Tutorial for Diagnostics

- Assertions
- Logging
- Performance Analysis > Introduction
- Performance Analysis > Howto Instrument for Performance Measurement

Assertions

Refer to FeatStd Assertions how to use and configure assertions.

Logging

Refer to FeatStd Logging how to use and configure FeatStd logging feature.

Candera Log Output

For a list of all LogRealms supported by Candera, refer to *Candera/System/Diagnostics/LogRealm.h*:

```
FEATSTD_LOG_DECLARE_REALM(CanderaEngine2D);
FEATSTD_LOG_DECLARE_REALM(CanderaEngine3D);
FEATSTD_LOG_DECLARE_REALM(CanderaEngineBase);
FEATSTD_LOG_DECLARE_REALM(CanderaAnimation);
FEATSTD_LOG_DECLARE_REALM(CanderaSystem);
FEATSTD_LOG_DECLARE_REALM(CanderaTextEngine);
FEATSTD_LOG_DECLARE_REALM(CanderaTransitions);

FEATSTD_LOG_DECLARE_REALM(CanderaPlatformDevice);
FEATSTD_LOG_DECLARE_REALM(CanderaPlatformOs);

FEATSTD_LOG_DECLARE_REALM(CanderaAssetLoader);

FEATSTD_LOG_DECLARE_REALM(CanderaGlobalization);

FEATSTD_LOG_DECLARE_REALM(CanderaMonitor);

FEATSTD_LOG_DECLARE_REALM(CanderaScripting);

FEATSTD_LOG_DECLARE_REALM(CanderaEntityComponentSystem);

FEATSTD_LOG_DECLARE_REALM(CanderaBehavior);
```

Player Logging Support

The Player supports logging output displayed in a seperate window that can be controlled from the CGI-Panel.

Starting Player with Log Output

To show the log output window of the Player, the CGIPanel.exe has to be started with option **-l**. This will open a console window in addition to the Panel and display windows to show log output. In the default configuration this window will only show the log output of the CgiApplication and default Candera level.

> If the Player is started via SceneComposer plugin notification, it will be automatically started with the -l option.

Logging Menu Controls

After loading the asset the logging menu controls will appear on the panel. You have two options:

- Enable log checkbox: Will enable/disable the log output via the CgiAppLogAppender
- Candera log level dropdown: Will set all Candera realms to the specified log level

> Menu controls will only appear if your DLL was built with FEATSTD_LOG_ENABLED option

