Performance Measurement Capabilities of VR Juggler: Real-time Monitoring of Immersive Applications

Christopher Just, Carolina Cruz-Neira, Albert Baker

(cjust@vrac.iastate.edu, cruz@iastate.edu, baker@cs.iastate.edu)

Virtual Reality Applications Center

Iowa State University

Abstract

Performance is one of the most critical aspects of creating an immersive application, affecting the comfort of users and their ability to interact with the environment itself. This paper discusses the importance of including application-extensible, real-time-viewable performance monitoring capabilities in Virtual Reality (VR) development systems. It describes how an extensible performance-monitoring system was designed and implemented for the VR Juggler[1] platform, and shows some of the ways performance monitoring has aided both the developers of VR Juggler itself, and those writing applications for it.

1 Introduction

Performance is a critical concern in the creation of immersive environments. A poor frame rate can reduce the most elaborate display system to a slide show, and high latencies can make it difficult to interact with a virtual environment. Overall performance is a combination of many factors, some of which are application-specific – the complexity of a scene to be rendered, for example. Yet our underlying toolkits – the building blocks for creating Virtual Reality (VR) applications – can also affect the performance of the application as a whole. For example, the system we use might impose a certain amount of per-frame overhead, or it might require serialization of certain tasks.

Optimizing an application is difficult without knowing where the performance bottlenecks lie. Therefore we must be able to measure the impact of our toolkits, libraries, or development systems. However, many traditional profiling tools are poorly suited to the heavily-threaded, multiprocessed, and highly interactive software of a typical VR application[2]. If these systems can provide capabilities for analyzing the performance of user code in our applications, so much the better.

There has been relatively little work to provide "embedded" tools for performance analysis of VR applications. Our research focuses on the design and implementation of performance monitoring capabilities in VR Juggler, a toolkit for writing immersive applications developed at Iowa State University's Virtual Reality Applications Center (VRAC). This paper also discusses some of our initial findings, as well as examples of how VR Juggler's performance monitoring capabilities have aided its own debugging and development process.

2 The Importance of Performance Measurements

There are many reasons that the performance of immersive VR applications is critical. Some are technical, others related to the cost of a system. Many affect the overall usability of an application or system.

Level of Engagement. VR applications need to be interactive and convincing – this is especially true for training applications, where users might become frustrated with an environment that

doesn't respond quickly to their commands. Moreover, a simulation that responds differently than the real world might cause the trainee to develop bad habits that do not transfer out of the simulator. In entertainment applications, users might simply grow bored with a poor framerate or slow movement. Users of scientific applications might be more forgiving of computational delays, but are still unlikely to use an immersive application that is much slower than a nonimmersive counterpart.

Physical and Psychological Impact. The effects of poor performance on a user can go far beyond simple inconvenience. For example, high latencies can be a contributing factor to "cybersickness" – the motion sickness-like nausea that affects some users of virtual environments [4]. We must remember that our applications can impact the comfort and health of our users.

Optimization. A thorough understanding of an application's performance can be a valuable tool for improving that performance. Optimizations to an application can reduce its hardware requirements, resulting in a significant economic benefit – especially for applications that will be widely deployed.

Debugging. Performance analysis can also be a tool for debugging. A typical VR application can involve numerous processes or threads with various timing requirements, update rates, and synchronization issues. By providing a display of "what happens when", performance data can improve the developer's understanding of such a complex system.

2.1 Throughput and Latency

The performance measurements we might wish to take in an immersive virtual environment can be divided into two broad categories: throughput and latency.

The most obvious example of throughput is the graphics framerate of the application. A sustained rate of 30 frames per second (fps) is generally considered adequate to create an illusion of smooth motion. Below this, viewers will begin to perceive frames individually. Very low frame rates can make it difficult to interact with objects in the environment, and navigation can become disorienting.

In computer graphics, the problem of achieving a particular frame rate is complicated by the need to synchronize buffer swapping with the video signal (typically between 60 and 120 Hz). This can exaggerate the effect of a small variation in rendering speed. Under certain circumstances, a small increase in the time needed to render a new frame can cut the overall framerate in half.

