

# **VR Juggler**

## **Performance Measurement Guide**

---

# VR Juggler: Performance Measurement Guide

Published \$Date: 2002/05/24 06:58:33 \$

---

---

---

## Table of Contents

Preface .....	vi
1. Introduction .....	1
2. Measuring application performance .....	2
Measuring latencies .....	3
3. Viewing performance data .....	4
Viewing performance graphs .....	4
Using the maximum stored samples .....	5
4. Configuring VR Juggler performance tests .....	6
5. Performance measurements built into VR Juggler .....	7
6. Adding performance tests to your own code .....	8
Adding new timestamps inside VR Juggler application classes .....	8
Putting names to numbers in VjControl .....	9

---

## List of Figures

2.1. Thread example - VR Juggler Kernel. ....	2
3.1. Performance data summary panel. ....	4
3.2. Example performance data graph. ....	
4.1. ConfigChunk for activating performance measurements. ....	
4.2. Configuring a log file for VR Juggler performance data. ....	

---

# Preface

This book describes how to use VR Juggler's performance measurement capabilities.

---

# Chapter 1. Introduction

This document describes the performance measurement features available in VR Juggler. It includes a discussion of how to configure performance collection and view the results in VR Juggler's GUI front-end, VjControl. It also describes how to add performance measurements to applications (including adding fine-grained performance measurement to VR Juggler application objects).

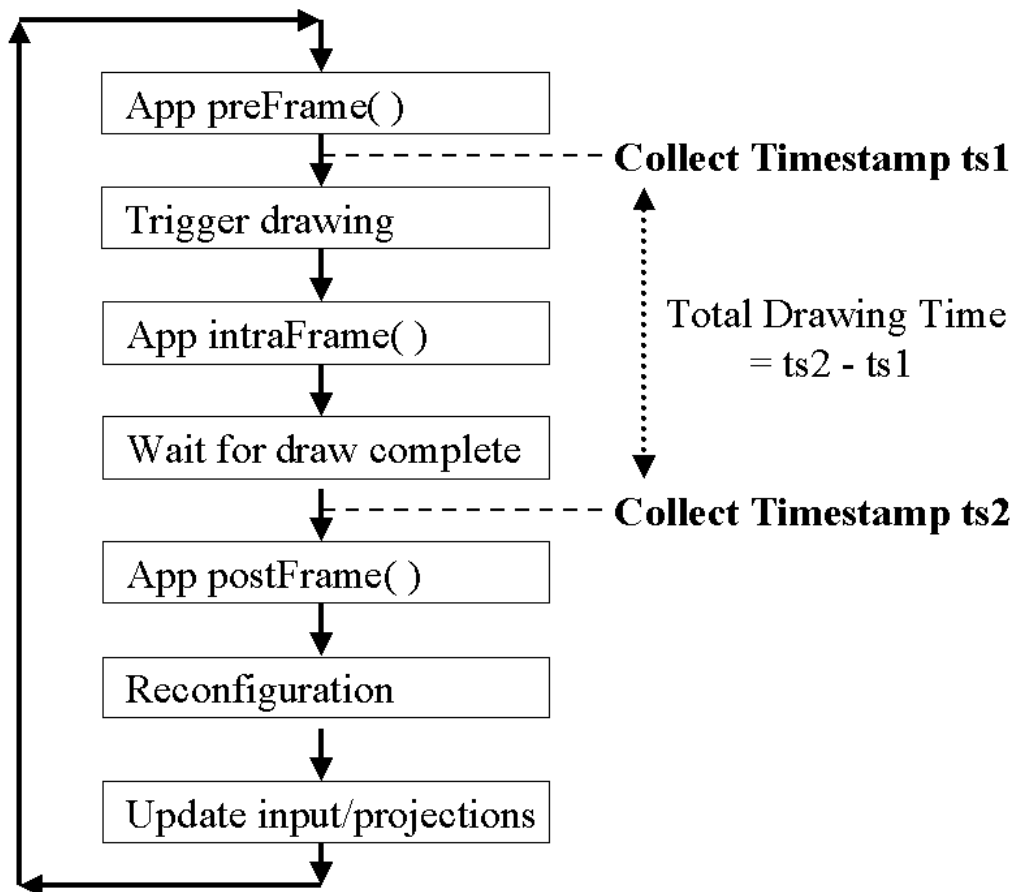
---

# Chapter 2. Measuring application performance

VR Juggler provides a timestamp-based system for measuring the overall performance of multithreaded applications. Some clarification is needed here: we measure the "wall-clock" time required by various portions of our application's threads. The measurements we take are coarse - we measure the time required for relatively large logical blocks of code, not for individual instructions (for the latter case, other tools such as **gprof** are more apposite).

