CodeXL Graphics Messaging Protocol

Contents

[CodeXL Graphics Messaging Protocol 1](#_Toc451258232)

[Overview 4](#_Toc451258233)

[Layers 5](#_Toc451258234)

[Other Global Layer commands 5](#_Toc451258235)

[GetAutoCaptureStatusCommand 5](#_Toc451258236)

[GetLayersCommand 5](#_Toc451258237)

[GetAppStatusCommand 5](#_Toc451258238)

[GetNumberCapturedFramesCommand 6](#_Toc451258239)

[GetApplicationNumberedImageCommand 6](#_Toc451258240)

[Layer Manager 7](#_Toc451258241)

[FrameCaptureWithSave 7](#_Toc451258242)

[SetSessionName.txt 8](#_Toc451258243)

[SetProjectName.txt 8](#_Toc451258244)

[GetFrameBufferImage.png 10](#_Toc451258245)

[GetCurrentFrameInfo.xml 10](#_Toc451258246)

[Frame Capture Layer 11](#_Toc451258247)

[ReleaseFrameCommand 11](#_Toc451258248)

[BreakOnFrameCommand 11](#_Toc451258249)

[Logger Layer (Timeline Trace) 12](#_Toc451258250)

[LinkedTrace.txt 12](#_Toc451258251)

[This command returns the LinkedTrace data from server. It supports usage in two modes: 12](#_Toc451258252)

[GetTimelineCommand 12](#_Toc451258253)

[TimelineData 12](#_Toc451258254)

[FrameProfiler Layer 15](#_Toc451258255)

[GetAvailableCountersCommand 15](#_Toc451258256)

[SetEnabledCountersCommand 15](#_Toc451258257)

[RunProfileCommand 15](#_Toc451258258)

[Directory Structure of the Capture Cache 16](#_Toc451258259)

[Capture and Replay demo logs 17](#_Toc451258260)

[Capture session 17](#_Toc451258261)

[Replay Session 18](#_Toc451258262)

[Server Status Return Codes 19](#_Toc451258263)

[When is GRAPHICS\_SERVER\_STATE\_ACTIVE returned from the server? 19](#_Toc451258264)

[When is GRAPHICS\_SERVER\_STATE\_STALLED returned from the server? 19](#_Toc451258265)

[When is GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING returned from the server? 19](#_Toc451258266)

[What do the return codes look like in the returned data? 20](#_Toc451258267)

[Using the AppInitDLL method to inject into games 21](#_Toc451258268)

[How do I use AppInitDLL from the CodeXL client? 21](#_Toc451258269)

[What happens if I do not specify --appinit-dll-filelist ? 22](#_Toc451258270)

[How do the registry settings get cleaned up? 22](#_Toc451258271)

[Some UI Design Implications 23](#_Toc451258272)

[Vulkan Support 24](#_Toc451258273)

[Vulkan Captures Session 24](#_Toc451258274)

[Loading trace data using the Capture Player 25](#_Toc451258275)

[Where is the command buffer handle in the LTR file? 26](#_Toc451258276)

[Is the third timestamp (top of pipe) included in the Vulkan data? 26](#_Toc451258277)

[A difference between DX12 and Vulkan 26](#_Toc451258278)

[What is the Vulkan equivalent of Present? 26](#_Toc451258279)

[API Trace Data Format 27](#_Toc451258280)

[GPU Trace Data Format 28](#_Toc451258281)

# Overview

CodeXL Graphics backend acts the same way as a web server. It understands HTML requests and sends appropriate responses back as text, XML, or HTML depending on what is requested. All requests sent to the server will cause the server to send a response. In the case where no data is requested from the server (request a state change, for example), the server will respond with a simple “OK”.

Since the server understands web requests, messages can be sent to the server using a standard web browser. As with any web message, the first part of the message is the IP address of the machine that the CodeXL Graphics server is running on (or localhost if running on the same machine as the browser). The term <IP> will be used to signify the IP address of the host machine.

<IP>/wrappers.xml – Return a list (as XML) of valid wrappers (plugins) which are available. For each wrapper, information such as name, API and description is returned

<IP>/log.txt – Send the log information back as text

<IP>/Version – Send back the version number of the Server as text

<IP>/ShutDown – Shut down the server. OK message sent back

<IP>/process.xml – Return a list (as XML) of processes which have the PerfStudio plugin attached, including the ProcessID (PID) and the active API used by that process. These parameters are used to build out more complex web commands directed at the application being run.

