requires higher-level control [Zel85], provided by hierarchical scene graphs.

Manipulation of transformations in such a hierarchy allows high-level control of articulated figures [MT85]. We generalize this approach to parameterized models. Attributes are made dependent on expressions of one or more parameters. If the parameter is changed, the attributes' values change accordingly. Parameters can also be made dependent on time, so a self-contained animation can be created, capable of running without any external control. The animation is implicitly created when the time- or parameter-dependent attributes are evaluated (e.g. for rendering).

The hierarchical data structure should also support instancing, so that multiple identical geometry representations do not have to be duplicated needlessly. However, there must be a possibility to distinguish multiple instances of one model. This can be done by using a directed acyclic graph (DAG) as the scene topology.

The parameters mechanism provides a convenient programmer's interface to the data structure: We introduce a naming scheme for scene graph nodes and parameters, and allow manipulation of the data only by standard functions that access the data using this naming scheme.

3.3 Object-oriented actor hierarchy

While 3-D graphics support is a fundamental for interesting virtual environments, actor behavior must be supported to be able to develop meaningful content. Our communication system is message based. If message passing is mapped onto procedure calls (such as in C++), the set of messages understood by each actor is fixed. In particular, an actor will not understand messages that have been created along with the coding of a more recent actor type. A naive message-passing system is not flexible enough to allow new actor types to be added to the system at runtime.

Another problem is to determine what can be understood by a particular actor. Something must be known about the actor to find out whether sending a particular message to that actor makes any sense or not.

Our approach to solve this problem is to construct an object-oriented hierarchy of actor types that can be queried and extended at runtime. A class server answers queries on the class hierarchy, so that the question whether a particular actor understands a particular message can be solved at runtime.

A suitable tool for the construction of the actor class hierarchy is an embedded interpreter for an object-oriented scripting language. The behavior of an actor is encoded in its methods, and these methods can be formulated using the scripting language. By exchanging strings containing method invocation written in the scripting language, the scripting language can be reused as the message protocol. As a side effect, the relative simplicity of such a scripting language makes it a good authoring tool for user-level programming.

Some of the methods will not contain any scripting code, but directly map to the basic functions built into the server. There are certain capabilities that every actor must have and that are therefore implemented in the basic actor class:

- **receive message**: if an actor receives a message, it must be able to carry out the command sent. If it does not understand the message, it may ignore it, or if there is external control in the form of a client connected to the actor, the message is handed on to the client.
- **send message**: sending a message just means passing an arbitrary string containing a statement onto the communication layer, so no restriction whatsoever needs to be imposed on sending messages.
- **tick message**: in regular intervals, the environment itself sends a "tick" message to each actor to drive its internal animation. The actor should then iterate its internal simulation process and act accordingly.
- **modify parameter message:** While arbitrary invocations of the actor's methods can be made, modification of parameters are particularly important. They happen very frequently and are therefore made a primitive message type.

4 Constraints

Communication in our system is **asynchronous**, i.e. if a message is sent, the sender does not wait for an answer. If an answer is required, for example as a result to a request, the receiver may send the answer back in a separate message. This communication scheme is employed so that the sender is not idle waiting for the answer while the message is on the

way. If synchronous communication is necessary, the sender still has the option to explicitly wait for the receiver. While this is efficient, it makes communication choreography difficult to implement. We therefore extend the communication system with a **constraints** mechanism.

Constraints can be defined on any parameter of the actor's data structure. An actor X can instruct another actor Y to watch a constraint on one or more of Y's parameters. If the parameter is modified, the constraint may be violated. Therefore, Y evaluates the constraint (a Boolean expression), and if it is found to be true, Y sends a callback message to X informing it of the new parameter value. X may then take appropriate actions.

With this constraint mechanism it is not only possible to constrain geometric relationships (e.g., keep a minimum distance between two objects), but also any other kind of condition (e.g., keep the bath-tube from overflow). An actor stores a list of all constraints it sets up, and every time it makes an update to a parameter, it checks for constraints defined on that parameter and evaluates those that apply.

5 Groups and worlds

If the receiver of a particular message is known, it may be addressed directly as the recipient of the message. If the same message is to be sent to multiple recipients, it may be more convenient for the user and more efficient for the environment to have a way of sending the message only once, but with multiple receivers. A grouping mechanism is needed because actors need not only to address a single other actor, but groups of actors. For example, if an actor is shouting, everybody in the vicinity can hear it. If the message was sent to all actors in the system (or more specific, all actors on the server), all actors need to examine it to find out whether it is relevant for them, and most of the messages will be unnecessarily sent, consuming bandwidth.