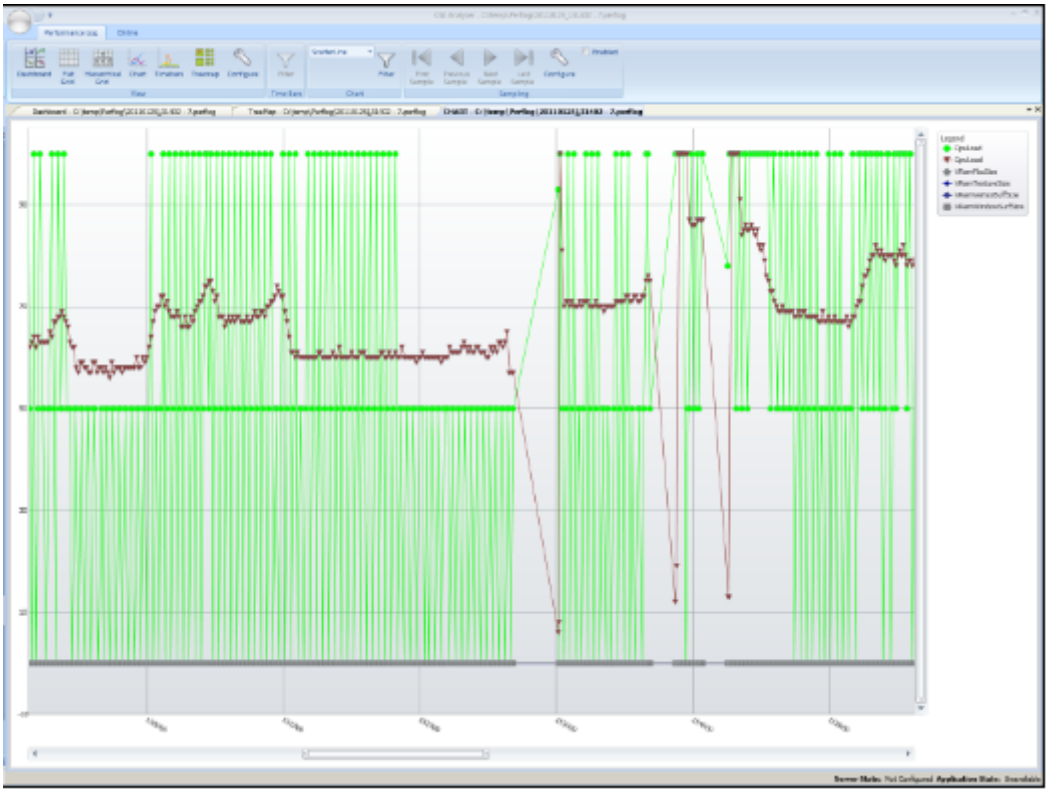
Log Control via SceneComposer

If the Player is started via SceneComposer plugin notification during asset generation, it is possible to use the SceneComposer logging menu options. The changed settings will be sent to the Player.

Performance Analysis > Introduction

Performance Analysis with CGI Analyzer

By enabling the Candera Monitor component (CMake: *FEATSTD_MONITOR_ENABLED*), Candera code is instrumented with performance traces, which are evaluated by the CGI Analyzer tool. In addition, also custom instrumentations can be added to applications, which will be visible in CGI Analyzer.



Offline Analysis

Performance logs are recorded with Candera Monitor and can be evaluated in CGI Analyzer regarding:

- frame rates
- CPU/GPU load
- CPU/GPU memory usage

Online Analysis

With online experiments, performance bottlenecks can be discovered: If framerate goes up after deactivating a dedicated part of the Candera Engine or the rendering pipeline, the bottleneck is located.

Following online experiments are supported:

- Replace textures by 2x2 texture
- Replace fragment shader with a very simple one
- Ignore draw calls

Besides online experiments, CGI Analyzer Online Analysis supports:

- Live display of key performance indicators (frame rate, CPU load, GPU load), where available
- Computation and transfer of Render Benchmarks from target to host (for multi frequency rendering)
- Online determination of scene graph characteristics (number of vertices, texture sizes, render order)
- Online measurement of render times (for each node on a per-camera base)
- Guards (FPS, unconditional)

For more information about CGI Analyzer, refer to CGI Analyzer User Manual.

Performance Analysis > Howto Instrument for Performance Measurement

Performance Measurement with Analyzer

This document shall convey the basics of performance measurement instrumentation.

General Setup

Project Setup

Copy your own project folder to your CGI Studio root folder (containing for example the Candera folder named cgi_studio_candera) named <CGI-STUDIO-ROOT>.

Point CMake GUI "Where is the source code" to the location of your source code.

Select an arbitrary folder for "Where to build the binaries".

After pressing "configure", following Candera Performance Measurement specific CMake flags can be configured:

FEATSTD_MONITOR_TYPE	Enables or disables Candera Monitor component and Candera performance monitoring instrumentation macros. By default, Candera is configured to not include instrumentation on graphics engine level.	On / Off
FEATSTD_MONITOR_TCPIP_ADDRESS FEATSTD_MONITOR_TCPIP_PORT	If FEATSTD_MONITOR_TYPE is set to FEATSTD_COM_TYPE_TCPIP , TCP/IP online connection to be established with Analyzer can be configured. Default: • localhost (127.0.0.1) • 13047	TCP/IP Address and Port

FEATSTD_MONITOR_SERIALPORT_ADDRESS FEATSTD_MONITOR_SERIALPORT_BAUDRATE	If FEATSTD_MONITOR_TYPE is set to FEATSTD_COM_TYPE_SERIALPORT , Serial port online connection to be established with Analyzer can be configured. Default: • empty • Baud: 9600	Serial COM Port and Baudrate
CANDERA_PERFORMANCE_RECORDER_DISABLED	Enables or disables Candera performance instrumentations. Default: Off	On / Off

Performance recording functionality is exposed by macros; hence no performance recording code will be included in Candera and Candera application builds, if performance recording is disabled.

