Intel® Graphics Performance Analyzers User Guide

Contents

Chapter 1: Intel® Graphics Performance Analyzers User Guide

Introduction to Intel® Graphics Performance Analyzers 4
Get Help 5
Game Optimization Methodology 6
Install and Launch Intel® GPA 12
Enable GPU Metrics Collection 13
Launch Application for Analysis 15
Start a Desktop Application with Graphics Monitor 16
Start a Windows* Mixed Reality Application 18
Connect to a Running Application (Windows* only) 18
Analyze Applications in the Network Mode 20
Configure Analysis Settings 21
Configure a Metric Set 21
Customize Keyboard Shortcuts 25
Set Up a Trigger 25
Set Up Tracing Options 27
Set Up Stream Options 29
Set Frame Delimiters 31
Analyze Desktop Graphics Applications 32
Manage Analysis with Graphics Monitor 33
Monitor Real-Time Performance with System Analyzer 35
Pause and Resume a Running Application 37
Export Metrics Data with System Analyzer 38
Identify Issues in Graphics Application Execution with Trace Analyzer 38
Analyze GPU-bound Applications with Graphics Frame Analyzer 46
Run Render State Experiments with Frame Analyzer 52
Use Advanced Profiling Mode 54
Graphics Frame Analyzer Plugins 57
Optimize Specific Domains 60
Optimize Shaders 60
Analyze Textures 68
Minimize Overdraw 75
Optimize Graphic API States 75
Identify Graphics Elements with Complex Geometry 77
Visualize Color Distribution 77
Quick Start with Profiling Unreal Engine* Games 80
Instrumentation and Tracing Technology API Support 89
Compile and Link with ITT API 90
Instrument Your Application 91
Minimize ITT API Overhead 93
Clock Domain API 94
Counter API 94
Domain API 95
Marker API 96
Metadata API 96
Relation API 97
String Handle API 98
Thread Naming API 98

Task API 99
Intel® GPA Reference 101
Performance Metrics Reference 102
CPU Metrics..... 102
Device I/O Metrics 103
GPU Metrics 103
Graphics API Metrics 121
Memory Metrics 127
User Interface Reference..... 128
Graphics Monitor UI Reference..... 128
Heads-Up Display (HUD) UI Reference 140
System Analyzer UI Reference 144
Graphics Frame Analyzer UI Reference..... 148
Graphics Trace Analyzer UI Reference 197
Notices and Disclaimers..... 211

Intel® Graphics Performance Analyzers User Guide

1

This documentation provides a comprehensive overview of Intel® Graphics Performance Analyzers (Intel® GPA) functionality, configuration settings, workflows, and instructions on how to use the set of tools for analysis of games and other graphics-intensive applications running on Intel hardware.

Start Here

Visit the official Intel® Graphics Performance Analyzers page to download the product for free.

- Introduction
- What's New
- Get Started
- Intel Graphics Performance Analyzers Cookbook

Introduction to Intel® Graphics Performance Analyzers

Intel® Graphics Performance Analyzers (Intel® GPA) is an application suite that contains tools for analysis and optimization of graphics-intensive applications, including but not limited to Microsoft DirectX*, Vulkan*, and OpenGL* games. With Intel GPA, you can test your applications on various platforms to detect bottlenecks and find the best optimization for each platform. You can also enable/disable certain features (such as detailed terrain features or additional interactive game elements) until you achieve the optimal game-playing experience.

Intel GPA tools display the application performance data, including GPU and CPU metrics, system events and their dependencies, which can help you optimize the application, save time and effort, and get your product to market faster. Intel GPA provides the following tools:



Graphics Monitor

Use the Graphics Monitor tool to access and arrange the set of metrics for graphics performance analysis of Microsoft DirectX, Vulkan, and OpenGL applications.

In Graphics Monitor you can choose desktop or Universal Windows applications for analysis, and open captured frames, streams and traces with the respective tools.



System Analyzer

Use System Analyzer as the first step on your way to application analysis to determine whether your code is CPU- or GPU-bound. If it is GPU-bound, keep on analyzing with Intel GPA tools.



Graphics Frame Analyzer

Use Graphics Frame Analyzer to explore the captured frame and understand the performance impact of specific API calls at different stages of the rendering pipeline. Experiment with API calls and various rendering pipeline settings to identify, locate, and resolve rendering issues in your Microsoft DirectX, Vulkan, and OpenGL applications.

Decrease frame rendering time, see into draw call issues and understand how they impact the FPS at different stages of the rendering pipeline.

Graphics Trace Analyzer

Use Graphics Trace Analyzer to identify problems with distributing the GPU and CPU resources and application data between functions.

Visualize the execution profile of the system events in your code over time across various flows, analyze application-level and system-level performance data, such as thread activity, context switches, API calls, identify synchronization and load balancing issues in your application.

Visualize the dynamics of resource sharing for application engines with multiple discrete tasks, such as collision detection and terrain generation, and more. Trace Analyzer can also help you identify whether your application is CPU- of GPU-bound.

NOTE
If your application is CPU-bound, consider using Intel® VTune™ Profiler, which is a tool to profile serial and multithreaded compute and media client and server applications that are executed on a variety of hardware platforms (CPU, GPU, FPGA). VTune Profiler supports local and remote target analysis on the Windows* and Linux* platforms.

Intel® Graphics Performance Analyzers Framework

Intel® Graphics Performance Analyzers Framework (Intel GPA Framework) is a cross-platform, cross-API suite of tools and interfaces that enables you to capture, playback, and analyze 3D graphics rich applications.

Use Framework in different ways: as a back-end, directly with console tools, and through linkage with libraries. Integrate Intel GPA Framework into CI automated frameworks to flag performance regressions and frame budget reports, automate various performance optimization experiments into scripts to quickly identify common bottlenecks. Capture multi-frame streams to spot performance issues that are hard to detect with single a frame capture, such as overall frame time variance, intermittent glitches, and so on.

See Also

Intel GPA Download Page
Intel GPA System Requirements
What's New

Get Help

Intel® Graphics Performance Analyzers (Intel® GPA) provides the following online resources to support the product users:

Get Started Guide Get Started Guide (Simplified Chinese)	Get an overview of the key product features and learn how to start using the tools on different host systems.
Context-sensitive Online Help	Get help for an active dialog box, pane, or window, by pressing F1. Your system should be connected to the internet. In Graphics Monitor, Graphics Frame Analyzer, and Graphics Trace Analyzer you can also access the online help by clicking the ? button.
Release Notes	Learn what's new in the latest release.
Support Forum	Ask questions, provide feedback on the product, and get help with using Intel GPA.
Intel GPA Cookbook	Find various performance analysis scenarios (recipes) to help you solve real-world specific performance issues using Intel GPA tools.
Documentation Library	View documentation for all the Intel GPA features.

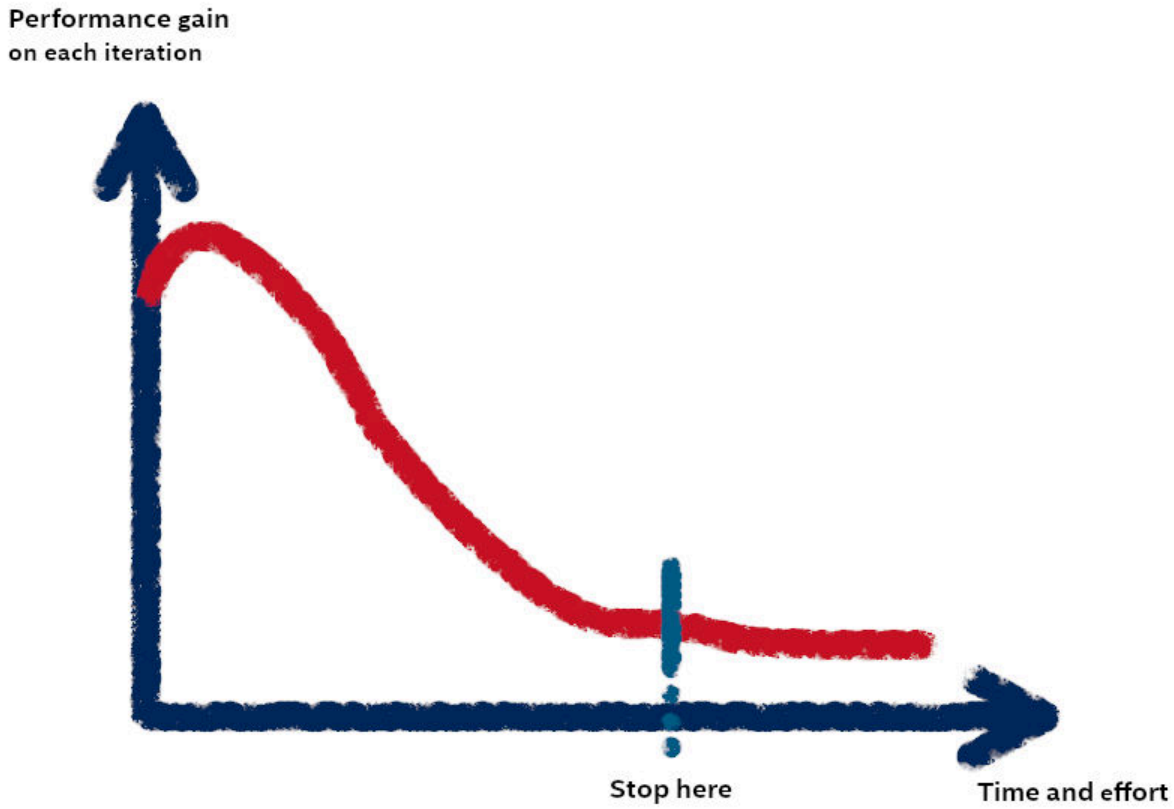
Documentation Archive	Find documentation for Intel GPA versions prior to 2023. Since version 2023 you can use the **Version** drop-down menu to switch between versions of the Intel GPA documentation.
Intel GPA Framework	Learn about Intel© GPA Framework - a cross-platform, cross-API suite of tools and interfaces, which allows you to capture, playback and analyze graphics applications.

Game Optimization Methodology

Optimizing a video game can be an overwhelming task when there is no clear direction. Without any starting information, it can be hard to even determine where to start. Learn how you can optimize your game in a clear and systematic manner using various Intel performance analysis tools.

Set Your Performance Goals

Strictly speaking, you can keep optimizing any piece of software endlessly, and the ratio of performance gain to time and effort keeps declining while you are optimizing an application. Therefore, it is necessary to set a clear performance goal and stop performing optimization when this goal is reached.



First, you need to consider the genre, style, and gameplay elements of your game. For a competitive first-person shooter, stable FPS and low input lag may be the key factors. For a slow-paced storytelling game, visual fidelity may outweigh the need for a >100 FPS value. A point-and-click adventure game with pre-rendered backgrounds may not require any graphics optimization at all.

You also need to consider your target audience and the kind of hardware these people might use. Are you targeting PC enthusiasts with top-level, high-performance hardware, or is your demographic comprised of gamers using laptops with integrated graphics? Several online game vendors can help, as they publish

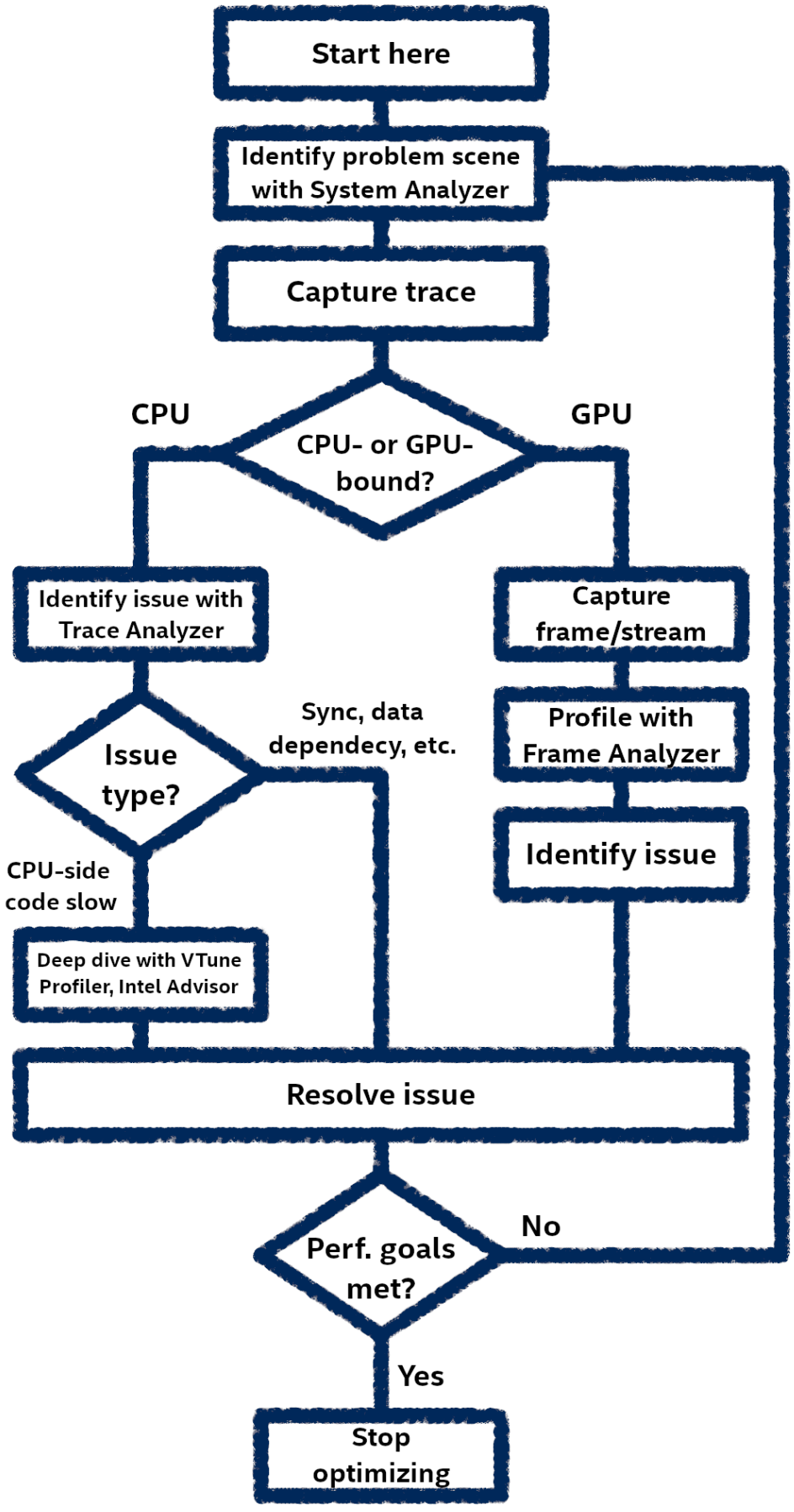
regular monthly and/or annual hardware and software survey results. One example of such global survey is the Steam* Hardware and Software Survey. Such surveys can offer great insight into the kinds of hardware and software real gamers use across the globe.

Together, these two factors will be deciding in setting your performance goals.

An example of a basic set of performance goals for a competitive first-person shooter may look like this:

- A stable FPS value of >=120 FPS for smooth experience on 120 Hz monitors
- Low input lag for precise targeting
- Good server-side performance for fair gameplay and scoring
- Flexible graphics options for maximum reach of people with different hardware
- Game performs well and provides a smooth experience when tested on configurations selected based on own ideas about target audience and hardware surveys

General Optimization Workflow



Optimization is an iterative process. When you have a game that does not perform up to expectations, your typical workflow may look like this:

1. Identify problematic scene.
2. Check if the game is CPU- or GPU-bound on this scene.
3. Identify main performance bottleneck.
4. Drill down and determine root cause.
5. Resolve issue.
6. Check if the game now meets your performance goals.
7. If it does, stop optimizing. If not, go to step 1.

How Intel Tools Can Help

Intel offers a variety of performance analysis tools that, together, cover all steps in this workflow.

The tools that can be the most useful are:

- Intel® Graphics Performance Analyzers

  A feature-rich set of tools that covers all graphics optimization and profiling needs.

  Download
- Intel® VTune™ Profiler

  A comprehensive tool for CPU performance analysis. Comes with a set of analysis types that cover all CPU optimization needs.

  Download

  Product Page
- Intel® Advisor

  A performance analysis tool with deep focus on threading and vectorization.

  Download

  Product Page

Before You Begin

Before you start optimizing a game, it is important to select an appropriate test system to use for performance profiling.

For performance profiling, use a balanced system where the CPU and GPU both belong to the same time period and price range. Align the test system specifications with your performance goals and target audience.

This is particularly important in one of the next steps, which is testing if the game is CPU- or GPU-bound. Profiling your game on a system with a top-notch CPU and an ultra-budget GPU will not yield a useful result, as the game will always be GPU-bound on such system. The same applies to powerful GPUs coupled with budget or obsolete CPUs: knowing that your game is CPU-bound on such systems is hardly representative of real-world scenarios.

Make sure to align the specifications of the test system with the target audience. A graphically simpler game oriented at storytelling should not be tested on ultra-high-performance hardware, as this unnecessarily cuts off a large portion of the audience with less sophisticated systems.

Identify Problematic Scene

Once you formulate your performance goals and configure a test system, you are ready to start with the optimization itself.

Most likely, you will realize that your game is only slow in certain situations.

Some possible examples are, ranging from obvious to more complex:

- Looking at the floor/ground boosts FPS, looking elsewhere drops FPS to unsatisfactory levels.
- Looking at a specific tree or bush model drops FPS dramatically.
- Moving further away from that tree does not improve the situation.
- Activating a light source, such as a flashlight, reduces performance significantly.
- Game is slow for a minute after you enter a new location, then improves.
- Game is slow to load specific models and textures.
- Seemingly sporadic performance drops with no obvious explanation.

One tool that can help make finding problem scenes easier is the System Analyzer of Intel GPA. You can launch your game with the System Analyzer HUD enabled and see the current FPS value and up to four select metric graphs until you find a problematic spot.

You can also track multiple metrics in real time using the System Analyzer window, quickly capture frames or traces when facing a problem, and switch between metric sets to analyze other aspects of performance.

> **NOTE**
> Depending on your test system, it might be useful to analyze the game remotely. In this case, you connect System Analyzer to a remote system running the game, and all metrics are displayed on your client machine. This reduces overhead and minimizes interference as much as possible and is especially useful when your test system is aligned to target audiences with less performing hardware.

To start a game and attach System Analyzer, launch your game through the Graphics Monitor window. Make sure to launch your game in **Trace** mode, since a trace capture is required for the next step.

If the performance issues are sporadic and you find it hard to capture a trace at the right time, set up a trigger. A trigger will automatically capture a trace of frame when a certain condition is met.

Once you find a problematic scene or area, the next step is to capture a trace to determine if the game is CPU- or GPU-bound in this scene.

What is CPU- or GPU-bound

Once you identify a problematic scene, the next important step is to determine whether the game is lagging due to CPU or GPU limitations.

There are 2 potential scenarios:

- **CPU-bound**
  - **Description:** the main limiting factor is the ability of the CPU to timely execute instructions that are supposed to be executed on each frame. The CPU is overwhelmed with one or more heavy tasks, such as game logic, physics, hit detection, etc.
  - **Perception:** in simplest form, this scenario is observed as combination of high CPU load and low GPU load.
- **GPU-bound**
  - **Description:** the main limiting factor is the ability of the GPU to timely execute tasks that are supposed to be executed by the GPU on each frame, such as geometry transformation, shading, post processing, etc.
  - **Perception:** in simplest form, this scenario is observed as combination of low CPU load and high GPU load.

In both cases, the FPS value is low, and subjective experience is unsatisfactory.

Technically, depending on the specific system, your game will always be CPU- or GPU-bound. Once you reach your performance goals, this is not important. However, while performing optimizations, knowing this eliminates a lot of guesswork and gives a clear direction.

Determine if Game is CPU- or GPU-bound

To determine if your game is CPU- or GPU-bound, use the Trace Analyzer tool of Intel® GPA.

You first need to capture a trace. A trace is a file that contains detailed information on the execution of a program.

To capture a trace, you can use the **Capture Trace** button of System Analyzer or press **Ctrl+Shift+T** during gameplay.

Once you capture your trace, open it using the Trace Analyzer tool and look at the main view. You should see all activity, both on CPU and GPU, while the trace was collected. This includes all logical processors, application threads, and multiple GPU metrics.

In the simplest of cases:

- **CPU-bound:** the logical processors are busy executing the application code most of the time, while the GPU is not loaded; for example, the **GPU Busy (%)** metric value is low.
- **GPU-bound:** GPU metrics related to GPU occupation are high (such as **GPU Busy (%)**, **EU Active (%)**), while the CPU cores are mostly idle.

Your case might not be as straightforward. For more details, see the full Trace Analyzer workflow.

If Game is CPU-bound

If your game is CPU-bound, it might be useful to try the following:

- Annotate your CPU-side code with the Instrumentation and Tracing Technology API (ITT API) to see tasks that take too long right in Trace Analyzer.
- Use Intel® VTune™ Profiler to find your hotspot and optimize CPU utilization. Refer to the User Guide and Cookbook to see how this tool can help in your case.
- Use Intel® Advisor if you suspect your application has a problem with threading and vectorization. Refer to the User Guide and Cookbook to see how this tool can help in your case.

Depending on the nature of your CPU-side issue, one or more of these tools can cover your CPU optimization needs.

Examples of possible issues that can lead to suboptimal CPU performance are:

- Suboptimal game logic code. For example, the function responsible for mapping NPC positions to points on a map is implemented poorly, and each map refresh causes a slowdown of the entire game.
- A multi-threaded game does not use synchronization properly, and CPU threads are idle most of the time.

Use the tools provided to isolate your hotspot and resolve the issue.

If Game is GPU-bound

If your game is GPU-bound, your next step is to capture a stream or a frame of the problematic scene to determine the root cause.

A **frame** is a capture of GPU activity during the process of rendering one frame. Open your frame in Frame Analyzer to get plenty of information on how a particular scene was rendered on the GPU.

If you find it hard to capture a frame at the right time, set up a trigger.

Frames are comprised of draw calls, which are calls to the graphics API to perform one action or another. Locate the lengthiest draw calls and determine their root cause using tips, experiments, and other features of Frame Analyzer.

Once you are looking at a frame in Frame Analyzer, you can experiment with the rendering pipeline to isolate a high-level bottleneck. For example, if you use the **2x2 Textures** experiment, which replaces all textures with a 2x2 pixel square, and frame time drops dramatically, this could indicate that there are overly complex textures in this scene.

You can also find the heaviest instructions in your shader code and experiment with it on the fly, without changing the original game code.

If you suspect that the issue persists across multiple frames, try to capture a **stream**. A stream is a sequence of frames, each of which can be opened individually in Frame Analyzer.

See the full Frame Analyzer workflow for details.

Balance Fidelity and Performance



Depending on the nature of your GPU performance problem, you might consider whether simplifying an object model or texture to improve performance will sacrifice visual quality. The answer really depends on the nature of the problem.

Consider one of the examples above, where performance drops when a specific tree appears in view and does not improve when moving away from the tree.

The most probable causes of such a situation are:

- The tree model is overly complex and has too many polygons. In this case, it might be best to simplify the model, sacrificing some visual quality for performance. It is highly likely that most players will not even notice the reduction in fidelity but will appreciate the performance increase.
- The LOD for this tree is not set up properly. In this case, it is best to ensure LOD is properly enabled. There is no sacrifice in visual fidelity here, since, by nature, LOD reduces the visual quality of the model with increasing distance to the viewer.

To summarize, you can find yourself in two situations:

- **Performance improvement requires a sacrifice in visual quality**

  In this case, consider tweaking an object to see how the performance changes with changes to quality. If an unnoticeable reduction in quality results in a large performance improvement, it might be best to adopt the change.
- **Performance improvement does not require a sacrifice in visual quality**

  In this case, consider if your performance goals are met and if the implementation is difficult. Decide if the ratio of gain to implementation efforts is good enough and implement the fix based on this.

Install and Launch Intel® GPA

Install and launch Intel® GPA on Windows or Ubuntu* systems.*

Visit Intel GPA download page to get the product installation package for your OS.

If your application is too heavy or you have multiple systems to analyze on, consider installing the product on both local machine and remote targets and analyze the application in the network mode.

Prerequisites

- Make sure your system satisfies hardware and software requirements.
- To smoothly profile Vulkan applications, make sure you have installed Vulkan SDK and runtimes.

Install and Launch Intel GPA on Windows*

To install the Intel GPA: from the product download page, click the **Windows* Download** button to run the installer.

The default installation path for Intel GPA is `C:\Program Files\IntelSWTools\GPA`.