VR applications – particularly those based on multiple-display projection systems – complicate the matter further. The application may need to render a scene multiple times – stereoscopic pairs of images for each display – and synchronize them all to swap buffers at the same time. In order to understand the performance of such a system, we need to know how much time is spent in application code (scene rendering for each display, physics simulation, etc.), how much time we lose waiting for video buffer swaps and other synchronization points, how much time the system spends on input updates and viewing matrix computation, and so on.

In addition to simple throughput, we are also concerned with issues of latency. In the simplest terms, latency is the time it takes the virtual environment to react to the user. If the user turns his head, how long does it take the view to adjust? If the user opens his hand, how long does it take before the ball he was holding starts falling to the floor? The latency of an application can be affected by many factors. The input hardware devices typically have some lag internal to their design; further delays can be introduced in the communication between the input device and the host computer (usually a serial or network connection). Finally, once the application has the data, it probably will not be used until the beginning of the next frame drawing step.

Since a portion of this latency is internal to the input device hardware, it is difficult to measure accurately in software[5]. We can timestamp the data when we start receiving it from the hardware, and set an additional timestamp once we have used that data to draw a frame and that frame has been displayed. Unfortunately, this does not include the processing delays internal to the input device itself.

2.2 The Need for Built-in Tools

There are many ways of measuring performance, and there are already a variety of profiling tools with various approaches and capabilities. Many of the commonly available profiling tools are poorly suited to the heavily-threaded, interactive nature of a VR application[2]. For example, knowing the average or worst-case frame rate of an application is insufficient – it is also vital to know if the frame rate drops when the user looks at a

particular part of the application's geometry or manipulates a certain object. Many more detailed tools could overwhelm the application developer with information about the complex inner workings of the underlying VR toolkit. In addition to gathering the necessary data, we must present it to the application programmer in a meaningful way, without drowning them in unnecessary detail.

This problem is emphasized in projection-based environments. For example, ISU's C6 draws six stereo pairs of images for each frame, and each screen uses a dramatically different view. The individual drawing times required for each screen might vary radically, depending on the application and the user's position in the virtual world. This makes it even more important to be able to match timing information with particular threads, graphics contexts, and so on.

Despite the difficulties of using conventional profiling tools with VR applications, there has not been a great deal of research on adding performance monitoring to VR toolkits themselves. The CAVE software library[6] can provide the current frame rate to the application. MR Toolkit[7] also provides support for collecting user-placed timestamps to measure performance. Perhaps the best example is SGI's Iris Performer software, which can acquire highly detailed performance information about graphics operations and display it interactively[3]. However, Performer is primarily a visual simulation package, and by itself lacks the interaction capabilities required for VR application development.

By including performance measurement capabilities in the development system itself, we can provide application developers with the information they need, in a format they can understand. If these capabilities are designed to be extensible, they can also interface with some of those other tools, using them where available and useful and providing all of that information to the developer as well.

3 VR Juggler – A Virtual Platform for VR Application Development

VR Juggler[1], developed at ISU's Virtual Reality Applications Center, is a recent addition to the ranks of VR development systems. As of January 2000, VR Juggler is available to the public under the GNU Library General Public License. It was designed to provide cross-platform support for a large array of VR hardware, including extremely flexible support for projection-based displays. It includes a set of performance monitoring capabilities designed around VR Juggler's central philosophies: dynamic control, extensibility, and portability.

VR Juggler attempts to provide a "virtual platform" for application development. Operating System and hardware specific details (process/thread management, input device access, window and display creation, etc.) are hidden behind common, abstract interfaces. In addition to facilitating portability, these abstractions allow us to perform dynamic reconfiguration – changing the configuration of a program without quitting or restarting it. For example, an application running on a single screen with simulated devices could be reconfigured to use a multi-screen stereo projection system with wireless trackers, and back again, with minimal interruption.

Given the level of flexibility and the layers of abstraction in the VR Juggler design, it should be clear that performance is a critical concern. No level of abstraction comes for free, and so the VR Juggler development team has needed to carefully weigh the performance impact of each of these design decisions. The abstractions have been kept as lightweight as possible. Features of C++ which could cause performance problems, such as virtual functions and exceptions, have been used sparingly.

To understand how the VR Juggler performance measurement system works, and to appreciate the resulting data, some understanding of VR Juggler's overall design is necessary. Figure 1 shows an overview of the major components, which are briefly described below.

