Utilizing Object-Oriented Databases for Concurrency Control in Virtual Environments

Damla Turgut, Nevin Aydin, Ramez Elmasri and Begumhan Turgut
Department of Computer Science and Engineering
The University of Texas at Arlington
P.O. Box 19015, Arlington, TX 76019-0015
E-mail: {turgut, aydin, elmasri, bturgut}@cse.uta.edu

Abstract

Virtual Reality Modeling Language (VRML) is widely used to represent, create, and display virtual reality objects and their environment. Some VRML applications require concurrent interaction by multiple users in a real-time distributed fashion. Such applications need a method for users to share and update the VRML objects in real-time. To allow concurrent shared real-time access, our approach is to store the VRML objects in an object-oriented database system (ObjectStore) in order to utilize the concurrency control mechanisms of the system. In this paper, we present an architecture that allows multiple users to interact in a non-trivial way in such a shared VRML environment. We outline how the VRML world can be saved in ObjectStore and implement a series of test cases demonstrating concurrency issues arising from simultaneous updates. Our architecture uses ordinary Java enabled web browsers with a VRML plug-in. A commercial web server routes client requests to a custom application server, which interacts with the object-oriented database. As users change the VRML world, our application server orders the requests and updates the master copy in the database.

Keywords: Object-Oriented Databases, Concurrency Control, Virtual Environments.

1. Introduction

Complex virtual reality applications are increasingly using VRML (Virtual Reality Modeling Language) to model the objects and their interactions [11, 3]. When dealing with VRML worlds and objects we often find the need to retrieve, manipulate and store the states of a VRML object as it changes over time. This is called VRML persistence and is gaining importance among VRML applications. As VRML applications become more complicated and process real-time data, the need for adequate persistence [9] capabilities increases. VRML, Java, and object-oriented databases are relatively new fields rapidly gaining importance and popularity. However, developers have not yet produced a comprehensive solution to VRML persistence.

The rest of the paper is organized as follows. We describe the traditional MAVE (Multi-agent Architecture for Virtual Persistence) architecture [1], introduce concepts from VRML and give an application where VRML persistence is useful in section 2. Section 3 gives an overview of PSE (Persistent Storage Engine) [8] for the ObjectStore [7] ODBS, which was used in our system to provide persistence. Section 4 illustrates five tests to determine how concurrency control in ObjectStore can support distributed VRML applications. Section 5 defines the integration of VRML with Java and introduces the EAI (External Authoring Interface), which offers a generalized method of accessing VRML nodes. Conclusions are drawn in section 6 by showing how to achieve VRML persistence within the MAVE architecture and future work.

```
Cone {
        field  SFFLOAT  bottomRadius  1
        field  SFFLOAT  height        2
        field  SFFLOAT  side          TRUE
        field  SFFLOAT  bottom        TRUE
}
```

Figure 1. Example Definition of a Node

2. VRML and MAVE Overview

VRML (Virtual Reality Modeling Language) [3] integrates 3D graphics, 2D graphics, text and multimedia into a coherent model. VRML allows us to use a computer generated 3D virtual environment which provides an intuitive interface to complex information. In VRML, a scene graph is a group of objects describing the structure of the virtual world that is being created. The primitive object types represent boxes, cones, cylinders, and spheres. The scene graph will be used to develop an EER (Extended Entity Relationship) model [2], which we will use to design the database for storing VRML objects in ObjectStore [7]. A VRML node is analogous to a structure definition in a high-level language. Figure 1 shows a node for "cone" which has four fields. Our database must store the names of the nodes, the names of the fields and their values, and must also store information on the shape, geometry, material, and appearance of the nodes.

Let us consider data from a geographic study. Suppose a forest fire has burned part of an area that is being surveyed. A persistence model should be able to store information on the location and extent of the fire. When we study the region again, we should record new geographic information, indicating which area was damaged by the fire. An alternate application domain, which needs persistence, is a multi user dynamic environment. Our persistent model makes it convenient for the user to record the current state of the world for future processing.

The objective of MAVE (Multi-agent Architecture for Virtual Persistence) [1] is to develop an agent-based architecture to support intelligent, reusable, distributed virtual worlds. MAVE is a two-tier architecture. The first tier is an object-oriented physical representation of the virtual environment that is designed to mimic the logical decomposition of the virtual world. The second tier is designed to support the needs for persistence, real-time interfaces to external data sources, distribution, and collaboration. Many of the virtual environment architectures employ a VRML representation of their content and provide a forum for multiple distributed users to congregate, communicate, and access the same type of information generally associated with web pages [10]. MAVE expands these capabilities by providing an architecture that can support the use of multi-user distributed virtual environments as advanced interfaces to distributed heterogeneous computer systems and databases. The object level representation is shown in Figure 2.