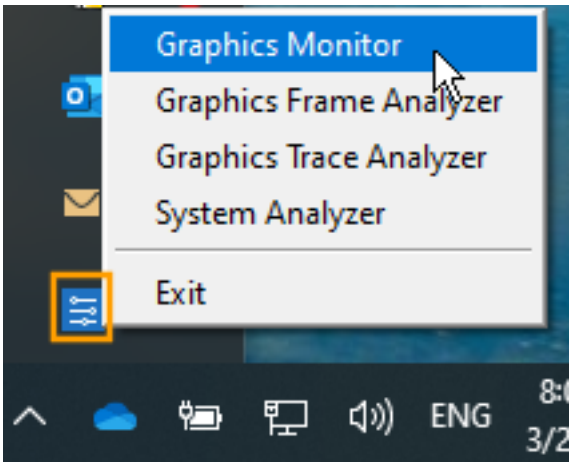
To launch the Intel GPA: from the Windows* **Start** menu, open **Intel® Graphics Performance Analyzers** entry and launch the



Graphics Monitor. Alternatively, you may right-click the



Graphics Monitor icon and select the tool:



Install and Launch Intel GPA on Ubuntu*

To install Intel GPA: from the product download page, click **Ubuntu* Download** button to run the installer with superuser privileges. The default installation path for Intel GPA is `/opt/intel/gpa`.

To Enable Metrics Collection by Intel GPA:

Refer to Enable Time-based and Query-based Metrics Collection.

To Launch Intel GPA:

- From the GUI: click the **Show Applications** button and choose the Graphics Monitor.
- From the CLI: enter `gpa-monitor`.

See Also

Manage Analysis with Graphics Monitor

Enable GPU Metrics Collection

Enable System-wide Time-based GPU Metrics Collection

Use these steps to enable System Analyzer and Trace Analyzer to collect Intel GPU metrics. Afterwards, the available metrics will also be shown in Graphics Monitor under **Options** on the **Metrics** tab in a top-level Intel category labeled with a suffix based on either the GPU name or graphics device ID.

1. Make sure you only have graphics drivers for an Intel GPU installed on the machine. If other GPU vendor's graphics drivers are installed, follow their instructions to remove them. Then confirm that Intel GPU graphics drivers are installed.

   a. Make sure that the Ubuntu* default open source Intel graphics driver package (`xserver-xorg-video-intel`) is installed and loaded. The following command should output text with the substring `xserver-xorg-video-intel`:

```
apt search xorg | grep intel
```

   b. Verify that the Intel GPU i915 kernel module is installed. The following command should output text with the module named `i915` in the left column:

```
lsmod | grep i915
```

2. Enable the Intel kernel module to allow user space applications to perform GPU based metrics collection. On boot, by default, it does not allow user space applications to do this.

   a. Use the following command to manually enable GPU based metrics collection:

```
sudo sysctl -w dev.i915.perf_stream_paranoid=0
```

   b. Optionally, use the following command to add an automated cron job that executes whenever the platform reboots:

```
sudo crontab -e # Add the following line at the end or as the 1st no comment line:
@reboot /sbin/sysctl -w dev.i915.perf_stream_paranoid=0
```

   c. Afterwards, confirm the `perf_stream_paranoid` mode 0 change by using the following command:

```
sysctl -n dev.i915.perf_stream_paranoid
```

3. Install the current version of the Intel metrics-discovery library (`libigdmd.so`). This is an open source library that is periodically updated to support Intel integrated and discrete GPUs. Intel Graphics Performance Analyzer (Intel GPA) tools use the library with its former name, `libmd.so`.

   NOTE These steps are validated on Ubuntu 20.04.

   a. Delete any old versions of the library used by the GPA tools. This helps to ensure that you have the latest version of the open source Intel metrics-discovery library. Run this command:

```
sudo rm /opt/intel/gpa/libmd.so*
sudo rm /opt/intel/gpa/libigdmd.so*
```

   b. Install the base packages required on your machine for compilation with:

```
sudo apt install g++ libdrm-dev
```

   c. Clone and build the Intel metrics-discovery library from the latest source:

```
cd ~/Downloads
git clone https://github.com/intel/metrics-discovery
cd metrics-discovery
mkdir build
cd build
cmake ..
make -j4 # Adjust based on the # of available CPU cores to improve compile time.
```

   d. Install the built Intel metrics-discovery library for the GPA tools to use:

```

sudo cp -a ../dump/linux64/release/metrics_discovery/libigdmd.so* /opt/intel/gpa/
sudo ln -sf libigdmd.so /opt/intel/gpa/libmd.so
```

4. Verify that the Graphics Monitor, Frame Analyzer, and other Intel applications are not running in the foreground or background as follows:

   a. Run the Ubuntu System Monitor.
   b. Search for the string `Intel`.
   c. If `GpaMonitor` is visible in the search results, right-click on it.

   d. Select **End** followed by **End Process**.
   e. Wait a few seconds for the application to shut down.
   f. Repeat steps a-f for any other Intel GPA applications you found.
5. Confirm that Graphics Monitor only shows a single Intel GPU connection in the upper-left drop-down menu.
6. In the **Metrics** tab in a top-level Intel category labeled with a suffix based on either the GPU name or graphics device ID, verify that Intel GPU metrics display under **Options**.

Enable OpenGL* Query-based GPU Metrics Collection in Frame Analyzer

After enabling the GPU metrics collection for System Analyzer and Trace Analyzer, you can perform the following steps to collect more accurate GPU metric values in Graphics Frame Analyzer by locking the GPU frequency:

1. Modify the rights to the following files to add write capabilities:

```
sudo chmod +w /sys/class/drm/card0/gt_max_freq_mhz
sudo chmod +w /sys/class/drm/card0/gt_min_freq_mhz
sudo chmod +w /sys/class/drm/card0/gt_boost_freq_mhz
```

2. Check and record what the current GPU frequency settings are:

```
cat /sys/class/drm/card0/gt_max_freq_mhz
cat /sys/class/drm/card0/gt_min_freq_mhz
cat /sys/class/drm/card0/gt_boost_freq_mhz
```

3. Write the same GPU frequency value in MHz for the three files. Do not exceed the GPU's maximum frequency, For example:

```
echo 1000 | sudo tee /sys/class/drm/card0/gt_max_freq_mhz
echo 1000 | sudo tee /sys/class/drm/card0/gt_min_freq_mhz
echo 1000 | sudo tee /sys/class/drm/card0/gt_boost_freq_mhz
```

After using the Graphics Frame Analyzer, restore the prior settings recorded by following the example in step 3 above, except replace `1000` with the original settings recorded. Alternatively, perform a reboot.

See Also

Manage Analysis with Graphics Monitor

Launch Application for Analysis

Use Graphics Monitor to specify and launch an application for analysis.

Choose the App Launching Mode

You may choose between the following Graphics Monitor options to launch your application:

- **Manual launch**: Start a desktop application or a Windows* Mixed Reality application via the Graphics Monitor.

  This is the recommended option.
- **Automatic detection of a launched app**: Connect to an already running application using the **Auto-detect Launched Applications** option of the Graphics Monitor.

  This mode is recommended for applications launched from game platforms (Steam*, Origin*, Epic Games*, and others) and IDEs (Visual Studio*, Unreal Editor*, and others).
- Ubuntu* only: Launch OpenGL* desktop applications using command line.

Choose Local or Network Usage Mode

With Intel® GPA, you can choose between local and remote (network) analysis of your game:

- In the local mode, your target application and Intel® GPA tools run on the same system.
- In the network mode, the target application runs on a remote system. The performance metrics are tracked on the system where the application runs, while the analysis system collects and displays the received data.

Choose the *local mode* in the following cases:

- System analysis with the HUD. This lightweight tool does not have significant performance impact.
- Frame analysis on high-performance systems.
- Trace analysis.

Choose the *network mode* in the following cases:

- System analysis with the System Analyzer. Running the System Analyzer on a separate system provides the following benefits:
  - Both the System Analyzer and the application can be viewed full-screen on their respective systems.
  - The accuracy of the measurements is not affected. When both the System Analyzer and the application run on the same system, System Analyzer requires CPU and GPU cycles, which can affect the target application performance and alter the accuracy of the measurements.
  - With System Analyzer, you can connect to multiple hosts and profile multiple applications simultaneously.

  NOTE
  If you choose to run system analysis locally, use the HUD to avoid the performance impact.

- Frame analysis if you analyze very complex workloads or target low-performance systems, such as netbooks or Ultrabook™ devices. In this case, it is recommended that you run the analysis in the network mode, with the Intel® GPA tools running on a 64-bit system with at least 4 GB of memory.

  NOTE
  - If the target system hibernates, the connection to the Graphics Monitor is aborted.
  - If the target system goes to the power saving mode or the screen saver starts, you can get incorrect data. To get accurate measurements, restart the Graphics Monitor and the Graphics Frame Analyzer/Graphics Trace Analyzer.

See Also

Manage Analysis with Graphics Monitor

Graphics Monitor UI Reference

Start a Desktop Application with Graphics Monitor

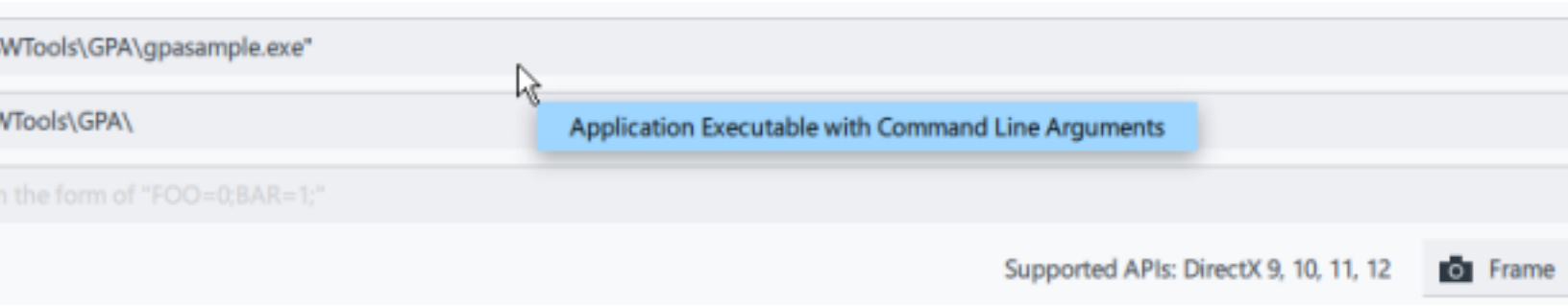
To start your application directly from the Intel® GPA's Graphics Monitor:

1. Launch the Graphics Monitor by double-clicking the icon  in the taskbar notification area.

   The Graphics Monitor Launcher Screen opens with the **Desktop Applications** tab active.
2. In the **Desktop Applications** tab, select an application for analysis:

- In the **Application Executable with Command Line Arguments** field in the lower part of the screen, browse to your application or enter the full application path.

  You can also enter any command-line options required to run your application. If the path to the application contains spaces, put the path in quotation marks.



- If you already analyzed your application with Intel GPA, you can select the application from the list of recently run applications.

3. (Optional) In the **Working Directory** field, type the directory in which the application should run.

   This is useful for games with a launcher or mission packs located in a different directory than the game files.
4. (Optional) In the **Environment Variables** field, type environment variables for the application.
5. Choose the mode to start the application:

   - **Frame** to monitor real-time metrics and capture a frame.
   - **Stream** to capture a sequence of frames until the application is stopped.
   - **Trace** to monitor real-time metrics and capture a trace.

   NOTE

   - You can configure the Graphics Monitor to capture a specific graphics API. This can be useful, if your application uses more than one API. To do this, enable the **Extended Profiling Modes** toggle in the Graphics Monitor **Settings** pane. The mode selector drop-down will feature entries that are specific to different graphics APIs.
   - **Stream** capturing is not supported on Linux* OS.

6. Click **Start** to launch the application with the HUD overlay.

Samples

If you are new to Intel GPA, consider using samples available in the list of applications and click **Start**. This runs a sample application with Intel GPA instrumentation, enabling you to quickly analyze the performance. With this sample, you can verify the correct installation and operation of Intel GPA. This is especially useful if your application appears to have problems when running with Intel GPA.

For example, on Windows use `gpasample.exe`. While working with this application, you can add more buttons to switch on/off 3D scene overrides: **/OVERRIDES** to change the GUI to display; **Toggle vsync**, **Toggle alpha blend**, and **Toggle depth** test buttons. By default, the application does not show these buttons.

See Also

Connect to a Running Application (Windows* only)

Analyze GPU-bound Applications with Graphics Frame Analyzer

Identify Issues in Graphics Application Execution with Trace Analyzer

Graphics Monitor UI Reference

Start a Windows* Mixed Reality Application

Use the Graphics Monitor to launch an application using Windows 10 Mixed Reality platform.*

Prerequisites:

- Make sure the Graphics Monitor is launched on your Windows* system.
- Set up the Head Mounted Display (HMD) used in the Windows Mixed Reality (WinMR) portal by launching WinMR from the start menu and running setup.

  Keep the portal open after the setup is completed.

To launch an application for analysis:

1. Double-click the  Graphics Monitor icon in the taskbar notification area, or right-click it and choose Graphics Monitor to start up the Graphics Monitor Launcher Screen.
2. On the Graphics Monitor Launcher Screen, click the **Universal Windows Applications** tab.
3. Select the desired application from the combo-box list or from the list of recent applications.
4. Click **Start** to launch the application with the HUD overlay.

   For Intel® HD Graphics and Intel® Iris® Graphics, the HUD appears in the HMD, as there is no mirrored display on-screen. For Discrete Graphics cards, the HUD is displayed on the mirrored game-view as well as the HMD.

NOTE
You can press **CTRL+F1** to toggle between different display modes, view real-time metrics and the available override modes.

See Also

Analyze Desktop Graphics Applications

Connect to a Running Application (Windows* only)

You may configure Intel® GPA to detect running Windows application(s), connect to them automatically, and analyze.*

Automatic detection may be useful in the following cases:

- When you analyze applications from game platforms (Steam*, Origin*, Epic Games*, and others) and IDEs (Visual Studio*, Unreal Editor*, and others).
- To save time on starting up Intel GPA tools when you launch your application.

NOTE Enabled **Secure Boot** setting blocks auto-detection. Either turn the feature off or consider starting the application directly from the Graphics Monitor (recommended).

Auto-Detect Launched Apps with Graphics Monitor

To enable automatic detection of a running application, do one of the following:

- Press **Ctrl** and click the Graphics Monitor icon 

in the taskbar notification area.

- Click the **Settings** button



in the upper right corner of the Graphics Monitor window and set the **Auto-detect Launched Applications** setting to **ON**.

As soon as you turn on automatic detection, the Graphics Monitor icon turns green:



.

Whenever any application starts, Intel GPA attaches to it and the HUD automatically displays metrics as an overlay on the application window.

Attach to Specific Processes

Once you enable the **Auto-detect Launched Applications** option, Intel GPA connects to all the processes that use graphics APIs (DirectX*, Vulkan*), including web browsers and non-game applications. To change this, specify the applications that you want/do not want to profile with Intel GPA, once you start them:

1. Turn off **Automatic detection** using Graphics Monitor.
2. Go to the Intel GPA installation directory. By default, it is `C:\Program Files\IntelSWTools\GPA`.
3. Add the application names you want to analyze to the `InjectionList.txt` file or add the application you do not want to analyze to `InjectionSkipList.txt`. When specifying file names, use the wildcard "*" to replace 0 or more characters, and skip the `.exe` extension. Example: "my*game", "basichlsl*".
4. Save the files.
5. In the Graphics Monitor **Settings** dialog box, turn on **Auto-detect Launched Application**.

As a result, whenever your application listed in `InjectionList.txt` starts, Intel GPA gets attached to that process and the HUD is enabled. The tool automatically displays metrics as an overlay on the application window. Applications listed in the `InjectionSkipList.txt` are not attached to Intel GPA.

> **NOTE**
> If the `InjectionList.txt` file contains at least one item, the `InjectionSkipList.txt` file and all other running applications are ignored.

Connect to a Running Steam* App

To run your Steam (or any other supported game platform) application for performance analysis with Intel GPA, do the following:

1. Shut down Steam.
2. In the Graphics Monitor **Settings** dialog box, turn on **Auto-detect Launched Applications**.
3. Re-launch Steam.

   Alternatively, you can browse to Steam from the Graphics Monitor's **Application Executable with Command Line Arguments** filed and click **Start**.
4. Run an application from Steam.

See Also

Manage Analysis with Graphics Monitor

Analyze Applications in the Network Mode

During remote analysis, the application runs on a *target* (remote) system, where the remote instance of Intel® GPA collects and displays the received data. The *host* version of Intel GPA tracks application and system metrics.

Remote analysis is useful in the following cases:

- To analyze application performance on different systems. Intel GPA tools display metrics from one target system at a time.
- To run a heavy application on a remote target but analyze the collected data on more powerful host system. Using system resources by Intel GPA may impact the accuracy of GPU and CPU metrics.

> **NOTE**
> Intel GPA uses 27072 and 27016 ports to enable communication between the host and target platforms. Both ports are secured with cryptographic and secure handshaking mechanisms.

To analyze a remote application:

Prerequisites: Install Intel GPA tool set both on the target and host system.

1. On your target system, launch the Graphics Monitor and start an application.
2. On the host system, launch the Intel GPA tools and connect them to the Graphics Monitor running on the target system as follows:

   For System Analyzer:

   a. At the top of the **System Analyzer** window, specify the hostname or IP address of the target system.
   b. Click the **Select Process** button to choose the application that you have started on the target system.

      Now you can view real-time application metrics and perform analysis.

   For Graphics Frame Analyzer:

   a. In the Graphics Frame Analyzer window, click the **This PC** button in the top left corner and enter the IP address of the target system.

      Optionally, you can also set an alias for the remote system. By default, Graphics Frame Analyzer is connected to the local system.
   b. Click the **Select Process** button to choose the application running on the host system.
3. If you connect to a target system for the first time, Graphics Monitor asks you to authorize the connection. Click **Accept Once** to allow the connection for the current session only, or **Accept Always** to add the system IP address to the list of authorized systems.
4. Once the systems are connected, proceed with the selected analysis type.

   The remote system collects metrics and provides access to performance statistics, while the host system is used to manipulate the analysis remotely (choose calls, resources, etc.).

To use the Graphics Trace Analyzer remotely, capture a trace on the target system and copy it to the host system manually.

> **NOTE**
> - Keep the Graphics Monitor open on the target system, while you are running the analysis, and do not remove the captured frame/stream from the systems.
> - If the target system hibernates, enters power saving mode, or the screen saver starts, you may get incorrect data. To collect accurate measurements, restart Intel GPA tools.

See Also

Analyze GPU-bound Applications with Graphics Frame Analyzer

Monitor Real-Time Performance with System Analyzer

Identify Issues in Graphics Application Execution with Trace Analyzer

Configure Analysis Settings

When profiling your applications, use the Graphics Monitor to configure the analysis settings:

- Configure a Metric Set
- Set Frame Delimiters
- Set Up a Trigger
- Set Up Tracing Options
- Customize Keyboard Shortcuts

Configure a Metric Set

Intel® GPA supports a wide variety of metrics. The metrics available for display might differ depending on your specific graphics device. With the GPA, you can collect two types of metrics:

- Metrics related to frames of the **target application**. For example, the quantity of draw calls or executed pixel shaders within a frame. In Trace Analyzer, you can use these metrics to understand characteristics of a separate frame such as the duration of pixel or vertex shader execution. In Frame Analyzer multi-frame view, you can use them to choose the frame for detailed analysis in profiling view.
- Time-based metrics related to the **system in general**. For example, the percentage of time during which the GPU is busy. You can use these metrics to understand whether your application is CPU- or GPU-bound.

To configure the metrics that are collected, do the following:

1. Start the Graphics Monitor.
2. On the Graphics Monitor Launcher screen, click the **Options** button.
3. Go to the **Metrics** tab and configure the available metrics options:
    - Select the GPU Sampling Interval
    - Enable/disable GPU metrics collection by the target application
    - Customize the Heads-Up Display (HUD) position
    - Configure the HUD metrics
    - Select the active metric set
    - Change a D3D Device for the Instrumented Process (Windows* only)

Select the GPU Sampling Interval

When you set **GPU Sampling interval** to **Frame**, metrics are only collected from the target application and are sampled every frame. For instance, if FPS is 60, Graphics Monitor collects metrics every 16.7 milliseconds.

Frame is the default value for **GPU sampling interval**. This mode in enough for high-level analysis, such as defining whether the application CPU- or GPU-bound or the overview of the rendering process with Trace Analyzer.

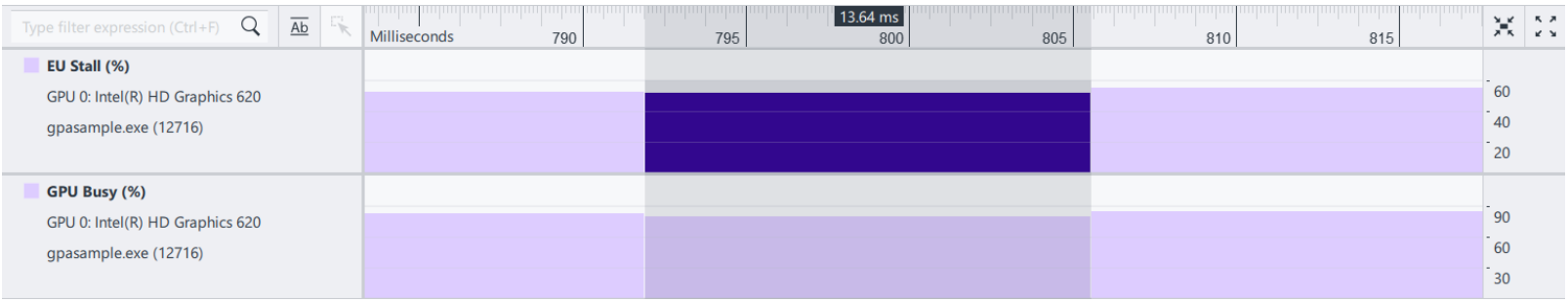
> **NOTE** You cannot collect metrics into a trace file when **GPU Sampling interval** is **Frame** and **Parallel Event Execution (DirectX 11)** toggle button in the **Trace** tab is ON.

When you set **GPU Sampling interval** to any other value (400, 200, 100, 50, 25, 12 microseconds), system-wide time-based metric collection is enabled. In this mode, Graphics Monitor collects metrics from the entire system with the selected frequency.

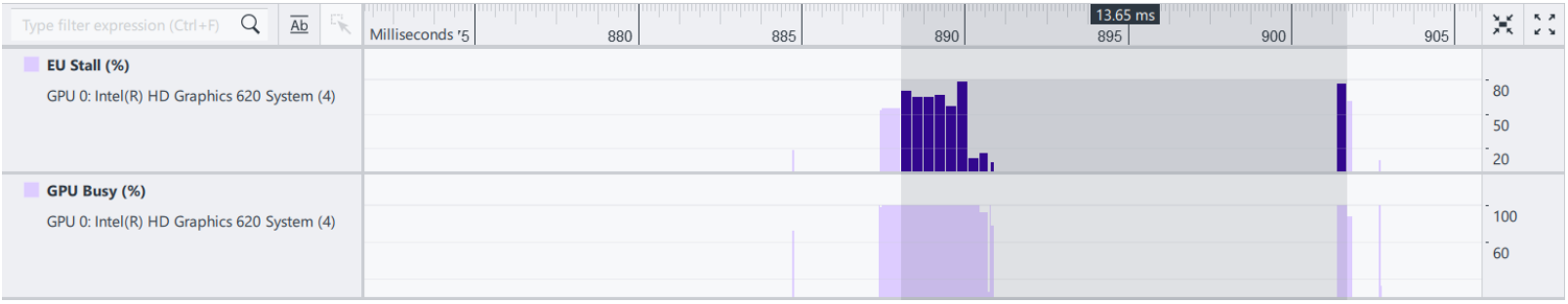
The interval in this case is much shorter than the frame duration, so that you can analyze metrics inside the frame. Use this mode for low-level analysis with high metrics granularity.

Below you can see an example of how the metrics for the interval of 13 milliseconds are displayed in the Trace Analyzer depending on the GPU sampling interval.

GPU sampling interval is Frame:



GPU sampling interval is 400 microseconds:



NOTE

- This mode may interfere with profiling if other GPU-intensive applications are running. Close other GPU-intensive applications when using any sampling interval other than **Frame**.
- Short sampling interval can lead to performance issues on some systems, although it allows you to collect more data. The larger the interval, the smaller the file size and the faster the file opens.

Enable/Disable GPU Metrics Collection by the Target Application

The **GPU metrics collection by the target application** setting is specific for applications that use Intel® Metrics Discovery Application Programming Interface directly, for example, Unreal Engine* apps. By default, this toggle button is ON. You need to turn it OFF to enable the GPA to collect metrics from such applications. Otherwise, the GPA is not able to collect metrics for any type of analysis.

Select the Active Metric Set

You can use the predefined sets of GPU metrics to focus your analysis on particular issues, for example, rendering or compute.