# Layers

Each component (TimeControl, Logger, FrameProfiler, FrameDebugger) is treated as a layer. The server holds a stack of layers and components can be pushed and popped as needed. The top layer is the active layer. To push a layer, use:

<IP>/<PID>/<API>/PushLayer=<component name>

Sends a response “OK” as a text string if OK.

To pop a layer, use:

<IP>/<PID>/<API>/PopLayer

Sends a text response of the name of the layer popped off the stack.

To show the contents of the stack, use:

<IP>/<PID>/<API>/ShowStack

Sends a response consisting of the stack as XML

## Other Global Layer commands

## GetAutoCaptureStatusCommand

To query the server as to whether AutoCapture is enabled, use:

<IP>/<PID>/<API>/AutoCapture

Sends a response as text, either “TRUE” or “FALSE”.

## GetLayersCommand

<IP>/<PID>/<API>/CommandTree.xml

Sends a response as XML showing all the commands available for all layers currently on the stack.

To use any layer-specific commands described below, the layer should be pushed onto the stack first.

## GetAppStatusCommand

<IP>/<PID>/<API>/AppStatus.xml

Sends a response as XML showing the current application status:

* Not running/Running/Suspended
  + Not running – the application was not yet executed or has been terminated. This is equivalent to checking whether the process exists.
  + Running – the application is currently executing
  + Suspended – the application has been executed and the server has interfered with the execution in order to capture frame data
* Time of run so far
* Number of frames rendered so far

## GetNumberCapturedFramesCommand

<IP>/<PID>/CapturedFrames?Application=executed application

Returns the number of captured frames for the executed application. Executed application is a full path of the application in order to distinguish two versions of the applications that might be executed from two different directories and therefor need to be treated as two different targets

## GetApplicationNumberedImageCommand

<IP>/<PID>/<API>/Picture?Application=executed application?index=n?width=x&height=y

Returns an image index n of a previously executed application with the width x and height y in a png(?) format. If width or height are not supplied then full size picture is returned in a png format.

Questions

\* Should another default be used?

\* Should we give the user an option to supply a returned format?

\* should this be a different command then the GetCurrentImageCommand? After all there is just another two parameters.

# Layer Manager

## FrameCaptureWithSave

To capture a frame, make sure the APITrace (logger) layer is pushed onto the stack:

The FrameCaptureWithSave command can receive two arguments in the URL, the capture type and the capture count (the number of frames to capture). In the following example the capture type is 3 and the capture count is 1. NOTE this command used to be FrameCaptureWithSave.txt – the “.txt” extension has been removed.

<IP>/<PID>/<API>/FrameCaptureWithSave?CaptureType=3&CaptureCount=1

Example:

http://localhost/7904/DX12/PushLayer=Logger

http://localhost/7904/DX12/FrameCaptureWithSave?CaptureType=3&CaptureCount=1

http://localhost/7904/DX12/PopLayer

This command has been moved up into the layer manager and is no longer in a feature of the frame capture layer. This is due to the fact that the command will eventually generate a full frame capture, or a linked trace, so needs to use the Frame Capture layer and the API Trace layer. Currently, the capture command only supports Linked Trace. The client specifies which type of capture to perform.

Available CaptureTypes are:

1 = API Trace (unsupported)

2 = GPU Trace (unsupported)

3 = Linked Trace (supported)

4 = Full Frame Capture (unsupported)

IMPORTANT: Only capture type 3 (Linked Trace) is supported as of December 15th 2015.

The XML returned from this command contains the following fields:

<XML>

<API>DX12</API>

<Location>

C:\Users\glcse\AppData\Local\Temp\GPUPerfStudio\D3D12Multithreading\Session1\Frame18673\description-D3D12Multithreading-2015-12-14-16-47-9.xml

</Location>

<Architecture>1</Architecture>

<APICallCount>7358</APICallCount><DrawCallCount>2053</DrawCallCount>

<TracedFramesCount>3</TracedFramesCount>

<TraceType>3</TraceType>

<Contents>

<LinkedTrace>

C:\Users\glcse\AppData\Local\Temp\GPUPerfStudio\D3D12Multithreading\Session1\Frame18673\LinkedTrace-D3D12Multithreading-2015-12-14-16-47-9.ltr

</LinkedTrace>

<FrameBufferImage>

C:\Users\glcse\AppData\Local\Temp\GPUPerfStudio\D3D12Multithreading\Session1\Frame18673\D3D12Multithreading\_FrameBuffer18673.png

</FrameBufferImage>

<FrameNumber>18673</FrameNumber>

<ElapsedTime>160924.671875</ElapsedTime>

<CPUFrameDuration>16.115320</CPUFrameDuration>

<FPS>102.211528</FPS>

</Contents>

</XML>

IMPORTANT: We are working to remove the requirement to push the logger onto the stack prior to calling FrameCaptureWithSave. The server will look at the capture type and automatically push the correct layer onto the stack.