For example, we might look at the kernel thread in VR Juggler. The kernel thread executes in a tight loop that synchronizes the drawing of various graphical displays (see loopdiagram). The framerate of a VR Juggler application is controlled by the total time required for an iteration of this kernel loop. Application developers using VR Juggler want to know the framerate, and also how much time is being spent on the actual drawing (as opposed to other tasks that add overhead to the kernel loop). To do this, we can store a timestamp just before starting to draw (location ts1 in the figure) and again after all the drawing threads have completed. The time between ts1 and ts2 is the time spent drawing the frame - that is, the time from when the various draw threads were told to start, and when all of them signalled the kernel that they had completed.

**Figure 2.1. Thread example - VR Juggler Kernel.**



The time between ts2 and the ts1 stamp in the next loop iteration is spent on other tasks such as updating the view#

ing transformations and (if necessary) calling various application-supplied callback methods.

We could take finer-grained measurements if we so desired, perhaps setting a timestamp after every major function call, or even stepping into some of those calls as necessary. The important thing to remember is that the time recorded for a stamp (such as `ts2`, the "time to draw" stamp in the VR Juggler kernel) is always the difference between that stamp and the stamp immediately preceding it.

## Measuring latencies

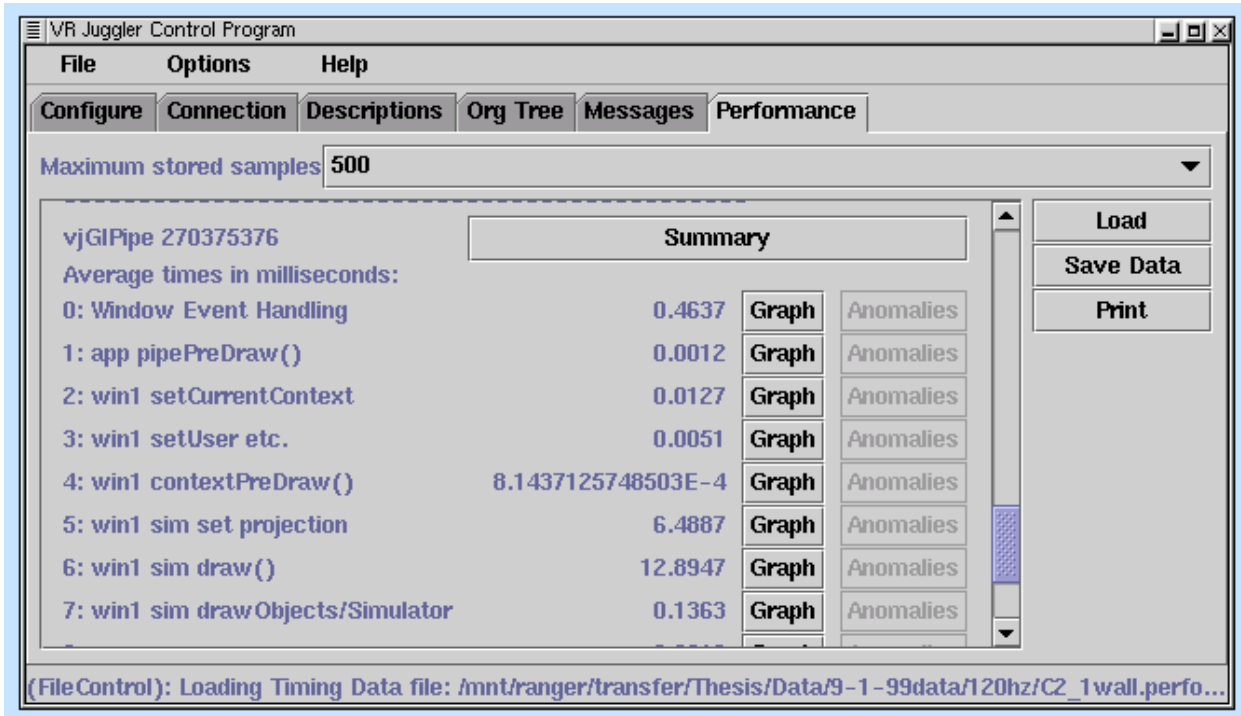
Sometimes - especially in interactive applications - raw speed is not the most pressing concern. Latency - the age of data when it is presented to the user - is more important. In a VR application, latency is usually thought of as the time between when a measurement of the user's body is taken, and when a scene generated using that information is presented.