GPU metrics are logically split into metric sets. For the analysis, you can use one metric set at a time due to GPU restrictions. The choice depends on the game and the focus of your analysis. For example, you can choose **RenderBasic** set to get rendering details and **ComputeBasic** set to look into computing issues.

NOTE

- Metrics may overlap within different metric sets.
- The list of available metrics depends on the hardware you use.

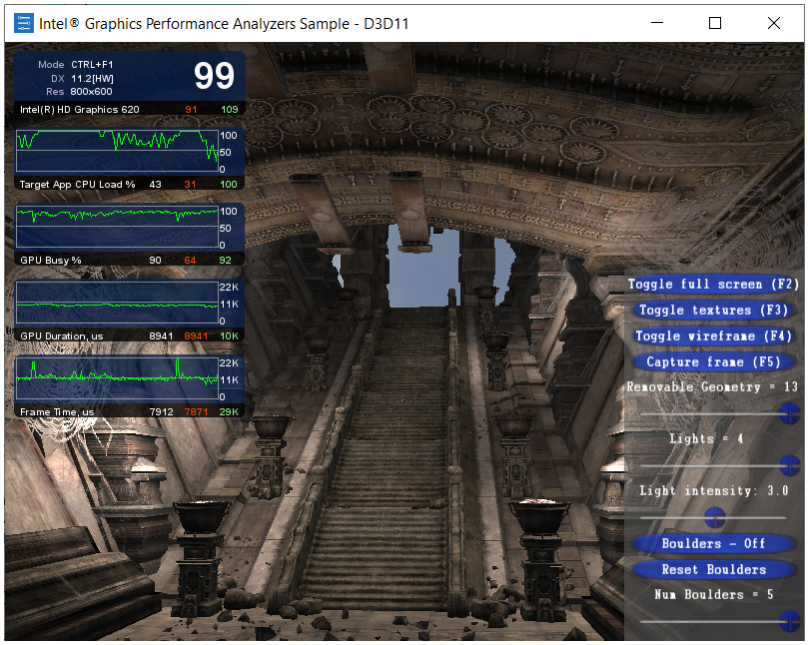
Select the active metric set by using the **Active Metric Set** drop-down menu in the GPU section of the metric tree.



Configure the HUD metrics

Heads-Up Display (HUD) is an overlay provided at the top left of the screen when you launch an application for analysis. The HUD displays system and application statistics on application performance using the GPU and CPU performance metrics of your choice.

The HUD enables you to see the metrics in real time, make first assumptions, and detect potential bottlenecks for further detailed analysis with Trace or Frame Analyzer. You can select up to four metrics that are overlaid on your running application. For the first time, you can choose basic metrics such as **Target App CPU Load**, **GPU Busy**, **GPU Duration**, **Frame Time**.



NOTE The HUD displays metrics only when you capture frames.

A basic scenario is to run the application with the HUD overlay, analyze the FPS rate, find a problematic scene, and capture a trace to find out whether your application is CPU- or GPU-bound. In some cases, you can determine whether your application is CPU- or GPU-bound immediately. For example, if the HUD displays that the CPU load is 100% while the GPU is only 30% utilized, your app is CPU-bound.

Configure HUD metrics the following way:

- Select a metric from the list of available metrics. Drag and drop a desired metric to the **Active HUD Metrics** list at the bottom of the **Metrics** tab.
- Reorder the Active HUD Metrics list by dragging and dropping a desired metric over another active metric.
- Remove a metric from the Active HUD Metrics list by dragging and dropping the metric back to the list of available HUD metrics. Or by hovering over the metric and clicking the X button.

Change a D3D Device for the Instrumented Process (Windows* only)

If your graphics application works with multiple D3D devices, you can choose the D3D device for the instrumented process to collect metrics and show the HUD.

For example, you can change a D3D device of the instrumented process for Mozilla* Firefox*, if the System Analyzer HUD is displayed on the title bar instead of the tab of interest.

1. Launch your application that works with multiple D3D devices, as described in Analyze GPU-bound Applications with Graphics Frame Analyzer.
2. Enable the HUD Display Mode by using the Ctrl+F1 keyboard shortcut.
3. Change D3D devices by using the Ctrl+F5 keyboard shortcut.

See Also

Performance Metrics Reference

Identify GPU-CPU issues with Graphics Trace Analyzer

Heads-Up Display (HUD) UI Reference

Set Up a Trigger

Set Up Tracing options

Customize Keyboard Shortcuts

Graphics Monitor Configuration Window

Graphics Monitor Options Window

Customize Keyboard Shortcuts

Applications can intercept some keyboard shortcuts. In this case, you can assign custom keyboard shortcuts, do the following:

1. In the Graphics Monitor Launcher screen, click the **Options** button.
2. Go to the **Shortcuts** tab and click the desired keyboard shortcut combination to modify it in the editable field. For example, to change the System Analyzer HUD Display Mode to Ctrl+Shift+A, use the Ctrl, Shift, and A keys on your keyboard.

Some games and applications trap all keystrokes, so keyboard shortcuts within the Intel® GPA do not work. In this case, to create frame/trace capture files you can:

- Use Capture Frame / Capture Trace buttons of the System Analyzer.
- Set up a trigger in the Graphics Monitor to automatically create frame/trace capture files when certain conditions occur, for example when FPS is 20.

See Also

Heads-Up Display (HUD) UI Reference

Configure a Metric Set

Set Up a Trigger

Set Up Tracing options

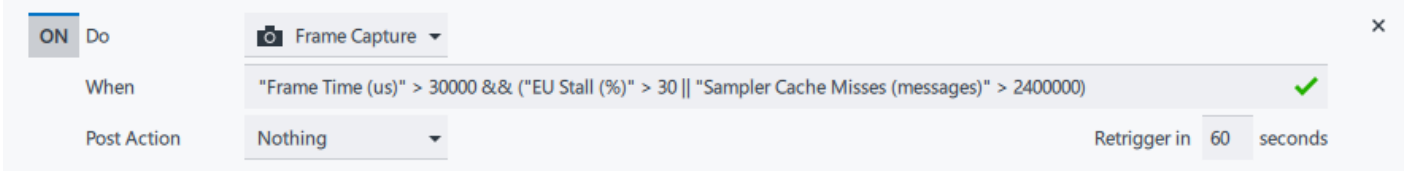
Graphics Monitor Configuration Window

Graphics Monitor Options Window

Set Up a Trigger

Triggers enable you to capture a frame or a trace automatically when a certain condition is satisfied. The conditions are specified in the form of custom logical expressions.

For example, you can configure the Graphics Monitor to capture a frame when the **Frame Time** is high and either of the **EU Stall** or the **Sampler Cache Misses** values are high. This combination that can be observed when occasional stutters occur within a game.



Such triggers can be useful for solving sporadic performance issues that are difficult to isolate manually.

Follow these steps to set up a trigger:

1. In the Graphics Monitor Launcher screen, click the **Options** button.
2. Open the **Triggers** tab and click the ON/OFF toggle to enable a trigger.
3. Select the desired action from the **Do** drop-down. You can choose between **Frame Capture**, **Trace Capture**, and **System Trace Capture**.
   - Select **Frame Capture** option if you aim at analyzing a problematic scene and need to understand GPU activity during the process of rendering a particular frame.
   - Select **Trace Capture** if you aim at analyzing a problematic scene in a particular application that can be related to synchronization issues or CPU and GPU activity during execution.
   - Select **System Trace Capture** if you need to analyze how the processes interact within the system. In this mode, data about CPU and GPU activity is collected from various applications at a time.
4. In the **When** box, specify your condition for the trigger. Conditions are true/false expressions that contain metrics, numbers, logical operators, and simple functions. To create an expression:
   a. Type "" (double quotes) to get a drop-down list of available metrics and choose a metric.
   b. Treat the metrics as operands in a Boolean expression and use the available operators and functions to create a custom logical expression. Available operators and functions:

+, -, *, /	Basic arithmetic operations
&&	Logical AND
||	Logical OR
!	Logical NOT
>, <	Greater than, Less than
>=, <=	Greater or equal, Less or equal
==	Equal
!=	NOT Equal
max(value1, value2)	Returns the largest of two values
min(value1, value2)	Returns the lowest of two values

NOTE

- The list of available metrics depends on the GPU sampling interval selected in the **Metrics** tab.
- The max() and min() functions can be nested and support metric values as parameters.
- Graphics Monitor checks the expression correctness dynamically and displays an error or warning icon in the **When** box if an issue is detected.
- In the expression, you can combine metrics from multiple sources, including both CPU and Intel GPU metrics.

5. Select the action to perform after the trigger execution from the **Post Action** drop-down menu. Available options are:
   - **Nothing**: no action is performed
   - **Terminate**: close the application
6. Specify the retrigger delay. The retrigger delay is the amount of time since the last trigger activation during which the trigger is ignored. This option can be used to prevent multiple subsequent trigger activations and to avoid capturing multiple duplicate frames or traces. The delay also provides time to write the capture files to disk.

7. (Optional) Add another trigger by clicking the



Add new trigger button to have multiple triggers working simultaneously.

NOTE This option is only available for Microsoft DirectX* and Vulkan* APIs.

See Also

Performance Metrics Reference

Identify GPU-CPU issues with Graphics Trace Analyzer

Heads-Up Display (HUD) UI Reference

Configure a Metric Set

Set Up Tracing options

Customize Keyboard Shortcuts

Graphics Monitor Configuration Window

Graphics Monitor Options Window

Set Up Tracing Options

With Intel® GPA, you can capture traces, records of activity on the CPU and GPU, and analyze them with Graphics Trace Analyzer.

To modify tracing options:

1. In the Graphics Monitor Launcher screen, click the **Options** button.
2. Go to the **Trace** tab and configure the available tracing options:
   - To capture a trace from the very first scenes, enable the application startup data capturing with the **Capture Application Startup** toggle button. Capture starts automatically at application startup and ends when the trace buffer is full.
   - Enable **Trace System View in System Analyzer**, if you are interested in system data rather than detailed application execution data such as API calls or debug regions. In this mode, only system data, for example, GPU utilization, is available for analysis.
   - To collect the necessary data for analysis and use disk space efficiently at the same time, set the duration of the application run to be captured in a single trace file. The default trace duration is 5 seconds.
   - Select predefined Intel® GPA domains and user ITT domains.
   - Add system Event Tracing for Windows* (ETW) providers.

NOTE The default settings represent the common use case.

Select Intel® GPA Domains and User ITT Domains

You can select predefined Intel® GPA domains - sets of data to collect about CPU and GPU activity. Available domains are the following:

- **Graphics API** domain enables you to collect calls from graphics APIs, such as DirectX, for all CPU threads.
- **ID3DUserDefinedAnnotation**, **ID3D11DeviceContext2 Events**, and **WinPixEventsRuntime** domains enable you to collect data from the debug annotation APIs. It is not recommended to use these domains in the code together and enable these domains simultaneously in the **Trace** tab.
- **OpenGL debug output** domain enables collecting debug information from OpenGL* API.
- **OpenCL API** domain enables you to collect OpenCL™ data: function calls, execution time for a particular task, and relation between the tasks of submitting and executing a particular kernel. The data is useful to spot synchronization issues or understand whether there is a problem in OpenCL if you use different APIs.
- **Intel Video Processing** domain enables capturing Intel® Media SDK and Intel® Video Processing Library (Intel VPL) events.
- **Parallel Event Execution (DirectX 11)** domain enables you to track parallel execution for DirectX* 11 applications. Parallel Execution track in Graphics Trace Analyzer visualizes how the driver parallelizes execution of submitted render events.
- **Metrics** domain enables collecting CPU and GPU metrics.

You can also select user domains in order to filter the subsets of instrumentation that are collected into the output trace capture file.

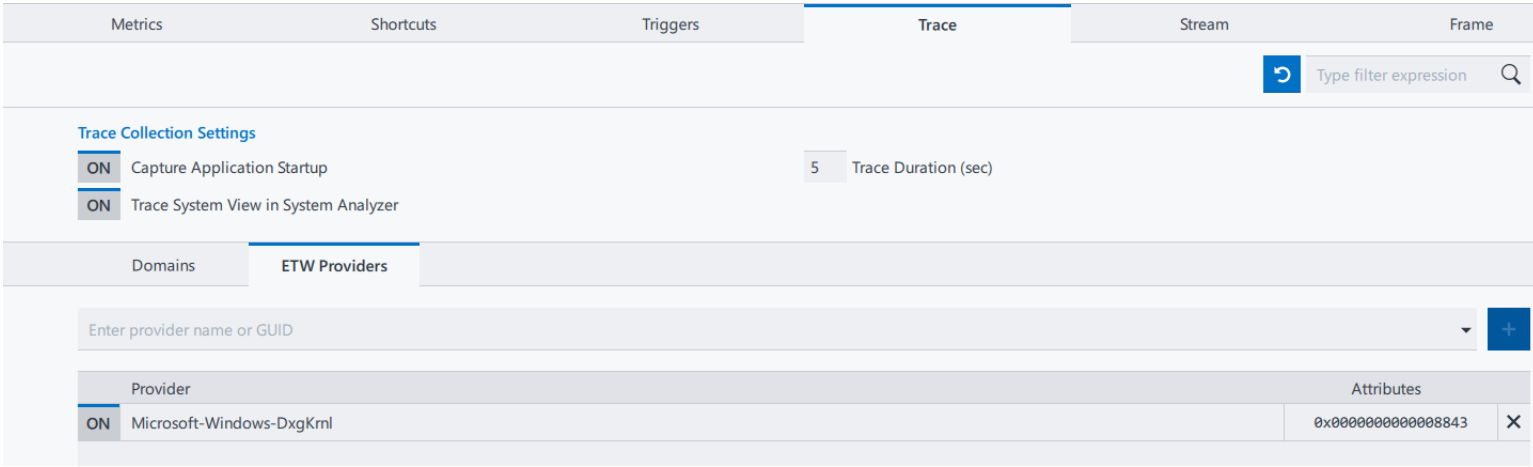
The list of user domains is gathered from the analyzed applications and is updated upon exit from the application. You can define user domain with Instrumentation and Tracing Technology API (ITT API). Intel® GPA supports ITT API in the same way as the debug annotation API.

Add System ETW Providers

You can use the Event Tracing for Windows technology to collect events from custom event providers. By default, Graphics Monitor displays a list of providers already registered in the system. You can select a default provider or create a custom one. You can also use the bitmask to filter specific events.

To configure ETW providers:

1. In the Graphics Monitor Launcher screen, click the **Options** button.
2. Navigate to the **Trace** tab and switch to the **ETW Providers** tab.



3. Click the **Enter provider name or GUID** field to get a drop-down list of available event providers.
4. Select the desired event provider and click the



Add button. The provider is added to the list of active providers.
5. In the **Attributes** field, specify the bitmask for the desired events.

NOTE

- You can enable or disable individual providers at any time by clicking the ON/OFF toggle.
- When you capture a trace with custom ETW providers enabled, the events from these providers will be grouped by provider name in the Trace Analyzer Trace Content pane.

See Also

Identify GPU-CPU issues with Graphics Trace Analyzer

Instrumentation and Tracing Technology APIs Support

Graphics Trace Analyzer: Platform View Window

Graphics Monitor Configuration Window

Graphics Monitor Options Window

Set Up Stream Options

With Intel® GPA, you can capture streams - sets of consecutive frames - and analyze them with Frame Analyzer. To modify the stream options:

1. In the Graphics Monitor Launcher screen, click the **Options** button.
2. Go to the **Stream** tab and configure the available stream options:
   - Enable/disable deferred stream capture (DirectX 12, Vulcan)
   - Create keyframes
   - Create screenshots
   - Enable/disable HUD layer
   - Enable/disable metrics collection

Enable/Disable Deferred Stream Capture (DirectX 12, Vulcan)

By default, stream capture starts when you start the application. The **Deferred stream capture** option allows you to choose a particular scene in your application and capture a stream manually from the beginning till the end of that scene.

When enabled, press L (default) to start or stop stream capture.

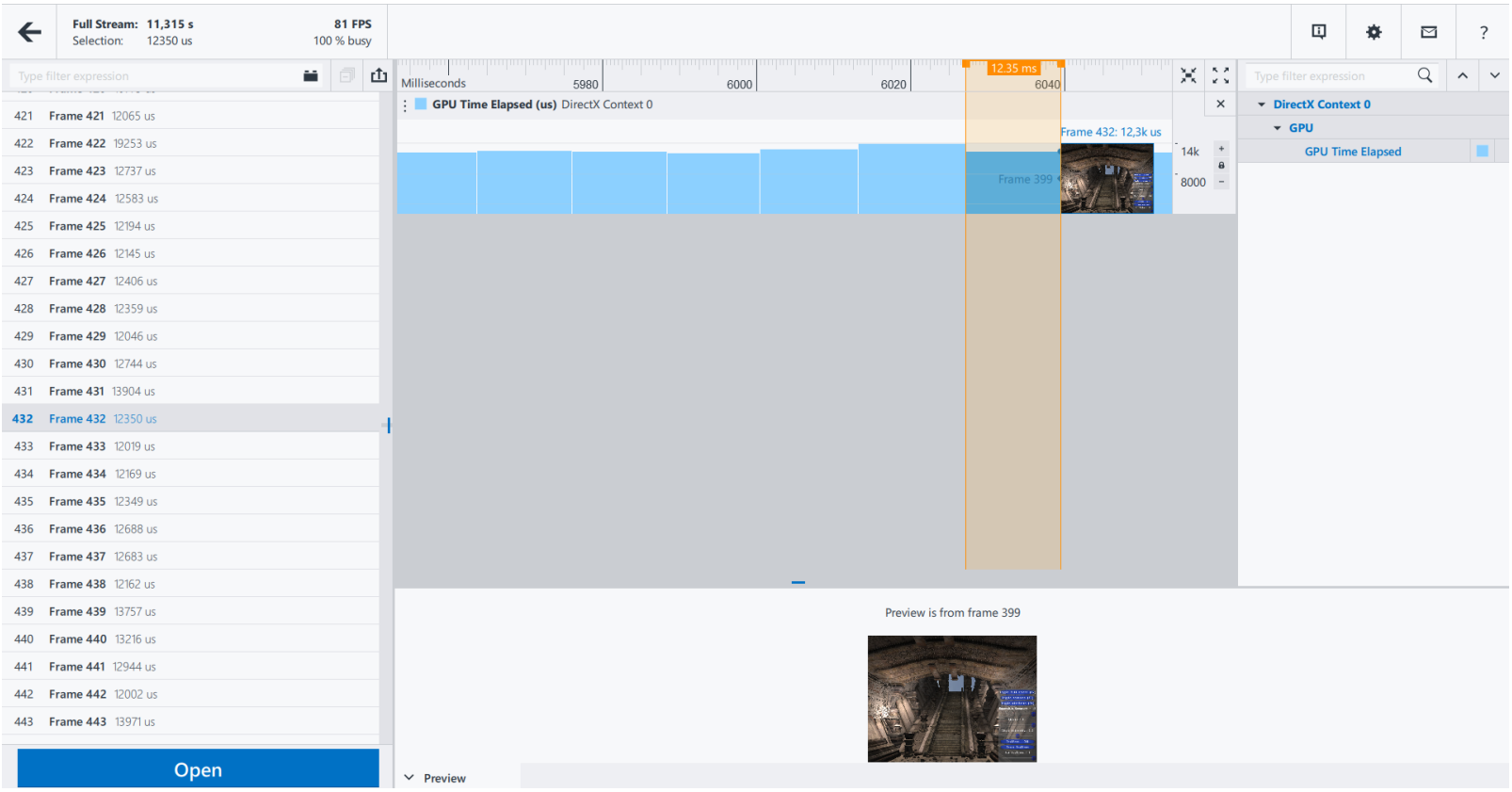
Create Keyframe at Time Interval or at Key Press

A keyframe is a saved state of an application, from which you can play back the captured stream. When you select a frame from the stream for analysis, Graphics Frame Analyzer plays this stream from the beginning till the selected frame to open it. When you set keyframes, Graphics Frame Analyzer starts playing the stream from the keyframe closest to the selected frame, which saves time.

When enabled, press K (default) to create a keyframe.

Create Screenshot Every N Frames or Milliseconds

The **Preview** pane of Frame Analyzer displays screenshots captured for the opened stream. This helps you find the frames you are interested in by seeing the picture.



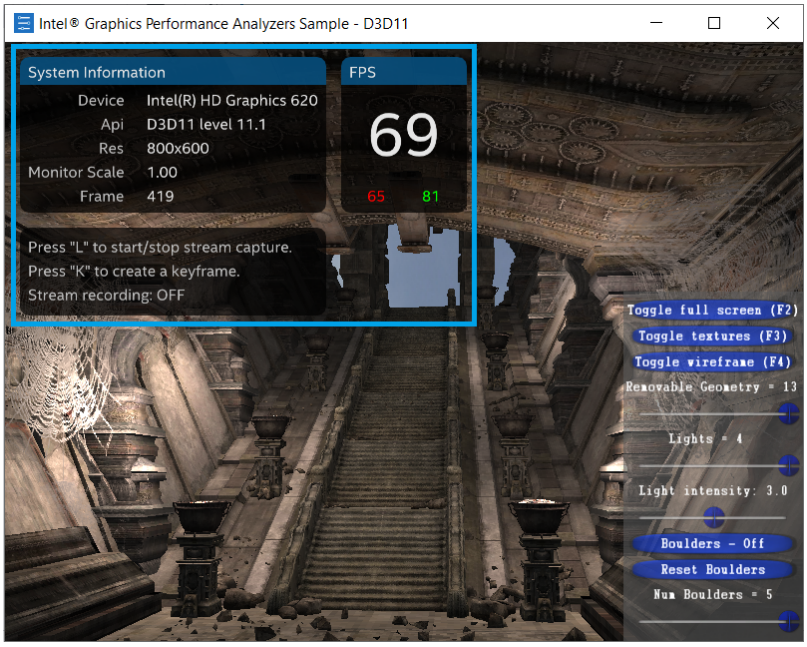
NOTE

- Screenshot creation can cause additional overhead. Disable this option if you suspect that there is excessive overhead.
- "No preview is available for current selection" message may appear for the first frames, if you enable screenshots creation.

Enable/Disable HUD Layer

HUD works in the background while your game is running and displays an overlay on your game, allowing you to quickly see the performance overview of your game.

With the HUD layer, you can find a problematic scene with low FPS and capture a stream of this scene for further in-depth analysis.



Enable/Disable Metrics Collection

When you enable GPU and CPU metrics collection, Frame Analyzer displays them in the Multi-frame View window, allowing you to find problematic frames.

See Also

Analyze GPU-bound Applications with Graphics Frame Analyzer Use Graphics Frame Analyzer to locate GPU bottlenecks, root-cause rendering issues, and increase performance.

Graphics Frame Analyzer Window: Multi-frame View

Set Frame Delimiters

By default, the delimiters for frame capturing of an OpenGL* application are Swapbuffers. But you can change the frame delimiters to suit the needs of your application.

To set frame delimiters for OpenGL applications, do the following:

1. On the Graphics Monitor Launcher screen, click the **Options** button.
2. In the **Frame** tab, enable the required OpenGL frame delimiter.

Available OpenGL Frame Delimiters:

- SwapBuffers
- SetCurrentContext
- glBlitFramebuffer
- glClear
- glFinish
- glFlush

> **NOTE** Although the tools may appear to work with the OpenGL API on a Windows* platform, this API is not officially supported on Windows* platforms.

See Also

Graphics Monitor Configuration Window

Graphics Monitor Options Window

Analyze Desktop Graphics Applications

When profiling desktop API graphics applications with Intel® Graphics Performance Analyzers (Intel® GPA), you can:

- Collect and display hardware and software metrics data from your application in real time and conduct Microsoft* Direct3D pipeline experiments using System Analyzer or System Analyzer HUD. This can help you understand the high-level performance profile of your graphics application, determine whether your application is CPU-bound or GPU-bound, and quickly isolate graphics bottlenecks.
- Create stream files for further analysis with Graphics Frame Analyzer (for DirectX 11 and Vulkan).
- Create frame capture files, which contain all Microsoft DirectX* context used to render the selected 3D frame, as well as GPU metrics per draw call/region.
- Collect real-time trace data during the application run for further analysis with Graphics Trace Analyzer.
- Understand the performance of your application at the frame level, render target level, and draw call level with Graphics Frame Analyzer:
  - Experiment with individual events (that is, any call that potentially renders one or more pixels to the frame buffer or does another GPU work) and various settings for the entire rendering pipeline.
  - Modify states and shader code to see whether it is possible to improve render time.
  - Determine whether texture bandwidth is a performance bottleneck.
  - Minimize overdraw by analyzing pixel history
  - Conduct "what if" optimization experiments without recompiling or rebuilding your application.
- Visualize the execution profile of the various tasks in your code over time using Graphics Trace Analyzer:
  - Explore GPU usage and analyze a software queue for GPU engines at each moment of time.
  - Analyze graphics API calls (draw calls, buffer locks, resource updates, presents).
  - Correlate CPU and GPU activity and identify whether your application is GPU or CPU bound.
  - Identify GPU and CPU application frame rate and how it depends on vertical synchronization.
  - Explore the performance of your application per selected GPU metrics over time.
  - Analyze GPU usage per DMA packet on a software queue.

