The JOTSA Animation Environment

Steven Robbins
Division of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

Abstract. JOTSA (Java On Time Synchronous Animation) is an environment for web-based animation of algorithms and data. On time refers to the moving of objects so that they complete their movement at a known time, independent of the hardware or operating system of the target machine. Synchronous refers to the coordination of the movement of several objects. JOTSA was motivated by the need to represent exact timing relationships in network protocols and other time-critical applications in a platform-independent manner. JOTSA provides a web-based user interface which is a natural vehicle for remote execution and wide dissemination. In addition to exact time animation, JOTSA supports multiple dependent or independent views, panning and zooming, linking of collections of objects, event-driven simulation, and synchronization. Applications have the full resources of the Java virtual machine and can be written to support interaction in a way that is familiar to the user.

1 Introduction

JOTSA (Java On Time Synchronous Animation) is a Java animation package for performing interactive animations. JOTSA's representation of movement is based on the path-transition paradigm [25]. On time refers to the moving of objects so that they complete their movement at a known time, independent of the hardware or operating system of the target machine. Synchronous refers to the coordination of the movement of several objects. If two objects are supposed to maintain a certain relationship as they move, that relation is exactly maintained on the display. JOTSA is written in the Java language [2], so it can be used on most modern platforms. JOTSA animations can be run over a network through a standard browser, and JOTSA has facilities which make it suitable for animation of user-written event-driven or time-driven simulations.

JOTSA was motivated by the need to develop interactive animated simulations of network protocols. In a typical network protocol, two or more processes communicate using handshaking. In the simplest case, one process sends a packet and waits for an acknowledgment. The next action depends on the timing of events that are not under the control of the sender, e.g. whether an acknowledgment arrives before the timeout. In an interactive simulation, the user should be able to affect the behavior of the simulation, say by destroying an acknowledgment before it reaches the sender. The user interaction makes interactive simulation nondeterministic in contrast to non-interactive simulations where the result of a given transmission is predetermined by the input to the simulator.

The design goals of non-interactive and interactive simulations are very different. A typical design goal for non-interactive simulation is to produce the results as soon as possible. The completion time depends either on the size of the time step (time-driven simulation) or the granularity of the events (event-driven simulation). In either case the execution time depends on the speed of the underlying hardware and software of the simulator. A design goal of interactive simulation is to have the simulated time appear to flow at a smooth, predictable rate. Ideally, this rate should be independent of the platform and the amount of computation required. Network protocols and many other simulations typically have a time scale on the order of milliseconds or microseconds. Rather than having these simulations run as quickly as possible, animation of such protocols must be scaled down to a human time scale of seconds.

The combination of an interactive simulation with animation for display poses certain difficulties. The animation must accurately reflect the movement of time. There are two aspects of this, the coordination of the animation with the simulation and the coordination of simulation time with real time. In many cases the time scale of the underlying system is so fast compared to the human time scale, that the simulation needs to be slowed down by a factor of 1000 or more to allow for human interaction. Performing the computations for the simulation fast enough is not a problem. The animation, on the other hand, poses severe restrictions on what can be shown. A smooth animation might require 30 frames per second, and such a display rate will tax even the fastest platform if the animation is complicated.

With hardware speed increasing exponentially and efficient Java implementations becoming available, animations may become unusable when moved to faster platforms. The author had just such a problem with an XTANGO application [17] which ran at about the right speed on the SPARC LX on which it was designed, but was much too fast when the lab upgraded to SPARC 5's and was lightning fast when run on a 200 Mhz Pentium Pro. While it was possible to adjust the speed while the animation was running, the user had to guess at the right setting for the particular hardware, and the rate at which the animation ran also depended on what other processing was occurring on the machine. JOTSA directly addresses these problems. A JOTSA animation runs at the same rate on both fast and very fast hardware. This exact time execution is particularly difficult in Java because the user does not have direct control of the screen.

Section 2 discusses the related work, and Section 3 gives an overview of JOTSA. Timing issues are discussed in Section 4. Some simple examples are given in Section 5, and the motivating network example is described in Section 6. Section 7 addresses some performance issues and Section 8 presents a discussion of open issues in Java animation.

2 Related Work

The extensive work on algorithm animation is described in the survey by Myers [13] and the definitive work of Brown [5]. XTANGO [26] and POLKA [27] are X-based animation libraries for C and C++ programs which motivated the creation of JOTSA. XTANGO's companion Animator is a standalone program that does animation controlled by ASCII strings sent to standard input. The Animator can be used with any program by having that program generate animation commands that are piped into the Animator. POLKA is a parallel animation library for animating the execution of parallel programs. POLKA animations must be written in C++ and the display is based on X. The Polka model has been extended to include real-time animations with Polka-RC [28]. In Polka-RC the program being animated and the animation routines run as separate processes and communicate using sockets.

A large number of simulation tools and languages with built-in animation tools are available. Most of these must be compiled for a particular system but are available on several platforms. GPSS [24] is a special-purpose simulation language oriented toward queuing systems which has had many reincarnations [4, 8]. SIMSCRIPT II [23] is a general programming language with features for building simulation models. SIMGRAPHICS does back-end animation and front-end graphical input for SIMSCRIPT II. SIMMAN/Cinema [15] is a general purpose simulation and animation language which has been used mainly in the manufacturing area. AweSim [14] is a general purpose simulation system for MS Windows systems which represents simulation objects by Windows bitmaps that can be moved around. Simulators without direct support of animation can use an add-on tool such as Proof Animation [10] to perform animations from trace data. Proof Animation works in a way similar to the XTANGO Animator, but the animation is done after the simulation has completed, and it is designed specifically for use with simulation languages.