This type of latency data can be collected by recording a timestamp when the input data - such as the user's body position - was read, and another when the result of that input data is presented to the user. The difference between these two stamps is the latency measurement.

# Chapter 3. Viewing performance data

Once we collect this sort of timing data from a VR Juggler application, we can examine it using the performance-monitoring features of VjControl. VjControl can load performance data from a log file or directly from a running application over a network connection. See Chapter 4, *Configuring VR Juggler performance tests*, below, for information on how to set up these options.

**Figure 3.1. Performance data summary panel.**



When a performance data file is loaded, VjControl displays a summary of the information (Figure 3.1). For each thread, it displays the average time associated with each timestamp (remember that this is the time between the named stamp and the stamp preceding it).

To simplify the display, VjControl tries to display the stamps in an organized, hierarchical fashion. For example, the three application callback methods that are called in our kernel frame are all collected under the heading "app", so that we can easily see the sum of the times for each of them, as well as the individual averages.

## Viewing performance graphs

Of course, averages do not tell the whole story - a few large anomalous readings could seriously skew the averages, for example. Therefore VjControl can also display a graph of the actual data over time. For example, see Figure 3.2. Each vertical line on the graph represents the time taken for drawing during one iteration of the loop. As you can see, the graph is mostly smooth, but there is a single anomalously large reading (approx. 295 ms) at the far left. This is a common occurrence when VR Juggler applications start up - the first few frames involve all of the application's configuration, initialization of device drivers, and so forth, before the application eventually settles into its regular routine.

It is also possible to view multiple samples in the same graph. In this case, the samples for an individual loop itera#

tion are "stacked" one on top of another - see Figure something or other. The lines are color-coded to the labels on the right side of the graph display. The checkboxes next to the labels can be used to select which samples are displayed and which are hidden. For example, we might want to display only the times associated with our particular application's code.

## Using the maximum stored samples

The performance measurement tools described here can produce huge amounts of data very quickly. This can make analysis difficult, just in terms of the memory required. Frequently, when monitoring an application - especially when doing so live - the average for a stamp over the entire run of the application isn't interesting. We might be more concerned with the recent behavior of our code. The main performance monitoring panel in VjControl includes a "Maximum stored samples" selector. This controls the number of samples (for each data buffer) that can be stored at one time. Once the maximum is reached, when new samples are added the oldest ones are eliminated. Only the currently stored samples are viewable in the graph panel, and only they contribute to the averages displayed in the summary panel.

---

# Chapter 4. Configuring VR Juggler performance tests

Enabling VR Juggler's performance monitoring capabilities requires two simple additions to the application's configuration file.

First, we need to tell the application which kinds of performance data to gather. Various components of VR Juggler create data buffers for collecting performance data, and each of these buffers can be turned on or off. The buffers are identified by their name (e.g. "Kernel" or "Head Latency"). Additionally, application code can create its own data buffers with unique names.

These categories can be individually activated and deactivated with a Performance Measurements ConfigChunk (which you can create under the Environment Manager folder in VjControl's config file editing window) - see Figure 4.1. Simply create an entry with the buffer's name and set it to true or false. If a buffer isn't named in the configuration, it defaults to false (disabled).

Sometimes, VR Juggler will create multiple buffers with similar names. For example, if there are two drawing threads, it will create buffers called "vjGIPipe 1" and "vjGIPipe 2". You can turn on the performance tests in both buffers by using the common prefix "vjGIPipe" in the configuration file.

Secondly, we must tell VR Juggler where to send the performance data once it is collected. This is determined by the Performance Target property in the Environment Manager ConfigChunk. If we are dynamically reconfiguring VR Juggler so that we can view performance while the application is running, the pulldown menu for the Performance Target property will list the network connection between VjControl and the application as a possible target. If that connection is selected, performance data can be viewed in a panel in VjControl (click on the Performance tab).

We can also send performance data to a file for later evaluation. To do this, first create an EM Connection ConfigChunk (found in the Environment Manager folder in VjControl). This chunk is illustrated in Figure 4.2. Here we can select the name of the output file and tell VR Juggler to "activate" it - that is, to open it and write to it. Note that the File Mode property should always be "output" for a log file.

Once we have created the EM Connection for our log file, we can return to the Environment Manager ConfigChunk and select it as our target for performance data. The output file that results from running the application with this configuration can be loaded into VjControl's Performance panel for analysis.

