Synchronous collaborative systems for distributed virtual environments in Java

Kevin Gorman, Daneyand Singley and Yuichi Motai*

Department of Electrical and Computer Engineering,
University of Vermont, Burlington, VT 05405, USA
E-mail: kgorman@cems.uvm.edu
E-mail: dsingley@zoo.uvm.edu
E-mail: ymotai@uvm.edu
*Corresponding author

Abstract: This paper presents a 3D-distributed virtual environment, which allows for the creation of applications to enable multiple users to collaboratively interact in, and communicate about, a virtual world. This virtual environment is used to primarily support a flight simulator application by providing communications routines and networking synchronisation between the client and the server. Four server–client synchronisation methods were developed and compared. The method of transparent synchronisation with non-blocking I/O has been found to be the best method for maintaining synchronisation among distributed virtual environments using Java. The successful creation of the distributed virtual environment and the positive results collected in the supporting experimental data, leads to the conclusion that Java, with the addition of the non-blocking I/O Java.NIO.channels API and the Java 3D API, can be successfully used to create high-performance distributed 3D-virtual environments.

Keywords: Java; non-blocking I/O; networking; synchronisation; virtual reality; distributed virtual environments.

Reference to this paper should be made as follows: Gorman, K., Singley, D. and Motai, Y. (XXXX) 'Synchronous collaborative systems for distributed virtual environments in Java', *Int. J. Computer Applications in Technology*, Vol. X, No. Y, pp.XXX–XXX.

Biographical notes: Kevin Gorman received his BA from Fairfield University, Fairfield, a and CT and a BS in Computer Engineering from Columbia University, New York, NY in 2000. He is currently pursuing an MS in Electrical Engineering at the University of Vermont, Burlington, VT. His primary interests lie in the design and verification of high speed digital circuits, design for test and built-in self test and repair of memory arrays.

Daneyand Singley received a BS in Computer Engineering from Rutgers University, New Brunswick, NJ in 2001. He is currently pursuing an MS in Electrical Engineering at the University of Vermont, Burlington, VT. His primary interests include high-performance microprocessor logic design, verification and processor architecture as well as Java/C performance programming and 3D-graphics architectures.

Yuichi Motai received a BEng in Instrumentation Engineering from Keio University in 1991, an MEng in Applied Systems Science from Kyoto University in 1993, and a PhD in Electrical and Computer Engineering from Purdue University in 2002. He is currently an Assistant Professor in Electrical and Computer Engineering Department at the University of Vermont. His research interests are computer vision, human–computer interaction, ubiquitous computing, sensor-based robotics and mixed reality.

1 Introduction

Distributed virtual environments enable geographically diverse users to communicate about and collaborate on various aspects of a virtual world as described by Roussou (2004). A distributed 3D-virtual environment can find many applications and an environment that is cost effective, portable, maintainable and extensible would be of particular use in both entertainment and academia as demonstrated by Burdea and Coiffet (2003), Campbell et al. (2002) and Hughston et al. (2003). However, for applications to be successful in either of these fields they need to be immersive and interactive and as such should be founded on a distributed virtual environment infrastructure that provides extensive graphic, control, communication and networking support as detailed in Bricken (1991), Burdea and Coiffet (2003) and Singhal and Zyda (1998). Construction of such virtual environments with these features is a non-trivial task.

Especially necessary for the success of such a distributed virtual environment is excellent networking support, which is essential for enabling cooperation and/or

collaboration between various participants. Many different models for building a distributed virtual environment with strong performance across a network has been presented (see Campbell et al., 2002; Cook et al., 2000; DeFanti et al., 1997; Han et al., 2002; Hughston et al., 2003). A particular difficulty in dealing with network communication between distributed virtual environments is maintaining synchronisation between the server and the client applications as shown by Benford et al. (1999), Cai et al. (2004), Cronin et al. (2002), Ishibashi and Tasaka (2003) and Lau et al. (2004). Users must have a consistent worldview and their actions must have concrete effects on their environment with minimal lag. Lau et al. (2004) further maintained that multiuser applications that do not provide for relatively consistent and synchronised worldviews run the risk of losing their interactivity and immersion. It would be highly desirable to utilise a method and group of programming tools that, along with the attributes listed above, can also facilitate the creation of a distributed 3D-virtual environment that provides good network performance. Java[1] and Java 3D were chosen as the programming tools for creating this environment for a variety of reasons. Cox and Novobilski (1991) argued that their object-oriented nature allows for more easily extensible and maintainable programs. The tools are freely available and a large open source movement exists to support the tools, reducing the overall cost of the project and accelerating the development schedule. In fact, most of the initial graphical engine (Java Flight Simulator, 2005) was obtained via open source code, reducing much of the earlier works to simply handling the integration of the graphical engine with basic communications and networking frameworks. The largest amount of effort was devoted to expanding these frameworks into robust elements that would allow for a full-featured networked virtual environment to be developed. In addition to these benefits of using Java, Java is being taught as a first programming language in many academic settings (see Stephenson and West, 1998), which means that the code base will be understood and easily usable and modifiable by a large percentage of researchers and/or game developers, extending the usefulness of the virtual environment. Finally, the fact that programs written in Java and Java 3D can be easily compiled and run on a variety of computing platforms, allows the virtual environment's applications to be ported to a number of systems (see Source for Java Technology, 2005).

However, Java initially lacked support for non-blocking I/O and 3D graphics. Since networking and graphics are possibly the most important aspect of a distributed 3D-virtual environment, Java's deficiencies in network support and 3D-graphics had somewhat limited its usefulness in this area. This difficulty was ameliorated with the availability of Java (SDK 1.4.1) and the new Java.NIO.channels API, which provides support for non-blocking I/O, along with the Java 3D API (version 1.3.1) (see Source for Java Technology, 2005). The Java programming tools, together with the VRML97 modelling standard (see VRML97 Standard, 2005), allow a 3D-virtual environment to be created in a cost effective manner that can support easy maintenance and extension across a variety of computing platforms. The current release of Java, when combined with these key APIs, now appears to provide everything necessary to support the creation of a high performance distributed 3D-virtual environment. A virtual environment was created to explore and possibly confirm, Java's suitability to the task.