Several Java-based simulation tools have recently been announced. JSIM [1] is a simulation library which is integrated with a database management system and is based on Query Driven Simulation. Simkit [7] is a class library for discrete simulation written in Java. Neither of these packages addresses the issue of exact time animation.

Project Horizon [11] is a cooperative agreement between NASA and the University of Illinois to enhance web technology to better support public access to earth and space science data. The Horizon Data Browser is a Java-based tool for browsing and visualizing scientific data. It is still in the alpha stage [30] of development but promises to provide a number of useful tools for animating data over the web.

Mocha [3] uses a completely different model for algorithm animation over the web that is not based on Java. The goal of the system is to provide a high level of security to protect algorithm code. As in the X Windows System model, Mocha programs run on the remote machine and the user interface runs locally. Mocha is still in the prototype stage.

The need to modify the display methodology when dealing with time-critical visualizations in the context of 3D modeling on high performance machines has drawn some recent attention [6, 12]. Here we are interested in more modest displays, but on commodity computers.

3 JOTSA Overview

JOTSA follows the object-oriented design paradigm of the Java language. The JOTSA class **JotsaAnimationObject** encapsulates all of the structures and methods needed to control a moving object. In the path-transition approach as implemented here, a path is a mapping from virtual time into a multidimensional space. At each point of time the object has a position (x-y coordinates), a shape (e.g. rectangle or oval), a size, an orientation, a color and other properties. By default, all of these attributes are constant. In most cases, the user will just specify how the position changes with time.

3.1 Shapes

JOTSA supports all of the geometric shapes supported by the underlying Java language including rectangles, ovals, character strings, arcs, lines and polygonal paths. In addition, regular polygons can be specified by the number of sides and a radius. Multi-line strings can be specified by single string containing newline symbols. JOTSA also supports externally created images and movies consisting of sequences of images. These movies can be displayed in real time in a frame-accurate way.

3.2 Position

The natural representation of an object may depend on the intended use. A standard way of specifying the position of a character string is by the coordinates of its lower left corner. This coordinate system is appropriate if the string is to be left justified. However, if the string is to be put in a circle, it is more convenient to specify the center of the string. The standard Java coordinate system defines (0,0) at the upper left corner of the applet window. Java supports specifying the position of a rectangle by its upper left corner, the position of an oval by the upper left corner of its bounding rectangle, the position of a string by its lower left corner and the position of a polygon by the coordinates of the vertices.

While the Java coordinate representations are appropriate for some applications, the selection of a particular vertex of a rectangle as its origin is somewhat arbitrary. JOTSA supports a centered object coordinate system in addition to the standard Java coordinate system. Specifying objects by their centers leads to a natural way of grouping objects together and manipulating them as a group. For example, a polygon can be represented by the coordinates of its center and the relative positions of the vertices from the center. Moving the polygon just requires changing the coordinates of the center position. JOTSA also allows the position of one object to be specified relative to that of another object so that objects can be moved as a group.

3.3 Size

Another attribute that can change along a path is the size of an object. Rectangles and ovals have a size parameter for each dimension, while character strings and regular polygons have one such value. The size of a string is the pointsize of its font, and the size of a regular polygon is its radius.

In JOTSA, the size of an object on the screen is determined by its size parameters and several scaling factors. The size parameters are integers representing a number of pixels. The scaling factors are doubles to allow for smoothly changing the size of an object along its path without accumulating roundoff error. A single scaling factor can be used to change the size of several objects in a coordinated way.

3.4 Collections

A JOTSA *collection* consists of a controlling object called the *master* and one or more additional objects called *slaves*. The relationship between a master and a slave in JOTSA is set up so that a master and all of its slaves act as a unit. Complex objects can be generated from simpler ones. When the master moves, the entire collection moves in a coordinated way. When the master is scaled in a given dimension, the slaves are similarly scaled and their positions are adjusted so that the unit is scaled.

Figure 1a) shows a circular master object and a number of slave rectangles. All scaling factors are one. The positions of the rectangles are specified by the positions of their centers relative to the center of the master. The center of each object is marked with a small dot, although in JOTSA it would not normally be shown. Figure 1b) shows the same collection when the scaling factors of the circle master object have be changed to 2.0 in the x-direction and 0.5 in the y-direction.

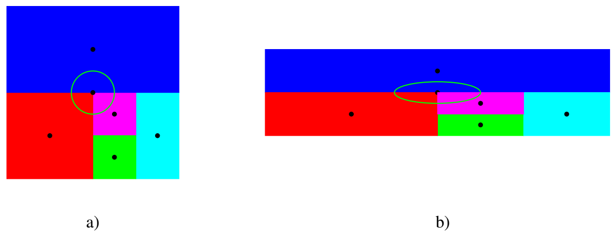

Figure 1: A collection of rectangles with a master circle.