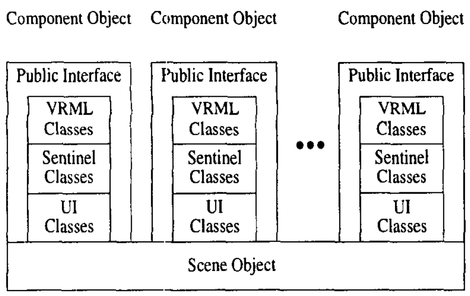

Figure 2. Object-level Architecture

The virtual environment provides a framework in which distributed users can collaborate and share resources, including a variety of multimedia types. In the object level architecture, VRML is used as one element of the overall virtual environment. To understand MAVE, it becomes important to discuss the hierarchical nature of a virtual environment. A virtual environment is composed of scenes and each scene is composed of objects. In MAVE, the VEC (Virtual Environment Component) architecture maintains a physical granularity that mimics the logical decomposition of the respective elements of the virtual environment. VEC is the lowest level element in a virtual environment that can stand alone as a useful entity. Each VEC in the virtual environment has a corresponding VRML visualization. The VEC architecture internalizes a programmatic representation of the VRML and then creates the visualization display. In response to a stimulus that would change the VRML visualization, the programmatic structure is changed first and then the visualization display is updated.

The system level architecture of MAVE is designed around an object-oriented database component. This component will augment the object-oriented structure of the virtual environment by providing a repository for persistent VECs. An object-oriented database is a natural choice for the MAVE architecture due to problems related to storing complex objects in a standard relational database. The system level architecture is illustrated in Figure 3.

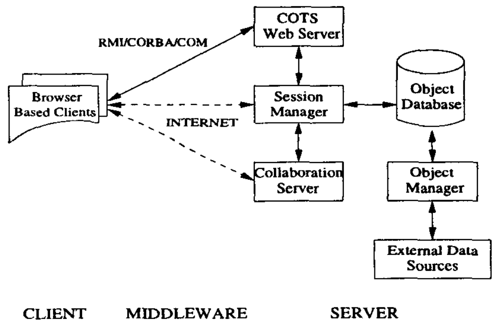

Figure 3. System-level Architecture

The client runs a traditional Web browser with a VRML viewer. We used the Netscape Plug-in Cosmo Player to execute VRML scenes described in this paper. The VRML data originally comes from a COTS (Commercial Off The Shelf) Server. Shared VECs are managed and updated by the session manager. Persistent VECs are stored in the object-oriented database and one or more object managers allow other programs or events to change the data. The object-oriented database will inform the session manager when any in-use objects are changed and the session manager can forward the new objects to the client in order to update the visualization display.

3. Introduction to PSE and ObjectStore

Many issues arise when considering how to add persistence to VRML. A VRML persistence scheme should distinguish between static VECs and dynamic VECs, and must consider concurrency issues during multiple simultaneous accesses. It should also make efficient use of the underlying storage management system.

Previous work on VRML persistence has used traditional file storage techniques. Traditional VR (Virtual Reality) applications have not required the power of object-oriented databases. Instead, they store just a static VEC state in a file. Recent VR applications needed more complicated and dynamic VECs to implement their environments. Limitations of static file storage include: no decision logic, no user defined nodes, and no external access. One way to allow these complexities is to use an object-oriented database. Object-oriented databases provide direct support for persistence of objects. Three object-oriented databases we considered for the MAVE project are ObjectStore [7], PSE (Persistent Storage Engine) [8], and PSE Pro. PSE is originally intended for small single user databases, and is not intended to support high volumes of updates or queries over a large collection of objects. PSE Pro does support large databases. ObjectStore provides object storage for both Java [4] and C++ objects, and supports very large databases and multiple users concurrently accessing multiple databases.