We attack this problem with group actors. Groups can be built on a spatial or functional criterion. If the group is implemented as a special actor, it can run a particular application. Behavior can be attached to the group actor, and it can send the results of the simulation to all group members. For example, if the group runs a roulette game, it can take bets from players, and inform them of the result. All roulette players would then be member in that group.

Groups can be built implicitly, or by explicit joining. For application groups, it may be necessary that an actor has specific capabilities. Our group server can determine if the actor is suitable for joining by the querying the class of the actor. For example, a dance class group may require the member to have feet. Humans, animals and other actors qualify for membership by being subclasses of "mobile being". Actors that do not fulfill this requirement may not join. In that way inconsistent situation are avoided.

Groups can be persistent, or be built on the fly. Sometimes a group may be required just for a single message, such as firing a shoot in a particular direction. Other cases, such as the roulette game, may require the group to exist for a longer time, or without any expiration at all.

6 Actor database management

The database is management is based on the spatial arrangement of the data. Server connections are based on neighborhood relation (every server occupies a box-shaped region in the simulated environment). A server's region is again divided into a regular voxel array (fig. 2). For every voxel, the server keeps track of all contained actors. This is similar to raytracing acceleration schemes like [FTI86].

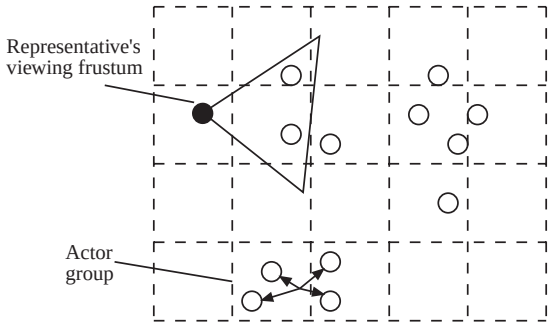

Fig. 2: A grid allows visibility culling and group-based communications

Unlike objects in raytracing, actors change their position over time, so the voxel data structure must be updated if an actor moves over a voxel boundary. The classification is based on bounding boxes for the actor's geometry.

This spatial data structure is used for efficient computation. The first goal is to increase the efficiency of message passing. Messages may be sent to a particular actor, but more often, they are sent to a group. Often group membership is based on presence within a particular region of space, which can be represented as an enumeration of voxels. The message is then delivered to all actors associated with these voxels. The number of actors in this group is much smaller than the total number of actors, so communication bandwidth is preserved.

The second goal is to minimize communication with the client, that expects updates on the visual representation of the environment as it dynamically changes. Based on the camera parameters provided by the client, the server determines the portion of the environment that is visible to the client. Updates need only be transmitted to the client if they lie within the visible range or viewing frustum (compare [Fun95]). Via the voxel array, the server has direct access to those actors that have to be examined for updates.

7 Implementation

We have developed a prototype implementation of the system architecture outlined here. The software is written in C++ under UNIX on SGI workstations. The rendering portion is done in OpenGL and uses the object-oriented modeling toolkit OpenInventor. [SC92]. With this toolkit, efficient modeling of the geometric portion of the actor, including animation and interaction description, is possible. Currently the whole environment is running on one server allowing multiple clients to interact with each other and with autonomous actors.

Each client provides simple mechanisms to move its Avatar through the environment by sending messages to the server. The appropriate portion of geometry lying within the viewing frustum of the Avatar is transported from server to client to update the clients local scene description which is the basis for real-time rendering. Simple animations are carried out by the client independently from the server.

The object-oriented behavior specification of actors is done in Python [Ros93]. Python interfaces to C/C++, so that those portions of the actor's behavioral methods that need efficient processing (like access to its geometric data structure) can be written in C++. For the specification of actor behavior, Python is more than powerful, which makes the construction of applications fast and simple.

Because the actors behavioral description is given in interpreted form (Python source), it can easily be stored on the server side, transported across the network, and reused by other servers. Messages between actors are Python statements. This has two major advantages: Messages can directly be executed as method invocations of the receiving actor, and messages can also be handled as normal data structures (strings), that are easy to manage (logging, network transport etc.). Furthermore, we are able to modify actors and actor classes at runtime. We are also able to transport actors as a whole over a network in their Python form, which is important for actor migration between environments.

The system is still an incomplete prototype that will need refinement. In particular, we will have to experiment with larger numbers of actors and participants to see how well our design scales given more demanding situations.