In order to preserve a collection under scaling, JOTSA defines two positions for each object. The position that an object appears on the screen is called its *absolute position*. Each object also has a pair of (x, y) coordinates called its *natural position* which by default is the same as the absolute position. When an object is a slave within a collection, its natural position represents the position of the object relative to the master. In the same way, each object has absolute and natural size parameters and other scalable attributes.

Suppose that the master has absolute x-coordinate X_m and scaling factor in the x-direction, s_m. If a slave has natural x-coordinate x_s, scaling factor in the x-direction, s_s, and natural size (width) in the x-direction, w_s, then the absolute x-coordinate of the slave, X_s, and the absolute width of the slave, W_s are:

$$X_s = X_m + s_m x_s$$

$$W_s = s_m s_s w_s$$

Notice that the scaling factor of the master affects both the position and size of the slave, while the scaling factor of the slave affects only its size. Figure 1 shows why this scaling method produces the desired result. When the master is scaled by a factor of 2.0 in the x-direction, the difference in the x-coordinates of the master and each of its slaves must be doubled. Similarly, when the master is scaled by a factor of 0.5 in the y-direction, the difference in the y-coordinates of the master and each of its slaves must be halved. Usually the master object is just used to control the slaves and is not displayed.

When two collections are merged into one, all of the slave objects in one collection are linked to the master of the other collection and have their natural position and size modified so that after the relinking, their absolute position and size remain the same. The other master is then deleted.

When a master object is rotated, the position of its slaves are similarly rotated. If the slave object is rotatable, the slave is also rotated about its center so that the rotation of the master and slave behave as a rigid rotation of the two objects. Certain JOTSA objects (such as ovals) cannot be rotated. In this case the position of the slave is rotated, but its orientation stays fixed relative to the screen when the master is rotated.

A more detailed description of JOTSA collections can be found in [19].

3.5 Multiple Independent Synchronized Views

JOTSA animation is done in one or more rectangular portions of the screen called canvases. Each canvas is associated with a set of objects which are to be displayed in that canvas. JOTSA synchronizes the canvases so that all canvas displays correspond to the same virtual time. Because of this synchronization, different canvases can show different views of the same scene.

3.6 Multiple Dependent Views

JOTSA supports panning and zooming. In fact, each canvas can have any number of scalable windows associated with it. Each such window shows the same objects as the parent canvas, but the view can be scaled (zoomed in or out) and translated (panned). While Java supports scaling of any image on the fly, rescaling an image of a moderate size takes on the order of a second on a moderately fast machine. This is at least an order of magnitude too slow for animation in real time. JOTSA redraws the objects with the appropriate size in the scaled window instead of using the intrinsic Java scaling. As long as the number of objects is not too large (say, less than 100), this approach is considerably faster than scaling the image produced by drawing the objects.

3.7 Data Animation

In addition to supporting simulation and animation of algorithms, JOTSA supports animation and visualization of data with time critical features that must be preserved independent of the platform. The animation in [16] describes an experiment in which physicists made a videotape of an experiment. The videotape was digitized and the critical features were animated. The animation incorporates fading, overlaid cues and linking of objects across views to improve visual persistence. JOTSA allowed the animation to be adjacent to or to be superimposed upon the original movie.

4 Timing Issues

In traditional animation paradigms, virtual time is monotonically increasing and changes at a rate determined by the amount of processing required at each time step or event and by the speed of the hardware that is running the program. Unlike other virtual-time systems, JOTSA's virtual time is directly linked to real time. The user specifies the speed of virtual time by a rate that connects virtual time to real time. When the rate is 1.0, virtual time and real time run at the same rate. When the rate is less than one, virtual time runs more slowly. For example, a rate of 0.5 indicates that virtual time runs at half the speed of real time.

A JOTSA animation specifies how an object moves in terms of virtual time. In the simplest case, an object moves along a straight line at a constant rate. The movement is specified by the endpoints of the path and the length of time (in virtual time) it takes to traverse the path. The user is then guaranteed that the object will be done moving after the given amount of virtual time. If the virtual time rate is 1.0, the user knows the real time at which the movement will be complete. The rate at which the object moves on the screen is independent of both the speed of the hardware and how much other processing (other objects moving or other process activity) is taking place.

Exact time execution comes with a price. If the hardware is too slow to handle the processing that needs to be done, the motion may look jumpy. If this is unacceptable to the user, virtual time can be slowed down. From the point-of-view of the developer of the animation, the animation is written in terms of motion in real time, and the algorithms are written in term of real time.

The basic assumption in JOTSA animations is that the display takes up most of the processing time and that the CPU can easily keep up with the processing necessary to do everything other than the display. Under these circumstances, JOTSA ensures that the non-display processing can be done fast enough to keep up with the flow of virtual time by limiting the display updates.

4.1 When to Repaint

One difficulty with implementing time-appropriate animations in Java is the lack of user control of when the screen is painted. Each Java canvas has its own **paint** method which cannot be called directly by the user. The user must request that the run-time system call **paint** by executing the canvas's

repaint method. By giving **repaint** an optional parameter, the application can suggest the number of milliseconds before the **paint** occurs. Unfortunately this value is just a suggestion.