Described in detail in this paper is a distributed virtual environment, complete with a communications package and robust networking routines, which supports many easily modifiable applications. The virtual environment, dubbed Whiskey, is designed to primarily support an application that can be used in an academic setting to allow for the collaborative exploration of Digital Elevation Maps (DEM), (see DEM Standard, 2005), through a flight simulator-like interface. This particular application of the Whiskey environment allows for a number of users to simultaneously investigate geographic phenomena in an open-ended learning environment, communicates about their impressions and adapts their viewpoint and orientation in response to other users' inputs in real time. Alternatively, via the same application, a group of users can be provided with a guided tour of a DEM to obtain education about a particular geographic area for a more directed educational experience. A possibly more entertaining use of the virtual environment would be a modification to the application that allowed various users to compete in aerial dogfights above the realistic landscape provided by the DEM. The foundation the Whiskey distributed virtual environment provides is sufficient to easily enable both of these applications together with a wide range of other uses.

The aspect of the primary application of the Whiskey environment that is described in detail in this paper will be the use of the virtual environment to provide a guided tour of a DEM. The networking issues related to using Java for creating distributed virtual environments will be examined. The communication and networking support necessary for creating an environment that can enable this application will be the main focus of the description of the design of the distributed virtual environment. The performance of the various methods used for networking synchronisation will be compared qualitatively and quantitatively. Finally, conclusions will be drawn on the current suitability of Java to the creation of distributed virtual environments and the best methods for maintaining client–server synchronisation in a Java-based environment will be recommended.

2 Java and distributed virtual environments

Java has been used to provide support for the creation of distributed 3D-virtual environments in the past (such as the effort made by Campbell et al. (2002)), synchronous collaborative environments like the Java Application Sharing in Multi-user Interactive Environment (JASMINE) has been outlined (see Georganas et al., 2003; Shirmohammadi and Georganas, 2001; Shirmohammadi et al., 2003) and the Java Adaptive Dynamic Environment (JADE) has even been proposed to provide support for

dynamic extensibility for virtual environments in run-time (see Crowcroft et al., 2000, for details). Today, application developers, especially those concerned with entertainment-oriented distributed virtual environments, readily appreciate the many benefits of using Java. Java provides many standard extensions and APIs that enhance developer's abilities to create graphics, control and communication routines for supporting their applications (as described by Goldberg et al. (2004)).

However, there have been issues with using Java to support distributed virtual environments related to Java's networking abilities. Java originally did not support non-blocking I/O. An attempt to recreate a virtual environment in Java that was originally created using C++ highlighted this issue. The best solution found to provide networking support was to implement non-blocking and polled socket I/O in C++ and to simply provide Java with a channel class to communicate with the C++ networking support code. The difficulties encountered during this conversion attempt directly led to the conclusion that networking services that are only capable of blocking calls are not sufficient in complex real-time applications as described in detail by Bangay (2001).

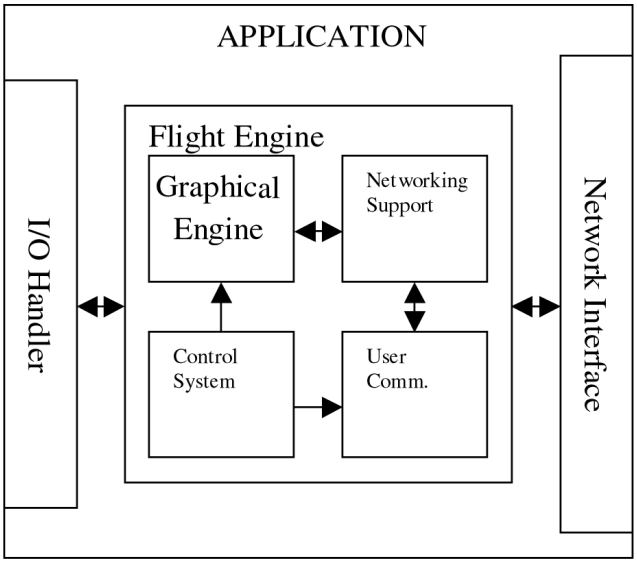
The Java networking support issues led to the development of custom non-blocking I/O APIs which seemed to address many of the obvious deficiencies with the original Java implementation (see Brewer et al., 2001). The official Java standard has since been updated to provide support for non-blocking I/O starting with the Java 1.4 release. However, limited work has been done to test and compare and contrast threaded (blocking) networking services with the new non-blocking I/O APIs.

3 Design of the Whiskey virtual environment in Java

The Whiskey environment consists primarily of a 'Flight Engine' which can be broken into four main pieces: 'Graphical Engine', 'Networking Support', 'Control System' and 'User Communications'. Thread-based client–server interaction is used for the communication system and is not discussed in detail since the implementation is fairly straightforward and is not dependent on any newer Java features or APIs. However, networking support for the virtual environment, which is much more performance critical, was explored via diverse thread and non-blocking I/O-based methods for network socket access, together with multiple client–server synchronisation techniques to determine whether the new Java.NIO.channels API makes a significant difference in the implementation of networking support.

The interactions between the main pieces of the Whiskey environment are given in Figure 1. An application is depicted, which could run in either client or server mode. The application contains the Flight Engine with the Graphical Engine, Networking Support, Control System and User Communications routines. The flight engine communicates with 'I/O Handler' subroutines, which handle the interfaces to the input and output devices, while the 'Networking Interface' consists of the subroutines necessary to handle the marshalling/ unmarshalling of data and the interactions with the socket itself. The following subsections describe in detail the implementation and significant challenges associated with each of the main aspects of the Whiskey distributed 3D-virtual environment, with particular focus on the Flight Engine components and the Networking Interface.