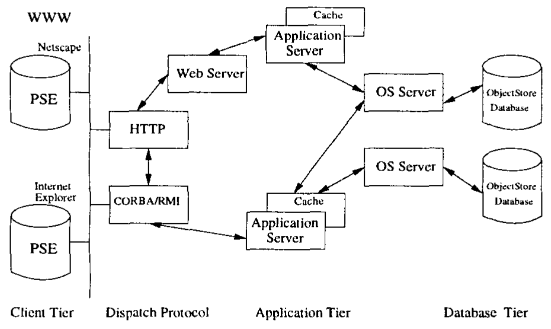

Figure 4. Initial Database Server Architecture

MAVE must support multiple users with concurrent accesses. The clients must always have the most recent state of the object being used and VECs must be stored and accessed efficiently. Figure 4 shows our initial database server architecture. The client tier uses an ordinary web browser with a VRML plug-in. In addition, each client must run a copy of PSE to locally cache objects relevant to the local visualization display. The PSE API is a subset of the Object Store API. Thus, ObjectStore server can freely use PSE features and functions on the client. The VECs are downloaded from the ObjectStore server to the PSE. When the client wants to commit changes to the VECs, the modified objects are saved from PSE to ObjectStore. This approach will reduce network traffic and increase performance. The protocol tier is responsible for passing requests and responses among clients and applications. The actual protocol chosen could be CORBA, RMI, Serialization, or Sockets. Our full architecture will use Netscape Enterprise server as a protocol tier; for the tests in this paper, however, we used custom socket protocols. The application tier provides a buffer between the client requests and the database. MAVE applications often have many users accessing the same database through the same web server. To prevent the web server from becoming a bottleneck, we introduced application servers. The web server can forward client requests to any number of application servers. The database tier is responsible for ensuring that all application components share access to distributed data. The MAVE architecture allows us to decouple the GUI logic and the persistence storage logic.

4. Demonstration of Concurrency Cases

The MAVE database server architecture must support several features: multiple concurrent access, browser (Web) based clients, notification by the database of any changes in data, locking and synchronization, and multi-threading support. We used five different tests to evaluate ObjectStore's ability to meet these requirements.

4.1. Terminology Used

Before the test cases are presented, the definitions from common ObjectStore and Java terms are described [4, 7, 8].

Java VM: One feature of Java is that the compiler targets the Java Virtual Machine (JVM) rather than a specific hardware platform. Thus, the Java byte code can be executed on any Java compatible platform without recompilation. A user can have multiple Java processes by invoking multiple instances of a JVM. Unfortunately, each JVM consumes 10 Mb of memory.

Session: A session is a distinct view of ObjectStore. A collection of persistent objects plus a portion of a database form a session. Two or more independent transactions can be placed in separate sessions to facilitate concurrency. PSE Pro allows multiple sessions in a single JVM.

Transactions: A transaction is the atomic number of steps needed to change the data in the database from one consistent state to another. A session can have only one active transaction. Transactions in different sessions can run concurrently, but they must access separate parts of the database. Databases offer different types of locking (database level, page level, object level), which affects the behavior of transactions. In some cases, transactions can deadlock, which forces one transaction to abort.

Concurrency and Multi-Version Concurrency Control (MVCC): Multi-Version Concurrency Control allows read-only transactions, which access the same portion of the database to execute concurrently. In addition, an update transaction can execute without blocking any of the readers. When an application accesses a database opened in MVCC mode, transactions from that application will never deadlock. For MAVE, we often have many users making concurrent transactions. MVCC is not appropriate for this use because more than one user may be making an update. Therefore, MAVE opens the database in update mode and uses concurrency control for read and update accesses.

Multi-Threading: Multi-Threading offers a better alternative than multiple processes. Each process (JVM) consumes approximately 10 Mb of memory; multi-threading allows us to have more than one thread of execution in a single Java process. For example, one thread could listen for external events while other threads update the different portions of the database. ObjectStore allows a single session to have several Java threads; each thread can belong to a single session at a time. All threads in the same session cooperate with each other. It is the developer's responsibility to prevent threads in the same session from illegally concurrently updating the same object. MAVE creates one thread per user, which are all part of the same session. Each thread handles the need of a single MAVE user's needs.

Event Notification: Event Notification facilitates multiple clients accessing the same portion of the same database. A thread, which updates a database, also sends a notification. Other threads that are reading the database receive the notification and thus always have a current view of the database.

4.2. ObjectStore Demonstration

We conducted five tests on ObjectStore each of which uses an application called persistent counter. The program manages a "hit counter" which can be updated by multiple users. The database is simple; there is just one persistent object-the integer counter.

Demonstration 1: Concurrency control using ObjectStore

Two transactions which update the same object must not run simultaneously. ObjectStore must lock out one of the transactions while the other completes. When the first action is committed, ObjectStore should release the lock and allow the second transaction to continue processing. For our example, this means the counter should be implemented twice. Figure 5 shows this situation. We executed our example in a Windows machine using Netscape's JVM. The execution resulted in a Transition In Progress Exception error. The error is caused by having multiple threads in a single session, which are not allowed unless the programmer adds explicit synchronization instructions. This is because transactions to the same portion of the database must be in separate processes (JVMs) in the version of the ObjectStore we used. However, multiple JVM for client requests can degrade the performance of the application server.