A possible repaint strategy is to try to maintain a reasonable frame rate of about 30 frames a second. This is enough the ensure that the animation will look fairly smooth. However, this strategy would probably use up most of the CPU capacity of a moderately fast machine. One of the design goals of JOTSA is to be CPU-friendly: unless it is needed, the CPU should be available to other processes. This goal is not completely altruistic. Java is naturally threaded, and a 30 frames per second display rate would leave little processing power for the non-display threads of the simulation.

CPU friendliness implies that the display should not require much processing power when nothing is changing on the display. However, when objects are moving rapidly, it is acceptable to use the full power of the CPU, especially if this rapid movement is short-lived. As long as an object is already being displayed at each pixel position along its path, redisplaying more often would not improve the quality of the display of this object. In a typical animation there will be times when objects are moving and times when they are stationary. The optimal display rate is therefore dynamic.

Information about the path is part of the object's class, so each object should determine its own redisplay rate. Having each object cause a redisplay would be too inefficient so this is done by a master thread. The master thread gets the recommended redisplay rate from each object and takes the maximum rate within a certain predetermined interval. Each time an object is drawn to the screen, the optimal redisplay rate for that object is calculated and delivered to the applet.

4.2 How to Paint

Since JOTSA supports multiple canvases, the issue of how to coordinate the paint methods of each canvas must be addressed. Here are five approaches for coordinating paint methods:

Method 1: The **paint** methods are completely independent. Each canvas has a thread that periodically repaints the screen at a rate determined by the objects displayed in that canvas. While simple to implement, this method does not provide any guarantees of synchronization between canvases.

Method 2: The **paint** methods are independent, that is, each executes as a result of a **repaint** request for that canvas, but all repaints use the same virtual time. A master thread determines when to repaint based on all objects displayed. All of the objects calculate their positions based on the same virtual time until the master thread calls the **repaint**s again.

Method 3: The canvas's **paint** method copies a temporary image to the screen. A master thread requests each canvas in turn to fill the temporary image with objects based on a common virtual time. When the image is complete, the master thread calls each canvas's **repaint**. This approach is a minor modification of Method 2 that requires some additional synchronization, since **paint** should not access the temporary image while it is being modified.

Method 4: Each canvas steals the graphics context from its **paint** method allowing it to paint directly without calling **repaint**. The master thread has all canvases paint serially in a predetermined order using the same virtual time.

Method 5: Each canvas draws all objects to a temporary image rather than directly to the screen. When all objects have been drawn to the temporary images of all canvases, the canvases draw that image to the screen. This minor modification of Method 4 minimizes the delay between the updating of the screen for the various canvases.

Methods 2, 3, 4 and 5 have been implemented, and a study is being conducted to determine which produces the best results.

4.3 Time Driven Simulation and Movies

In time driven simulation, virtual time is incremented by a fixed amount at each time step. Similarly, movies are usually shown at a fixed frame rate. JOTSA supports the ability to display at a given frame rate. If the animation for a given movie frame cannot be displayed in time for the next frame, some frames will be lost. If an animation time step cannot be displayed fast enough, the display of some time steps will not appear, but the computation for each time step is done so that the simulation is correct. It is assumed that most of the processing is due to the display rather than the simulation and the simulation processing can be done fast enough to keep up with the passage of virtual time.

4.4 Synchronization and Events

Java supports synchronization though the use of monitors. Each object is potentially a monitor, and the key word *synchronized* is used to include methods in the monitor. Instead of condition variables, each monitor has its own **wait** and **notify** methods. Notify events are not queued, and a **notify** sent while the corresponding thread is not waiting is lost.

JOTSA supports synchronization through the display objects. When an object is done moving it can notify a thread that it has finished. JOTSA allows a thread to atomically start an object moving and wait for the object to reach its final position. This capability allows an animation to avoid the race condition in which an object finishes its motion and attempts to notify the thread before the thread has begun waiting.

In addition to the wait-notify mechanism implemented directly using the corresponding Java methods, JOTSA also supports sleeping for a given virtual time and a queued event list in which events are put in a queue and taken out using a FIFO discipline. These features are convenient in more complicated simulations in which events can be generated asynchronously to the waiting thread while the waiting thread is doing other work. Events can also be generated by an arbitrary number of JOTSA timers.

4.5 Scheduling

While JOTSA events and synchronization methods are not difficult for the programmer familiar with thread programming, experience with teaching advanced undergraduate computer science majors to use Java indicates that programming with threads is a difficult and time consuming process for the inexperienced programmer. Java programming (without threads) is actually quite simple, and students who have programmed before pick it up quickly. The Java environment uses threads to wait for common events such as keystrokes, mouse clicks or mouse movements, but these are easy to use as the implementation is transparent to the user. Such actions generate events that call an event handler which the user overrides to handle these events. Thus, although the Java programming environment is naturally threaded, this aspect of Java is mostly hidden from the programmer.

JOTSA provides a method for handling sequences of object motions without the need to deal with thread programming. An object can be set to generate an event captured by the standard Java **handleEvent** handler. These new events indicate completion of object movement. A second interface under development is a scheduling class that schedules object movements using a procedural interface.