Figure 1 Architecture of an application of the Whiskey virtual environment



3.1 Graphical engine

The graphical engine is constructed in Java 3D. The example application for the exploration of DEMs, currently imports 1^0 DEMs to provide the geographical landscape that can be examined via the application. A user can select from a variety of VRML97 airplane models, which will serve as the user's avatar in the virtual environment. The avatar is useful because it provides orientation and movement cues for the user in a form that is familiar to most people and requires little explanation. To aid in navigation around the geographical model a simple box is used to represent the atmospheric boundaries of the simulation world along with navigation posts that are placed at the corners of the DEM. No physics model is currently included, but the object-oriented nature of the tools and the modularity of the environment would allow for the easy addition of such a feature. A single light source is used to represent the sun with a fixed location. The 3D-environment is displayed in a resizeable window that offers a simple GUI for selecting avatars/DEMs via the 'Options' button, displaying help information (including avatar controls) via the 'Help' button, enabling networking features via the 'Client Connect' and 'Start Server' buttons, activating communication features via the 'ipKchat' and 'ipKtalk' buttons, and reporting on avatar movement speed and location via the 'Speed' and 'Azimuth'/'Altitude' fields (see Figure 2).

Figure 2 Application screenshot showing GUI

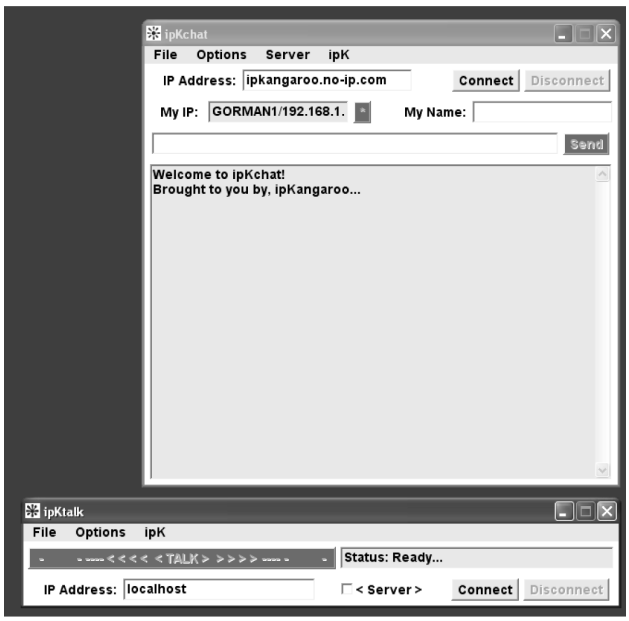
3.2 Control system

The control system allows the users to drive the simulation with the mouse, the keyboard or a combination of the two devices. Many specialised devices such as 3D-mice, data gloves and other tracking devices support keyboard or mouse emulation modes, which allow for the simple integration of new devices into the environment. The control system has a straightforward construction that could be easily modified to take advantage of the Java interface APIs to allow for direct integration of drivers that support all the features of 3D-input devices. The current mouse/keyboard-based interface provides simple pitch/roll/yaw controls along with a method for increasing or decreasing forward speed. The controls have a built in damping effect such that the motion of the avatar due to pitch/roll/yaw will decrease over time until the avatar's orientation has stabilised. This was determined to be necessary because of the propensity for many of the application's users to overcompensate for motion in the 3D-environment until they become familiar with the controls. Finally, a method for stopping all avatar motion and a reset to initial orientation/position are included to prevent the user from becoming lost in the virtual world due to disorientation.

3.3 User communications

The user communications system is activated via buttons on the main GUI. The user can launch either a text-messaging window that runs the ipKchat routine or a voice-based discussion window that runs the ipKtalk routine. From the newly launched window, the user connects their client system to the server and then begins communicating. A user may engage in a text messaging-based discussion with all other simulation users if desired or they can limit their discussion to a particular person or group of persons. A user may also engage in a discussion via voice with another user provided their local client machines are supplied with the requisite speakers and microphones. The GUIs used to support the user communication routines are depicted in Figure 3.

Figure 3 ipKtalk and ipKchat GUI's



3.3.1 ipKtalk routine

This routine supports walkie–talkie type communication over the network. The audio format used is 8 bit, 8 kHzA, mono, signed Pulse Code Modulation (PCM). Two machines are needed to support this communication method: a client that initiates the connection and a server that waits for the connection. Once a connection is made between the client and server a user can press the 'Talk' button to record a voice note. Pressing the 'Talk' button again will first zip the audio stream and then send it to the other party for immediate playback. There is an audio stream playback buffer running on both the client and the server to cache incoming audio streams for playback. A 'Compatibility Mode' can be enabled via the 'Options' menu. This mode addresses audio errors that occur on certain hardware configurations. Finally, a 'Reset Line' action can be chosen from the 'Options' menu. This action reinitialises all classes and resets the server/client connection if a serious error is detected due to audio encoding, a zip compression malfunction or any other extraneous errors. The 'Reset Line' action was added during early development of the ipKtalk routine using Java 1.3. Java audio support seems to have become more robust with later releases, but the 'Reset Line' action has been left active in the event that a user experiences any issues.

3.3.2 ipKchat routine

This routine supports text-based chat communication over the network. At minimum, two machines are needed to support this communication method: a client that initiates the connection and a server that waits for the connection. Once a connection is made between the client and server a simple GUI, along with text base commands, is used to issue instructions to the server. The number of client connections to a particular server is only limited by the server system's processing capacity. When a client connects to the server, a thread is spawned on the server to

listen for and communicate to the client. Each thread associated with a particular client contains the client's socket, the client's name, the time when the connection was established and either the chat room or client with which the user is interacting. The threads are tracked by the unique client name they are associated within the server system. The client name corresponds to the user invoking that particular client connection and the user name is how that client's user is identified to other users. All communication threads are actively managed and optimised for performance via general garbage collection and the closing of terminated connections. The chat application supports both general chat rooms, in which any number of connected users can talk to each other, along with private chat rooms that can be established for any two users. A status of 'Available' or 'Unavailable' is associated with each client. An audio alert is also provided to notify the user that another application user is trying to establish a chat session with them. This alert can be customised to play any wave file that the user supplies to their client application.