In this tutorial, instead of using a TCP/IP online connection, performance log data is being written to a local file. In this case, performance log data needs to be transferred to Analyzer tool manually, i.e. via TFTP from target to host.

And the following cmake setting were chosen:

```
FEATSTD_MONITOR_TYPE = FEATSTD_COM_TYPE_FILE_DUMP
```

Program Skeleton (Recording to File)

For this tutorial, the following program skeleton is used:

```
#include <stdio.h>
#include <time.h>
#include <Candera/System/Monitor/PerfMonPublicIF.h>
#include <FeatStd/Monitor/PerformanceRecording/FileDumper.h>

using namespace Candera;
using namespace FeatStd::PerfMon;

int main(int argc, char *argv[])
{
    // activate recording
    CANDERA_PERF_SET_ENABLED(true);

    FileDumper::Open("c:\\temp\\tutorial.perflog");

    // BEGIN do the recording here

    // END do the recording here

    // Flush recording buffer at regular, small intervals to prevent buffer overflow
    FileDumper::Flush();
```

```
    // Close file and open new file in order to keep performance log files small
    FileDumper::FlushAndClose();

    fgetc(stdin);

    return 0;
}
```

Important is the activation of recording. Without this line, no performance recording will be written to the recording buffer.

The call to `FeatStd::PerfMon::FileDumper::Open` opens the file to write to. `FeatStd::PerfMon::FileDumper::Flush` writes the data recorded in the buffer to the file, if possible. `FeatStd::PerfMon::FileDumper::FlushAndClose` calls `DumpFile::Flush`, just in case, and closes the file.

The performance log API was designed with different storage media and transport mechanisms in mind. As you can see from the following code snippet, the function `FeatStd::PerfMon::FileDumper::Flush` calls the function `FeatStd::PerfMon::Controller::DumpPerfData` and passes a pointer to a member function to it. This function is called in return to actually write data to the provided mechanism, in this case, to a file.

```
static void Flush()
{
    FEATSTD_PERF_DUMP_DATA(FlushLog, false, 0);
}
```

Program Skeleton (Recording to Memory)

The following code snippet shows the program skeleton for recording to memory:

```
#include <stdio.h>
#include <time.h>
#include <Candera/System/Monitor/PerfMonPublicIF.h>
#include <FeatStd/Monitor/PerformanceRecording/MemoryDumper.h>

using namespace Candera;
using namespace FeatStd::PerfMon;

int main(int argc, char *argv[])
{
    const UInt32 perflogLength = 256;
    UInt8 perflog[perflogLength];

    // set destination for recording
    MemoryDumper::Initialize(perflog, perflogLength);

    // activate recording
    CANDERA_PERF_SET_ENABLED(true);
```

```
    // BEGIN do the recording here

    // END do the recording here

    // Flush recording buffer at regular, small intervals to prevent buffer overflow
    MemoryDumper::Flush();

    // Print performance log location and size to provided stream
    // (default: std::cout) and disable performance log recording
    // A breakpoint has to be set here to prevent recorded data from being overwritten
    MemoryDumper::Dump();

    // Reset the dumper to restart recording
    MemoryDumper::Clear();

    // re-activate recording;
    CANDERA_PERF_SET_ENABLED(true);

    fgetc(stdin);

    return 0;
}
```

Recording to memory is done similar to recording to file. Instead of closing the old file and opening a new one, the storage location along with the length of the performance log is dumped (written) to a specified stream (defaulting to `std::cout`) using the function `FeatStd::PerfMon::MemoryDumper::Dump()`. Performance log recording is automatically disabled at this point. A breakpoint has to be set here to stop the application and copy the performance log from memory to a file for opening with Analyzer. To continue recording, the performance log dumper has to be reset using the function `FeatStd::PerfMon::MemoryDumper::Clear()` to re-start recording at the beginning of the user-provided buffer.