5 A Simple Example

An applet which uses JOTSA is a class that extends **JotsaAnimationApplet**. A minimal JOTSA applet must perform the following steps in its **init** method:

- Call **super.init();**
- Set up a layout which includes **JotsaDefaultCanvas** as one of its components.
- Make sure the components have been laid out by calling **validate();**
- Call **JotsaInitImages();**

In the simplest case, to move an object requires the following steps:

- Create the object using **new JotsaAnimationObject**;
- Set the type of the object to be displayed.
- Set the path the object is to move along.
- Set the virtual time it takes to move the object.
- Insert the object in the list of displayable objects.
- Activate the object to start it moving.

When a **JotsaAnimationObject** is created, its initial position, a level number and a key are given. The **level** number determines the order in which objects are displayed and thus which objects cover other objects. The **key** can be used at a later time to destroy the object. The level and key can be omitted and JOTSA will automatically choose unique ones. An example of code to create an red oval that fits in a rectangle 100 pixels wide and 50 pixels high is given below. The oval is moved so that its center travels along a straight line from the point (150, 200) to (250, 300) in 5000 milliseconds.

```
JotsaAnimationObject obj;
obj = new JotsaAnimationObject(150, 200, this);
obj.SetFillOval(100, 50, Color.red);
obj.SetPositionCentered();
obj.PathCreateAlongLine(150, 200, 250, 300);
obj.TimesSet(5000);
JotsaInsertObject(obj);
obj.Activate();
```

A complete applet illustrating this action is about a page in length and can be found on the web [20]. A more complicated example that illustrates most types of JOTSA objects can be found in [21]. These two examples are described in [18].

To move an object and wait for it to complete its motion requires a thread, since a Java applet is not allowed to sleep. JOTSA provides a class called **JotsaWaitingThread** to simplify this operation. The user creates a thread that extends this class, creates the object, and instead of activating the object, executes: **JotsaWait(obj)**. This atomically starts the object moving and suspends the thread until the object has completed its motion. It handles the synchronization necessary to avoid the race condition in which the object finishes its motion before the thread is suspended. An example illustrating this can be found in [22].

6 The Motivating Example

JOTSA was motivated by the need to perform an animated simulation of network protocols. The concept was proven by implementing interactive animated simulations of the data link layer protocols described in a standard computer networks text [29].

Figure 2 shows the initial display for a unidirectional version of protocol 5 of [29], a sliding window protocol. The windows of the sender and receiver are shown as well as statistics for the sender and receiver. The user can pull up a control window and adjust the various parameters such as error rates and timeout values. At any time the simulation can be paused and parameters can be adjusted. In this simulation, the sender and receiver are separate, independent threads. Each thread implements its part of the protocol, and the JOTSA environment does the event handling and the animation. The code for each thread closely matches the network algorithm. The user controls the animation by making packets available to the sender from the network layer and by controlling the type of errors that occur.

Figure 3 shows a snapshot of the display after a group of frames has been sent. As each frame is sent, a JOTSA timer is set to generate a timeout event for the sender thread. The simulation does not determine at this point whether the transmission will be successful or not. The error mechanism can be independent of the transmission, and a frame or acknowledgment can be destroyed at any time due to a statistically driven automatic error mechanism or by the user clicking on a frame to destroy it while in transit. The transmission of the frame is represented by a JOTSA object in motion. When the motion finishes, the object generates an event that notifies the receiver thread of a frame arrival. The receiver then generates an acknowledgment frame. If the acknowledgment frame arrives, it generates a frame arrival event for the sender.

Since event generation can be tied to the motion of a JOTSA object, the simulation closely parallels the actual transmission of data.

For each of the six protocols that have been implemented, a number of scenarios have been developed illustrating features of that protocol. For example, in Protocol 5 the receiver has a window of size one. This means that frames cannot be accepted out of order. If a frame is lost, as is frame number 3 in Figure 3, the receiver must discard all subsequent frames and all frames after the lost one must be resent. Clicking on the **Commentary** button will bring up a running dialog box containing a commentary which is synchronized with the running of the protocol. Optionally, an audio description is available. The audio commentary is particularly effective because it allows the user's eyes to focus on the main animation display.

A drawback of the Java security model is that it only allows applets to read files from the server from which the applet was obtained. For example, if the applet resides on a machine called *appletserver*, and a user is running a browser on a machine called *appletclient*, then the applet can read files stored on *appletserver*, but it cannot read files stored on *appletclient*. The scenarios are configured by files which are read in by the applet. Generally, the user will not have direct access to the *appletserver* machine. The Java security model thus prevents the user from writing his own scenarios. To enable this, the user would have to have a web server and load the JOTSA applets directly on this web server. This would defeat some of the main advantages of using the web.

The entire application and the JOTSA environment must be loaded onto the client machine before the applet can be run. While this is automatically done when the web page is accessed, it can take a while if the network connection is slow. The JOTSA code is about 150K bytes in size and so is the application in this example. The first time it is run, 300K bytes must be downloaded. JOTSA need only be downloaded once, and additional JOTSA applets can be run without the JOTSA part being downloaded again. This example application is quite large, and a typical JOTSA applet might only be about 10-20K bytes in size.

While the audio description of the scenarios can greatly add to their usefulness, the sound files must be downloaded over the network. A typical scenario might require 10 to 20 sound files of about 20K bytes each. Thus the sound data may far exceed all of the rest of the network traffic of a particular application.