NOTE: The Frame Buffer PNG files are saved to disk at the same resolution as the actual frame buffer in the application.

## SetSessionName.txt

This command is used to set the session name that will be used in the location of the capture data on the graphics server.

<IP>/<PID>/<API>/SetSessionName.txt=MySessionName

Example:

http://localhost/94048/DX12/SetSessionName.txt=MySessionName

The command sets the session name in the path to the capture directory (e.g. in bold red below)

C:\Users\gselley\AppData\Local\Temp\CodeXL\D3D12Multithreading\**MySessionName**\_2015\_12\_22\_18\_18\_14\Frame\_0000001128

NOTE: If the session name command is not used by the client the server will default to using “Session” as the session name.

The command returns “Ok” for success or “Failed” for failure.

## SetProjectName.txt

This command is used to set the project name that will be used in the location of the capture data on the graphics server.

<IP>/<PID>/<API>/SetProjectName.txt=MyProjectName

Example:

http://localhost/94048/DX12/SetProjectName.txt=MyProjectName

The command sets the project name in the path to the capture directory (e.g. in bold red below)

C:\Users\gselley\AppData\Local\Temp\CodeXL\**MyProjectName**\MySessionName\_2015\_12\_22\_18\_18\_14\Frame\_0000001128

NOTE: If the project name command is not used by the client the server will default to using application’s name as the project name.

The command returns “Ok” for success or “Failed” for failure.

**Frame Debugger Layer**

## GetFrameBufferImage.png

<IP>/<PID>/<API>/FD/GetFrameBufferImage.png?width=x&height=y

Example:

http://localhost/7904/DX12/PushLayer=FrameDebugger

http://localhost/7904/DX12/FD/GetFrameBufferImage.png?width=300?height=200

http://localhost/7904/DX12/PopLayer

Returns the image of the current frame with the width x and height y in a png format.

NOTE: if the user uses a width of 0 and a height of 0, then server will generate an image at the same resolution as the application’s frame buffer (e.g. 3840x2160)

IMPORTANT:

1. If called when the graphics server is running with an application it will extract the frame buffer image of the running application and return it as a png.
2. If called when the CapturePlayer.exe is running with the graphics server, the command will return the frame buffer image that was generated at the time of the capture (this image is part of the capture files that were save to disk).

This server resizes the output image to match the aspect ratio of the frame debugger.  For example if a 800x400 (aspect ratio 2) frame buffer image is requested for a destination image size of 300x200 (aspect ratio 1.5), the server will generate an image of 300x150 (aspect ratio of 2).

## GetCurrentFrameInfo.xml

<IP>/<PID>/<API>/FD/GetCurrentFrameInfo.xml

Example:

http://localhost/7904/DX12/PushLayer=FrameDebugger

http://localhost/7904/DX12/FD/GetCurrentFrameInfo.xml

<http://localhost/7904/DX12/PopLayer>

This command returns the frame information for the currently running session. It returns the following XML to the caller:

<XML src="GetCurrentFrameInfo.xml">

<FrameNumber>9144</FrameNumber>

<ElapsedTime>79516.460938</ElapsedTime>

<CPUFrameDuration>15.438769</CPUFrameDuration>

<FPS>120.226537</FPS>

</XML>

The ElapsedTime and CPUFrameDuration are in milliseconds.

# Frame Capture Layer

Various client-side settings are sent along with the capture command. Sends a response “OK” as a text string if OK.

## ReleaseFrameCommand

To release a captured frame and resume the application, use:

<IP>/<PID>/<API>/FC?Release=1

Sends a response “OK” as a text string if OK.

## BreakOnFrameCommand

<IP>/<PID>/<API>/FC?Break=*Frame\_Number*

Instruct the server to break when the specified frame number is reached.

# Logger Layer (Timeline Trace)

The Logger layer is used to get the API trace data. Must be on the stack to use.