3.4 Networking support and interface

Buttons on the main GUI also activates the networking support. The user specifies the server they would like to connect to and their simulation environments are connected and synchronised with the press of the 'Client Connect' button. Much effort was spent in making sure that the networking capabilities were easy to use, robust and scaleable. The network relies on communication via a TCP/IP link. Changes in the server environment are communicated to the clients via a simple multicast transmission. The main issue is maintaining synchronisation between the clients and the server. To enable this, the following four different options were considered.

3.4.1 Server-based synchronisation with threads

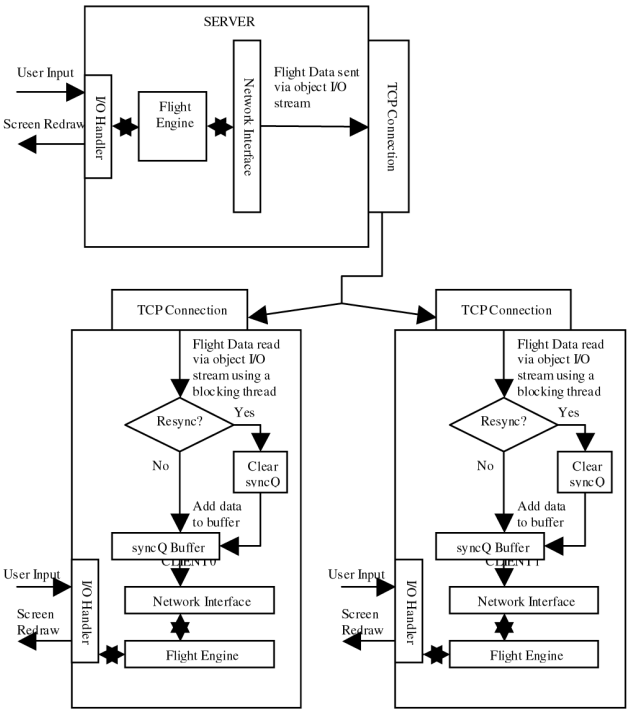
The first synchronisation method implemented to provide networking support was a server-based scheme that uses threads. The server periodically collects information (including position, direction of travel, distance travelled, command type and update rate) about each avatar into a class used for flight data management and transmits the class directly as a packet of information, with no reformatting, to a client machine via Java's support for object input/output streams. The frames per second setting in the GUI determine the frequency at which the packets are sent. Figure 4 describes the packet structure in detail. The client machine is running a thread that is constantly reading from the machine's socket. As data arrives it is read by the thread, which returns the packet data. This data is then copied to a FIFO, referred to as the 'syncQ' buffer on the client end, which is filled with flight data management classes as they are populated with data from the server. Whenever the client machine updates its display, it simply pulls the next packet of data from the FIFO and uses the data in the packet to update the local environment information.

Figure 4 Flight data packet describing an avatar

```
public class flightPacket implements Serializable
{
  public Point3d     m_newPos = new Point3d(0,0,0);
  public Vector3d    m_travel = new Vector3d(0,0,0);
  public double      m_distanceTraveled = 0;
  public boolean     sync = false; //command type
  public int         fps = 0;
}
```

The issue with this method is that the thread will wait until data is available prior to returning and allowing the syncQ FIFO to update. This allows the client application's environment to seriously diverge from the server's environment over time. Synchronisation is maintained by a resync command that the server can issue. A resync command is periodically transmitted to each client that indicates which packet they should be using to update the display. If the client is not properly synchronised, the client's FIFO is emptied and the packet that they received with the resync command is drawn immediately. Figure 5 illustrates the client–server interaction in detail.

Figure 5 Server-based synchronisation with threads



The result of this synchronisation scheme is that the client and server environments can diverge over time until a resync command is issued. If the applications are running on networks with minimal latency and systems with evenly matched computing power the number of display frames that client and server will diverge by is kept to a minimum. In non-ideal conditions, however, either a number of packets will be queued while waiting for a resync command to be issued. Also, under some conditions the thread will be left waiting for new packets to display, the display will just be continually updated with the last frame that was received and system resources will be squandered while the thread is left waiting for data to arrive. Both of these circumstances will cause the objects

in the client's virtual environment to move in a periodic jumping/stuttering fashion when the display is updated after an in-opportunely timed resync or a long wait for data.

3.4.2 Client-based synchronisation with threads

The second synchronisation method implemented to provide networking support operates very similarly to the first. However, instead of allowing the server to direct the resynchronisation of the client environment, the client handles this directly. The client machine is still running a thread that is constantly reading from the machine's socket. The thread is used to transfer data from the socket to the FIFO and whenever the client machine updates its display it simply pulls the next packet of data from the FIFO and uses the data in the packet to update the local view perspective.

To maintain synchronisation between the client and the server, the client periodically issues resync commands to itself. The resync command causes the client's FIFO to be emptied and the next packet that is received will be drawn immediately. Since the client is handling the synchronisation the resync command is asserted on a more regular basis, as it is not subject to network lag. However, the final result is much the same as the first method. Under non-ideal conditions a number of packets can be queued prior to the periodic resync being issued and the displayed scene will still appear to stutter as avatar locations are updated by large amounts when the resync is finally asserted. The converse case of the thread waiting for data to arrive will also have deleterious consequences identical to the issues described under the first synchronisation method. This particular method was not explored in detail.

3.4.3 Server-based synchronisation with non-blocking I/O

The next synchronisation method that was examined as a way to provide networking support takes advantage of a new feature of Java: support for non-blocking I/O. The non-blocking I/O provides a method of obtaining data from the socket without the cost of waiting for a threaded read of the socket to return with data. Non-blocking I/O, instead, allows for the transfer of data from the socket into a buffer whenever data is available on the socket, supporting the ability to immediately terminate the socket read and return if there is no data available. Periodically, an update is requested via the non-blocking I/O and regardless of whether data is available or not, the client's environment is allowed to continue executing the application immediately afterwards. Like the previous thread-based methods synchronisation is still maintained via a resync command.