7 Performance Issues

Java performance varies considerably between platforms. Standard Java applications will run more slowly under a slower implementation, but JOTSA applications run at the same speed on all platforms as long as the display is the limiting factor to the speed.

The main consequence of slow platforms on JOTSA applications is a jerky display.When a moving object cannot be displayed at almost every pixel position along its path, its motion does not appear to be smooth. Most JOTSA applications will have a slider to control the rate of virtual time and the user can slow down the flow of time if the display quality is not sufficient. In the example shown in Figures 2 and 3, such a slider is brought up by pushing the **Controls** button.

The time it takes JOTSA to display a frame depends on a number of factors including the size of the window to be displayed and the number and types of objects to be displayed. The time for a single update is almost independent of the motion of the objects, but the speed of movement determines how often the display should ideally be updated.

JOTSA allows the display to be broken up into several rectangular pieces, called canvases. The display of the individual canvases can be either synchronous or independent. In the example in Figure 3, all of the moving objects are in a rectangular region between the sender and receiver boxes. This small area (about 150 by 150 out of 630 by 325) is a separate canvas and is the only part that needs to be updated often. The other six canvases only need to be updated when an event occurs, typically less than once a second.

The maximum frame rate for a simple JOTSA applet on different platforms is shown in Figures 4 and 5. In addition to processor type and speed, the frame rate depends on a number of factors such as the particular Java implementation, the operating system, the display hardware and the amount of memory on the target machine. The first of these

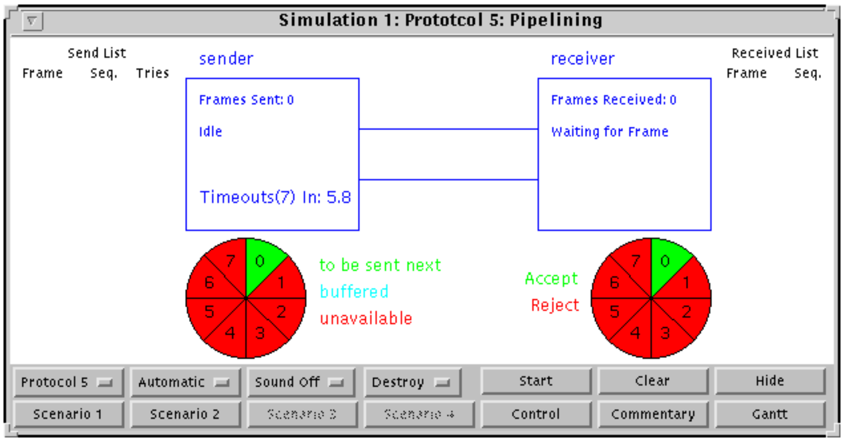

Figure 2: The initial display after protocol 5 is chosen.

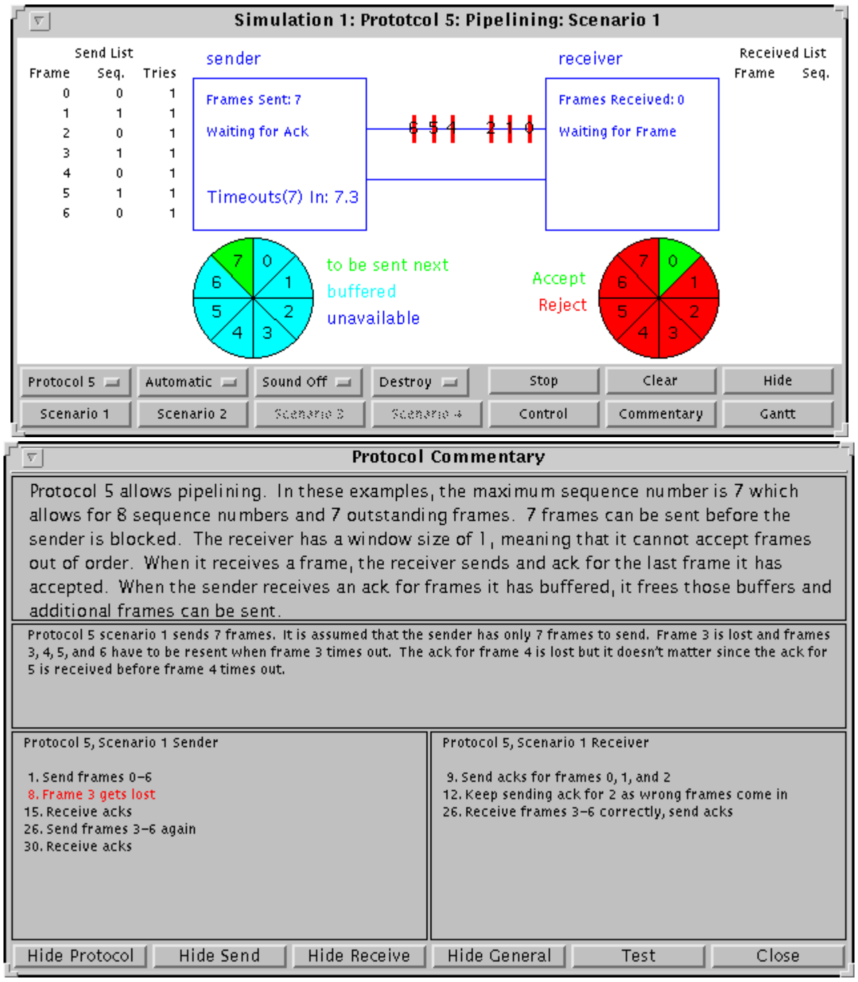

Figure 3: A snapshot of the display and a commentary dialog after several frames have been sent. Frame 3 has been lost.