## LinkedTrace.txt

<IP>/<PID>/<API>/LOG/log.txt

Example:

http://localhost/7904/DX12/PushLayer=Logger

http://localhost/7904/DX12/FD/LinkedTrace.txt

http://localhost/7904/DX12/PopLayer

## This command returns the LinkedTrace data from server. It supports usage in two modes:

1. If called when the graphics server is running with an application it will return the live LinkedTrace data (i.e. live data generated in the current session)
2. If called when the graphics server is running the CapturePlayer the following will accours:
   1. If the CaptureType is 3 then the LinkedTrace data that was saved to disk will be returned.
   2. If the CaptureType is 4 then the LinkedTrace data will be generated live from the looping frame.

## GetTimelineCommand

<IP>/<PID>/<API>/LOG/log.txt

Will get a list of the API calls (as text) and any interface information.

## TimelineData

* Default File->Save filename updated to include frame number after process name:
  + Format should match the following: ProcessName\_Frame-NNNNN-YYYY-MM-DD\_HH-MM-SS.ltr
* Meta-data is preceded by “//”

The format specified below is an updated unified trace response format based on new requirements and change suggestions for CodeXL Graphics. CodeXL will provide the “CodeXL” argument to the current “LinkedTrace.txt” CommandResponse, and will receive a response string that includes everything specified below.

1. Will include the generated header contents at the top of the response.

//AMD CodeXL Frame Trace  
//ProcessExe=C:\Users\salgrana\Documents\AMDAPP\samples\opencl\bin\debug\x86\D3D12HelloTexture.exe  
//TraceDateTime=1/1/0001 12:00:00 AM  
//TraceFileVersion=1.0  
//ApplicationArgs=  
//WorkingDirectory=C:/Users/salgrana/Documents/AMD APP/samples/opencl/bin/debug/x86

1. Will include generated system information section.

//==System Info==  
//Device AMD FX(tm)-8150 Eight-Core Processor Platform Vendor = Advanced Micro Devices, Inc.  
//Device AMD FX(tm)-8150 Eight-Core Processor Platform Name = AMD Accelerated Parallel Processing  
//Device AMD FX(tm)-8150 Eight-Core Processor Platform Version = OpenCL 2.0 AMD-APP (1756.4)  
//Device AMD FX(tm)-8150 Eight-Core Processor CLDriver Version = 1756.4 (sse2,avx,fma4)  
//Device AMD FX(tm)-8150 Eight-Core Processor CLRuntime Version = OpenCL 1.2 AMD-APP (1756.4)  
//Device AMD FX(tm)-8150 Eight-Core Processor NumberAppAddressBits = 32  
//Device Barts Platform Vendor = Advanced Micro Devices, Inc.  
//Device Barts Platform Name = AMD Accelerated Parallel Processing  
//Device Barts Platform Version = OpenCL 2.0 AMD-APP (1756.4)  
//Device Barts CLDriver Version = 1756.4 (VM)  
//Device Barts CLRuntime Version = OpenCL 1.2 AMD-APP (1756.4)  
//Device Barts NumberAppAddressBits = 32

1. Will update APITrace preamble to the following format:

//==API Trace==  
//API=DX12  
//ThreadID=9500  
//ThreadAPICount=245

1. Breakdown of the updated API Trace response format is as follows:

APIType APIFuncId InterfacePtr APIInterface\_FunctionName(Arguments) = ReturnValue StartMillisecond EndMillisecond SampleId

4 120 0x0000000E1CABBAD0 ID3D12CommandQueue\_Signal(+0x0000000E1CAF2E60, 2845) = S\_OK 43022.227685 49866.672999 0

* The initial “ThreadId” token is removed, as it’s made redundant by the updated API Trace preamble in #3.

1. Will update GPU Trace preamble to the following format:

//==GPU Trace==  
//CommandListEventCount=98

* Cannot include a CommandList Pointer in the preamble, as multiple unique CommandQueues / CommandLists can exist in a single GPU Trace.

1. Breakdown of the updated GPU Trace response format is as follows:

CommandQueuePtr D3D12\_COMMAND\_LIST\_TYPE CommandListPtr APIType FuncId APIInterface\_FunctionName(Arguments) = ReturnValue StartMillisecond EndMillisecond SampleId

0x02B7B12E210 0 0x02B7B12E9E0 128 5 ID3D12GraphicsCommandList\_DrawIndexedInstanced(3, 1, 6, 0, 0) = void 6122.031 6122.967 2

* The CommandQueue and CommandList pointers will be altered to include the 0x prefix.