No code modifications or special libraries are needed to determine whether your game is CPU or GPU bound and to figure out what is happening within a specific frame of your game.

Additionally, you can instrument your application with the Instrumentation and Tracing Technology API (ITT API) to visualize the execution profile of the various tasks in your code over time in the Graphics Trace Analyzer: just add calls within your game code to designate logical tasks in your game. You can also use System Analyzer to profile non-DirectX* applications instrumented by the ITT API. In this case, the Metric Tree Control pane shows only CPU metrics and only the Capture Trace button is enabled. For detailed information regarding the ITT API, see the Instrumentation and Tracing Technology API Support topic.

NOTE Intel® GPA is not supported on 32-bit systems.

Performance Analysis Workflow

Graphics Monitor is the main hub tool for application analysis. You must have Graphics Monitor running on your target system to start an application for analysis.

1. Launch the Graphics Monitor to start the application and configure analysis settings.
2. Connect your Intel GPA tools installed on the host system to the target system using the target system IP address. If you are analyzing an application locally, you can specify "This Machine" or "localhost" instead.

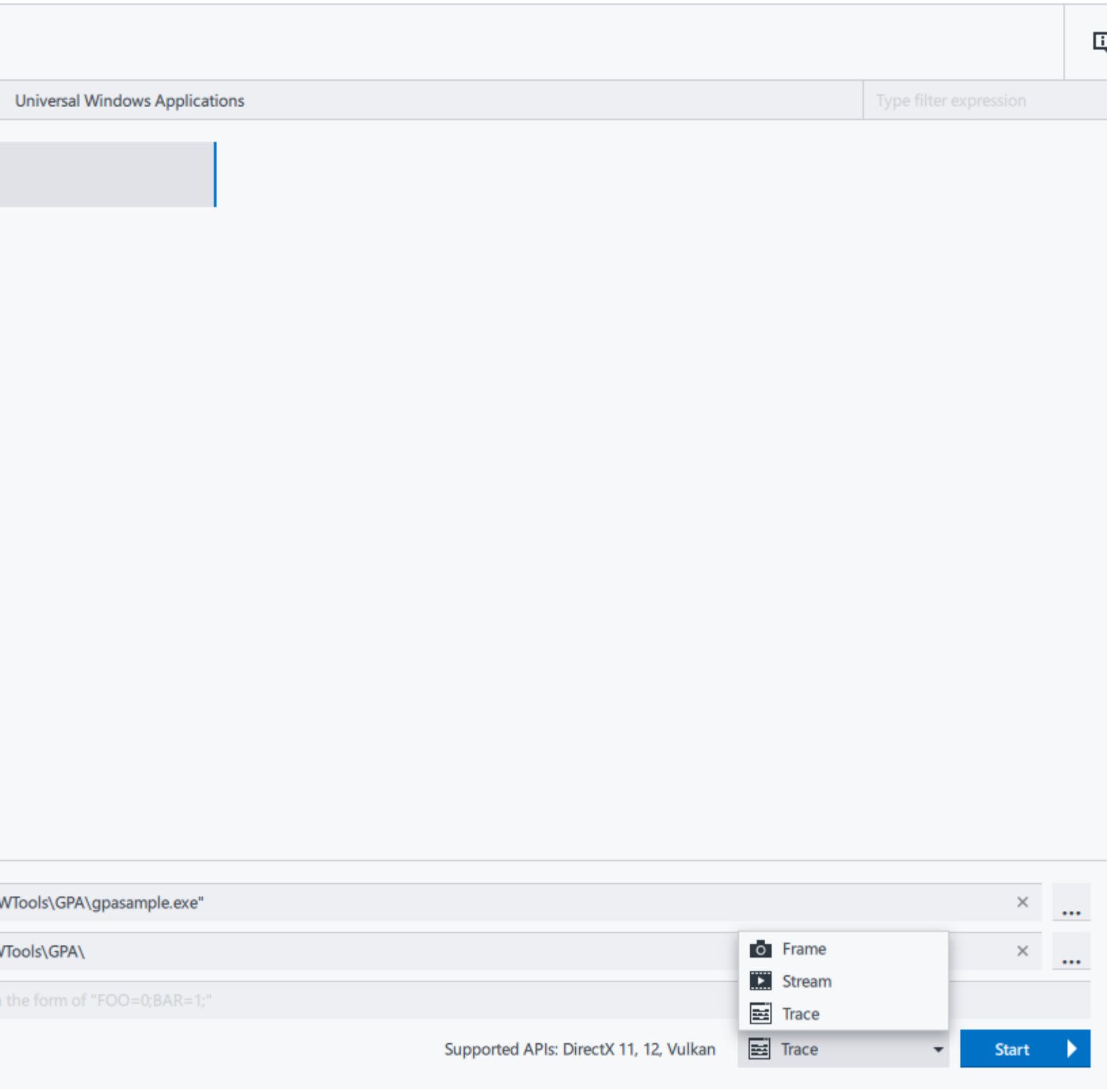
NOTE When you connect to a target system for the first time, Graphics Monitor asks you to authorize the connection. Click Accept Once to allow the connection for the current session only, or Accept Always to add the device IP address to the list of authorized devices.

3. Perform system analysis with System Analyzer or HUD.
4. Capture stream, frame or trace files for further in-depth analysis:
   - Perform stream analysis to detect frames with potential performance bottlenecks .
   - Perform frame analysis to understand the performance impact of specific draw calls at different stages of the rendering pipeline.
   - Perform platform analysis to analyze your application performance with respect to CPU and GPU utilization.
5. Change your game code and re-run Intel GPA to verify that your changes achieve the expected performance improvements.

Manage Analysis with Graphics Monitor

Graphics Monitor is the hub tool of the Intel© GPA suite. It enables you to configure analysis settings, capture frames, streams and traces and open them using the respective tools.

You can use Graphics Monitor to launch applications for analysis and manage different aspects of profiling.


Universal Windows Applications
Type filter expression
WTools\GPA\gpasample.exe"
VTools\GPA\
the form of "FOO=0;BAR=1;"
Frame
Stream
Trace
Supported APIs: DirectX 11, 12, Vulkan
Trace
Start

Manage Analysis Options

Before you begin your analysis, it is recommended to configure your analysis options using the **Options** window to align the tool behavior with your needs.

See the Configure Analysis Settings section for details.

Launch Applications for Analysis

To launch your applications for analysis using Graphics Monitor:

1. Select an application for analysis. To do that, browse to the application binary and working directory using the text boxes. Alternatively, select a previously launched application from the application history.
2. Select the application startup mode from the **Application Startup Mode** drop-down menu at the bottom. Select the mode that corresponds to the type of performance data you want to capture. There are three modes available:

   - **Frame:** choose this mode to monitor real-time metrics and capture a frame. This mode supports DirectX* 9, 10, and 11 applications.
   - **Frame [DX12]**: choose this mode to monitor real-time metrics and capture a frame for DirectX 12 applications. This mode supports DirectX 12 Ultimate features, like Mesh Shading and Ray Tracing.
   - **Trace:** choose this mode to monitor real-time metrics and capture a trace.
   - **Stream:** choose this mode to capture a sequence of frames. This can be useful for isolating sporadic performance problems that only affect a subset of frames.

   NOTE

   - If your application uses multiple graphics APIs, you can use the **Extended Profiling Modes** toggle in the Graphics Monitor **Settings** pane. When enabled, the launch modes drop-down menu is extended, allowing you to explicitly specify the graphics API to use.
   - **Stream** capturing is not supported on Linux* OS.

3. Press **Start** to run the application.

   Your application starts with the Heads-Up Display (HUD) enabled. HUD is an overlay provided at the top left of the screen when you launch the application. HUD displays system and application statistics on application performance using the GPU and CPU performance metrics.

Next Steps

Once you launch your application, use the HUD to track performance in real-time. Capture a frame, trace, or a stream to investigate the performance issue deeper using other tools of the Intel© GPA toolset.

NOTE
Monitoring options in the HUD for DirectX 12 applications differ from the previous API versions and will be evolving with the next product versions.

Monitor Real-Time Performance with System Analyzer

Use System Analyzer to locate problematic areas and to determine if your application is CPU- or GPU-bound.

A top-down approach to analysis and optimization suggests that first you have to determine whether your performance issues are with the CPU or the GPU. Understanding your performance issues at this highest level is extremely important. Otherwise, you could spend a lot of time optimizing parts of your game that end up being secondary to much larger performance issues.

A good rule is to have a balance between the utilization of the CPU and the GPU resources - do not have the CPU running at 100% while the GPU is only 20% utilized, and vice versa.

To find out whether your application is CPU-bound or GPU-bound, you can use heads-up-display (HUD) or System Analyzer running in a host/target mode.

System Analyzer vs Heads-up Display (HUD)

HUD works in the background while your game is running and displays its results as an overlay on your game, quickly allowing you to see the performance overview of your game. HUD displays the following data: system information, FPS, and four performance metrics selected in the Graphics Monitor Options.

System Analyzer is a separate tool that displays a variety of performance metrics at a time. System Analyzer offers the following advantages over HUD:

- Uses host/target (network) mode, enabling you to monitor performance on a remote system. This reduces performance impact on the application. You are recommended to use System Analyzer in the network mode to analyze graphics applications on Ultrabook™ devices, tablets, netbooks, or other devices with limited resources.
- Provides more detailed metrics resolution and the possibility to view more than four metrics at the same time, while also providing counter set selection and multiple other options. Using System Analyzer, you can compare an unlimited number of metrics with same measurement units by displaying them in the same chart, easily switch between metrics, and resize metrics charts for better analysis.

If you connect the System Analyzer to the application with HUD enabled, HUD is switched to the lightweight mode that only shows FPS rate.

System Analysis Workflow

1. Start your game from the Graphics Monitor

   Open Graphics Monitor and select the application to run in the Graphics Monitor Launcher screen.

   This runs your game with the HUD overlay, quickly allowing you to analyze the performance data and capture frame and trace files.
2. (Optional) Launch System Analyzer

   Return to the Graphics Monitor and click the

   Connect System Analyzer button next to your application.

   System Analyzer opens and attaches to your application. By default, System Analyzer starts in System View mode, which displays general system metrics. When it's attached to the application, System Analyzer provides more detailed metrics.
3. Analyze metrics data

   Intel GPA supports a wide variety of metrics. The metrics available for display might differ depending on your specific graphics device. Using real-time metrics, you can determine if your application is CPU- or GPU- bound, or identify the problematic area for further detailed analysis. For example, you can detect scenes, where:

   - Frame rendering takes too long: analyze the **Frame Time** metric.
   - CPU and GPU workload is imbalanced: evaluate CPU and GPU load by analyzing **GPU Busy** and **Target App CPU Load** metrics. Using these metrics, you can assume whether your performance issues are with the CPU or the GPU.
4. After analyzing real-time metrics, you can proceed with more detailed analysis and try the following:

   - If you suspect that your game is GPU-bound, capture frame files and use the Graphics Frame Analyzer to perform deeper analysis into exactly what is happening within a specific frame – see where in the rendering pipeline your game is spending its time, down to the render target or draw call level of detail. Check Frame Analysis for Desktop API Graphics Applications.

- If you suspect that there are issues in balancing your workloads across the CPU and GPU, capture and analyze trace files to get a task-based overview of your game across both the CPU and GPU domains.
- If you know that your game is CPU-bound, you might try using some CPU-specific tools to help improve your performance on the CPU, such as Intel® VTune™ Profiler. In particular, with multiple cores now being the rule rather than the exception, parallelizing your code can have a big impact on overall CPU performance.

Capture Frame/Stream/Trace for Deeper Analysis

To perform deeper analysis with Frame Analyzer or Trace Analyzer, capture frame, stream, or trace files:

1. In Graphics Monitor, specify an application for analysis.
2. Choose **Frame**, **Stream**, or **Trace** mode from the launch modes drop-down menu on the lower right.
3. Click the **Start** button to launch the application. A window with the game running opens.
4. Choose one of the following methods to capture frame, stream, or trace:

   - With HUD:
     - To capture frame, press **CTRL + SHIFT + C**
     - To start/stop stream capture, press **L**
     - To capture trace, press **CTRL + SHIFT + T**

   NOTE Hot keys may interfere with game keyboard usage. In this case, you can:

   - Customize shortcuts
   - Set up a trigger in Graphics Monitor to automatically create frame/trace capture files when certain conditions occur, for example when FPS is 20

   - With System Analyzer:
     1. Return to the Graphics Monitor and click the  **Connect System Analyzer** button next to your application.
     2. Click  **Capture Frame** or  **Capture Trace** buttons on the top left of the System Analyzer window.

Heads-Up Display (HUD) UI Reference

System Analyzer UI Reference

Pause and Resume a Running Application

When running the application with the System Analyzer HUD or System Analyzer, use the **Ctrl+Shift+P** keyboard shortcut to pause and resume your graphics application.

While capturing a frame or trace file (by default **Ctrl+Shift+C** and **Ctrl+Shift+T** respectively) in the pause mode, you will see no visual indication that the capture was successfully completed. Resume your application with the **Ctrl+Shift+P** keyboard shortcut to see the indication.

(Windows* only) On some systems, resume attempts may take about five seconds. This issue occurs only if the analyzing application uses a keyboard hook procedure of the type `WH_KEYBOARD_LL`, and the registry key `LowLevelHooksTimeout` is set to a larger value than the default (300 ms). You may regularly use Pause/Resume and resuming may take longer than you are comfortable with. In this case, use a registry editor such as RegEdit to modify the value of this registry key back to the default value of 300 ms.

See Also

Customize keyboard Shortcuts

Export Metrics Data with System Analyzer

With the System Analyzer you can export metric values to a CSV file to post-process the analysis of your application.

To export metric values:

1. Launch the graphical application with the System Analyzer.
2. Make sure that the application window is active.
3. Press the **Export Metrics to CSV Files** button

   to start gathering metrics data. The button is highlighted while the CSV export is in progress.
4. To stop populating the CSV file, press the **Export Metrics to CSV Files** button

   again.

The System Analyzer automatically generates a unique file name for saving the metrics data. The filename consists of two fields: the name of the game executable and the export date and time, in the format <executable name>_<date_time>. For example: StateManager_2012_02_17__11_25_14.csv. All created CSV files are saved on the client system with the .csv extension, and by default reside in the user's home directory. On Windows*, the default home directory is %USERPROFILE%\Documents\GPA.

See Also

System Analyzer UI Reference

Identify Issues in Graphics Application Execution with Trace Analyzer

Use Graphics Trace Analyzer to:

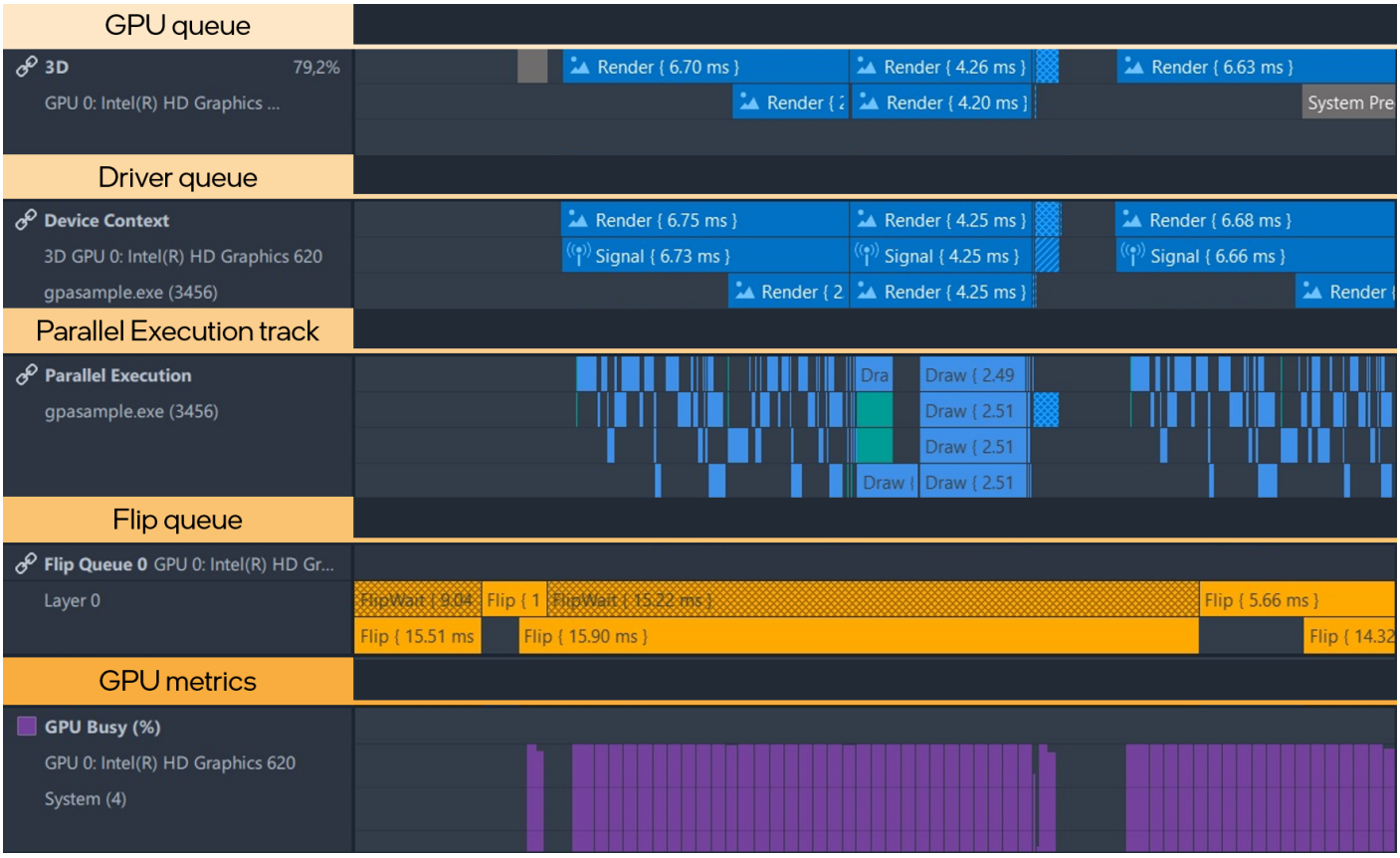
- Identify problem areas in graphics application execution: analyze calls to graphics APIs such as Microsoft DirectX*, Vulkan* or OpenGL* , review user-defined debug markers, threads, queued GPU commands.
- Perform a high-level analysis of synchronization and parallelism efficiency, relations between dependent threads and objects.
- Evaluate workload performance across the CPU and GPU.

Graphics Trace Analyzer captures a *trace*, which is a record of activity on both the CPU and the GPU during application execution.

Trace Data

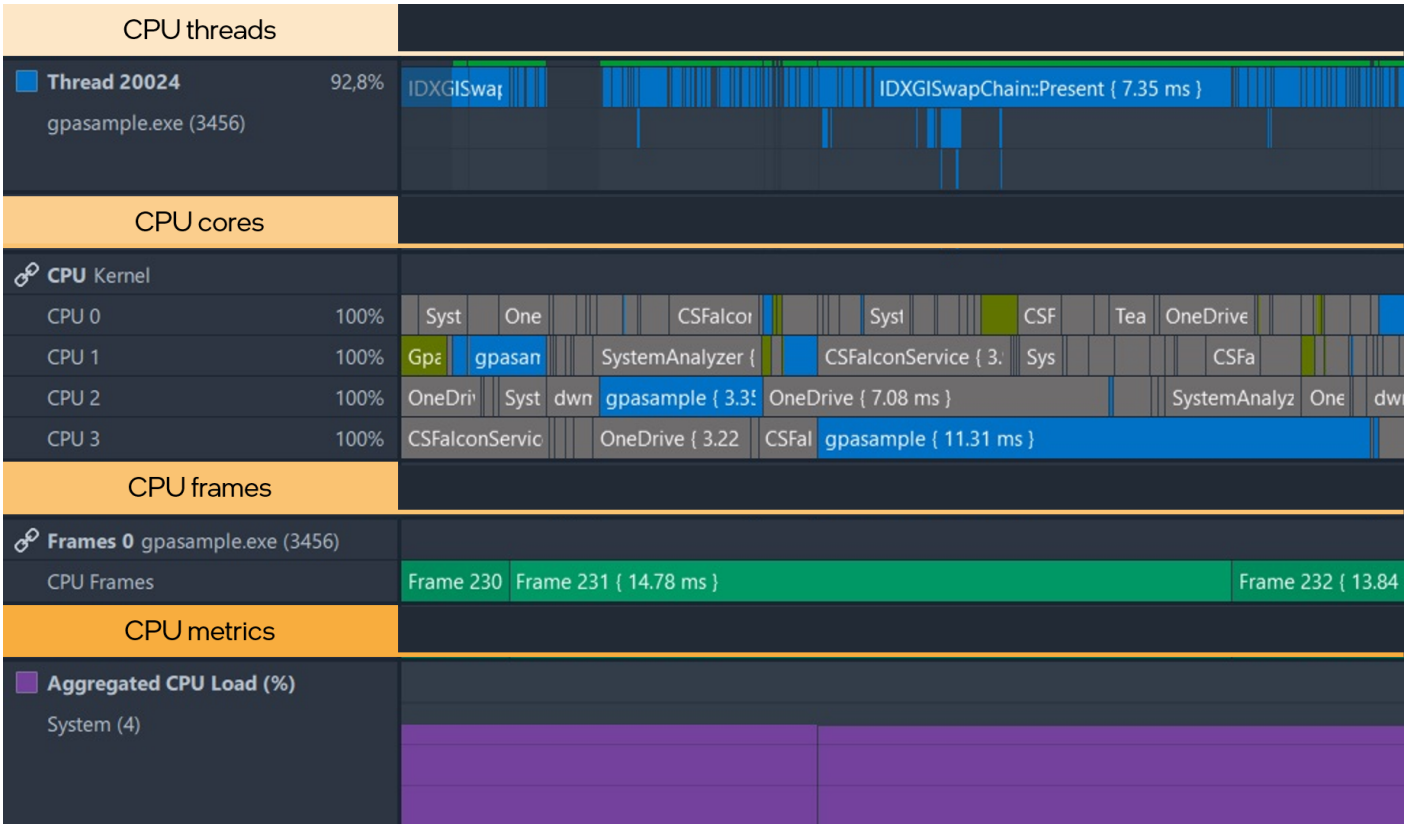
Trace represents data captured from graphics applications. During the rendering process, applications submit hundreds of graphics commands from different threads. The graphics driver interprets the commands for the GPU, puts them into command buffers, pushes the buffers into a CPU queue, and schedules the commands for execution on the GPU, forming a final frame on the screen.

GPU Activity Data



- **GPU queue** shows how the GPU executes commands forming a final frame on the screen. The GPU queue indicates whether the GPU is busy or idle.
- **Driver queue** shows how the graphics driver schedules graphics commands for execution on the GPU. The driver queue shows how many graphics commands are submitted, and how many of them are waiting for execution.
- **Parallel Execution** track shows how the driver parallelizes execution of submitted render commands (draw, clear, dispatch, resource barriers). The track is available for DirectX apps.
- **OpenCL Execution** track visualizes execution of OpenCL kernels on a GPU or a CPU.
- **Flip queue** reflects the relationship between the application present calls, present packages of GPU/CPU queues, composition work performed by the Desktop Window Manager (DWM), and Vertical Synchronization (VSync) events. Flip queue data allow you to roughly estimate the frame rendering latency, which includes present, flip, and VSync events.
- **GPU metrics** show GPU performance for the selected metrics set. Place the metrics track next to the GPU queue to see the correlation between application execution and GPU workload. For example, identify whether the GPU was busy during the processing of a certain package.