Recording of Performance Values

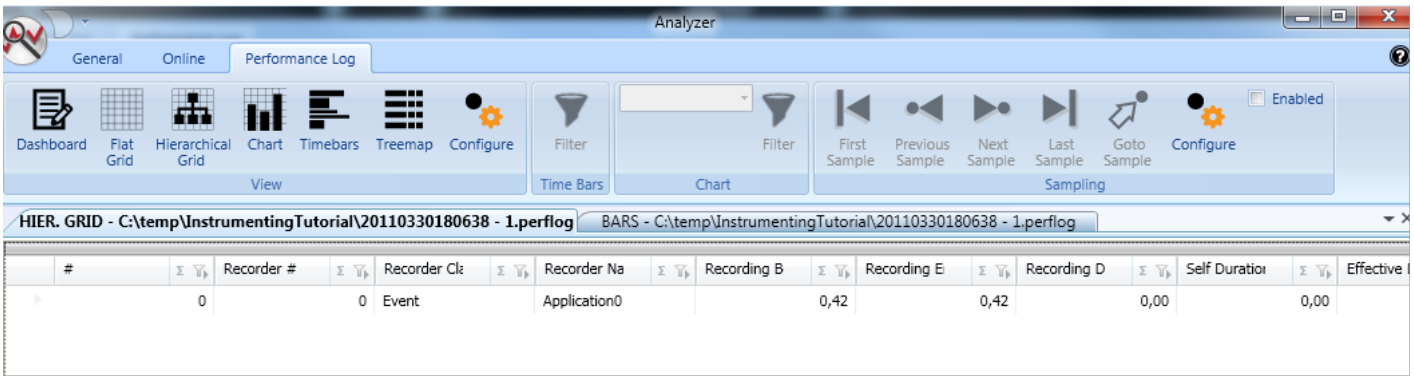
Using Event Recorders

Event Recorders are the simplest recorders. They just record events at given points in time. Their usage is straight forward:

```
...
CANDERA_PERF_RECORD_EVENT(EventRecId::Application0);
...
```

Here, an event with event id Application0 is recorded. The namespace EventRecId is defined in the file Recorder.h. If we compile and run our application, a performance log file is created with just this one recording.

We can now open this file in Analyzer. In the Flat / Hierarchical Grid View (Figure 1), as well as in the Time Bars View, the recording can be inspected. It is possible to change the predefined recorder name for display using the Configuration Dialog from the View Ribbon Bar. You have to reopen the performance log for the changes to take effect.



Using Value Recorders

Value recorders can record integral numbers (positive and negative). The example below shows a succession of value recordings.

```
...
CANDERA_PERF_RECORD_VALUE(ValueRecId::Application0, 10);
busywait();
CANDERA_PERF_RECORD_VALUE(ValueRecId::Application0, 10);
busywait();
CANDERA_PERF_RECORD_VALUE(ValueRecId::Application0, 10);
busywait();s
CANDERA_PERF_RECORD_VALUE(ValueRecId::Application0, 10);
busywait();
...
```

As with event recorders, value recorders are identified by their id. The namespace ValueRecId is again defined in the file Recorder.h. In addition to the id, the value to record is passed to the recording function. The function busywait() waits in a loop for some time to pass, just to make things more interesting.

Opening the new performance log in Analyzer now shows five recordings (one event and four values). The "Value" column shows the value 10 for each value recorder (Figure below).

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Value	Absolute Val
1	0	Event	Application0	200,50	200,50		
2	0	Value	Application0	300,50	300,50	10	10
3	1	Value	Application0	400,50	400,50	10	10
4	2	Value	Application0	500,50	500,50	10	10
5	3	Value	Application0	600,50	600,50	10	10

"The Absolute Value" column shows also the values 10. This is the case, because as a default, Analyzer assumes as relationship between recorded values "Unrelated". To regard the values of a value recorder as related, the value relationship "Differential" has to be set. This can be accomplished again via the Configuration Dialog from the View Ribbon Bar. Remember to close and reopen the performance log after changing settings. Figure 3 shows the value recordings of recorder Application0, interpreted as having the relationship "Differential". The "Value" column is still the same, but the "Absolute" column shows, that each recording has its recorded value added to the value of its predecessor.