# FrameProfiler Layer

## GetAvailableCountersCommand

<IP>/<PID>/<API>/FP/CounterInfo.xml

Will send a list (as XML) of all available counters for the current hardware that the server is running on. Different hardware will have a different list of counters. Counter 0 is GPUTime.

## SetEnabledCountersCommand

<IP>/<PID>/<API>/FP/CounterSelect.txt=<comma delimited list>

Will instruct the server to select the provided counters. Counters are passed in as a comma delimited list ie CounterSelect.txt=0,1,2 will select the first 3 counters, including GPUTime. Leaving the field blank will select all counters.

The command also returns the number of passes required to perform this profile.

## RunProfileCommand

<IP>/<PID>/<API>/FP/Profiler.xml

Will run the profiler for the selected counters and send back an XML string consisting of the timing values for each counter for each draw call.

# Directory Structure of the Capture Cache

The Capture Cache is located within the CodeXL temporary directory e.g.:

C:\Users\yourname\AppData\Local\Temp\**CodeXL**

The name of the application being debugged is hardwired by the server to be the root of all captures for that application, e.g.:

C:\Users\yourname\AppData\Local\Temp\CodeXL\**D3D12Multithreading**

Each time the graphics server is run a new session name is generated to store the captures for that session. If the user does not specify a session name using the SetSessionName command the server will default to using “Session” as the name string. The server appends the Year, Month, Day, Hour, Minute, and Second to the session name as follows:

C:\Users\yourname\AppData\Local\Temp\CodeXL\D3D12Multithreading\**Session\_2015\_12\_22\_18\_18\_14**

If the user sets their own session name by using the SetSessionName command the default “Session” part of the name string will be replaced as follows:

C:\Users\yourname\AppData\Local\Temp\CodeXL\D3D12Multithreading\**MyDebugName\_2015\_12\_22\_18\_18\_14**

The index of the frame being rendered is used to generate the name of a sub-directory to store the actual capture data in:

C:\Users\yourname\AppData\Local\Temp\CodeXL\D3D12Multithreading\MyDebug\_2015\_12\_22\_18\_18\_14\**Frame\_0000001128**

# Capture and Replay demo logs

## Capture session

**Startup**

22:16:37.369806: 0: 192.168.1.10 - /Process.xml

22:16:37.847423: 1: 192.168.1.10 - /Process.xml

22:16:38.070950: 2: 192.168.1.10 - /106576/DX12/ShowStack

22:16:38.109455: 3: 192.168.1.10 - /106576/DX12/PushLayer=TimeControl

22:16:38.124956: 4: 192.168.1.10 - /106576/DX12/TC/Settings.xml

22:16:38.147480: 5: 192.168.1.10 - /106576/DX12/AutoCapture

**Heartbeat commands**

22:16:40.814887: 6: 192.168.1.10 - /106576/DX12/PushLayer=FrameDebugger

22:16:40.823907: 7: 192.168.1.10 - /106576/DX12/FD/GetCurrentFrameInfo.xml

22:16:40.842435: 8: 192.168.1.10 - /106576/DX12/FD/GetFrameBufferImage.png?width=300?height=200

22:16:41.936079: 9: 192.168.1.10 - /106576/DX12/PopLayer

**Set the session name**

22:16:46.572595: 22: 192.168.1.10 - /106576/DX12/SetSessionName.txt=FallOfDuty-Debug

**Capture a Linked Trace**

22:16:47.911249: 23: 192.168.1.10 - /106576/DX12/PushLayer=Logger

22:16:47.924309: 24: 192.168.1.10 - /106576/DX12/FrameCaptureWithSave.txt=3

22:16:48.192127: 25: 192.168.1.10 - /106576/DX12/PopLayer

**Shutdown**

22:16:53.476463: 39: 192.168.1.10 - /Shutdown

## Replay Session

CapturePlayer.exe is started with the location of the capture’s XML metadata file as a command line argument, e.g.:

**D:\depot\devtools\main\GPUPerfStudio\Build\VS2015\bin\Release\CapturePlayer-x64.exe** **C:\Users\yourname\**

**AppData\Local\Temp\GPUPerfStudio\D3D12Multithreading\Session\_\_2015\_12\_22\_18\_18\_14\Frame\_0000001128\ description-D3D12Multithreading-2015-12-22-18-18-14.xml**