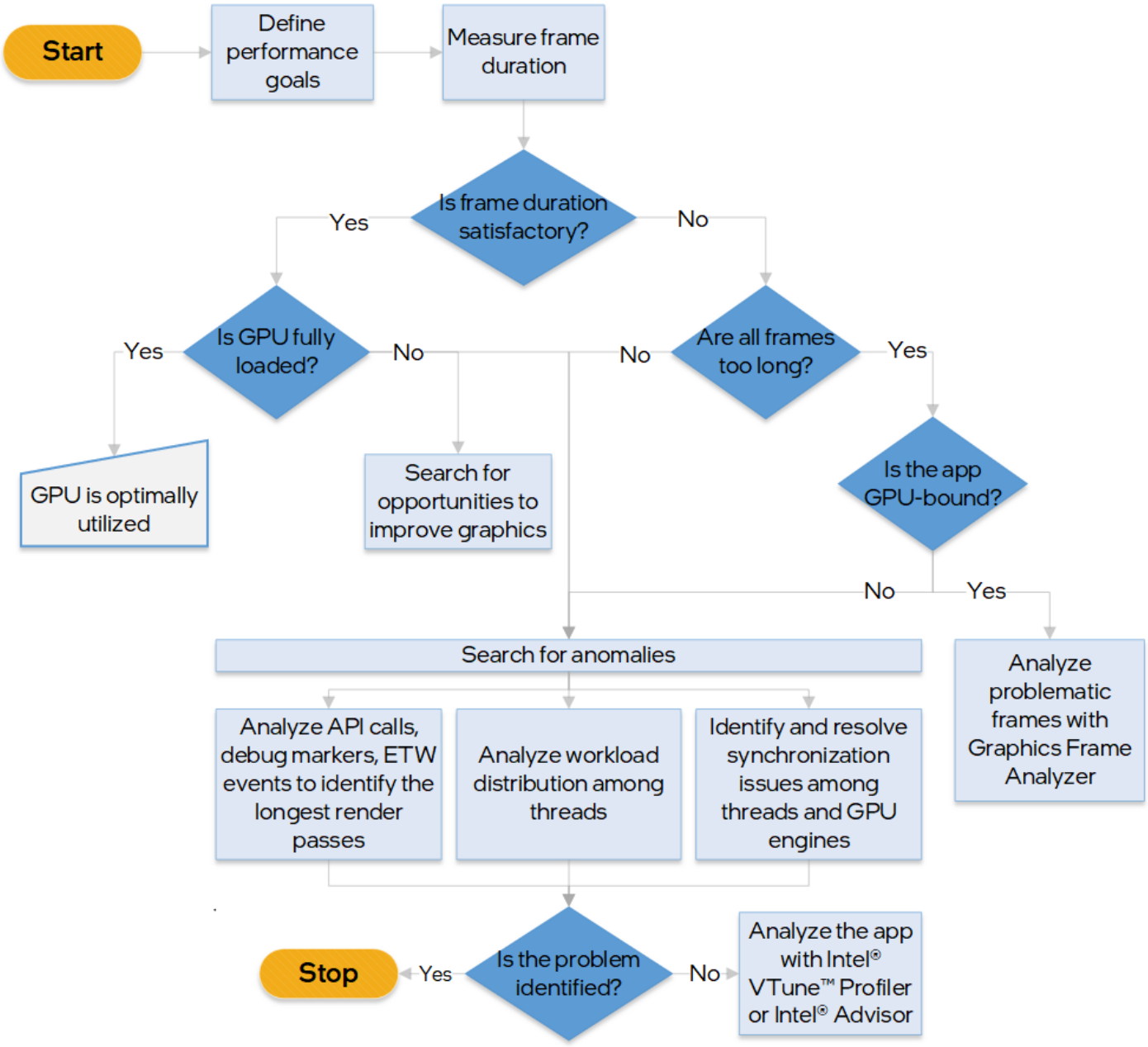
CPU Activity Data



- **CPU threads** track represents the activity of each thread: graphic API calls (draw calls, buffer locks, resource updates, presents), and user-defined debug annotation markers (Microsoft PIX, Instrumentation and Tracing Technology API (ITT API)).
- **CPU cores** track shows how threads from different processes including your profiled application are executed.
- **CPU frames** track shows the range containing graphics commands between two successive frames' buffer swap calls.
- **CPU metrics**show CPU performance for the selected metrics set. CPU and GPU metrics help you compare CPU and GPU utilization, and spot problematic areas.

Identify Performance Issues

Your workflow may look like this:



The proposed workflow focuses on game analysis. The steps may differ if you aim to optimize content creation applications.

Define Performance Goals

Set clear optimization goals based on the style, dynamics of the game, and the hardware your audience might use.

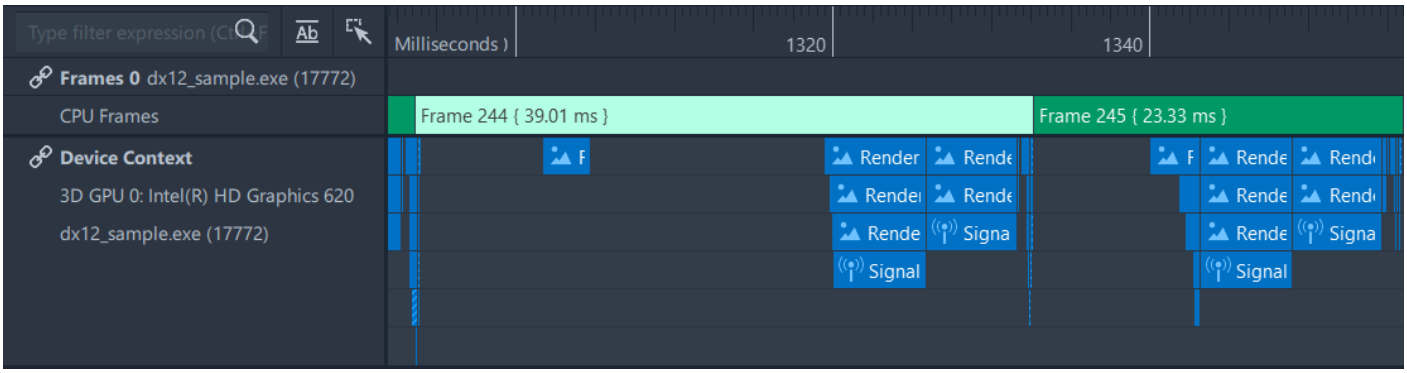
Different game types such as shooters or storytelling games imply different optimization goals, for example:

- **Increase frame rate.** The more dynamic the game is, the shorter the frames should be. At the same time, if the frame rate of the game is too high, the user may not see some of the rendered frames.
- **Optimize visual content representation.** Depending on the game type, you may be interested in identifying additional GPU resources for better detailing, for example, for elaborate landscapes or textures.

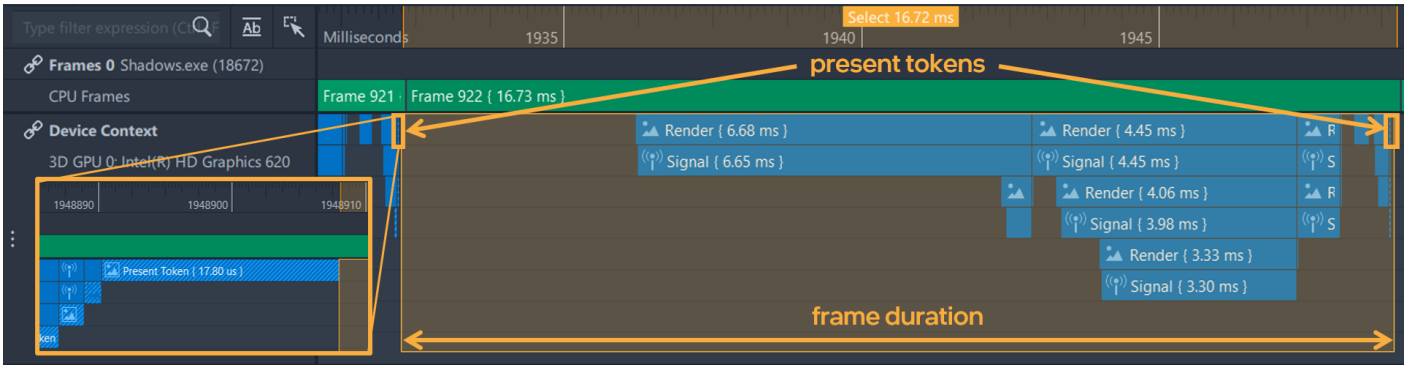
- **Reduce frame duration for cloud gaming.** Applications developed for cloud gaming have a restricted budget for each frame. In this case, there is a complex process behind the frame rate: receiving the user input, sending it to the server, processing, frame rendering, compression, sending data over the network, decompression, and displaying the frame on the screen.

Measure Frame Duration

You can use either frame rate or duration as a metric for analysis. For more precise results, start performance profiling with frame duration. The frame duration is measured in milliseconds and shown in curly braces for each frame in the **CPU Frames** track.



If the **CPU Frames** track is not available for your application, estimate the frame duration using present tokens. On the **Driver queue** track, select a range from the right border of a present packet to the right border of the next one. You can see the frame duration on the timeline.



Make sure the frame duration is consistent and meets your performance goals. For example:

- If frame duration is satisfactory, you can analyze whether the GPU is optimally utilized and inspect available GPU resources to incorporate more state-of-the-art graphics in the game.
- If all frames take longer than expected, you can identify whether your application is GPU-bound and inspect issues with Graphics Frame Analyzer.
- If frame duration varies greatly, you can spot anomalies with Graphics Trace Analyzer: analyze API calls, parallelization, synchronization, ETW events, and debug API markers in more detail.

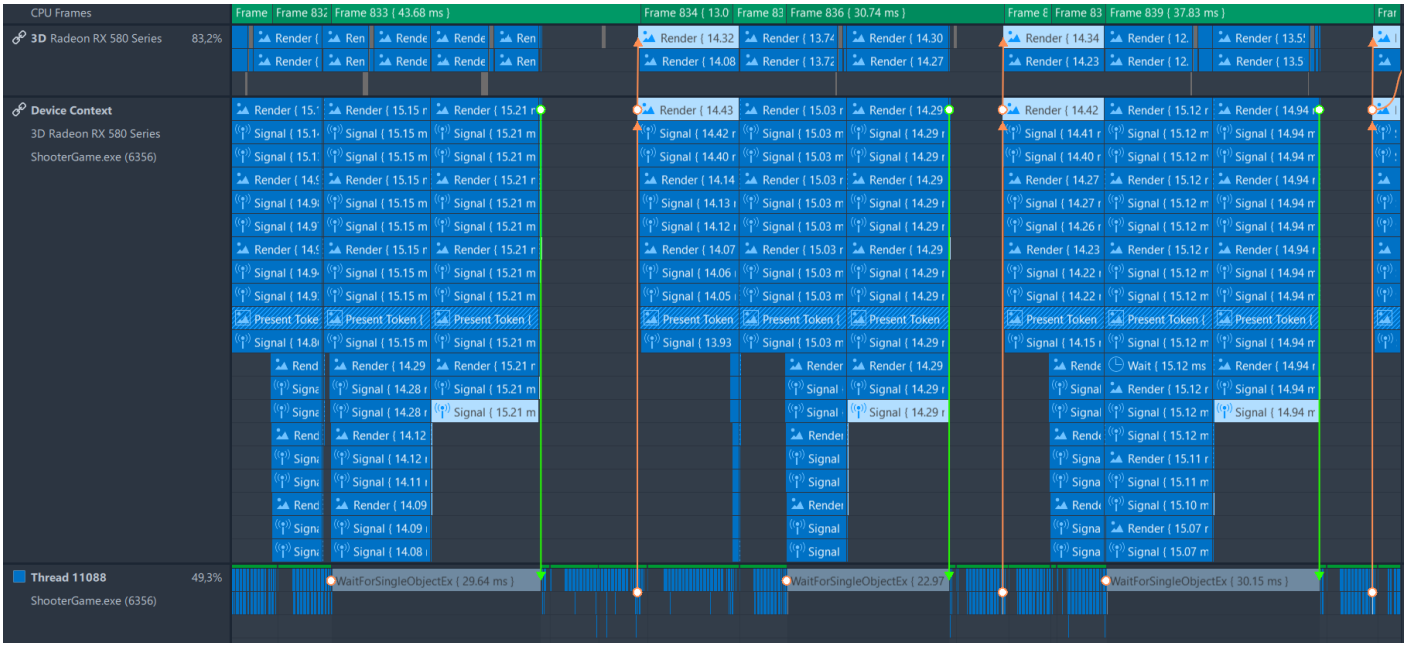
Analyze a Game with Satisfactory Frame Duration

If the frame duration is satisfactory and the GPU is loaded with instructions all the time, your game probably utilizes the GPU optimally.

If the frame duration is satisfactory and the GPU is not loaded all the time, visible gaps in the GPU queue may indicate the following:

- Underutilized GPU resources
- Improper graphics workload balancing
- Synchronization issues.

First, analyze how graphics workloads are distributed across CPU threads, and check GPU-CPU and GPU-GPU synchronization. For example, in the screenshot below, the gaps in the **GPU** and **Driver** queues indicate that the CPU is waiting for a signal from the GPU to resume processing and prepare the job for the GPU. In this application, the frames are rendered in triplets, and GPU-CPU synchronization increases the duration of the first frame in each triplet nearly fourfold. GPU-CPU synchronization is visualized with green arrows in Graphics Trace Analyzer:



In such cases, check whether there is a good reason for synchronization, whether you can change this and how your improvements will affect the gameplay.

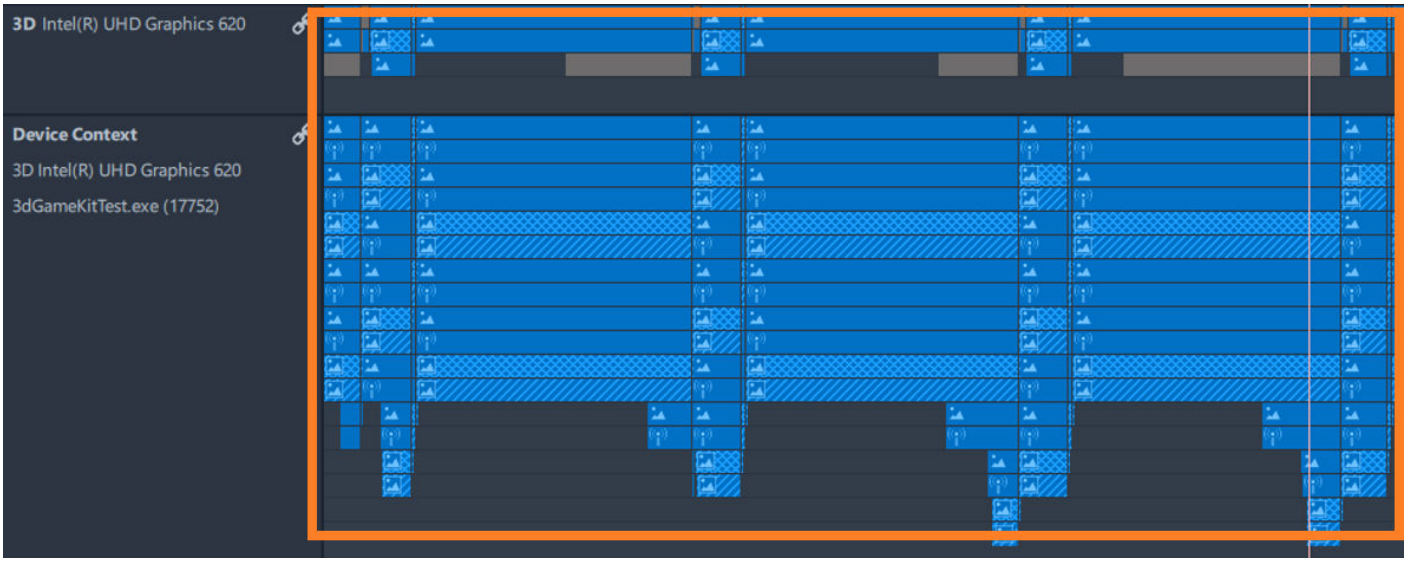
Tip: Refer to the video "What Do I Do If the GPU Shows Idle Time" to learn how to analyze a game where the GPU is underutilized.

If a GPU queue is not full and synchronization works properly, the GPU has resources that you can use to incorporate state-of-the-art graphics effects in a game without decreasing the frame rate. For example, you can add beautiful post-processing or textures.

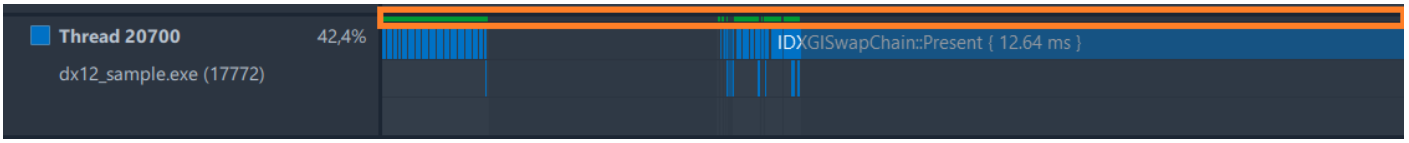
Analyze a Game with Unsatisfactory Frame Duration

Consistently long frames may indicate that your game is GPU-bound. You can identify a GPU-bound game by the following criteria:

- The GPU is busy the entire time and the GPU queue has no visible gaps.
- The Driver queue continuously accumulates command buffers waiting for execution on the GPU. In this case, the Driver queue size is long.
- Average DMA buffer execution time exceeds the desired limit based on the expected frame duration.



- CPU threads are inactive most of the time. The thread activity zone above the **CPU thread** track contains green or grey intervals indicating whether the thread was active or inactive during a particular period.



If your application is GPU-bound, capture a stream or a frame of a problematic area and analyze rendering performance in-depth using Graphics Frame Analyzer.

In other cases, when frame duration varies, search for anomalies with Graphics Trace Analyzer:

- Analyze API calls, debug markers and Event Tracing for Windows* (ETW) events to identify the longest render passes
- Analyze workload distribution and synchronization among threads.

If your case needs more in-depth analysis, use other CPU-side performance analysis tools offered by Intel:

- Use Intel® VTune™ Profiler to find your hotspots and identify issues related to CPU utilization
- Use Intel® Advisor for a deep focus on threading and vectorization.

Capture and Open a Trace

Before the analysis, stop all irrelevant applications that utilize GPU. You cannot identify performance issues accurately when several apps are competing for the GPU resource.

To capture trace data during the application run, do the following:

Configure Analysis Settings

1. Launch Graphics Monitor.
2. Click the **Options** button on the lower left of the Graphics Monitor configuration window.
3. In the **Trace** tab, configure tracing options as needed: set trace duration, choose data domains, enable data capturing on application startup.
4. Optionally, configure other analysis settings:
   - In the **Metrics** tab, select a set of default GPU/CPU metrics to monitor for your application
   - In the **Triggers** tab, you may specify a condition to start a trace capture automatically.
5. Exit the **Options** screen by clicking the



Back button on the upper left of the screen.

Run Analysis

1. In the Graphics Monitor Launcher screen, specify an application for analysis.
2. Choose **Trace** mode from the launch modes drop-down menu on the lower right.
3. Click the **Start** button to launch the application. A window with the game running will open.

Capture a Trace

Choose one of the following methods to capture the trace data to a file:

- HUD (recommended)

  In the window with the target app running, press **Ctrl+Shift+T** (default). When the capture is complete, a message is displayed just below the HUD with the filename or possible errors, if any.

  NOTE Hot keys may interfere with game keyboard usage. In this case, you can:

  - Customize shortcuts
  - Set up a trigger in Graphics Monitor to automatically create frame/trace capture files when certain conditions occur, for example when FPS is 20
  - Use

    Capture Trace button in System Analyzer

- System Analyzer

  1. Return to the Graphics Monitor configuration window and click the

     Connect System Analyzer button next to your application. The button becomes available after you start the application.
  2. Click the

     Capture Trace button to capture a trace. When the capture is complete, the System Analyzer displays a message with the filename or possible errors, if any.

- System View Trace

  Use system view capture from System Analyzer, if:

  - The methods above do not work for you
  - You are interested in system data rather than detailed application execution data such as API calls or debug regions. In this mode, only system data, for example, GPU utilization, is available for analysis.

  1. Click the **Options**button in the Graphics Monitor.
  2. In the **Trace** tab, set **Trace System View in System Analyzer** toggle to ON.
  3. Open System Analyzer.
  4. Start your application:

     - From Graphics Monitor, hover over the application and click the **Run**button.
     - Run the application from a file manager.
  5. Return to the System Analyzer and click the

     Capture Trace button.

View Collected Data

To view the collected data:

1. From the Graphics Monitor context menu, launch Graphics Trace Analyzer.
2. In the Open Trace Capture window, select and open the captured trace file.

See Also

Set Up Tracing Options

Graphics Trace Analyzer UI Reference

Analyze GPU-bound Applications with Graphics Frame Analyzer

Use Graphics Frame Analyzer to locate GPU bottlenecks, root-cause rendering issues, and increase performance.

Graphics Frame Analyzer is the tool to use once you have determined your application to be GPU-bound and have ruled out other issues, such as CPU-side code inefficiency and synchronization, using System Analyzer and Graphics Trace Analyzer.

Frames and Streams

To profile your game using Graphics Frame Analyzer, you first need to capture a frame or a stream.

- A **frame** is a capture of all GPU activity and resources involved in rendering one frame.

  Use frames when you are only interested in the rendering performance of one specific frame.
- A **stream** is a consecutive set of **frames**, each of which contains all information that a typical frame contains, as well as pipeline and hardware metrics.

  Use streams when you would like to analyze the behavior of a frame relative to other frames, select the slowest frame out of a scene, or troubleshoot frame time fluctuations.

Capture and Open a Frame

To capture and open a frame:

1. Run Graphics Monitor on your system and specify an application for analysis.
2. Select the **Frame** startup mode for DirectX* 9, 10, or 11.

   Select the **Frame [DX12]** mode for DirectX 12 applications.
3. Click **Start** to run your application.

   The application launches with the HUD Overlay enabled.
4. Press **Ctrl+Shift+C** to capture a frame.

   You can capture as many frames as needed without restarting the application.

   NOTE
   If you find it hard to capture a frame at the right moment, set up a trigger to capture frames automatically when a custom condition is satisfied.

   If HUD hot keys interfere with game keyboard usage, you can:

   - Customize shortcuts.
   - Use

     Capture Frame button in the System Analyzer.

5. Once you are done, open the frame using either of these options:

- Use the **Open File** ribbon of the Graphics Monitor window.

NOTE New [DX12] frames are saved in folders with a `.gpa_frame` extension, not as separate files.

- Launch Graphics Frame Analyzer and select a frame to open.

Capture and Open a Stream

NOTE
To change tool behavior related to stream capture, configure Stream Capture Options.

To capture and open a stream:

1. Run Graphics Monitor and specify and application for analysis.
2. Select the **Stream** startup mode.
3. Click **Start** to run your application.

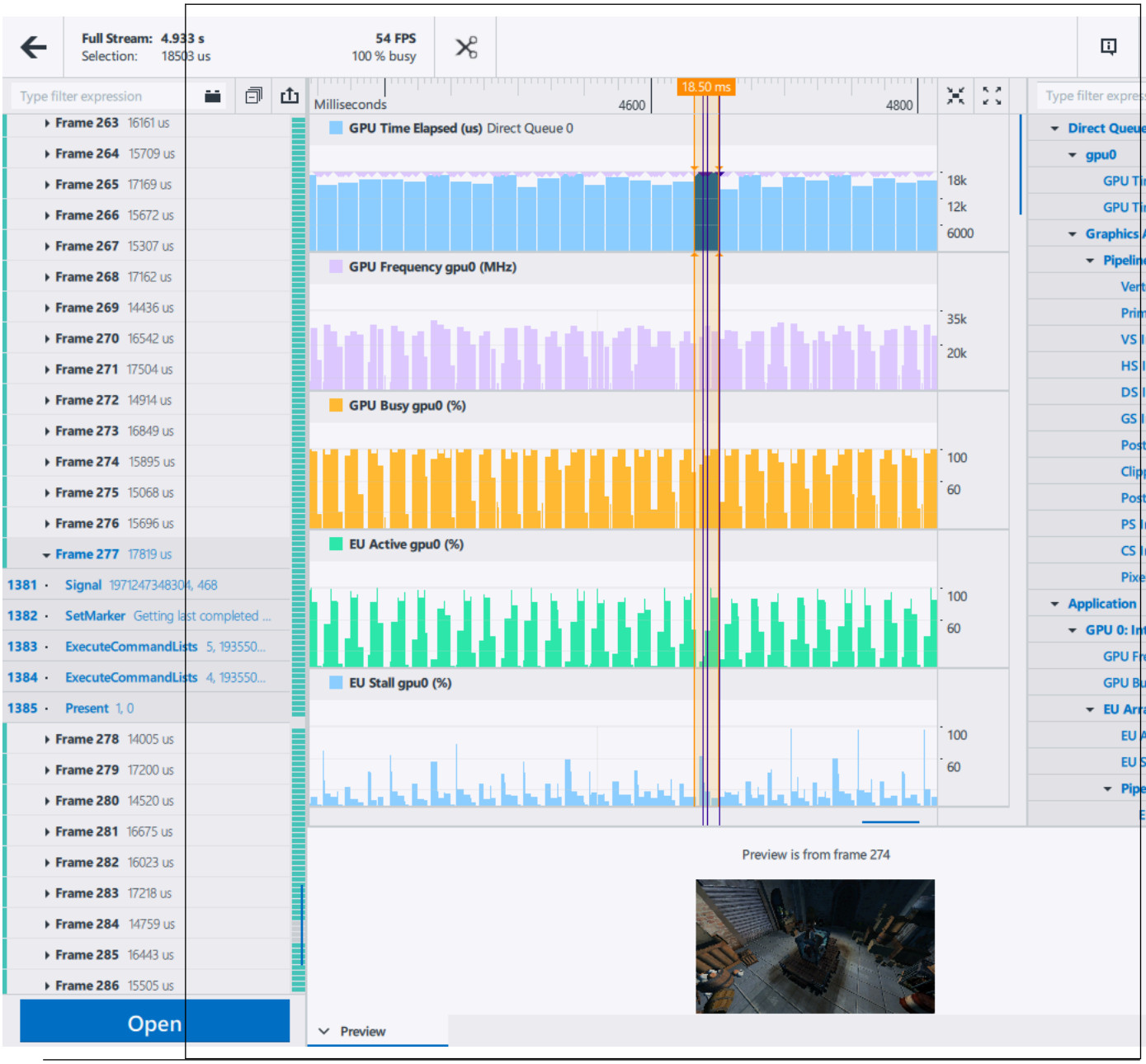
   The application launches with the HUD Overlay enabled.
4. By default, stream capture starts immediately when the application is started.