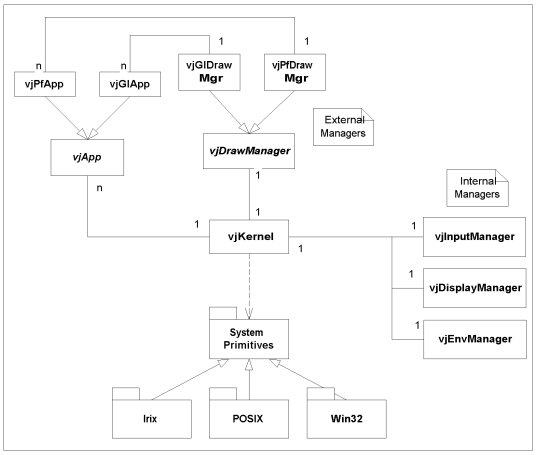

Figure 1. VR Juggler structural overview

3.1 Input Manager

The Input Manager owns and controls access to all input devices. Many of the Input Manager's device drivers will spawn threads to perform I/O. Luckily, most of these threads will require fairly few computational resources – for example, a device driver responsible for reading data from a low-speed serial port will spend most of its time waiting. Nevertheless, the total impact of multiple device drivers needs to be considered.

3.2 Display Manager and Draw Manager

The Display Manager component of VR Juggler is responsible for creating and managing graphics windows and drawing threads. The Draw Manager works in concert with the Display Manager, and provides functionality specific to the graphics API in use (currently OpenGL or Iris Performer).

VR Juggler provides considerable flexibility for creating multiple display threads, and assigning graphics windows to them. In a multiprocessor system, it might be reasonable to create a separate thread for each graphics window to make best use of the available processors, but having multiple threads accessing the graphics hardware simultaneously might actually decrease drawing speed.

There are a number of performance issues in the display loop. First, how many times does the application-supplied draw function get called, and how long does it take each time? How much time is spent waiting for synchronization and other overhead?

3.3 Environment Manager

The primary responsibility of VR Juggler's Environment Manager component is to receive dynamic reconfiguration commands from the outside world. It creates a network socket through which other applications (such as our VjControl GUI tool) can query the current configuration and send instructions for changing the configuration.

The Environment Manager is also responsible for collecting performance data from the rest of the VR Juggler system. It can send that data to an external program like VjControl or write it to a file for later evaluation.

3.4 Kernel

VR Juggler has a small Kernel object which acts as a central communication broker between the application and its various components. The Kernel also contains a single thread which controls several aspects of the system, such as triggering drawing and input device updates. The Kernel thread is responsible for a small amount of per-frame overhead which must be measured and, if possible, minimized.

3.5 VjControl

VjControl is the counterpart to VR Juggler's Environment Manager component. It provides a graphical user interface (GUI) written in Java for portability. VjControl allows users to connect to VR Juggler applications and view and modify their configurations. It also allows users to collect and display performance information interactively.

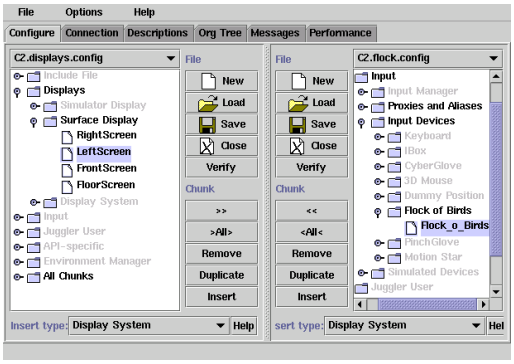

Figure 2. VjControl, VR Juggler's configuration and monitoring tool.

4 Performance Measurement Capabilities of VR Juggler

One of the major goals of VR Juggler is extensibility. In the case of its performance monitoring features, this means that support for new measurement techniques can be added easily. In the simplest case, the application developer can add performance measurements inside the application code – breaking the draw function into logical pieces, or measuring the time required by various aspects of a physics modeling routine. However, it is also possible to add completely new kinds of performance information to the system. Sources of performance information can register with the Environment Manager, which collects the data and writes it to a file, or sends it across a network socket to the VjControl program, which can display it interactively. Developers can write new modules for VjControl to display these additional performance metrics.