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Value	Absolute Val
1	0	Event	Application0	200,50	200,50		
2	0	Value	Application0	300,50	300,50	10	10
3	1	Value	Application0	400,50	400,50	10	20
4	2	Value	Application0	500,50	500,50	10	30
5	3	Value	Application0	600,50	600,50	10	40

Let's add another value recorder:

```
...
CANDERA_PERF_RECORD_VALUE(ValueRecId::Application1, -50);
busywait();
...
```

If we run our program again and view the new performance log in Analyzer, we see the following:

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Value	Absolute Val
0	0	Event	Application0	100,46	100,46		
1	0	Value	Application0	200,46	200,46	10	10
2	1	Value	Application0	300,46	300,46	10	20
3	2	Value	Application0	400,46	400,46	10	30
4	3	Value	Application0	500,46	500,46	10	40
5	0	Value	Application1	600,46	600,46	4294967246	4294967246

What happened? The problem here is, that Analyzer interprets a recorded value by default as positive long value. In order to interpret it correctly, we have to help Analyzer by using the Configuration Dialog from the View Ribbon Bar. By checking the checkbox labeled "Data values are signed" for the value recorder Application1, the values will from now on be interpreted correctly. Remember to reopen the performance log for the changes to take effect (Figure below).

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Recording D	Value
0	0	Event	Application0	100,46	100,46	0,00	
1	0	Value	Application0	200,46	200,46	0,00	10
2	1	Value	Application0	300,46	300,46	0,00	10
3	2	Value	Application0	400,46	400,46	0,00	10
4	3	Value	Application0	500,46	500,46	0,00	10
5	0	Value	Application1	600,46	600,46	0,00	-50

Using Cumulative Value Recorders

The listing below shows the usage of a Cumulative Value Recorder with an annotation. An annotation is a way to differentiate between different flavors of the same recorder.

```
...
{
    CANDERA_PERF_RECORDER(CumulativeValue, (ValueRecId::Application2, "cumval"));
    busywait();
    CANDERA_PERF_RECORD_VALUE(ValueRecId::Application2, 10);
    busywait();
    CANDERA_PERF_RECORD_VALUE(ValueRecId::Application2, 20);
    busywait();
    CANDERA_PERF_RECORD_VALUE(ValueRecId::Application2, 30);
    busywait();
    CANDERA_PERF_RECORD_VALUE(ValueRecId::Application2, 40);
    busywait();
}
...
```

A Cumulative Value Recorder has to be instantiated. This is done with the first statement. All further references to this recorder add the respective values to the existing recording entry and do not instantiate new recording entries. In order for this to work, the references to the Cumulative Value Recorder have to be in the same block, or a sub-block. The image below shows the new result of the recording in Analyzer.

Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Call Count	Value	Absolute Val
0	0	Event	Application0	100,44	100,44		
1	0	Value	Application0	200,44	200,44	10	10
2	1	Value	Application0	300,44	300,44	10	20
3	2	Value	Application0	400,44	400,44	10	30
4	3	Value	Application0	500,44	500,44	10	40
5	0	Value	Application1	600,44	600,44	-50	-50
6	0	Value	Application2	700,44	1.200,44	4	100	100

We see, that the values are summed-up and the value of "Call Count" is 4, the number of recordings.

Using Timing Recorders

Timing Recorders are, as the name implies, used to measure time. The time is measured in relation to the execution block where they are used. The life of a Timing Recorder ends with its execution block, as does the time measurement. Listing 8 shows the usage of a Timing Recorder.

```
...
{
    CANDERA_PERF_RECORDER(Timing, (TimingRecId::Application0));
    busywait();

    CANDERA_PERF_RECORD_EVENT(EventRecId::Application0);
    busywait();
}
```

```
...
```

Timing Recorders have to be instantiated. Everything (direct or called) in the same execution block contributes to the measured time. Also, all recordings in this execution block are regarded as being nested, being child recordings of the timing recording. The figure below shows the Hierarchical Grid View in Analyzer.