   You can enable the **Defer Stream Capture** option in the Stream Capture Options to start and stop stream capture manually.

   If this option is enabled, locate a scene of interest and press **L** to start and stop stream capture.
5. Once you are done, open the stream using either of these options:
   - Use the **Open File** ribbon of the Graphics Monitor window.
   - Launch Graphics Frame Analyzer and select a stream to open.

Perform High-Level Stream Analysis

Once you open a stream, the Graphics Frame Analyzer Multi-Frame View is shown.



There are several sources of information available in this view:

- **Stream Viewer** pane:
  - **Available Information:**

    This pane shows a bar chart representation of frames that were captured into this stream, as well as multiple hardware and pipeline metrics.

    Each bar on the top track represents a frame, with its height representing the **GPU Elapsed Time**, the time it took to render this frame.

    The combination and granularity of metrics in this view is influenced by the **Metric Set** and the **GPU Sampling Interval** options of the Graphics Monitor options Metrics tab. A smaller sampling interval allows for finer metric granularity, but increases overhead.

The pane also features timestamps for end points of execution of each command list within a frame. The timestamps are shown in the form of vertical lines that span across the **Stream Viewer** pane.

- **Potential Insights:**
  - You can use the frame time bar chart to select the longest frame and open it for deeper analysis. You can also use screenshots to keep track of notable scene changes.
  - If there are sporadic performance issues across several frames, or if you suspect that a previous frame influences the next one in a negative way, you can select multiple frames and open them simultaneously, concatenating all information. This can also be useful if there are calculations done every other frame.
  - If you decided to collect metrics with fine granularity, you can zoom into a frame and correlate low-level activity on the GPU with debug regions and command lists execution within a frame. If you decide that the metric values within a debug region look abnormal, you can open the frame for analysis and focus on the problematic region straight away.

- **Stream Content** pane:
  - **Available Information:**

    This pane shows a list of all frames collected into this stream, as well as separation by command queues. For example, if your application features separate Render and Compute queues, both queues are shown as tabs on the **Stream Content** pane.

    You can expand each frame to see debug regions and command lists in each frame. Click on a debug region or command list to highlight it on the bar chart.
  - **Potential Insights:**
    - You can use this pane to navigate around the stream and track execution of debug regions and command lists within them.

For more information on this view, see the Multiframe View UI Reference topic.

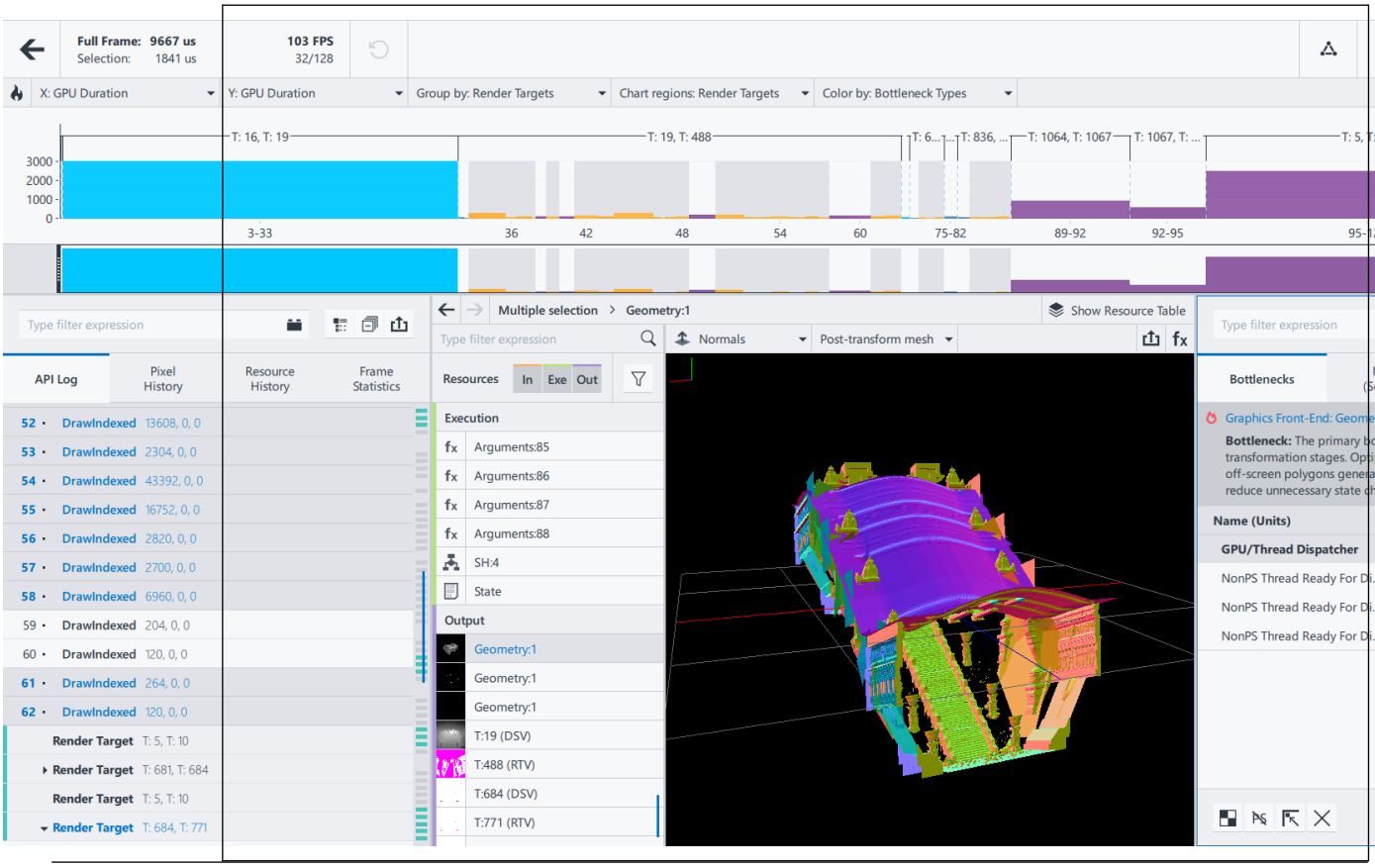
NOTE
You can trim your stream by pressing the



Trim Stream button, for example, if the stream only contains a few frames you are interested in. Trimming a stream saves disk space and processing time.

Perform High-Level Frame Analysis

Once you have a frame to analyze, either open this frame directly or select it from a stream.



Once you open a frame, note the following elements:

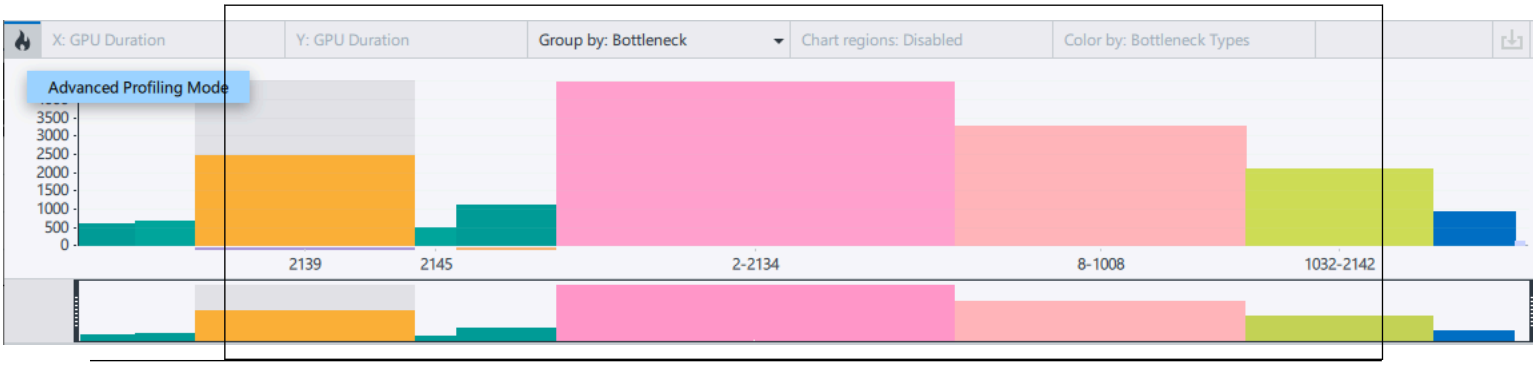
- The **Bar Chart** shows all draw calls within a frame. By default, the **Y** axis is the **GPU Duration**, so the higher the bar, the longer the draw call took to execute.
- The **Main Toolbar** contains data about the duration of the entire frame and the current selection. These values are recalculated dynamically when you make any changes that save or add frame time, such as applying experiments or modifying shader code. Pay attention to this block when applying experiments or trying shader code changes in real-time, as a potential time gain will be shown there.
- The **Resources** pane shows all resources related to the current selection.
- The **Metrics** pane shows the most likely bottleneck for the currently selected draw call or group, and offers advice on resolving the issue. Bottlenecks are marked with a colored hot spot icon on the left:
  -  indicates a primary bottleneck.
  -  indicates a secondary bottleneck.

Next Steps

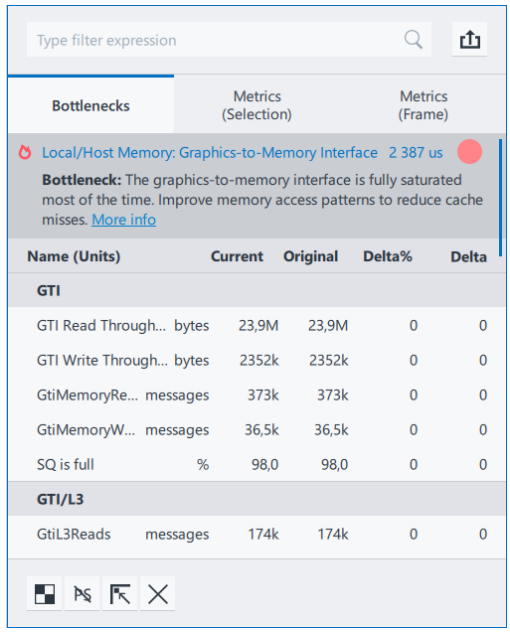
Depending on your profiling needs, there are multiple ways to approach a problematic frame:

- **Start optimizing based on hardware bottlenecks:**

  To do this, enable the Advanced Profiling Mode. This mode groups draw calls by hardware bottlenecks, which enables you to focus on issues that are common across multiple draw calls.



Click on a bottleneck group to see the bottleneck in the **Metrics** pane and get advice on resolving the issue:



You can then work over the groups of draw calls and resolve the root causes of the bottlenecks. This may produce a large positive performance impact, since you can apply optimizations that commonly affect a large group of draw calls each time.

- **Start optimizing based on debug regions:**

  If your application features debug regions, you can enable their display in Frame Analyzer and work by optimizing known logical portions of frame rendering.

  You can also group draw calls by debug regions by using the **Group By** drop-down menu, thus combining all draw calls from this region into one bar on the bar chart. This enables you to focus on entire logical portions of your frame, such as light rendering or shadows.
- **Grouping-independent approaches:**
  - You can use Render State Experiments to experiment with your graphics pipeline without code modifications. For example, you can replace a texture with a simple 2x2 texture to check if an overly complex texture has a large performance impact.
  - If you suspect a problem with a specific domain, such as Shader Execution, you can focus on optimizing this domain. For example, for shaders, you can edit shader code in real time without recompiling the application, and instantly see the performance and visual impacts in Frame Analyzer.

Run Render State Experiments with Frame Analyzer

Use Advanced Profiling Mode

Optimize Specific Domains

Run Render State Experiments with Frame Analyzer

Modify render states of Graphics API to isolate potential performance bottlenecks in your application by running experiments.

While in single frame view of Graphics Frame Analyzer, select a draw call from the **Bar Chart** and then look at the bottom of the **Metrics panel** where you can view different render state experiments you can apply to your selected call.

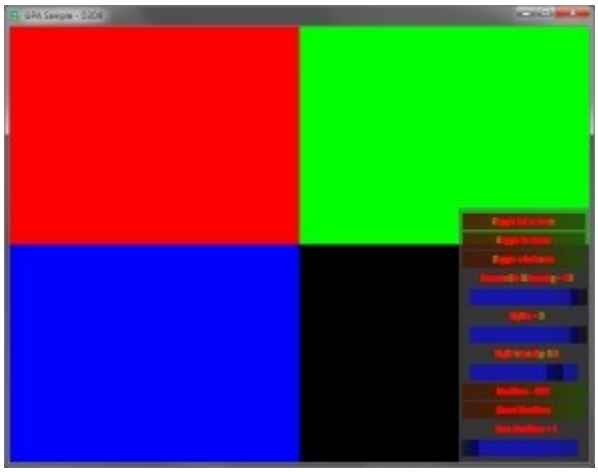


2x2 Textures

Apply the two by two texture override experiment to replace all textures for a selected event by a simple two by two texture, using a simple halftone or colorized bitmap. If applying this experiment to a draw call improves the framerate significantly and decreases GPU duration, it means that your game may be texture-bound and needs optimizing textures.

There are several ways to optimize your textures:

- Reduce the size of your textures
- Reduce the number of textures used in the scene
- Reduce the texture filtering setting, specifically minimizing anisotropic filtering
- Reduce the number of texture fetches in your shaders

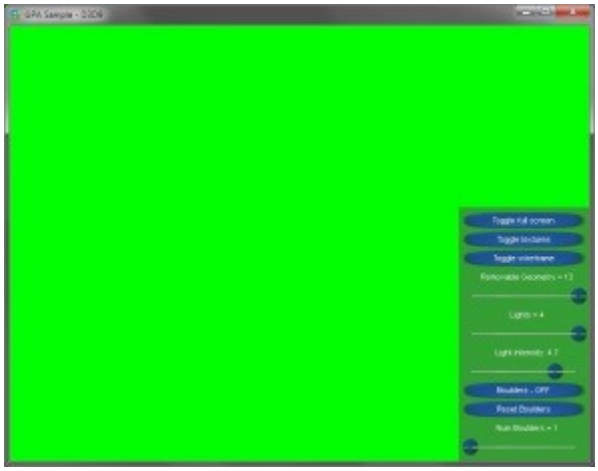
Normal Picture	Force 2x2 Texture
	



The Simple Pixel Shader

Replace every pixel/fragment shader with the pixel/fragment shader that writes a constant color to all render targets.

If applying this experiment significantly improves the frame rate, it means that the performance slowdown is caused by pixel shader computation and/or texture performance. Inspect your shaders further to identify and rework the most expensive calls to improve scene rendering performance.

Normal Picture	Force Simple Pixel Shader
	



One by One Scissor Rect

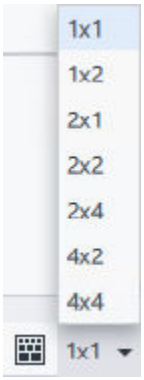
One by One Scissor Rect Bypasses pixel processing from the rendering pipeline. If the frame rate does not increase with this experiment, complex geometry or vertex shader is a bottleneck. Using the resource viewer, you can analyze the geometry of your frame or experiment with shader code to locate and solve potential issues. If the GPU duration significantly increases, then rasterization is likely to be the bottleneck. It's important to note that the effect of this experiment is driver and graphics hardware vendor-specific and depends on whether scissoring occurs prior to or after the pixel shader stage.



Disable Events Experiment

Prevents the selected events from rendering. This experiment can help you test scene efficiency.

Shading Rate Experiment



Adjust the shading rate for a specific resource using one of the available presets. With this experiment, you can identify objects that do not contribute meaningfully to the visual quality of the scene, such as distant objects.

By lowering the shading rate for such objects, you can trade off insignificant details for performance improvements. One example when an adjustment of the shading rate can help is a texture sampler bottleneck. Graphics Frame Analyzer allows you to adjust the shading rate dynamically and view the results and potential performance gains in real time to determine whether some detail can be sacrificed without a noticeable loss in quality.

NOTE This experiment is only supported on systems that satisfy certain hardware and software requirements. For more information, see the corresponding entry in the Release Note

Use Advanced Profiling Mode

Use Advanced Profiling mode of Graphics Frame Analyzer to group events within a frame by bottleneck or state.

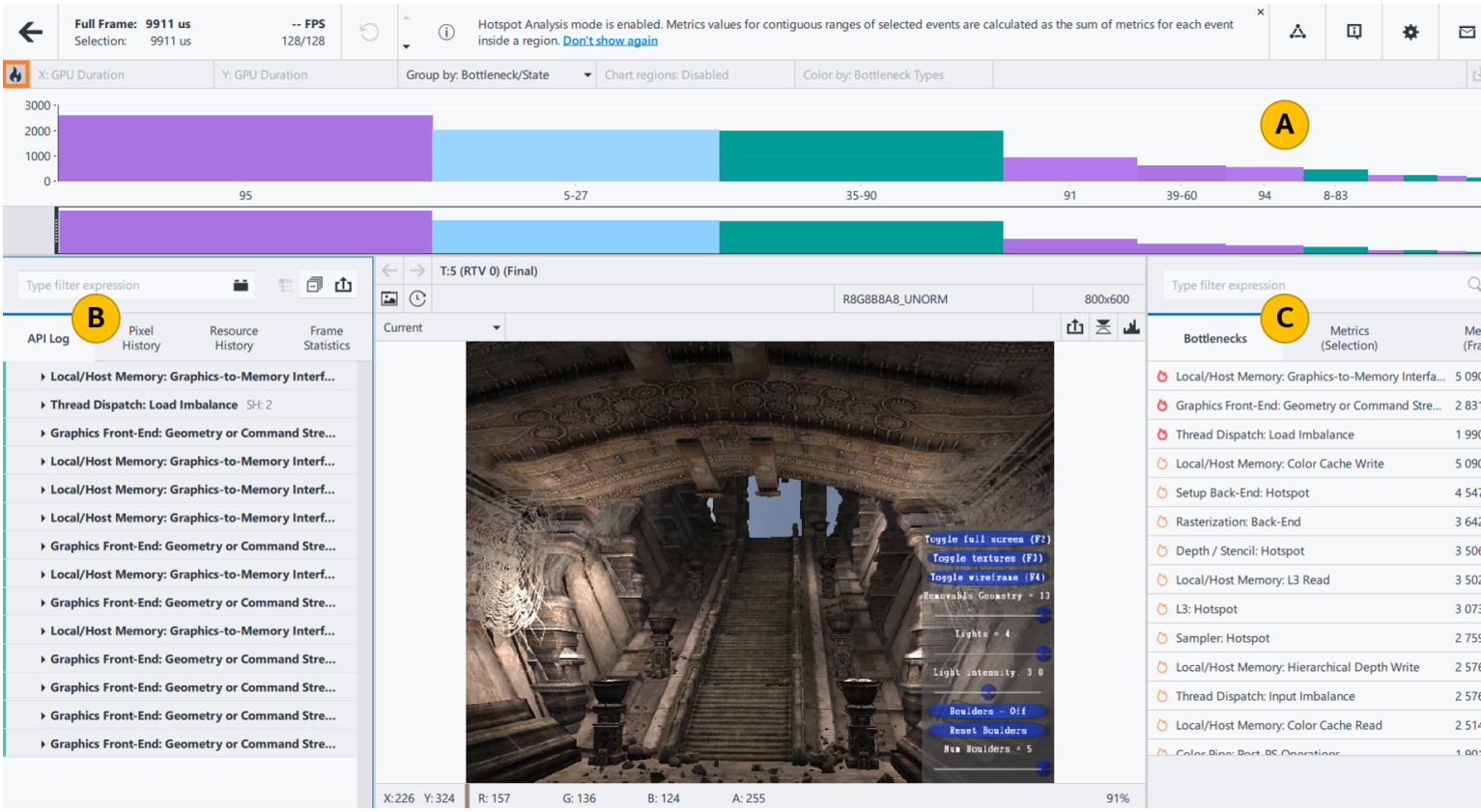
By default, the Visualization pane shows the bar chart in a time-based view. Advanced Profiling mode groups the events within a frame by bottleneck and/or state, where states are shader sets in DirectX* 11, pipeline state objects (PSO) in DirectX 12, and pipelines in Vulkan. By enabling Advanced Profiling mode, you can identify exactly which bottleneck or state is affecting the GPU most. Then you can explore bottlenecks in more detail: each grouping expands to reveal which events contribute to that specific bottleneck/state.

To indicate bottlenecks with Advanced Profiling mode:

1. Click the



Advanced Profiling Mode button on the top left.



The API calls are grouped by bottlenecks and/or states in the **Visualization** (

A

) and **API Log** (

B

) panes. The first group of draw calls is the most impactful and time-consuming. In the Visualization pane, the X-Y axis is set to GPU Duration by GPU Duration, which allows the tallest and widest bars to represent the events that take up the most time on the GPU.