Portability is another key concern. VR Juggler has been compiled and used on more than half a dozen platforms – notably Irix, Windows NT, and

Linux. The collection of timestamp data, detailed below, is a good example of how VR Juggler compromises between portability and quality.

Current work on the VR Juggler performance monitoring abilities concentrates on two areas. The first is providing a detailed breakdown of time required per frame by various parts of each VR Juggler component (Display Manager threads, Kernel threads, etc.). The second is providing a method for measuring input device latency (timestamping input samples when they are received in order to calculate their age when used).

4.1 High-Precision Timestamps

The support for high-precision timing varies from one platform to another. The resolution of the available timing facilities varies, as do the access methods and underlying implementations. There are portable solutions, but these are seldom the best (most optimal or most precise) available.

In VR Juggler, these sorts of nonportable annoyances are hidden behind abstract interfaces. In the case of timer access, the interface is a class called `vjTimeStamp`. An instance of `VjTimeStamp` stores a single timing datapoint, expressed in microseconds since the application was initialized. The class includes facilities for recording the current time and comparing and subtracting timestamps. `VjTimeStamp` also stores information about the resolution of the timer, if available.

In C++, the usual way to provide several variant versions of a single class interface is to use inheritance and virtual functions. Because of concerns about virtual function call overhead (and because typically only one such class can compile on a particular architecture anyway), VR Juggler instead uses the technique of writing multiple unrelated classes, implementing identical interfaces, without using virtual function calls. When the VR Juggler library is compiled, a single class is chosen and `typedef`ed to the generic name (e.g. `vjTimeStamp`).

Three versions of `vjTimeStamp` have been written. The first is simply a dummy class – `vjTimeStampNone` – which can be used on any system. It has no actual capabilities, but is useful when porting VR Juggler to an operating system where none of the expected timing facilities exist.

`VjTimeStampPosix` is a portable (across POSIX systems) version which uses the standard `gettimeofday()` function. Unfortunately, while widely supported, the precision of `gettimeofday()` varies widely between systems ("never worse than 10 milliseconds" according to several systems) and is often not very good at all.

A better solution exists on most SGI systems, and is taken advantage of by the `vjTimeStampSGI` class. The *cycle counter* is a high-precision hardware timer built into the processors of most SGI systems. The resolution of this timer varies per system, but can be queried at runtime – typical resolutions for recent hardware are in the range of 20-80 microseconds. When VR Juggler is initialized, the cycle counter is mapped into application memory, where it is read whenever a time sample is taken.

4.2 Storing and Collecting Performance Data

Ultimately, the performance data we collect needs to be gathered and written to a file or network port. We wish to minimize the performance impact of that I/O, keeping it in a separate process, preferably running on a separate processor.

`VjPerfDataBuffer` is an example of a performance metric data source, designed to collect multiple uniquely-indexed timestamps from a thread's execution loop. For example, we might want to collect timestamps at several points inside the main loop of the VR Juggler kernel, which executes once per system frame. We want to store a timestamp at the top of the loop and another after swapping graphics buffers. The time between the first and second timestamps is the time spent drawing the frame. The time from the second stamp to the top of the next loop is spent on other tasks such as updating the tracker information and viewing transformation.

The kernel creates an instance of the class `vjPerfDataBuffer`, which is essentially a circular array of `vjTimeStamp`s. Each buffer entry also has an index associated with it, which lets us distinguish between the various places in the loop where we collect our timestamps.

At the top of the kernel loop, we take a sample with an index of zero, and after swapping buffers we take another with the index set to one.

Periodically, another process will read the data from the `vjPerfDataBuffer` and send it to a file or a network port. This reader process is created by a `vjTimedUpdate` object, which is part of VR Juggler's Environment Manager.

It is possible for the process writing to the `vjPerfDataBuffer` to outrun the `vjTimedUpdate` process and run out of room in the buffer. When this happens the timestamp set function will return without storing any value. The buffer does keep track of the number of entries that are lost in this fashion.

4.3 Storing Latency Information

The drawing threads created by VR Juggler's Display Manager are also instrumented to provide input latency data. When the Input Manager reads a new sample from an input device, it takes a timestamp and stores it alongside the sampled data. When that input data is used, its age can be compared with the current time.