	Number of Objects										
Platform	1	2	4	8	16	32	64	128	256	512	1024
Sparc LX 50 Mhz	23	22	19	16	15	13	11	6	4	2	1
Sparc 4 110 Mhz	50	49	47	43	36	28	19	11	9	6	3
Sparc 20 60 Mhz	102	95	89	80	64	59	39	23	15	8	5
Sparc Ultra 167 Mhz	158	143	95	93	61	34	40	32	18	11	9
486 66 MHz Linux	28	26	25	21	19	14	9	5	3	1.5	0.7
Pent. 100 MHz Linux	78	82	74	64	54	37	25	16	12	7	3
Pent. Pro 200 Win95	18	18	18	18	18	18	18	14	7	3	3
Pent. Pro 200 NT	93	94	94	90	93	90	89	58	40	26	14

Figure 4: A table showing maximum frame rates when using a small window size of 100 by 100 on different platforms.

	Number of Objects										
Platform	1	2	4	8	16	32	64	128	256	512	1024
Sparc LX 50 Mhz	9	10	9	9	8	7	8	5	3	2	1
Sparc 4 110 Mhz	13	14	13	13	11	11	9	6	6	5	3
Sparc 20 60 Mhz	25	25	25	25	25	25	21	19	13	10	6
Sparc Ultra 167 Mhz	25	26	25	27	28	26	17	16	11	9	7
486 66 MHz Linux	16	17	15	14	14	9	7	4	3	1.4	0.7
Pent. 100 MHz Linux	15	15	14	13	13	11	10	8	10	5	3
Pent. Pro 200 Win95	18	18	18	18	18	17	15	9	6	3	1
Pent. Pro 200 NT	14	14	15	15	14	13	13	12	11	10	8

Figure 5: A table showing maximum frame rates when using a large window size of 800 by 800 on different platforms.

figures shows the maximum frame rate for a small window of size 100 by 100 pixels, and the second one is for a large window of size 800 by 800. The larger window represents 64 times as many pixels as the smaller one. In most cases for the small window, the number of objects is the main determining factor on the display rate. For the larger window, the number of objects does not significantly affect the display rate until it exceeds some threshold.

The Pentium Pro platform is an exception to this. While it is faster than most of the other systems as shown by its NT performance, the Pentium Pro under Windows 95 has a maximum frame rate of less than 20 frames per second, even for one object in a small window. This surprising result was traced to the Java time function **System.currentTimeMillis()**. This function is supposed to return the system time in milliseconds, and JOTSA uses it to compute when to issue the next display request. Consecutive calls to this function should return values which differ by 0 or 1, and this behavior was confirmed on the Sun systems and on the Intel systems running Linux. Under NT differences were either 0 or 10. However, under Windows 95 the nonzero differences are either 50 or 60. This coarse granularity of time prohibits JOTSA from making the precise calculations needed for higher frame rates.

8 Discussion

Prior to the introduction of Java, many web-based animations used a model of execution based on the X Window System [9]. In this model the software runs on the remote machine (the X client), and the display appears on the local machine (the X server). There are three major disadvantages to this model. The computing burden is on the remote machine requiring the software provider to supply sufficient computing power for all users. Secondly, the system puts a heavy load on the network while the programs are running, making the speed of the animation dependent on the network traffic. Thirdly, while X servers can be obtained for most systems, they are not normally installed on the most ubiquitous machines, those running Microsoft operating systems.

The Internet community has focused on Java as the language for the web. Java holds the promise of platform independence based on a model of compile once, run anywhere. To some extent this has been already achieved with the core of the Java language. In the Java model, the program is compiled into an intermediate form and stored on the remote machine (the server). It is downloaded to the local machine (the client) when it is accessed. The Java program is run on the local machine, usually with an interpreter. Once the program has been downloaded there is no longer any demand put on either the remote machine or the network. Since the Java program is interpreted (or compiled on the fly at run time), the same program will run on any system. However, many problems still exist before true platform independence is achieved.

While the Java Application Window Toolkit (AWT) has the same features on all platforms, the look and feel varies. The differences may actually be desirable under some circumstances, because the Java environment behaves in a way that is familiar to the user on a particular platform. However, certain aspects of this variability make it difficult to achieve a satisfactory appearance on all platforms.

Fonts pose a particular problem when there is a need to place text accurately among other displayed objects. The size and shape of characters is different on different platforms, since Java uses the text capabilities of the underlying window environment. Consequently, the same character string will take up a different amount of space when viewed on different platforms. In Java, the width of a box is specified by a number of pixels, while the width of a string is specified by the font style and size. A Java program can determine the width in pixels of a given string in a given font, but there is no convenient method for ensuring that a string will fit inside the box boundaries, other than by trial and error.

Sound support in Java is rather rudimentary. While audio files can be played, audio control is limited. Sound clips can be started and stopped, but there is no convenient way to tell when a sound clip is finished. This makes it difficult to properly sequence sound clips. A solution to this problem has been promised in the upcoming Java Media Toolkit.