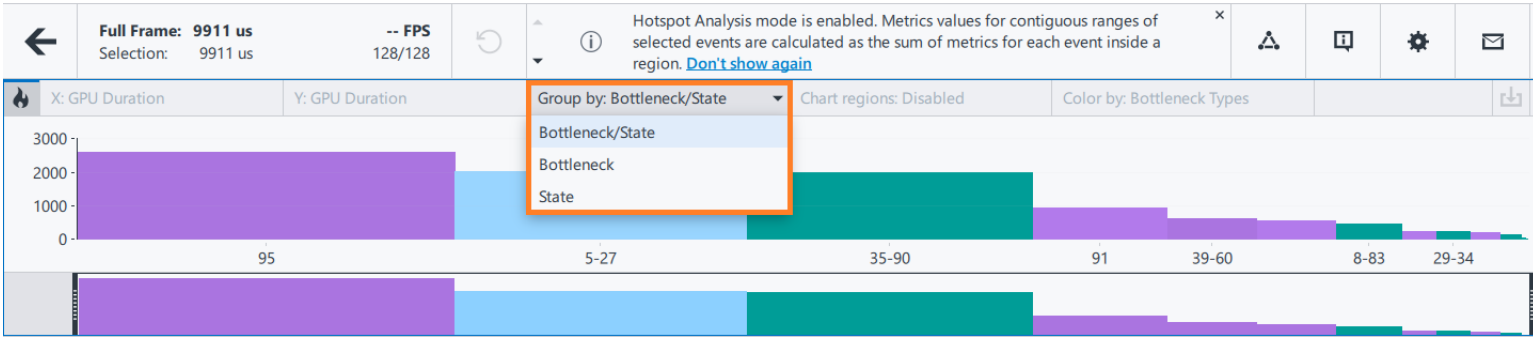
The **Bottlenecks** tab of the Metrics pane (

C

) presents color-coded bottlenecks. Red hot spot icons next to the bottleneck indicate main bottlenecks, and orange indicate secondary bottlenecks. Secondary bottlenecks are potential and may disappear once you resolve primary bottlenecks. On the right of the bottleneck name, you can see:

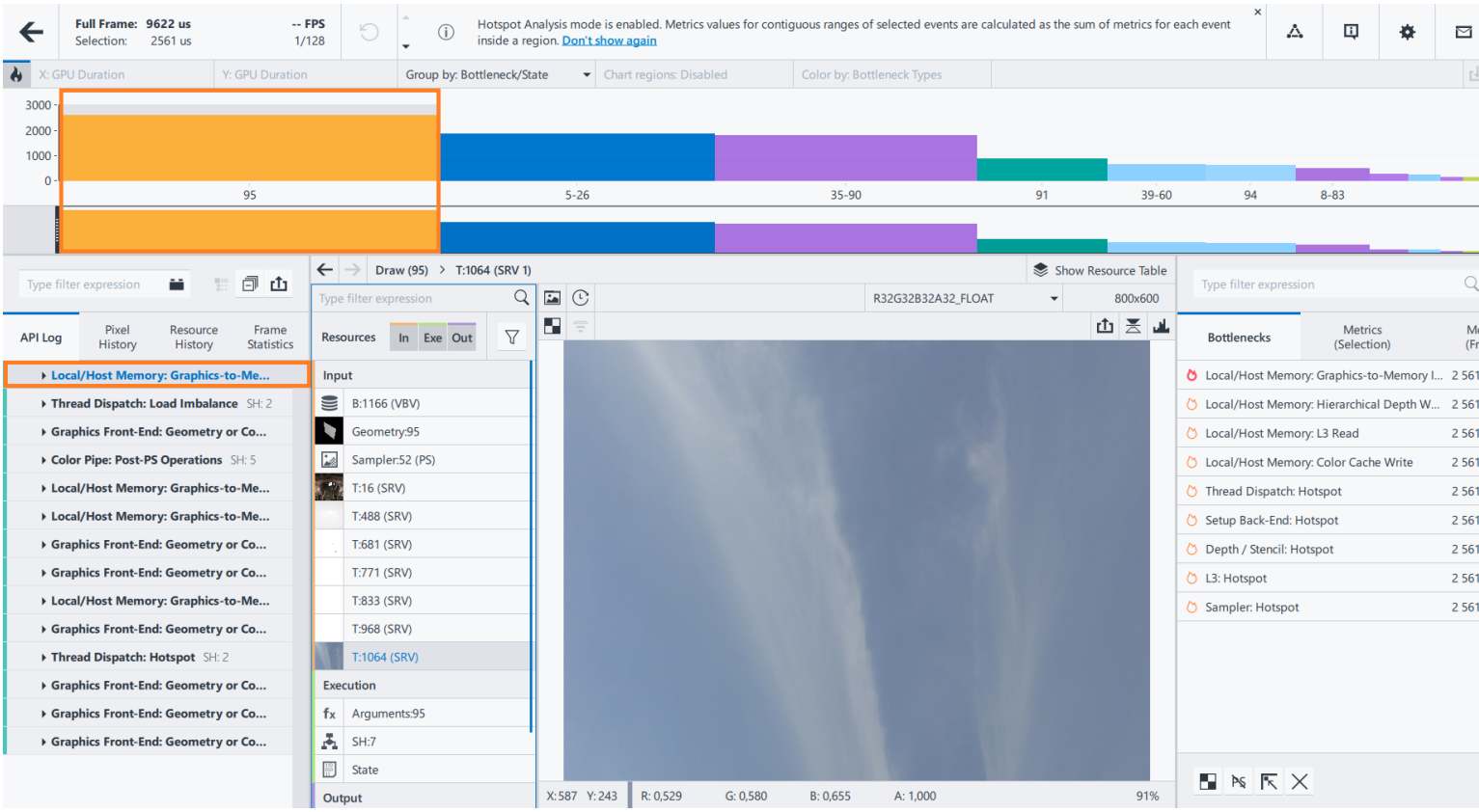
- Total duration of all API calls contributing to this bottleneck, in microseconds.
- Pie chart showing the percentage of time spent on API calls that have this bottleneck in relation to the time spent on all API calls with primary (red) or secondary (orange) bottlenecks.

2. (Optional) You can change the categorization used to determine hot spots by selecting any of the attributes in the **Group By** drop-down menu:



- **Bottleneck/State** - groups events that share the same state and bottleneck.
- **Bottleneck** - groups events that share the same bottleneck; shared states are used as sub-categories.
- **State** - groups events that share the same states; the bottleneck is used as a sub-category.

3. Start resolving issues with the most impactful bottleneck. Click the first group of draw calls either in the **Visualization** or **API Log** pane.



The resources associated with this group of calls appear in the **Resources** pane. The **Bottlenecks** tab of the Metrics pane presents color-coded bottlenecks for the selected group of calls.

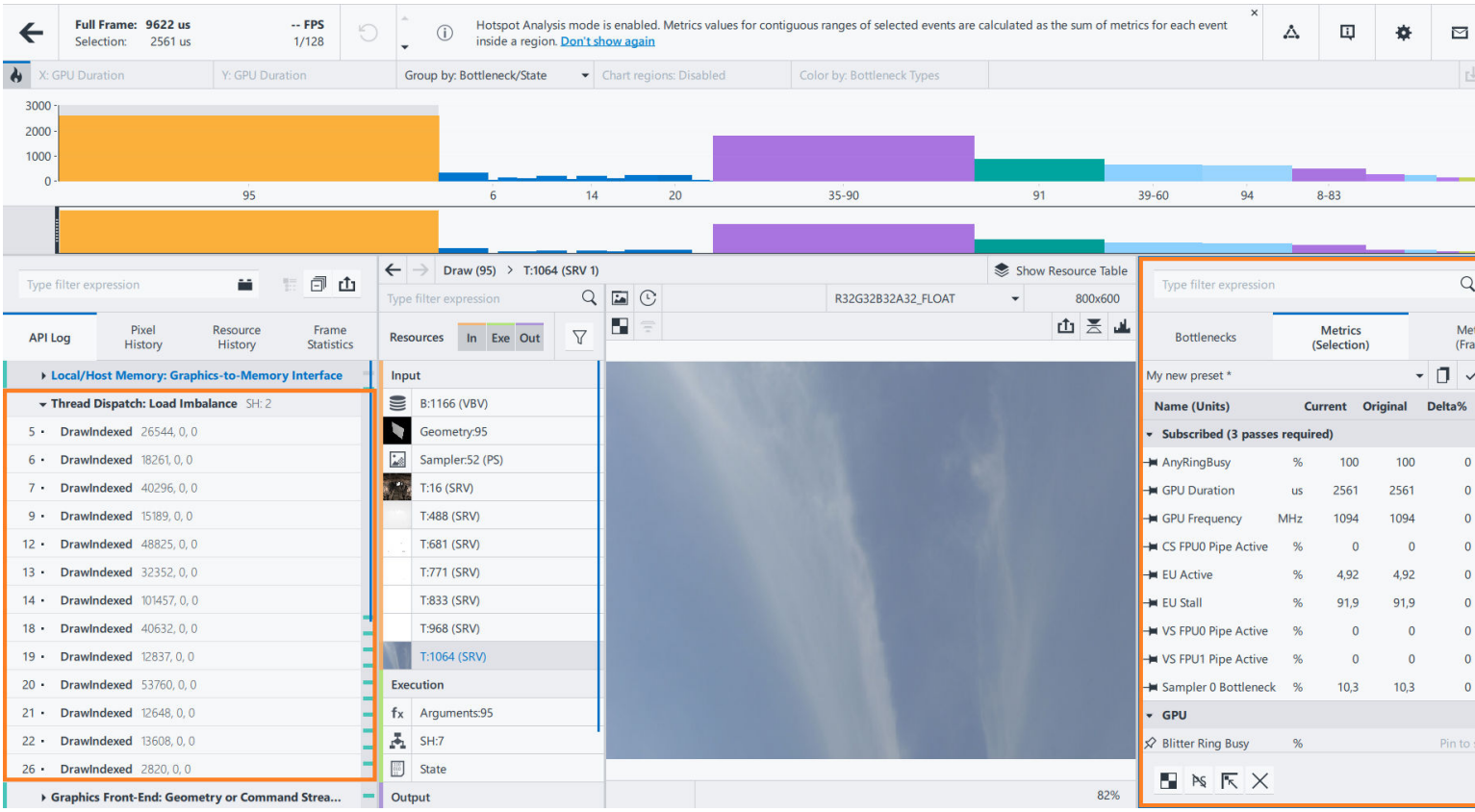
4. In the **Bottlenecks** tab, click the red bottleneck, for example, **Local/Host Memory: Graphics-to-Memory Interface.**

   The bottleneck cause, hints, and related metrics appear.

5. Work over the groups of draw calls and resolve the root causes of the bottlenecks. This may produce a large positive performance impact, since you can apply optimizations that commonly affect a large group of draw calls each time.

   In the API Log pane, each grouping can be expanded to reveal all of the events contributing to the hot spot.

   In the Metrics pane, you can explore metrics in detail. Metric values are an average/weighted value for the selected region.



For the Advanced Profiling mode back end flowchart and detailed bottleneck descriptions, see the Intel® Graphics Performance Analyzers Cookbook.

Visualization Pane

API Log Pane

Resources Pane

Graphics Frame Analyzer Plugins

Use Graphics Frame Analyzer Plugin Interface to mark specific events and resources.

Intel® GPA provides default plugins and enables you to use custom plugins.

> **NOTE** Graphics Frame Analyzer Plugin Interface is compatible with Python* 3 only.

Default Graphics Frame Analyzer plugins:

- **Event by Metric Value:** searches for API calls within the specified metric range.
- **Format**: searches for a resource of the specified format. The plugin is available only in the Resource Table.
- **Frame Time:** searches for a frame of the specified duration. The plugin is available only in the Multiframe View.
- **Subsequent Barriers:** searches for subsequent ResourceBarrier calls. The plugin is available only for the DirectX* 12 based applications.
- **Transitions to Common State:** finds barriers that transform into common state. The plugin is available only for the DirectX* 12 based applications.

- **Zero Pixel Write:** searches for draw calls that write 0 pixels into Render Target. The plugin is available only for the DirectX* based applications.

Apply a Graphics Frame Analyzer plugin:

1. Open a frame for analysis in the Graphics Frame Analyzer Profiling View Window.
2. Enter "=" into the **Type Filter Expression** field in the **API Log** or the **Resource Table**.
3. Select a plugin from the drop-down list of available plugins.
4. Define arguments of the selected plugin.
5. Press **Enter**.

Create a Custom Plugin for Graphics Frame Analyzer

Integrate your own plugin into the Graphics Frame Analyzer Plugin Interface.

The following workflow is used for creating a plugin for Graphics Frame Analyzer:

1. Write a __init__.py script containing desc() and run() methods.
2. Write additional scripts, if needed.
3. Copy the plugin folder to %USERPROFILE%\Documents\GPA\python_plugins.

> **NOTE**
> A plugin requires a separate folder, as each plugin is considered a separate Python module.
>
> Folder name must be PEP-8 compatible and is used for plugin invocation from the Type Filter Expression field in the Graphics Frame Analyzer.

desc() method

The method is used to obtain the following meta information about a plugin:

- **name***(optional)*: plugin name shown in the Graphics Frame Analyzer Plugin Interface. If the name is not defined, a directory name can be used.
- **description***(optional)*: plugin description.
- **apis***(optional)*: supported APIs.

  The following values are available:

  - DirectX
  - DirectX 11
  - DirectX 12
  - OpenGL
  - Vulkan

> **NOTE** If no value is defined, all the available values apply.

- **plugin_api_version***(required)*: plugin API version. The available values are **1.0** and **1.1**.
- **applicabilities***(optional)*: Graphics Frame Analyzer pane(s) where the plugin is used.

  The following values are available:

  - Apilog
  - Resources

> **NOTE** If no value is defined, all the available values apply.

An example of the desc() method:

```
def desc():

return {

 "name" : "My sample plugin",

 "description" : "Sample plugin is written for educational purpose",

 "apis" : ["DirectX"],

 "plugin_api_version" : "1.1",

 "applicabilities" : ["Apilog"]
}
```

run() method

The method returns items which are found by the defined criteria. The method uses the **node_to_result()** method or the **resource_to_result()** method of the utils.common module distributed with the Graphics Frame Analyzer.

An example of the run() method:

```
def run():

 api_log = plugin_api.get_api_log_accessor()

 calls = [x for x in api_log.get_full() if x.get_description()['name'] ==
'ClearRenderTargetView']

 return [common.node_to_result(x, 'info', 'This is a clear call!') for x in calls]
```

NOTE If a call is not defined, the detected call is marked, but no information message appears.

You can use the following types of marks:

- informational
- warning
- erroneous

The run() method supports arguments that are passed from the Graphics Frame Analyzer. Each argument can be decorated with function decorators in compliance with PEP-3107. Argument descriptions are shown in the Graphics Frame Analyzer.

```
def run(name: "Draw call name" = 'ClearRenderTargetView'):

 api_log = plugin_api.get_api_log_accessor()

 calls = [x for x in api_log.get_full() if x.get_description()['name'] == name]

 return [common.node_to_result(x, 'info') for x in calls]
```

plugin_api

To access different methods of a custom plugin API, import plugin_api into the plugin that you created, and then get accessors for the required data.

plugin_api is located at <Intel® GPA installation directory>/python_plugins/plugin_api.

The available accessors are the following:

- **get_api_log_accessor()** - processes the API Log: gets calls, arguments, bindings, etc.
- **get_resources_accessor()** - processes resources: gets resource descriptions, views, shaders, etc.
- **get_metrics_accessor()** - processes metrics: gets metric values, metric descriptions, etc.

An example of an accessor:

```
import plugin_api

def run():

 metrics_accessor = plugin_api.get_metrics_accessor()

 descs = metrics_accessor.get_metrics_descriptions()

 # some code
```

Optimize Specific Domains

With the Graphics Frame Analyzer, you can optimize specific domains:

- Optimize Shaders
- Analyze Textures
- Minimize Overdraw
- Optimize Graphic API States
- Identify Graphics Elements with Complex Geometry
- Visualize Color Distribution

Optimize Shaders

Review shader source code and assembly code with Graphics Frame Analyzer to identify inefficiencies in shader execution. Change the shader code inside Graphics Frame Analyzer and check the performance effect without recompiling your code.

1. Check for shader bottlenecks
2. Profile shader source code
3. Analyze shader resources
4. Profile shader assembly code
5. Experiment with shader code
6. Evaluate performance
7. Evaluate final picture

Check for Shader Bottlenecks

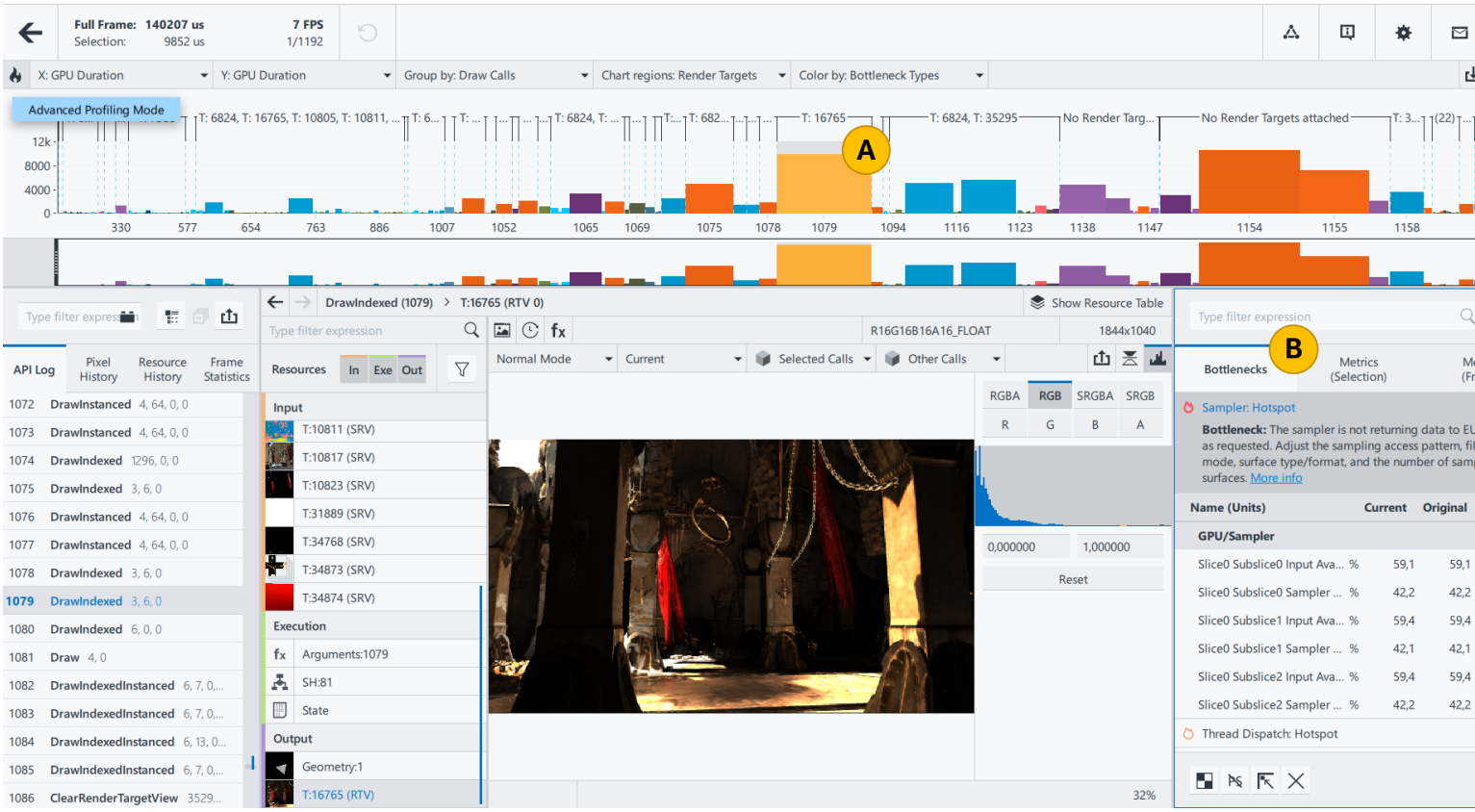
Select a group of draw calls for analysis either using



Advanced Profiling mode or choosing the calls that take more time on the GPU (



).



To determine whether shaders cause a performance bottleneck for the selected draw calls, refer to the **Bottlenecks** tab of the **Metrics** pane (



). Shader algorithms may cause performance issues in various cases, but the main ones are the following:

- 

  Shader Execution bottleneck is caused by shaders that perform complex computations and are executed many times. In most cases, the bottleneck is caused by pixel and compute shaders.

  In this case, EU Thread Occupancy and EU Active metrics are close to 100%.

  To optimize the shader execution bottleneck, you can simplify a shader: reduce unnecessary computations in a shader code and avoid using complicated arithmetic functions.

- 

  Sampler bottleneck appears when the sampler is not able to generate output data with the requested speed.

  In this case, there's a huge difference between Input Available and Sampler Output Ready metrics.

  To optimize the sampler bottleneck, you can: identify which sampling instructions take more time, optimize inefficient functions by changing resource parameters (format, size, filter parameters), or reduce the amount of sampling.

- 

  L3 bottleneck appears when graphics cache (L3) is not able read or write data with the requested speed.

  To optimize L3 bottleneck, you can reduce memory traffic or improve memory access patterns to Shared Local Memory (SLM), Unordered Access Views (UAV), textures, and constants.

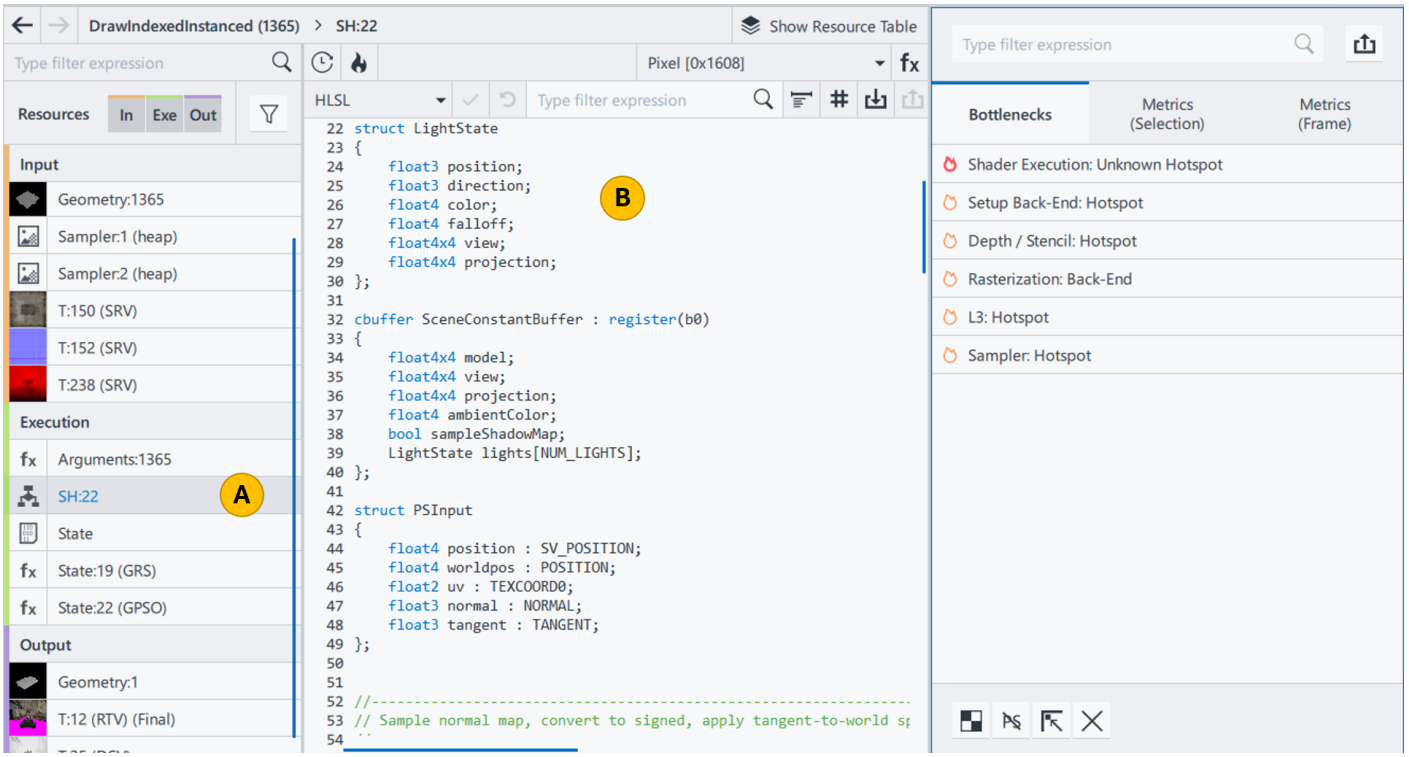
This topic covers basic steps for shader profiling with Graphics Frame Analyzer. To get detailed solutions for bottlenecks, refer to Cookbook recipes Optimize Shader Execution and Optimize Sampler.

Profile Shader Source Code

1. Once you selected draw calls for analysis, choose the shader used in these draw calls from the Resource List (

   A

).



   Graphics Frame Analyzer displays the shader source code in the **Shader Editor** (

   B

).

2. From the **Shader Type** drop-down list on the top right (

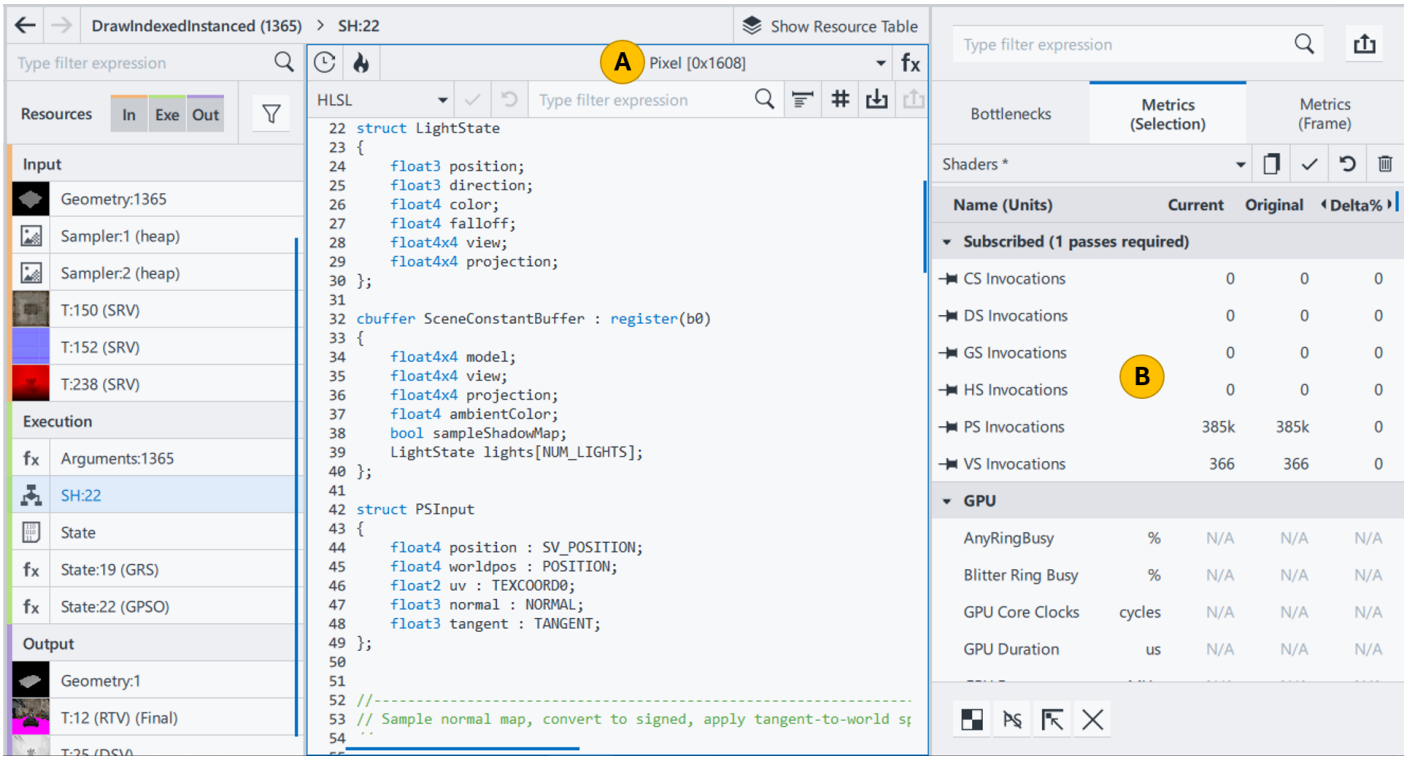
   A

   ), choose pixel or vertex shader for profiling.