**Startup**

22:25:20.507597: 0: 192.168.1.10 - /Process.xml

22:25:21.005879: 1: 192.168.1.10 - /Process.xml

22:25:21.297154: 2: 192.168.1.10 - /107092/DX12/ShowStack

22:25:21.323141: 3: 192.168.1.10 - /107092/DX12/PushLayer=TimeControl

22:25:21.343159: 4: 192.168.1.10 - /107092/DX12/TC/Settings.xml

22:25:21.365280: 5: 192.168.1.10 - /107092/DX12/AutoCapture

**Request the Linked Trace**

22:25:22.917572: 6: 192.168.1.10 - /107092/DX12/PushLayer=Logger

22:25:22.926463: 7: 192.168.1.10 - /107092/DX12/LOG/LinkedTrace.txt

# Server Status Return Codes

The graphics server can now return server status codes as a response to a request for data. The status values can be found in the following header file:

\devtools\main\GPUPerfStudio\Server\Common\GraphicsServerState.h

The file contains the server status enumerations.

enum GRAPHICS\_SERVER\_STATE

{

GRAPHICS\_SERVER\_STATE\_ACTIVE = 0,

GRAPHICS\_SERVER\_STATE\_STALLED = 1,

GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING = 2

};

#define GRAPHICS\_SERVER\_STATUS\_STALL\_TIME\_THRESHOLD 500

#define GRAPHICS\_SERVER\_STATUS\_STALL\_LOOP\_SLEEP\_TIME 100

// Uncomment the following to allow the graphics server to return server status codes

//#define USE\_GRAPHICS\_SERVER\_STATUS\_RETURN\_CODES

To activate the server to return the status codes USE\_GRAPHICS\_SERVER\_STATUS\_RETURN\_CODES must be uncommented and the project rebuilt.

## When is GRAPHICS\_SERVER\_STATE\_ACTIVE returned from the server?

Never. If the application has not crashed and is still rendering a request for data from the client will always be returned with the actual data. This is the success case so GRAPHICS\_SERVER\_STATE\_ACTIVE is never sent.

## When is GRAPHICS\_SERVER\_STATE\_STALLED returned from the server?

If the application has never called Present() or if a Present() has not been called for more than 500ms, a request for data will be returned with GRAPHICS\_SERVER\_STATE\_STALLED in XML, HTML, or PNG form. The exact data returned is described below.

## When is GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING returned from the server?

If the web server detects that the game process that was initially attached to by CodeXL has crashed, the GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_ RUNNING state will be returned in XML, HTML, or PNG form. The exact data returned is described below.

## What do the return codes look like in the returned data?

The actual data returned to the client depends on the type of data request that was sent to the server.

Requests for XML data will be sent:

<XML>

<GRAPHICS\_SERVER\_STATE>GRAPHICS\_SERVER\_STATE\_STALLED</GRAPHICS\_SERVER\_STATE>

</XML>

<XML>

<GRAPHICS\_SERVER\_STATE>GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING</GRAPHICS\_SERVER\_STATE>

</XML>

Requests for TEXT data will be sent an html format message with the status value:

<html>

<body>

<GRAPHICS\_SERVER\_STATE>GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING</GRAPHICS\_SERVER\_STATE>

</body>

</html>

<html>

<body>

<GRAPHICS\_SERVER\_STATE>GRAPHICS\_SERVER\_STATE\_STALLED</GRAPHICS\_SERVER\_STATE>

</body>

</html>

Requests for IMAGE data (e.g. png) will be sent a single byte that will contain the numerical value of the enumeration (e.g. 1 for GRAPHICS\_SERVER\_STATE\_STALLED or 2 for GRAPHICS\_SERVER\_STATE\_PROCESS\_NOT\_RUNNING). Please note that the returned data will not be a viable png format image, it will not have a header, it will consist of a single byte only. This case is very easy to test for by checking the size of the data returned from the request. Read the byte value to get the server status enumeration value.

# Using the AppInitDLL method to inject into games

A new dll-injection method has been added to the Graphics Server of Code XL to enable global injection of MicroDLL.dll. It has been found in some cases that the current method of injection doesn't process-hop correctly through certain third-party game launchers and into the final game executable which results in not being able to get inside the game process. This could be caused by a game launcher detecting unauthorized dll's being loaded into their process and terminating or crashing as part of their copy protection/DRM protection system (e.g. with Hitman Steam version).