However, non-blocking I/O does not support the transmission of data in discrete packets of flight data management classes, because object input/output streams are not available. Instead, due to the requirements of the non-blocking I/O standard, the flight data must be marshalled into a sequence of bytes that can be transmitted from the server to the client. The method chosen to marshal the data for transmission is to transform the data into a string of a particular format (see Figure 6), which is then divided into bytes. The non-blocking I/O on the client system reads the data from the socket into a byte buffer as soon as the data is available. The byte buffer, referred to as the 'dst' buffer, is a piece of memory of constant size located at a fixed address. The contents of the byte buffer are in turn decoded into a string buffer and the string buffer is parsed into a FIFO that consists of a linked list of strings, which is referred to as the 'syncQ' buffer. Each string in the list contains the information that was previously transmitted in a packet. When new data is required to update the client's environment the next link in the list is accessed and the string contents are transferred back into the normal packet of the flight data management class (as described previously in Figure 4), completing the unmarshalling of the transmitted data. This method of marshalling/unmarshalling the data, while inelegant, was found to be very robust.

Occasionally, the group of bytes transmitted will only constitute a partial packet of information. In this event, the bytes are returned to the dst buffer, where they will reside until the next successful read of the socket. Only, after enough bytes have been stored in the dst buffer to enable reconstitution of a full packet of flight data will the data be decoded into the string buffer. Periodically, the server issues a resync command, which causes the syncQ to be cleared and the next string of data to be processed immediately and used to update the display. Figure 7 illustrates the client–server interaction in detail.

The non-blocking I/O addresses the main issue of waiting for data to arrive at the socket, which is inherent in the threaded approach, at the small expense of needing to maintain a number of buffers for handling the marshalling and unmarshalling of data for exchange between the server and the client. However, this method is not perfect, because in a non-ideal system there is still the chance that a number of packets will be queued while waiting for a resync command to arrive. After the arrival of the resync command, a number of packets will still be dropped, which will cause the displayed image to jump or stutter. So, while this method does offer a distinct improvement over the threaded methods in that it addresses the issue of wasting system resources waiting for data, it is not the most ideal solution.

Figure 6 Marshalling of flight data packet into string form for non-blocking I/O, synchronisation method

```
String send = "/*" + toSend.m_newPos + toSend.m_travel + toSend.m_distanceTraveled + "," + toSend.sync + "," + toSend.fps + "*/";
```

Figure 7 Server-based synchronisation with non-blocking I/O

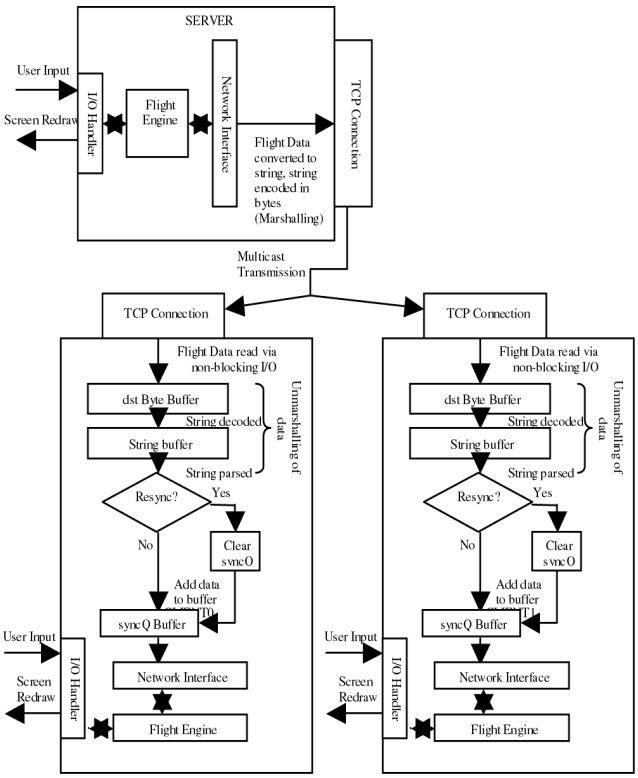
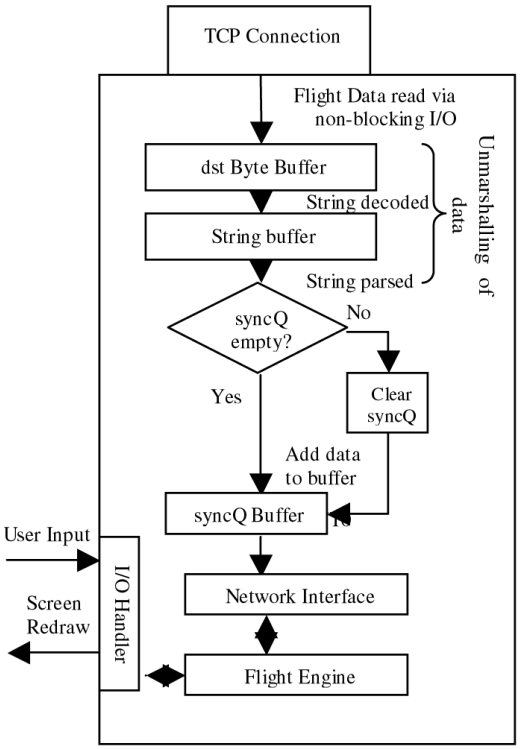


Figure 8 Transparent synchronisation with non-blocking I/O, client only



3.4.4 Transparent synchronisation with non-blocking I/O

The final synchronisation method that was developed also used non-blocking I/O. The non-blocking I/O was used to provide a synchronisation method that did not rely upon a discrete resync command, but instead operated continuously to maintain synchronisation between the client and server environments in a more transparent fashion. Data transmission and transformation from packet into byte and byte back into packet still follows the steps illustrated in Figure 7.