This latency data can be stored in a `vjPerfDataBuffer`. When we wish to take a latency measurement, we copy the input data's timestamp into a buffer element and follow it with the current time. Subtracting these values gives us the age of the tracker data.

In the VR Juggler display system, we record the tracker latency at the beginning and end of drawing each graphics window.

4.4 Viewing the Data

Once VjControl has read performance data from a socket or from a log file, it needs to display it to the human user in a meaningful way. The simplest way is to simply present the average times for each part of each loop over a given period of time. Figure 3 shows such a data summary as presented by VjControl.

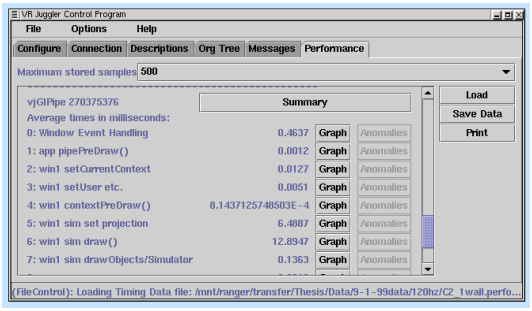

Figure 3. VjControl performance data summary for the main loop of a VR Juggler drawing thread.

Of course, this display is not always sufficient. Sometimes the mere averages can oversimplify the reality of an application's performance, hiding important facts. For example, sometimes the variation in timings can be important. VjControl can also graph the individual timing samples for the interior of a loop over a period of time, as illustrated in Figure 4:

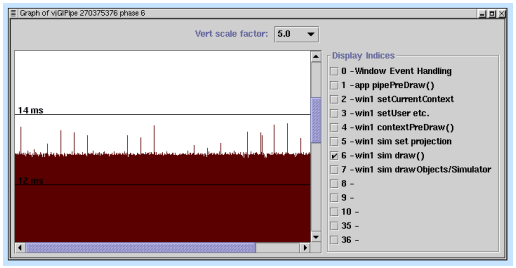

Figure 4. Graph of per-frame timings for the application-supplied draw() function over the course of several hundred frames.

The figure shows a graph of the time spent in the application's `draw()` function. This uses the same data set as shown in Figure 3, and we can see that the time required for most calls to `draw()` is about 12.9 milliseconds, as the average reports. But in the graph we can also see spikes where a single call to the `draw()` function can take significantly longer. The graph also suggests a pattern to the anomalies, which appear to occur at 250 ms intervals (but not with complete regularity). This information could be useful for tracking down the cause of the anomalies, which could be internal to the application (such as an extra computational step taken once every 20 frames) or something in the system itself (for example, an interrupt being called, or another task interfering with our drawing thread).

5 Initial Tests and Surprises

The VR Juggler performance testing experiments uncovered a bug in the Juggler implementation before the timing code was even completed. The first run of timing measurements in the full immersive environment – ISU's C2 – was meant as a pilot test of the data buffers in a multiprocessing environment. The most that was hoped for was to gather sample data for experimenting with graphs and visualization.

The test used the C2's tracking system and wand for input. The number of active displays windows was varied, but each window displayed the same full-screen stereo view. The test was merely designed to record the frame rate with each configuration. This is a summary of results:

4/19/99 Test	One Window	Two Windows	Four Windows
Time per frame	33.33 ms (30 fps)	78.35 ms (12.8 fps)	133.99 ms (7.5 fps)

The 30 frames per second measured in the first test is a reasonable figure, based on the amount of geometry the test program was rendering. However, as additional windows were added, the performance dropped precipitously. Comparing these results with our expectations, it appeared that something was seriously wrong.

The machine used for this test had two separate graphics engines, and twelve processors. VR Juggler was configured to use two drawing processes – one driving each graphics board. In run 1, one process drew the single window, and the other was idle. In run 2, each process was configured to draw one window, and in run 3, each process was responsible for two windows. The expected results were that run 1 and run 2 would have approximately the same frame rate, and that run 3 would be slower.

The experiment showed a severe degradation when a second window was added, considerably in excess of the additional amount of work required to render it. The lower frame rate was visible to the naked eye and verified with a stopwatch. The problem was also seen with a backup copy of the VR Juggler library compiled without the performance measurement code – showing that the performance code was not itself causing the problem.