To get around this, there is a system-wide dll injection system available through the registry in Windows. This technique avoids the need to process hop by directly injecting into all new processes as they startup. The registry keys are:

1) HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit\_DLLs

- which contains a list of dll's to inject into any newly started process by windows

2) HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\LoadAppInit\_DLLs

- is set to 1 to enable system-wide dll-injection

When the AppInitDLL mechanism is activated by the Graphics server it will write the full path of the 32 and 64 bit Graphics server MicroDll.dll files to the AppInit\_DLLs registry setting. It will also write a '1' to the LoadAppInit\_DLLs key.

Note: This method is potentially very dangerous to the operating system since every process started by the OS will be injected with the dll's specified in the AppInit\_DLLs registry setting.

Every effort has been made to reduce the impact to the OS as much as possible. This includes but is not limited to: Only injecting MicroDLL into processes when the GPUPerfServer is running in the system.

## How do I use AppInitDLL from the CodeXL client?

By default, the option to use AppInitDlls is disabled. To enable the feature use the “--use-appinit-dll” or “-A” GPUPerfServer command line option.

For example:

GPUPerfServer --use-appinit-dll

**IMPORTANT:** The GPUPerfServer must be started with elevated privileges for APPInitDLL to work.

To limit the number of processes that MicroDLL is injected into, use the “--appinit-dll-filelist” GPUPerfServer command line option to specify a list of comma-delimited executables that MicroDLL should be loaded into. Anything else not on this list will be ignored.

For example:

GPUPerfServer --use-appinit-dll --appinit-dll-filelist=hitman-beta.exe,hitman-release.exe

Specify 'all' if the executable name is not known and MicroDLL will be loaded into any new process started by the system while the Graphics Server is running.

For example:

GPUPerfServer --use-appinit-dll --appinit-dll-filelist=all

## What happens if I do not specify --appinit-dll-filelist ?

**IMPORTANT:** MicroDLL will not be loaded into any processes

## How do the registry settings get cleaned up?

The Graphics server will clear out the registry settings once an application has been connected to or when the graphics server detects a startup error and has to close down early. "Connected" means that the registry will be cleaned when the graphics server receives the first command that is directed to a specific process ID.

**WARNING:** It won't clear out the setting if the application or Graphics server crashes or closes down unexpectedly.

It's probably good practice to clear out the registry setting manually by running the Graphics server with the --clean option.

For example, with elevated privileges call:

GPUPerfServer --clean

## Some UI Design Implications

There needs to be an option on a per-project basis to enable this feature, a checkbox for "system-wide dll injection (AppInit\_DLLs)".

A disclaimer that this is an advanced option and that you need to know what you're doing.

If this option is chosen, the Graphics server will need to be run with elevated privileges (having a warning dialog box pop up informing the user that this is happening is OK). Otherwise the registry entries won't be written to. Have a "Run Graphics Server with elevated privileges" checkbox.

The UI needs to be able to supply the "-all" option to --appinit-dll-filelist or a list of exe's to inject into.

We recommend a cleanup button that would call: GPUPerfServer --clean (with elevated privileges).

# Vulkan Support

## Vulkan Captures Session

For Vulkan capture sessions in CodeXL it will be necessary to change the URLs of the commands sent to the Graphics Server so that they target the Vulkan server. For example:

/5944/DX12/ShowStack

Becomes:

/5944/Vulkan/ShowStack

A full capture session will appear as follows:

/Process.xml

/7392/Vulkan/ShowStack

/7392/Vulkan/PushLayer=TimeControl

/7392/Vulkan/TC/Settings.xml

/7392/Vulkan/SetSessionName.txt=May%E2%80%9306%E2%80%932016\_15%E2%80%9306%E2%80%9306

/7392/Vulkan/SetProjectName.txt=D3D12Multithreading

/7392/Vulkan/PushLayer=FrameDebugger

/7392/Vulkan/FD/GetCurrentFrameInfo.xml

/7392/Vulkan/FD/GetFrameBufferImage.png?width=512?height=512

/7392/Vulkan/PopLayer

/7392/Vulkan/PushLayer=Logger

/7392/Vulkan/FrameCaptureWithSave.txt=3

/7392/Vulkan/PopLayer

/ShutDown

## Loading trace data using the Capture Player

The CapturePlayer now supports a DX12 render loop and a Vulkan render loop but currently defaults to using DX12 in all cases. We are going to fix this so that the player can switch between the two API render loops depending on the API used in the original trace/capture. The CapturePlayer is passed the meta-data XML as a command line argument and we have added an API data field to the XML to specify the API used in the capture. Previously, the API data field was only in the LTR and not easily accessible to the player.