Synchronisation is maintained in this method by simply replacing all data in the syncQ string buffer with new data whenever it arrives at the socket. If no new data arrives the client application continues to update its environment based on the information contained in the syncQ buffer. As soon as new data appears an automatic resync-like operation takes place that enables the application to immediately begin using the new data and discard any remaining, unused packets. Figure 8 illustrates the changes to the client system.

This method essentially allows for resync to occur only at optimal points in time, since it is the most advantageous to resync whenever data has arrived before all the previous data has been used to update the display. When this is coupled with the fact that the non-blocking I/O allows for socket access that avoids waiting for data to appear on the socket (and avoids the misuse of system resources, unlike the threaded method), the foundation for a solid networking support system is established.

4 Comparison of synchronisation methods

The four network synchronisation methods described above each provide a means of maintaining synchronisation between the server and client environments. To determine which method is best suited for creating a Java-based 3D-virtual environment with networking support, a number of experiments were run and both qualitative and quantitative results were recorded.

4.1 Quantitative results

A number of procedures were implemented to facilitate the collection of quantitative data under all modes. A means of easily switching the synchronisation and socket access methods was installed. For the methods that rely on an explicit resync signal, a data collection routine was established that recorded the server's currently drawn frame number the resync command corresponded to, along with the client's local frame that was being processed and drawn when the resync command was received. For the method that uses the transparent (or automatic) resync method, the number of frames was dropped when the new packet of data arrived was recorded. This data provides a quantitative means of comparing the synchronisation methods by allowing for the contrasting amounts of divergence between the client and the server environments to be recorded. These measurements were taken using various resync intervals.

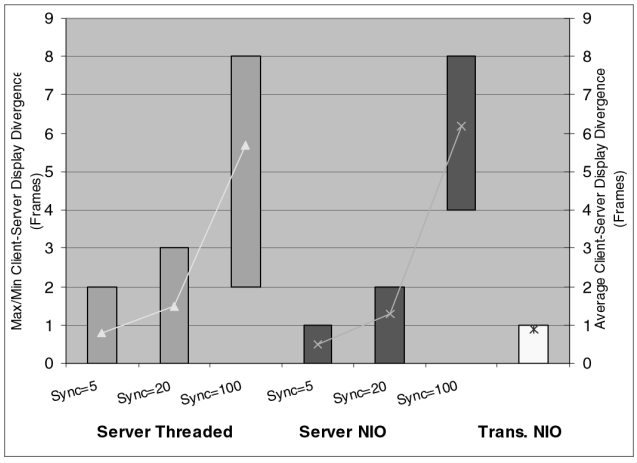
Each simulation on the client system consisted of a standard test that guided the user's avatar through a set sequence. The client and server frames per second were

both set to 30. The simulation was started in full screen mode. From the starting position the avatar's speed is increased to a simulated 300 mph as displayed by the GUI. The control key corresponding to a yaw to the left is pressed and held to cause the avatar to turn to the left. The resync command is set to occur after every 5, 20 or 100 frames for both of the server-based synchronisation approaches and the amount of divergence is recorded. Then, the transparent synchronisation approach is used and the amount of divergence recorded.

This sequence of tests was performed using a dual processor 700 MHz Intel Pentium 3 class machine with 1 GB of memory as the client. The second processor and extra RAM were instrumental in enabling correct video capture of the test sequences. The server was running on a 700 MHz AMD Athlon system with 256 MB of memory. Both machines were running Windows XP and communicated via a 100 Mb/s LAN. The tests were performed on these machines and the maximum number of frames discarded was recorded. Video recordings of typical test sequences used for testing the performance of the application can be accessed from the project website (see Whiskey Source Code, 2005).

The client–server frame divergence data was collected in sets of 10 samples over 10 simulation runs for each synchronisation method and resync interval and then compared. Figure 9 illustrates the results. The server-based synchronisation method with threads shows a trend towards undesirable increasing divergence between the client and server applications as the resync interval increases. The same trend is seen when the server-based synchronisation method with non-blocking I/O is used, although the maximum divergence tends to be lower at the point in time when the resync command is received by the client. The transparent synchronisation method using non-blocking I/O shows a maximum of one frame being discarded by the client at any one point in time, which indicates that the divergence in the client and server applications is kept to a minimum. It should be noted that the only other method that provides an equivalent maximum number of frames discarded and a slightly better average number of frames discarded, is the server-based synchronisation method using non-blocking I/O with a resync interval of five frames.

Figure 9 Client-server divergence in frames for a variety of synchronisation and resync intervals



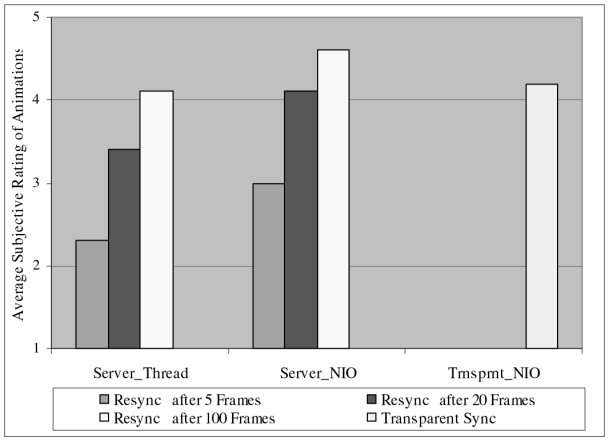
4.2 Qualitative results

In addition to the quantitative comparison, a qualitative comparison was also performed. Users of the client's virtual environment were asked to rate the smoothness or fluidness of the motion of the environment in the graphical display on a scale of 1–5, with 5 being the best. The subjective ratings were averaged and recorded for each synchronisation method and resync interval to record the users' subjective impressions. The impressions of 15 users were recorded.

The standard test sequence that was described above for collecting quantitative results was used again for observation and qualitative assessment. Each test was also performed using the same computer systems described above. The test sequences were repeated for each user and the application users taking part in the qualitative comparison were then asked to rate the application's performance. Video recordings of typical test sequences used for qualitative assessments of the application can be accessed from the project website (see Whiskey Source Code, 2005).