---

# Chapter 5. Performance measurements built into VR Juggler

Several components of the VR Juggler software have been instrumented for performance analysis. These performance checks are primarily designed for testing the performance and overhead of VR Juggler itself; however, they also provide an easy way to get an overview of application performance. Using these performance checks, an application developer can easily see how much time is being spent in each of the publicly callable methods of a VR Juggler application object. These tests include:

- Timing of the VR Juggler kernel thread, which calls application methods such as `preFrame()`, `postFrame()`, and `intraFrame()`. To activate these tests, set the "Kernel" tests to True in the Performance Measurements ConfigChunk.
- Timing of the drawing threads for OpenGL-based applications, which call methods such as `draw()`, `contextInit`, `pipePreDraw()`, etc. Activate these with the string "vjGIPipe".
- Latency measurements for the positional input used to draw each viewport in OpenGL-based applications. The string is "Head Latency".

---

# Chapter 6. Adding performance tests to your own code

Sometimes, developers of VR Juggler applications may want to use finer-grained measurements than can be accomplished simply by turning on Juggler's built-in measurements. Applications can create their own performance buffers to collect timing information inside application-specific code.

## Adding new timestamps inside VR Juggler application classes

Adding detailed performance measurements inside application code is a relatively simple process. For our example, we'll assume that we want to instrument our application's `preFrame()` method. We'll also assume that `preFrame()` just calls four other methods - call them `work_a()` through `work_d()` - which do all the actual work of the application.

The first thing we need to do is create a buffer for storing the performance data. The declaration looks like this:

```
#include <Performance/vjPerfDataBuffer.h>
vjPerfDataBuffer* perf_buffer;
```

Of course, the actual variable declaration should be located inside the application class' definition. Next, we need to allocate the buffer itself. The following lines need to be called before the application's `preFrame()` method - the application's `init()` method is a good place to insert it.

```
perf_buffer = new vjPerfDataBuffer ("My Buffer", 500, 5);
vjKernel::instance()->getEnvironmentManager->addPerfDataBuffer (perf_buffer);
```

The arguments in the first line above warrant examination. "My Buffer" is the name of the data buffer we are creating; it is also the string we'd use in the configuration file to identify this buffer and activate it. 500 refers to the number of samples that can be stored in the buffer before it overflows. While the application is running, the Environment Manager will periodically write out the data in the buffer. If this value is too low, the buffer may fill up before it is written out. While this is not dangerous, it will result in missing samples in our output data. On the other hand, using a too-large value will waste memory that might be better used elsewhere.

The third argument is the number of unique "indices" which will be stored in this buffer. The indices are used to determine where in our code a particular timestamp is set. Knowing this value in advance allows us to manage memory in the data buffer very efficiently - after all, we wouldn't want our performance measurements to unduly impact performance themselves. Again, using a too-large value could result in wasted memory.

The second line of code registers our buffer with VR Juggler's Environment Manager, which is responsible for writing out the performance data.

Next, we need to actually gather our timestamps. An example of an instrumented `preFrame()` method follows.

```
void MyApp::preFrame (void)
{
    perf_buffer->set(0);
    work_a();
    perf_buffer->set(1);
    work_b();
    perf_buffer->set(2);
    work_c();
}
```

```
perf_buffer->set(3);  
work_d();  
perf_buffer->set(4);  
}
```

We collect five timestamps each time VR Juggler calls this method; the stamps are uniquely identified by the integer argument for the set() command. For example, stamp 2 records the time required for calls to work\_b(), while stamp 4 records the time for calls to work\_d(). Stamp 0 records the time since the previous call to preFrame completed (i.e. the time since the last timestamp - 4 - was collected).

We could rewrite this example to make the tests even finer grained; for example, we could also collect timestamps inside of the work\_c() method. However, we do have to make sure that all the stamps are uniquely identifiable. For example, we might collect stamps with the indices 3, 4, and 5 within work\_c(). In that case, the stamp collected between the calls to work\_c() and work\_d() would have the index 6. To make this more maintainable, it may be helpful to store the current index in an integer variable, and simply increment it each time we collect a stamp.

Finally, our application's destructor should be sure to unregister the data buffer:

```
vjKernel::instance()->getEnvironmentManager->removePerfDataBuffer (perf_buffer);
```

## Putting names to numbers in VjControl

Integer indices are well-suited for recording performance information, but difficult to understand when viewing that information. VjControl can be configured to associate descriptive names with the integer indices of a performance data buffer.

To do this, create a PerfData ConfigChunk in VjControl. The name of the ConfigChunk should be the name (or the first part of the name) of a data buffer (e.g. "vjGIPipe"). This ConfigChunk has only one property called Labels. This property is a series of text strings which are associated with the numeric indices in the order they appear. These text labels are then used when displaying the performance data in VjControl's Performance panel.