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Recording D	Se
0	0	Event	Application0	100,50	100,50	0,00	
1	0	Value	Application0	200,50	200,50	0,00	
2	1	Value	Application0	300,50	300,50	0,00	
3	2	Value	Application0	400,50	400,50	0,00	
4	3	Value	Application0	500,50	500,50	0,00	
5	0	Value	Application1	600,50	600,50	0,00	
6	0	Value	Application2	700,50	1.200,50	500,00	
7	0	Timing	Application0	1.200,50	1.400,48	199,98	

#	Recorder #	Recorder Cla	Recorder Na	Recording B	Recording E	Recording D
8	1	Event	Application0	1.300,50	1.300,50	0,00

A Timing Recorder can have further Timing Recorders in its execution block. What we gain here is a call stack-like hierarchy of recordings. The listing below shows the code snippet and the next figure shows the performance log opened in Analyzer.

#	Recorder #	Recorder Cla	Recorder Na	Recording Begin	Recording End	Recording Duration	Self Duration	Ef
0	0	Event	Application0	100,50	100,50	0,00	0,00	
1	0	Value	Application0	200,50	200,50	0,00	0,00	
2	1	Value	Application0	300,50	300,50	0,00	0,00	
3	2	Value	Application0	400,50	400,50	0,00	0,00	
4	3	Value	Application0	500,50	500,50	0,00	0,00	
5	0	Value	Application1	600,50	600,50	0,00	0,00	
6	0	Value	Application2	700,50	1.200,50	500,00	500,00	
7	0	Timing	Application0	1.200,50	1.400,48	199,98	199,98	
9	0	Timing	Application1	1.400,50	1.700,50	300,00	100,00	

#	Recorder #	Recorder Cla	Recorder Na	Recording Begin	Recording End	Recording Duration	Self Duration
10	1	Timing	Application1	1.500,50	1.600,50	100,00	100,00
11	2	Timing	Application1	1.600,50	1.700,50	100,00	100,00

Using Cumulative Timing Recorders

Cumulative Timing Recorder, as Cumulative Value Recorder, store one value per left execution block only. The listing below shows the usage and the next figure shows the result of this instrumentation in Analyzer. Cumulative Timing Recorders have to be instantiated.

```
...
{
    CANDERA_PERF_RECORDER(Timing, (TimingRecId::Application3, true));
    busywait();

    {
        {
            CANDERA_PERF_RECORDER(Timing, (TimingRecId:: Application3));
            busywait();
```

```
        }
        {
            CANDERA_PERF_RECORDER(Timing, (TimingRecId:: Application3));
            busywait();
        }
    }
}
...
```

Recorder Cl	Recorder Na	Recording Begin	Recording End	Recording Duration	Self Duration	Effective Duration	Call Count
Event	Application0	100,52	100,52	0,00	0,00	0,00	
Value	Application0	200,52	200,52	0,00	0,00	0,00	
Value	Application0	300,52	300,52	0,00	0,00	0,00	
Value	Application0	400,52	400,52	0,00	0,00	0,00	
Value	Application0	500,52	500,52	0,00	0,00	0,00	
Value	Application1	600,52	600,52	0,00	0,00	0,00	
Value	Application2	700,52	1.200,52	500,00	500,00	500,00	4
Timing	Application0	1.200,52	1.400,50	199,98	199,98	199,98	
Event	Application0	1.300,52	1.300,52	0,00	0,00	0,00	
Timing	Application1	1.400,52	1.700,52	300,00	100,00	100,00	
Timing	Application1	1.500,52	1.600,52	100,00	100,00	100,00	
Timing	Application1	1.600,52	1.700,52	100,00	100,00	100,00	
Timing	Application3	1.700,52	2.721,42	1.020,90	1.020,90	920,90	2

Using Asynchronous Timing Recorders

Asynchronous Timing Recorders are the only timing recorders whose recordings can outlive them. An Asynchronous Timing Recorder has to be instantiated and the end of the recording operation has to signaled explicitely (Listing below).

```
...

{
    // Start recording of ansychronous timing.
    CANDERA_PERF_RECORDER(AsyncTiming, (AsyncTimingRecId::Application0));
    busywait();
}

...

// End of asynchronous recording, outside of creation scope
CANDERA_PERF_ASYNCREC_FINISHED(AsyncTimingRecId::Application0);

...
```