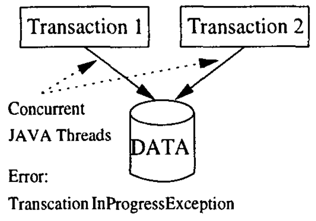

Figure 5. Demonstration 1

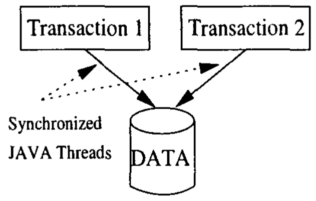

Figure 6. Demonstration 2

Demonstration 2: Concurrency control using Java synchronization

Using Java synchronization techniques, we can ensure that only one thread at a time from a JVM can modify the database. That is, the client rather than the database handles synchronization. Figure 6 shows the situation. This demonstration correctly updates the counter twice. By adding synchronization code, the developer ensures that threads in the same session maintain consistent execution. Unfortunately, this places the burden on the developer to manage thread blocking, thread synchronization, and access priority.

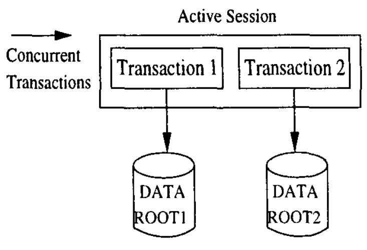

Figure 7. Demonstration 3

Demonstration 3: Concurrency control for different parts of the same database

ObjectStore offers different kinds of locking mechanisms. By switching from database level locking to page level locking, we can allow concurrent updates to different parts of the same database. We executed this demonstration and found that Java synchronization is required even though the threads were accessing different parts of the database. We repeated the demonstration using PSE Pro and found that PSE Pro does support multiple transactions; two counter values are correctly updated even in the absence of Java synchronization instructions as seen in Figure 7.

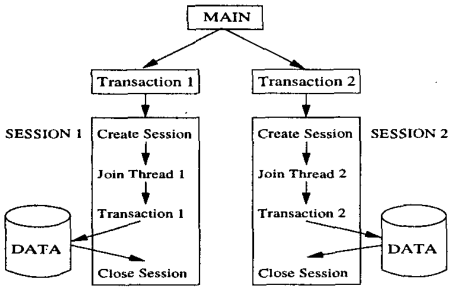

Figure 8. Demonstration 4

Demonstration 4: Concurrency control with separate sessions

We spawned two threads, which join separate sessions. Concurrent transactions from the sessions should be synchronized by the database. This demonstration represents our preferred organization for MAVE applications. Figure 8 shows this organization. We executed this demonstration using PSE Pro as the database engine. PSE Pro successfully handled concurrency control. Thus, PSE Pro supports our preferred architecture from MAVE.

Demonstration 5: Concurrency control with multiple clients

MAVE is designed to support multiple accesses by separate clients. Since the version of ObjectStore we used supported only a single session, we needed to add code to the server to synchronize requests from multiple clients. We simulated multiple clients by spawning several Java threads that communicate with the synchronization thread. The synchronization thread is the only thread that communicates directly with the ObjectStore database. In an actual MAVE application, the server would have a single synchronization thread and one thread per client to handle client requests. The synchronization thread manages concurrency issues for incoming requests. A series of read requests can be handled concurrently. The synchronization thread must synchronize a series of write requests. Writes to the same segment of the database should execute serially one at a time. Writes to different segments of the database could be handled concurrently, but the version of ObjectStore we used allowed only one transaction per session. Therefore, our current synchronization thread serializes all thread requests. For a combination of read and write requests, our synchronization thread executes update threads before executing any read threads.

5. Integrating VRML with Java

An important part of the MAVE architecture is the ability to transfer information between VRML and Java [5, 6]. To implement dynamic behavior in VRML worlds, we need the ability to query the state of the VRML world, to make decisions based on the state, and change the VRML world appropriately. In this section, we consider two methods of communication: *Scripting*, and *External Authoring Interface* (EAI). In many cases, either technique can be used to implement the desired behavior.

Script nodes can bridge the gap between VRML and Java. The fields of a script node are user extendable and events arriving at the script nodes are directed to an external application. Script nodes are ideal for handling events internal to the VRML world. The events and the fields of the script nodes must be defined in advance.

In contrast, the EAI offers a generalized method of altering and accessing nodes and events of the VRML world. EAI is best suited for integrated multi-media presentations, which include VRML as one media type. The EAI is implemented through a web browser plug-in and allows arbitrary dynamic changes. A VRML browser window embedded in a web page can be controlled from a Java applet on the same page. We implemented an example using EAI to allow the user to change the color of a VRML sphere as depicted in Figure 9. Our example offers a simple demonstration of using EAI to manipulate persistent VRML data.