The CapturePlayer needs to be modified to use the newly added API field in the meta-data XML file to instantiate the Vulkan render loop to retrieve the Vulkan trace data.

BDC will modify the player to switch between DX12 and Vulkan.

The Command sequence for retrieving a Vulkan trace will be:

/Process.xml

/6828/Vulkan/ShowStack

/6828/Vulkan/PushLayer=TimeControl

/6828/Vulkan/TC/Settings.xml

/6828/Vulkan/PushLayer=Logger

/6828/Vulkan/LOG/LinkedTrace.txt

/6828/Vulkan/PopLayer

/ShutDown

## Where is the command buffer handle in the LTR file?

The command buffer handle is highlighted in yellow below. This only applies to GPU trace data.

0x0 0 **0x000002144E3D4F10** 2 104 Vulkan\_vkCmdDraw(0x000002144E3D4F10, 3, 1, 0, 0) = void 4151704.4830884304828942 4151704.5085699120536447 6

## Is the third timestamp (top of pipe) included in the Vulkan data?

No, not yet. The Vulkan timestamp data is currently the same as in DX12. It consists of two values that measure the bottom of pipe delta (in red below). Vulkan also supports the top of pipe timestamp but we do not currently include this data yet.

0x0 0 0x000002144E3D4F10 2 104 Vulkan\_vkCmdDraw(0x000002144E3D4F10, 3, 1, 0, 0) = void **4151704.4830884304828942** **4151704.5085699120536447** 6

## A difference between DX12 and Vulkan

In Vulkan, there aren’t “interfaces” really… it’s more like a “device function” or a “queue function” etc. Example:

DX12 you do this: **pCmdListPtr->Draw(nVerts);**

In Vulkan you do this: **vkCmdDraw(cmdBufHandle, nVerts);**

## What is the Vulkan equivalent of Present?

vkQueuePresentKHR

## API Trace Data Format

//==API Trace==

//API=Vulkan

//ThreadID=24324

//ThreadAPICount=221

64 21 0x0000000000000000 Vulkan\_vkMapMemory(0x0000000006110FA0, 0x0000000005E8FF30, 0, 64, 0, 0x0000000003E4A7F0) = VK\_SUCCESS 18.0297356228235728 18.0336467344313327 0

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Field Index** | **Field Value** | **Field Description** |
| 0 | 64 | Entry point category. See VktTraceAnalyzerLayer::GetAPIGroupFromAPI() |
| 1 | 21 | FuncID |
| 2 | 0x0000000000000000 | Not used. In DX12 this used to be interface pointer. |
| 3 | Vulkan\_vkMapMemory(0x0000000006110FA0, 0x0000000005E8FF30, 0, 64, 0, 0x0000000003E4A7F0) = | Call plus parameters. Can omit the “Vulkan\_” part. |
| 4 | VK\_SUCCESS | Return value |
| 5 | 18.0297356228235728 | CPU timestamp start |
| 6 | 18.0336467344313327 | CPU timestamp end |
| 7 | 0 | Sample ID |

## GPU Trace Data Format

|  |  |  |  |
| --- | --- | --- | --- |
| //==GPU Trace== | |  | |
| //CommandBufEventCount=51 | |  | |
| 0x0 0 0x000000000604A140 2 105 Vulkan\_vkCmdDrawIndexed(0x000000000604A140, 4800, 1, 0, 0, 0) = void 19.2326690852642059 19.2355490922927856 9 | | | |
|  |  | |  |
| **Field Index** | **Field Value** | | **Field Description** |
| 1 | 0x0 | | Queue index. Should just display it as a non-hex value in UI. |
| 2 | 0 | | Not used. This used to be DX12 command list type, which doesn’t exist in Vulkan. |
| 3 | 0x000000000604A140 | | Command buffer handle |
| 4 | 2 | | Entry point category. See VktTraceAnalyzerLayer::GetAPIGroupFromAPI() |
| 5 | 105 | | FuncID |
| 6 | Vulkan\_vkCmdDrawIndexed(0x000000000604A140, 4800, 1, 0, 0, 0) = | | Call plus parameters. Can omit the “Vulkan\_” part. |
| 7 | void | | Return value |
| 8 | 19.23266909 | | GPU bottom timestamp start |
| 9 | 19.23554909 | | GPU bottom timestamp end |
| 10 | 9 | | Sample ID |