Looking at the collected data more closely, it was found that the frame drawing times in each run are fairly consistent. There were wide variations during the first few seconds of execution, as Juggler loaded and configured itself, but this data was omitted from all of the averages.

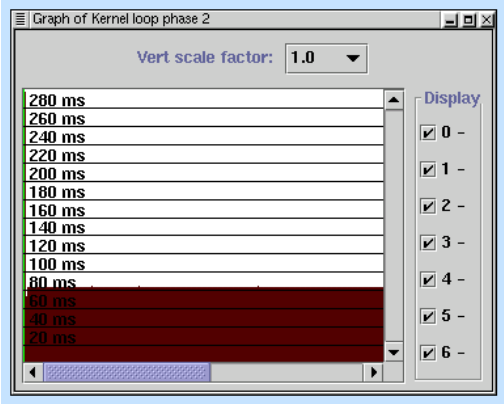

Figure 5. Framerate variation at application startup. Note the single 295 ms sample on the far left.

For the bulk of the run, there was only a small variation from one frame to the next, and a handful of larger variations, likely due to other system activity. The averages were an accurate measure of the time needed to draw each frame.

The timestamp collection placement was refined, and a second test indicated that the slowdown was entirely limited to the application's drawing routine. This indicated that the delay was not caused by any of the (known) synchronizations inside VR Juggler.

The next step was to modify the Juggler configuration. The program was tested with simulated devices and with monoscopic instead of stereo displays with no change in the results. Rendering multiple windows with a single drawing thread did not evidence the problem.

Finally, we compared tests with the OpenGL drawing code using display lists (as the application was originally written) and without, and discovered that the unusual behavior only happened when using display lists. With this hint, one of the VR Juggler developers was able to pinpoint the exact cause of the delay.

All memory in VR Juggler applications is shared between processes. While generally this simplifies programming, it means that a special interface is needed to access data specific to a particular OpenGL context, such as display list indexes. The original implementation of this interface used a semaphore that had to be acquired every time the context-specific data was accessed, and in the test application this happened approximately 2000 times per window per frame – with disastrous results. The interface was rewritten to go through the synchronization point once per context per frame, and the following results were recorded:

5-20-99 tests	**One Window**	**Two Windows**	**Four Windows**
Total Frame Time	33.33 ms (30.00 fps)	33.33 ms (30.00 fps)	66.67 ms (15.00 fps)
Per draw( )	12.88 ms	12.96 ms	12.81 ms

With VR Juggler now reacting as expected, and with the performance testing code substantially debugged in the process, work could proceed. The moral is that code hidden behind a layer of abstraction really can hurt you.

6 VR Juggler Performance Summaries

In this section we present a detailed look at the performance of a particular VR Juggler

application under a number of configuration and compilation conditions. The system used is that described in the preceding section: ISU's C2 device, attached to a computer with twelve processors and two graphics cards (each driving two displays).

The test application used is one of the sample applications included in the VR Juggler software distribution, which draws a mostly static environment using a large number of untextured polygons. Note that these results are illustrative, and not definitive. The goal is to present an overview of how VR Juggler's performance can be affected by configuration options and the demands of various VR systems, and to show how some of the possible compilation options on one platform can alter overall performance.

6.1 Basic Tests

The first set of tests was performed with a stock Juggler build. The number of active C2 walls was varied to simulate the behavior of an application on single- or multiple-screened devices.

Timing information was gathered for the Kernel loop and the main loop of each drawing thread. In the two-window and three-window tests, only one of the two drawing threads is shown for clarity; the other showed similar results. All times are in milliseconds.