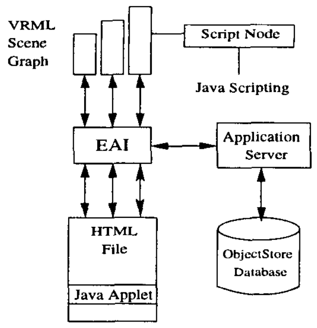

Figure 9. Sphere Example

In our example, the client sends its position to the server and the server changes the position of a scene object. Specifically, we implement the road mirage example. This example models the experience of a user driving down a highway on a hot day. The heat waves rising from the road ahead offers the illusion of water. We implement this illusion by moving the mirage as the user approaches it. On the server side, we have an application server that the clients connect to. The application server talks to ObjectStore and stores the position and orientation of the client. The client side has two programs: the VRML scene and the Java program. The VRML scene has two nodes: a sphere representing a mirage and a proximity sensor. The Java client connects to the server and wakes the proximity sensor to send the data via script node. The client sends this data to the server which calculates a new position for the mirage sphere. The client updates the sphere's position based on information from the server.

Our next example simulates a multi-user system. We implement a world with presence. *Presence* means that each VRML viewer is represented as a scene entity. Thus, users are aware of each other's movements through the scene. The VRML object which represents a user is called an avatar. The MuClient script node holds information about orientation and position of the user, and communicates the user's position to the server. Each client also has a MuReceiver class, which provides information on the position and orientation of the other users. The MuServer class organizes clients and creates one thread per connected user. This thread is called MuDispatcher and it will receive position information from its client and forward this information to other clients. MuDispatcher is also responsible for updating the ObjectStore database as shown in Figure 10.

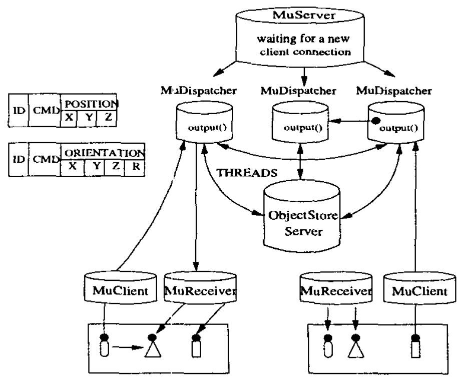

Figure 10. Multi-user Example

6. Conclusions and Future Work

This paper has incorporated VRML persistence into the MAVE architecture. This offered an initial description of the use of ObjectStore for persistent storage of VRML objects. We have implemented several demonstrations of the interactions between Java, VRML, and the object-oriented database. Our final architecture will include a web server, which dispatches transactions to one or more application servers. The application servers route client request to a synchronization thread. The synchronization thread orders requests based on the concurrency control capabilities of our database engine. A database stores persistent VRML objects. Using event notification features, all clients using a VRML object will learn of changes to that object in real-time, in order to refresh their visualization display. Future work includes storing an array of VRML scenes using event notification features and optimizing consistent access to the database.

Acknowledgements

We would like to thank Vivek Vijayaraghavan for his work during the implementation phase of this paper.

References

[1] J. Coble and K. Harbison, "MAVE: A Multi-agent Architecture for Virtual Environments" *Proceedings of 11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, June 1998.

[2] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 3rd Ed., 2000, Benjamin-Cummings.

[3] J. Hartman and J. Wernecke, *The VRML 2.0 Handbook*, Silicon Graphics Inc, August 1996.

[4] Java Programming with ObjectStore, *Object Design Inc*, www.odi.com, 1998.

[5] R. Lea, K. Matsuda and K. Miyashita, *Java for 3D and VRML Worlds*, New Ride Publishing, 1996.

[6] M. McCarthy and A. Carty, *Building 3D Worlds in Java and VRML*, 1st Ed., Prentice Hall, 1998.

[7] ObjectStore User Reference, *Object Design Inc*, www.odi.com, 1998.

[8] OSJI and PSE Java discussion lists, *Object Design Inc*, www.odi.com, majordomo@odi.com, 1998.

[9] R. Sessions, *Object Persistence: Beyond Object Oriented Databases*, 1st Ed., Prentice Hall, 1996.

[10] Singhal, S. and Zyda, M. *Networked Virtual Environments: Design and Implementation*, Addison-Wesley, 1999.

[11] VRML, VRML-EAI, VRML-dbwork discussion lists, maintained by VRML Group, www.vrml.org, majordomo@vrml.org, 1998.