Another more important problem for animation is that the speed of the hardware and the efficiency of the Java runtime environment affect the speed at which Java programs run. Unless care is taken, motion which is very slow on one machine will be very fast on another. JOTSA addresses this aspect of platform-independent web-based animation. It is particularly difficult in Java, since the user does not have direct control of the repainting of the display. JOTSA has been shown to be a powerful tool for web-based simulation and animation of algorithms and physical processes. It has also been used for animation of data in systems where exact time is a critical feature.

References

[1] R. S. Anir, J. A. Miller and Z. Zhang, "Java-based query driven simulation environment," *Proc. 1996 Winter Simulation Conference,* pp. 786–793, 1996.

[2] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[3] J. E. Baker, I. F. Cruz, G. Liotta and R. Tamassia, "Algorithm animation over the world wide web," **http://www.cs.brown.edu/people/jib/Papers/mocha.ps**.

[4] J. Banks, S. Carson, and J. N. Sy, *Getting Started with GPSS/H*, Wolverine Software Corporation, Annandale, Va, 1989.

[5] M. H. Brown, *Algorithm Animation*, MIT Press, Cambridge, 1988.

[6] S. Bryson and Sandy Johan, "Time management, simultanaeity and time-critical computation in interactive unsteady visualization environments," *Visualization 96*, pp. 255-261, 1996.

[7] A. H. Buss and K. A. Stork, "Discrete event simulation on the world wide web using Java," *Proc. 1996 Winter Simulation Conference,* pp. 780–785, 1996.

[8] S. W. Cox, "GPSS world: A brief preview," *Proc. 1991 Winter Simulation Conference*, pp. 59–91, 1991.

[9] E. Cutler, D. Hilly and T O'Reilly, *The X Window System in a Nutshell, 2nd edition,* O'Reilly and Associates, Inc, 1992.

[10] N. J. Earle and J. O. Henriksen, "The power and performance of PROOF animation," *Proc. 1995 Winter Simulation Conference,* pp. 494–501, 1995.

[11] M. Folk and R. E. McGrath, "The horizon project," *Federal Webmasters Workshop,* Aug. 7, 1996. **http://hdf.ncsa.uiuc.edu/horizon/Webmaster.7.Aug.96/**.

[12] T. A. Funkhouser and C. H. Sequin, "Adaptive display algorithm for interactive frame rates during visualization of complex environments," *Computer Graphics: Proceedings of SIGGRAPH 93*, pp. 247–254, 1993.

[13] B. Myers, "Taxonomies of visual programming and program visualization," *J. of Visual Languages and Computing,* 1, pp. 97–123, 1990.

[14] A. A. B. Pritsker, and J. J. O'Reilly, "AweSim: The integrated simulation system," *Proc. 1996 Winter Simulation Conference,* pp. 481–484, 1996.

[15] D. M. Profozich and D. T. Sturrock, "Introduction to SIMMAN/Cinema," *Proc. 1995 Winter Simulation Conference,* pp. 515–518, 1995.

[16] K. A. Robbins and S. Robbins, *Using exact time animation to show nonperiodicity,* UTSA Computer Science Technical Report, CS 97-4, 1997.

[17] S. Robbins, *A microprogramming animation,* UTSA Computer Science Technical Report, CS 95-10, 1995. **http://vip.cs.utsa.edu/personnel/srtechreps.html**.

[18] S. Robbins, *A JOTSA example,* UTSA Computer Science Technical Report, CS 96-13, 1997. **http://vip.cs.utsa.edu/java/jotsahome/**.

[19] S. Robbins, *JOTSA collections,* UTSA Computer Science Technical Report, CS 97-5, 1997. **http://vip.cs.utsa.edu/java/jotsahome/**.

[20] S. Robbins, "A simple applet which moves an oval along a line," **http://vip.cs.utsa.edu/java/jotsahome/**.

[21] S. Robbins, "An applet illustrating many JOTSA features," **http://vip.cs.utsa.edu/java/jotsahome/**.

[22] S. Robbins, "An example illustrating splits and merges," **http://vip.cs.utsa.edu/java/jotsahome/**.

[23] E. C. Russel, "SIMSCRIPT II and SIMGRAPHICS tutorial," *Proc. 1993 Winter Simulation Conference,* pp. 223–227, 1993.

[24] T. J. Schriber, *Simulation using GPSS*, John Wiley, New York, 1974.

[25] J. T. Stasko, "The path-transition paradigm: A practical methodology for adding animation to program interfaces," *J. of Visual Languages and Computing,* 1, pp. 213–236, 1990.

[26] J. T. Stasko, "Animating algorithms with XTANGO," *SIGACT News,* 23(2) pp. 67–71, 1992.

[27] J. T. Stasko and E. Kraemer, "A methodology for building application-specific visualizations of parallel programs," *J. Parallel and Distr. Computing*, 18(2) pp. 248–264, 1993.

[28] J. T. Stasko and D. S. McCrickard, "Real clock time animation support for developing software visualizations," *Australian Computer Journal*, 27(3) pp.118–128, 1995.

[29] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, Third Edition, 1996.

[30] "The horizon image data browser," **http://imagelib.ncsa.uiuc.edu/imagelib/Horizon/**.