The subjects' ratings were recorded and averaged together. The results are compared in Figure 10. The graph clearly shows that users much preferred either the transparent synchronisation method with non-blocking I/O or either of the other server-based synchronisation methods provided the resync interval was set to 100. The synchronisation methods that rely upon an active resync command produce the most fluid and subjectively pleasing graphic animations when the resync interval is maximised. This is because fewer resyncs result in fewer jarring updates to the objects being displayed. It should be noted that the use of the non-blocking I/O, even with the server-based synchronisation method, appears to better optimise the use of system resources due to the improved qualitative ratings when compared to the threaded approach.

Figure 10 Subjective rating of fluidness of animations for a variety of synchronisation methods and resync intervals



4.3 Discussion

The synchronisation methods that performed the best both in a quantitative and qualitative sense are the transparent synchronisation with non-blocking I/O and the

server-based synchronisation with non-blocking I/O using a resync interval of either 20 or 100. Arguments have been made by Pettifer and West (1999) that a user's subjective experience is more important than the virtual environments objective reality. It is thus possible to relax synchronisation to optimise a user's subjective experience. This would make the synchronisation approaches that rely upon a discrete resync interval more viable. A better subjective experience can be obtained by simply increasing the resync interval at the expense of maintaining synchronisation. If the user is truly willing to sacrifice some of their objective experience by decreasing synchronisation, then the server based synchronisation with non-blocking I/O using a resync interval of 20 becomes attractive.

It should be noted that it has been postulated that as the resync interval is decreased below 5 frames the server-based synchronisation methods should start to behave similar to the transparent synchronisation method. This does hold true for the amount of divergence realised, but the transparent method produces a much more subjectively pleasing experience. It is likely that inefficiencies in the server-based synchronisation code are the cause, but further experimentation would be necessary to truly understand this inconsistency.

In the end, a method that provides a very good subjective experience while still maintaining excellent synchronisation is clearly the better choice overall. The method of using transparent synchronisation with non-blocking I/O provides the best subjective and objective experience in a Java-based distributed virtual environment.

5 Conclusion

Construction of a distributed 3D-virtual environment that is cost-effective, portable, maintainable and extensible is a non-trivial task. Such environments must provide acceptable graphic, control, communications and networking support. A method and a group of programming tools to create a distributed 3D-virtual environment that provides good network performance along with the other attributes mentioned would be highly desired. The work on the Whiskey distributed virtual environment has shown that Java is now a useful tool to enable the creation of an environment that contained these features. Java, until recently, did not provide adequate networking support, making effective synchronisation between network and client system difficult, but the release of the official support for non-blocking I/O has addressed this issue.

The simplification of early effort provided further validation for the decision to develop the project in a Java-based programming environment because of its broad open source support. The development work on graphical support for the Whiskey environment was greatly minimised due to the solid foundation provided by the open source work. Earlier project work was devoted to simply integrating the disparate code sources together into a working whole. This provided more time for developing all the features necessary to transform the virtual environment into the desired application. The graphic engine was simply updated to provide the level of detail and feature support that was preferred, the control system was adjusted to enhance usability and the communication package's infrastructure was developed. This allowed for much time to be spent establishing a networking support system to tie various user environments together.

A great deal of effort was expended to determine what the best method was for maintaining synchronisation of virtual environments across a network using a Java-based toolset. The transparent method for synchronisation that utilises non-blocking I/O loses the fewest number of frames and receives excellent average subjective scores for the fluidness of graphical display animations. This transparent synchronisation method, via both qualitative and quantitative measurements, was found to be the best method for maintaining client–server synchronisation in a distributed virtual environment. This result corresponded well with expectations of the designers and further supported the fact that Java can be used to provide the foundation for a networked 3D-virtual environment.

The graphical engine, user communications, control system and networking support, when functioning together, provided a stable base for the collaborative geo-exploration flight simulator application that was developed. The platform required minimal network/computing overhead, but provided maximum collaborative functionality via 3D-graphics, intuitive controls and multiple communication avenues. Thus, the non-blocking I/O support now available in Java 1.4 and later, allowed for the easy development of virtual environments that can support a variety of networked applications. Java's many benefits that make it an attractive language to develop environments for educational and entertainment purposes are now bolstered by an API that makes the promise of robust networking support an easily realised actuality. This networking support can now be used to provide real time interaction among multiple users, finally allowing Java to become a good tool with which to develop real-time networked virtual environments.

The source code for the Whiskey application is open source and freely available from the Whiskey project website (see Whiskey Source Code, 2005). It is the developers' hope that future users will both learn from and improve upon the framework that has been developed. ipKtalk and ipKchat can be used as stand-alone applications or as subroutines. They are simple, yet full-featured programs, which can be easily expanded and/or modified (with permission) to fulfil many different requirements. They are packaged with the Whiskey application. All code is available under the General Public Licence (GPL).

Acknowledgement

We would like to thank Sergei Grichine who provided the open source Java code that served as the foundation for the graphical engine (see Java Flight Simulator, 2005).

We would also like to thank Soji Yamakawa for his many VRML97 models that helped to populate our environment (see VRML97 Airplane Models, 2005).

Finally, the authors would like to thank IBM for their continous support for our education.

References

Bangay, S. (2001) 'Experiences in porting a virtual reality system to Java', *Proceedings of the First International Conference on Computer Graphics, Virtual Reality, and Visualization (Cape Town, South Africa, 2001)*, New York, NY: ACM Press, pp.33–37.

Benford, S., Greenhalgh, C. and Vaghi, I. (1999) 'Coping with inconsistency due to network delays in collaborative virtual environments', *Proceedings of the ACM symposium on Virtual Reality Software and Technology (London, UK, December 1999)*, New York, NY: ACM Press, pp.42–49.

Brewer, E., Culler, D. and Welsh, M. (2001) 'SEDA: an architecture for well-conditioned scalable internet services', *Proceedings of the 18th ACM Symposium on Operating Systems Principles (Banff, Alberta, CA, October, 2001)*, New York, NY: ACM Press, pp.230–243.