# Windows	One	Two	Four
Kernel Loop			
App preFrame	0.039	0.040	0.038
Trigger draw	0.049	0.043	0.045
App intraFrame	0.001	0.001	0.001
Draw sync	41.367	41.340	62.183
App postFrame	0.005	0.007	0.006
Check reconfig	0.008	0.008	0.009
Update input devs	0.161	0.173	0.214
Update frame data	0.035	0.055	0.107
Draw Process #1			
Window events	0.518	0.597	0.756
App pipePreDraw	0.009	0.010	0.08
First Window			
Prepare	0.036	0.037	2.101
Left setProjection	12.225	12.246	0.688
Left draw	14.150	14.017	13.902
Left drawObjects	0.135	0.165	0.161
Right setProjection	0.023	0.022	0.024
Right draw	14.178	14.055	13.850
Right drawObjects	0.123	0.157	0.147
Second Window			
Prepare			0.152
Left setProjection			0.816
Left draw			13.890
Left drawObjects			0.159
Right setProjection			0.024
Right draw			13.831
Right drawObjects			0.153
Sync	0.037	0.043	0.014
SwapBuffers called	0.236	0.326	2.615
Total Frame Time	41.666	41.666	62.603

The very similar performance between the one-window and two-window tests is noteworthy. It shows how adding another draw process, running on a separate processor and accessing a separate graphics board, has very little overall impact on VR Juggler's performance.

Adding a second window to each drawing thread does impact performance, as expected, though it is interesting to note that in this case the program wastes less time waiting for buffer swaps. It does twice as much work, but the frame rate only drops from 24 to 16 fps. Adding the second window does appear to impact some of the other aspects of the system such as, reasonably enough, handling X events associated with the windows.

6.2 Using Additional Draw Processes

In the third test show above, we split four graphics windows between two drawing threads, out of concern that having multiple threads attempting to access the same graphics hardware simultaneously would cause contention. VR Juggler's configurability allows us to test that assertion directly.

In a test conduced with four graphics windows active, divided among four display threads, the time required for each call to the application draw

function increased slightly (into the 15-16 ms range).

Unfortunately, large delays appeared elsewhere – in waiting to swap buffers and when first accessing the graphics hardware for each OpenGL context. Overall framerate dropped to 14.3 fps. Additionally, examination of the individual data showed a large number of single-frame anomalies in various stages of the drawing threads.

For this application, at least, it appears that one drawing thread per graphics card is the optimal configuration. It is possible that a program with a more CPU-bound drawing routine would have different results.

6.3 Compilation Variations

VR Juggler has several features that can be selected at compilation time, including the threading model and instruction set to use. On the SGI platform, we can select between POSIX threads and IRIX `sproc()` processes, and also between the MIPS3 and MIPS4 instruction sets.

The tests in the previous sections were all done using `sproc`-based versions of the application. A POSIX threads version of VR Juggler was tested using the same configuration as in the four-window test in section 6.1. The results for the drawing threads were interesting:

Draw thread (first window only)	**POSIX**	**SPROC**
Prepare	8.374	2.101
Left set projection	0.698	0.688
Left draw	6.543	13.902
Left drawObjects	0.582	0.161
Right set projection	0.017	0.024
Right draw	6.120	13.850
Right drawObjects	0.508	0.147

The improvement in time required for application draw is very significant. Although this is mitigated somewhat by increases in other parts of the code, it still resulted in an easily noticeable increase in the frame rate – from 16 to 24 fps.

When compiling VR Juggler on IRIX systems, we can also select between the MIPS3 and MIPS4 instruction sets. We expect better performance with MIPS4, but it is only available on more recent machines. Compiling with MIPS3 would allow us to use the same binary on all machines in our lab, but at what performance cost on the higher-end machines?

The tests in 6.1 were conducted with a MIPS4 version of the library. Here we compare the four-wall configuration with a MIPS3 binary:

Draw thread (first window only)	**MIPS3**	**MIPS4**
Prepare	18.701	2.101
Left Set projection	0.711	0.688
Left draw	14.716	13.902
Left drawObjects	0.164	0.161
Right Set projection	0.022	0.024
Right draw	14.732	13.850
Right drawObjects	0.152	0.147

As can be seen, the MIPS3 version of the application is slightly slower at drawing. Unfortunately, in this instance the increase was enough to force a significantly longer wait before the application could swap buffers (noticeable under the "Prepare" line). Overall performance of the MIPS3 application was 12 fps, as opposed to 16 fps for the MIPS4 version.

6.4 Latency Variations

All of the above tests also gathered latency data for the user's head tracker, provided by a magnetic tracking device with a serial connection to the test computer. Latency measurements were taken in the drawing threads just before and just after drawing each window.