8 Conclusion

While current software architectures for virtual environments allow interactive immersive multi-user application, they suffer from limitations with respect to performance, persistence of simulation data, fault tolerance and ease of development.

Our system is aimed at the construction of a large-scale distributed simulated environment. Continuous simulation and user-defined simulations will help to move a step toward the vision of a global Cyberspace. A network of servers stores the simulation data base and performs the simulations steps. The data base is distributed using a geometric criterion, every server is responsible for a region of the environment. The items in the data base are autonomous actors that communicate with each other by message passing. An object oriented approach is used to define an actor hierarchy. Actors can be addressed in groups, so that applications make efficient use of the message system.

Users connect to the server network by invoking a client that monitors user action and communicates with the server. The client presents the environment to the user in an appropriate way. It also creates an Avatar for the user inside the virtual environment, thereby allowing the user to interact with other actors and Avatars inside the VE. Future work will involve more efficient communication of graphical data from the server to the client. We will also run tests with a larger number of users, and obtain quantitative data on the performance of the system.

References

[BF93] BENFORD S., FAHLEN L.: A spatial model of interaction in large-scale virtual environments. *3rd European Conference on CSCW*, 109-124, 1993.

[Bre95] BRECHNER E.: Interactive Walkthrough of Large Geometric Databases - Wrap Up and Future Directions. *SIGGRAPH'95 Course Notes #32, 1995.*

[BC94] BRICKEN W., COCO G.: The VEOS Project. *Presence*, Vol. 3, No. 2, 111-129, 1994.

[CH93] CARLSSON C., HAGSAND O.: DIVE- A platform for multi-user virtual environments. *Computers & Graphics*, Vol. 17, No. 6, 663-669, 1993.

[FTI86] FUJIMOTO A., TANAKA T., IWATA W.: ARTS: Accelerated Ray-Tracing System. *Computer Graphics and Applications*, Vol. 6, No. 4, 16-26, 1986.

[Fun95] FUNKHOUSER T.: RING - A Client-Server System for Multi-User Virtual Environments. *SIGGRAPH Symposium on Interactive 3D Graphics*, 85-92, 1995.

[HMR] HONDA Y., MATSUDA K., REKIMOTO J., LEA R.: Virtual Society: Extending the WWW to support a Multiuser Interactive Shared 3D Environment. To appear in *Proc. of VRML'95* Symposium, San Diego, 1995.

[MZP95] MACEDONIA M., ZYDA M., PRATT D., BRUTZMAN D., BARHAM P.: Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. *Proceedings of VRAIS'95*, 1995.

[MT85] MAGNENAT-THALMANN N., THALMANN D.: *Computer Animation - Theory and Practice*., 1985.

[PBC94] PAUSCH R., BURNETTE T., CONWAY M., DELINE R., GOSSWEILER R.: Alice: A Rapid Prototyping System For Virtual Reality. *SIGGRAPH Course*, No. 2, 1994.

[Ros93] VAN ROSSUM G.: An Introduction to Python for UNIX/C Programmers. *Proceedings of NLUUG - Dutch Unix User Group Conference*, 1993.

[SG95] SCHMALSTIEG D., GERVAUTZ M.: On System Architectures for Virtual Environments. Proceedings of the 11th *Spring Conference on Computer Graphics*, Bratislava, Slovakia, 1995.

[Sha93] SHAW C., GREEN M., LIANG J., SUN Y.: Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, Vol. 11, No. 3, 287-317, 1993.

[Sin94] SINGH G., SERRA L., PNG W., NG H.: BrickNet: A Software Toolkit for Network-Based Virtual Worlds. *Presence*, Vol. 3, No. 1, 19-34, 1994.

[Sin95] SINGHAL S., CHERITON D.: Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality. *Presence*, Vol. 4, No. 2, 169-194, 1995.

[SW94] SNOWDON D., WEST A.: AVIARY: Design Issues for Future Large-Scale Virtual Environments. *Presence*, Vol. 3, No. 4, 288-308, 1994.

[SC92] STRAUSS P., CAREY R.: An Object Oriented 3D Graphics Toolkit. *Proceedings of SIGGRAPH'92*, No. 2, 341, 1992.

[WGS95] WANG Q., GREEN M., SHAW C.: EM - An Environment Manager for Building Networked Virtual Environments. *Proceedings of VRAIS'95*, 1995.

[Zel85] ZELTZER D.: Towards an Integrated View of 3-D Computer Animation. *The Visual Computer*, Vol. 1, No. 4, 249-259, 1985.