Bricken, M. (1991) 'Virtual reality learning environments: Potentials and challenges', *ACM Computer Graphics: SIGGRAPH*, Vol. 25, No. 3, pp.178–184.

Burdea, G.C. and Coiffet, P. (2003) *Virtual Reality Technology*, Hoboken, NJ: John Wiley and Sons, Inc.

Cai, W., Lee, B.S., Turner, S.J. and Zhou, S. (2004) 'Time-space consistency in large-scale distributed virtual environments', *ACM Transactions on Modeling and Computer Simulation*, Vol. 14, No. 1, pp.31–47.

Campbell, B., Collins, B., Hadaway, H., Hedley, N. and Stoermer, M. (2002) 'Web3D in ocean science learning environments: virtual big beef creek', *Proceedings of the Seventh International Conference on 3D Web Technology (Tempe, AZ, 2002)*, New York, NY: ACM Press, pp.24–28.

Cook, J., Marsh, J., Pettifer, S. and West, A. (2000) 'Collaborative virtual environment: DEVA3: architecture for a large scale distributed virtual reality system', *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (Seoul, Korea, October 2000)*, New York, NY: ACM Press, pp.33–40.

Cox, B.J. and Novobilski, A. (1991) *Object-Oriented Programming: An Evolutionary Approach*, Boston, MA: Addison-Wesley Publishing.

Cronin, E., Filstrup, B., Jamin, S. and Kurc, A.R. (2002) 'An efficient synchronization mechanism for mirrored game architectures', *Proceedings of First Workshop on Network and System Support for Games (New York, NY, April 2002)*, New York, NY: ACM Press, pp.67–73.

Crowcroft, J., Oliveira, M. and Slater, M. (2000) 'Component framework infrastructure for virtual environments', *Proceedings of the Third International Conference on Collaborative Virtual Environments (San Francisco, CA, 2000)*, New York, NY: ACM Press, pp.139–146.

DeFanti, T.A., Johnson, A.E. and Leigh, J. (1997) 'Issues in the design of a flexible distributed architecture for supporting persistence and interoperability in collaborative virtual environments', *Proceedings of the 1997ACM/IEEE Conference on Supercomputing (CDROM) (San Jose, CA, November 1997)*, New York, NY: ACM Press, pp.1–14.

DEM Standard (2005) Available at: http://rockyweb.cr.usgs.gov/nmpstds/demstds.html. Accessed on 20 April 2005.

Georganas, N.D., El Saddik, A. and Yang, D. (2003) 'A lightweight multi-session synchronous multimedia collaborative environment', *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (Tunisia, Tunis, July 2003)*.

Goldberg, A., Kesselman, J., Melissinos, C., Petersen, D., Soto, J.C. and Twilleager, D. (2004) 'Java technologies for games', *ACM Computers in Entertainment*, Vol. 2, No. 2, Article 8.

Han, S., Lee, D. and Lim, M. (2002) 'ATLAS: a scalable network framework for distributed virtual environments', *Proceedings of the Fourth International Conference on Collaborative Virtual Environments (Bonn, Germany, September 2002)*, New York, NY: ACM Press, pp.47–54.

Hughston, R., Mine, M.R. and Shochet, J. (2003) 'Building a massively multiplayer gamer for the million: Disney's toontown online', *ACM Computers in Entertainment*, Vol. 1, No. 1, Article 6.

Ishibashi, Y. and Tasaka, S. (2003) 'Casuality and media synchronization control for networked multimedia games: centralized versus distributed', *Proceedings of Second Workshop on Network and System Support for Games (New York, NY, May 2003)*, New York, NY: ACM Press, pp.42–51.

Java Flight Simulator (2005) Available at: http://www.earthquakemap.com/javaflight/what-is-javaflight.html. Accessed on 20 April 2005.

Lau, R.W.H., Li, F.W.B. and Li, L.W.F. (2004) 'Supporting continuous consistency in multiplayer online games', *Proceedings of the 12th Annual ACM International Conference on Multimedia, Technical Poster Session 2: Multimedia Networking and System Support (New York, NY, October 2004)*, New York, NY: ACM Press, pp.388–391.

Pettifer, S. and West, A. (1999) 'Subjectivity and the relaxing of synchronization in networked virtual environments', *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (London, UK, 1999)*, New York, NY: ACM Press, pp.170–171.

Roussou, M. (2004) 'Learning by doing and learning through play: an exploration of interactivity in virtual environments for children', *ACM Computers in Entertainment*, Vol. 2, No. 1, Article 1.

Shirmohammadi, S., El Saddik, A., Georganas, N.D. and Steinmetz, R. (2003) 'JASMINE: A java tool for multimedia collaboration on the internet', *Journal of Multimedia Tools and Applications*, Vol. 19, No. 1, pp.5–28.

Shirmohammadi, S. and Georganas, N.D. (2001) 'An end-to-end communication architecture for collaborative virtual environments', *Computer Network Journal*, Vol. 35, Nos. 2–3, pp.351–367.

Singhal, S. and Zyda, M. (1998) *Networked Virtual Environments Design and Implementation*, Reading, MA: Addison-Wesley.

Source for Java Technology (2005) Available at: http://java.sun.com/. Accessed on 20 April 2005.

Stephenson, C. and West, T. (1998) 'Language choice and key concepts in introductory computer science courses', *Journal of Research on Computing in Education*, Vol. 31, No. 1, pp.89–95.

VRML97 Airplane Models (2005) Available at: http://ciel.me.cmu.edu/soji/aircraft/aircrafte.html. Accessed on 20 April 2005.

VRML97 Standard (2005) Available at: http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-IS-VRML97WithAmendment1/Accessed on 20 April 2005.

Whiskey Source Code (2005) Available at: http://www.cem.uvm.edu/~medialab/Fall04/ee214/Java3D/. Accessed on 20 April 2005.

Note

[1]Java and Java 3D are trademarks or registered trademarks of Sun Microsystems, Inc. in the USA and other countries.