VR Juggler triple-buffers most input data, so that the same data sample is used throughout the generation of a single frame. This results in certain obvious characteristics: latency reported at the end of drawing is the same as that before plus the time required to draw, and the second window drawn by a thread has higher (initial) latency than the first. While it is the final value reported that is relevant to the end user, we will focus on the first value reported for each frame, and ignore all costs associated with drawing.

In general, initial latencies reported in these tests fell in the range of 12.5 to 16.5 ms, with the slower tests reporting higher average latencies. Far more interesting was the variation in latency measurements within a given test run. Figure 6 is typical:

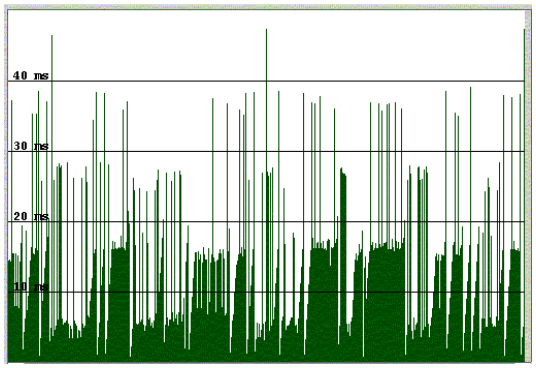

Figure 6. Variation in tracker latencies.

The variation we see, and the pattern, is related to the asynchronous nature of the I/O thread which reads data from the tracker. This degree of variability could result in a certain "jerkiness" of the environment's response to head motion, though it is too subtle to be easily noticed with the naked eye.

The latencies of simulated input devices provide an interesting contrast. VR Juggler's keyboard-based simulators are synchronous, processing data only when the Kernel thread's input update routine is executed. This results in very regular sub-millisecond reported latencies – but unfortunately does not count latencies in the windowing system's keypress reporting.

7 Conclusions and Future Work

While the framework for detailed performance measurements in VR Juggler is in place, and the initial results have already been helpful to its developers, there is much work left to do.

On the SGI IRIX platform, VR Juggler needs to be tested with a greater variety of applications and configurations. In particular, the tests discussed above always involved fewer threads than available processors.

There are also other platforms to consider. There is considerable interest in VR Juggler's performance on multiprocessor Linux and Windows NT systems, though so far we have not had adequate available hardware for that testing.

It must also be noted that the performance framework is designed to be extensible. In the future it could include tighter integration with existing system monitoring tools and the performance capabilities of the underlying hardware and software.

Nevertheless, VR Juggler's integrated performance metrics have already proven their value. By intelligently placing measurement points in library and application source code, developers can acquire an easily understood picture of program performance, without sacrificing useful detail. By collecting and displaying performance data interactively, they can better understand how their applications react to user commands.

Accurate performance information can be used to optimize applications, to influence hardware selections, or to measure the impact of a virtual environment on users. Measuring performance today is a first step toward improving performance tomorrow.

References

[1] http://www.vrjuggler.org

[2] Sharon Rose Clay, "Optimization for Real-Time Entertainment Applications on Graphics Workstations", *Designing Real-Time 3D Graphics for Entertainment*, Siggraph 95 course.

[3] J. Rohlf and J. Helman, “IRIS Performer: A High-Performance Multiprocessing Toolkit for Real-Time 3D Graphics.” Proc. Siggraph 94, ACM Press, New York, 1994, pp. 381-394.

[4] Randy Pausch, Thomas Crea, Matthew Conway, "A Literature Survey for Virtual Environments: Military Flight Simulator Visual Systems and Simulator Sickness." *Presence* 1(3), MIT Press, 1992, pp. 344-363.

[5] Mark R. Mine, "Characterization of End-to-End Delays in Head-Mounted Display Systems." Technical Report TR93-001, University of North Carolina at Chapel Hill Department of Computer Science, 1993. ftp://ftp.cs.unc.edu/pub/publications/techreports/93-001.ps.Z

[6] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE." ACM SIGGRAPH 93 Conference Proceedings. 1993, pp 135-142.

[7] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun, “Decoupled Simulation in Virtual Reality with the MR Toolkit.” *ACM Transactions on Information System*s, Volume 11, Number 3: 287-317, July 1993.