   In most cases, the pixel shader is causing the bottleneck, but you can compare *Shader Invocations* (

   B

) metrics to understand which shader was executed more times and needs optimization.



The shader code opens in the Shader Editor. For easier reading, you can click the



button to indent the code, and the



button to preprocess the selected shader and hide the code paths that do not get executed.

3. Click the



Shader Profiler button to view performance data per shader code line.

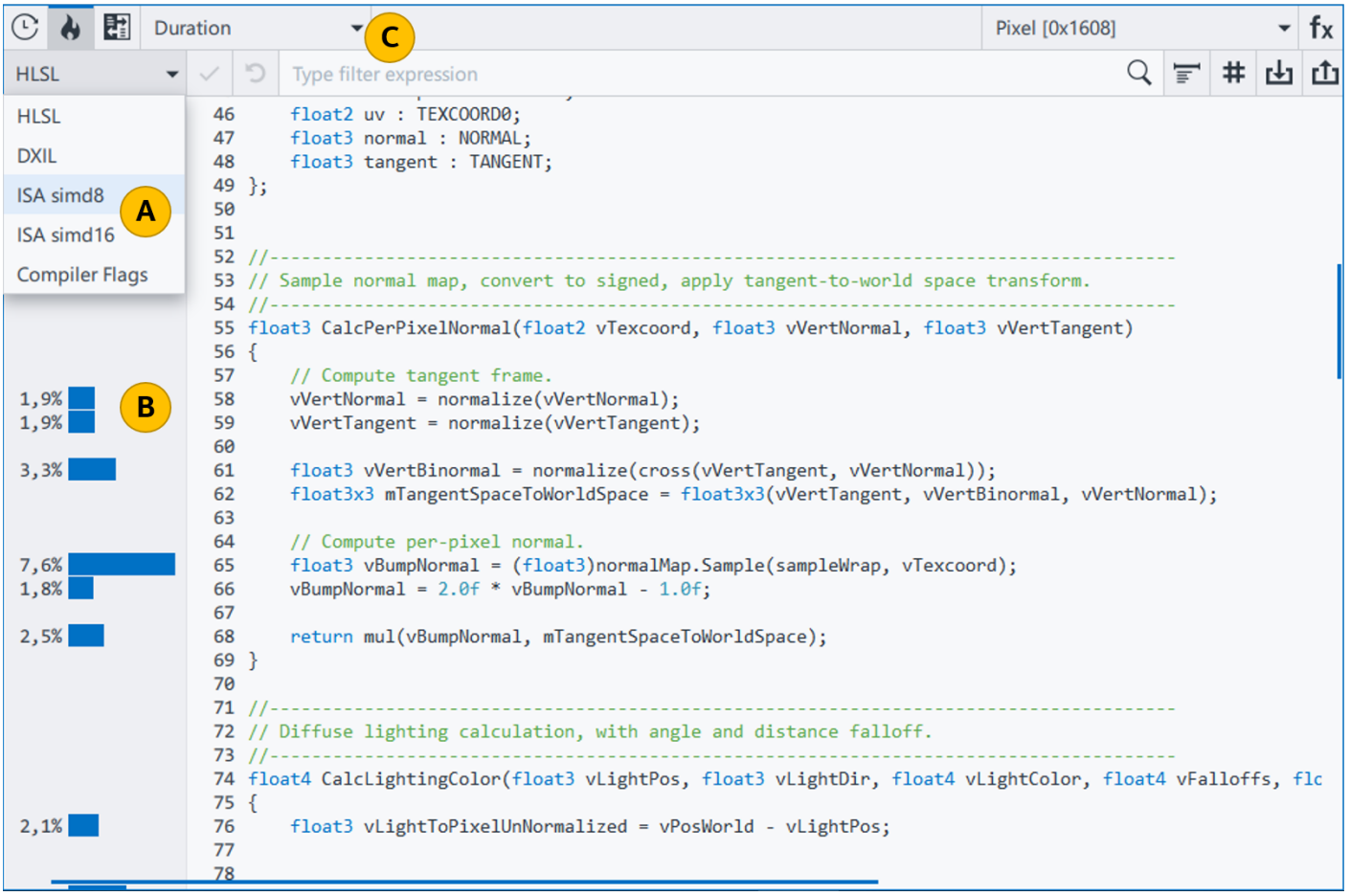
The shader viewer displays all generated versions of Gen ISA code in the drop-down menu on the top left (



), and the profiling data column on the left of the shader editor (



).



You can choose either duration or execution count to analyze the efficiency of your shaders (

C

).

- **Duration:** shows the estimated portion of time a line of code took in percent, relative to the execution time of all shader stages.
- **Execution Count:** shows the total number of times the exact line of code was executed.

At this step, you need to understand which parts of code take more time. Stalls may occur, for example, due to inefficient use of resources or redundant calculations.

As a next step, you can:

- Analyze shader resources
- Profile shader assembly code
- If you already have assumptions about bottleneck root cause and possible improvements, experiment with shader code and evaluate the impact on performance.

Analyze Shader Resources

To inspect resources bound to the shader, open the **Shader Resource List** by clicking the

fx

button.

In the **Shader Resource List**, resources used in the shaders are grouped by type:

- Render Target View (RTV)

- Sampler
- DirectX resources:
  - Shader Resource View (SRV)
  - Constant Buffer View (CBV)
  - Unordered Access View (UAV)
- Vulkan resources:
  - Access View (UAV)
  - Storage Buffer Object (SBO)
  - Storage Texture
  - Texture
  - Uniform Buffer Object (UBO)
  - Vertex Buffer View (VBV)

Graphics Frame Analyzer displays resource parameters, the shader type using the resource, and shader registers each resource is bound to. Resources are listed in the following format: <resource name> <register ID> <resource type>:<resource ID> (<view type><view ID>) <resource debug name>. For example, *cbPerObject b0 B:43 (CBV 1).*

To open a resource, click the desired resource name from the **Binding** column.

Profile Shader Assembly Code

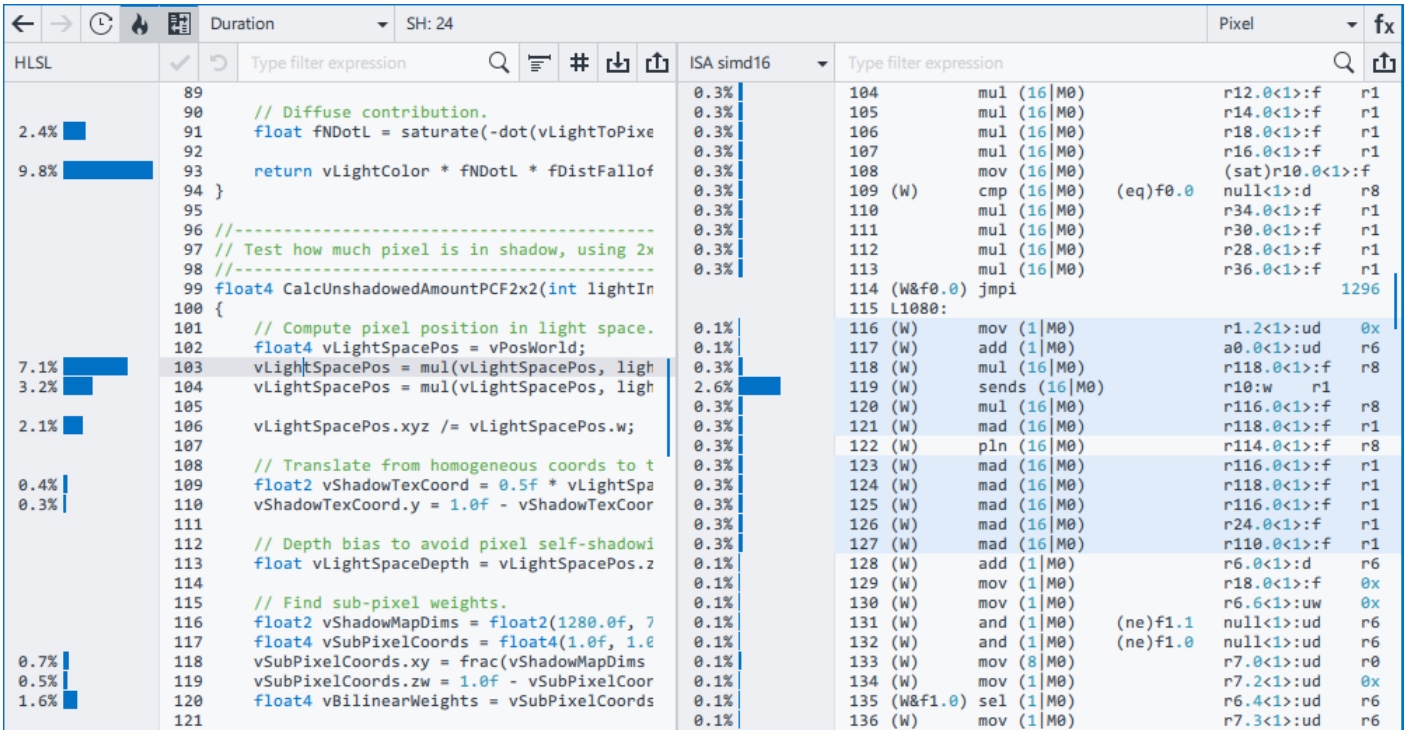
When optimization of shader algorithms reached its limits, you can step down to a low-level optimization and examine ISA shader code to use GPU resources more efficiently.

To view ISA instructions, select *ISA* from the drop-down menu on the top left. To view source-assembly mapping:

1. Click the 

   Shader Profiler button to view performance data per shader code line.
2. Click the 

   Show source-assembly mapping button to view the source code and the assembly code side-by-side, and to map individual source or assembly lines to their counterparts.



Source-assembly mapping is available for shaders compiled with debugging information. To enable source-assembly mapping, you can:

- Compile a shader with debugging information in your application.
- For DirectX frames, apply any modification to a shader in Graphics Frame Analyzer and click the

  button. Your shader recompiles with debugging information directly in Graphics Frame Analyzer.

If you are new to Gen assembly profiling, refer to Introduction to GEN Assembly to learn how to interpret ISA code and understand register region syntax and semantics. To get detailed descriptions of Gen ISA instructions, refer to Intel® Iris® Xe Graphics Open Source Programmer's Reference Manual.

Experiment with Shader Code

You can experiment with the HLSL code for DirectX*, and HLSL and GLSL for Vulkan*, if they are available for your shader. Edit the shader directly in Graphics Frame Analyzer and evaluate the impact on visuals and performance in real time:

1. Select HLSL or GLSL from the respective drop-down menu and edit the code in the Shader Editor.

   The shader recompiles on the fly. If you introduced any errors, you can see the corresponding message in the Notification pane below the Shader Editor.
2. Click the

   button to save the changes.

   Graphics Frame Analyzer recalculates all metrics and displays new data in the Metrics pane and in the Main bar chart.

NOTE When you click the



button, Graphics Frame Analyzer saves all the shaders. This enables you to write your own code and replace the whole shader to experiment.

3. If you want to undo your edits, click the



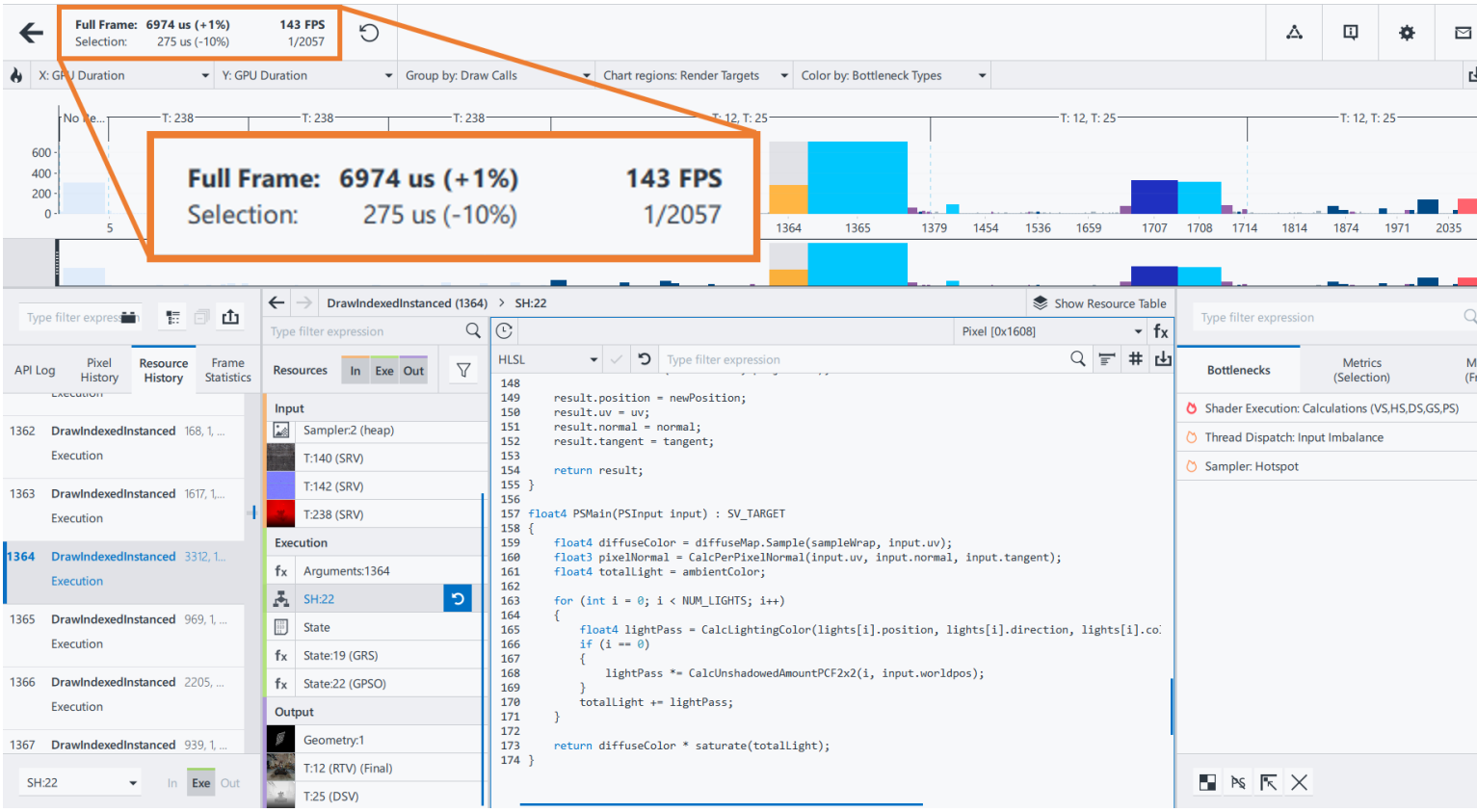
button.

The original shaders are restored.

Evaluate Performance

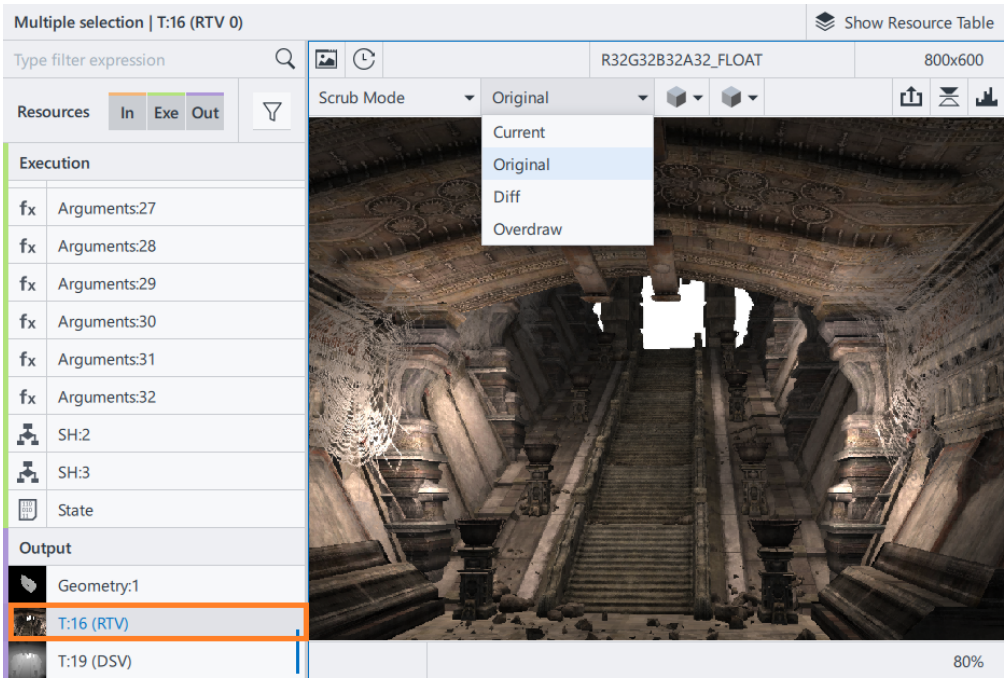
After you apply changes, metrics change automatically.

See how time spent on selected draw calls changes on the top left of the **Profiling View** window. The difference between current and original states is shown in braces.

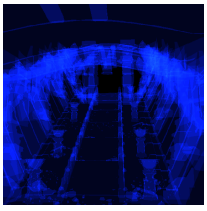


Evaluate Final Picture

To see how the render target changes, open the render target view (RTV) resource from the resource list.



To compare a new render target version and the original one, choose one of the modes from the **Output Texture Visualization Mode** drop-down list:

Current	Original	Diff	Overdraw
Shows the render target with modifications.	Shows the render target without modifications.	Shows the difference between the current and the original mode.	Shows the render target with an overdraw visualization.
			

NOTE

- For DirectX* 11 shaders without debug information, DXBC-ISA mapping is available instead of HLSL-ISA mapping.
- Source-assembly mapping is not supported for Shader Model 5 shaders on DirectX* 12 applications. To enable it, recompile the shaders for Shared Model 6.
- Shader Profiler requires Intel® Graphics Driver version 26.20.100.7755 or higher.
- This feature is supported on 9th Generation (code names Skylake, Coffee Lake, Kaby Lake), and 11th Generation (codenamed Ice Lake) Intel® Graphics hardware.

Analyze Textures

Review all textures for the selected events with Graphics Frame Analyzer to determine whether you can optimize texture usage and rendering algorithms.

Using high-resolution textures or multiple textures within a draw can negatively impact your application performance.

To speed up the rendering, you can:

- Reduce the physical size of textures
- Reduce the number of textures used in the scene, for example, use one texture for a set of objects
- Change the texture filtering setting
- Reduce the number of texture fetches in the shaders

Graphics Frame Analyzer enables you to experiment with textures and see how your changes impact performance and the final picture.

Experiments help to define whether the selected texture bottlenecks the sampler module. If a sampler block is highlighted red or yellow in the **Advanced Profiling** mode, there are issues with texture sampling.

Select Textures for Analysis

1. Click the draw call or the group of draw calls you are interested in (

).

   The list of resources associated with the selected region appears. Texture names are shown in the following format: *T:<texture ID> (<view type><view ID>),* for example, *T:325 (SRV 0).* Textures have four types of views:

   - Shader resource view (SRV)
   - Render target view (RTV)
   - Unordered access view (UAV)
   - Depth stencil view (DSV)

2. (Optional) Filter the list of resources (

   ):

   - Use the

     button to select only textures.
   - Turn off

     Execution resources, as textures are only used as

     Input or

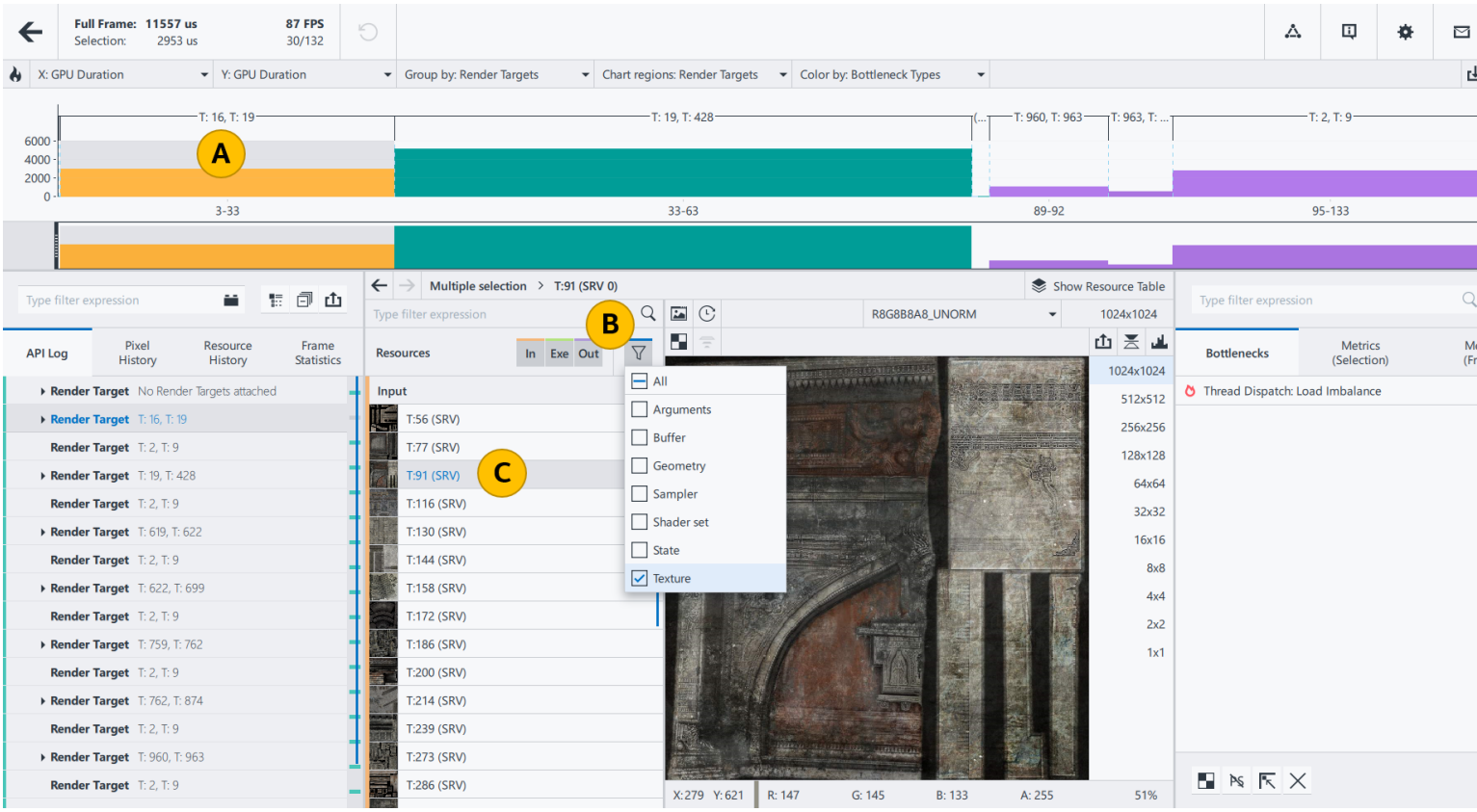
     

     Output.

3. Click the texture you are interested in (

).



4. (Optional) Prepare a texture for analysis:
   - **Open a texture in a new window** by clicking the

     button.

     For example, you can view the RTV texture on a separate display and continuously track changes after optimization activities. The texture in a separate window is updated dynamically.
   - **Flip the texture image** by clicking the

     button.

     This is useful if your texture is stored in memory in the reverse order and appears upside down (OpenGL, DirectX).
   - **Select MIP level or the image layer** for 3D textures and texture arrays.

Experiment with Textures

You can apply the following experiments to shader resource view (SRV) textures, which are used as input:

Texture 2x2

This experiment replaces the texture with a simple 2x2 pixel texture for the selected events.

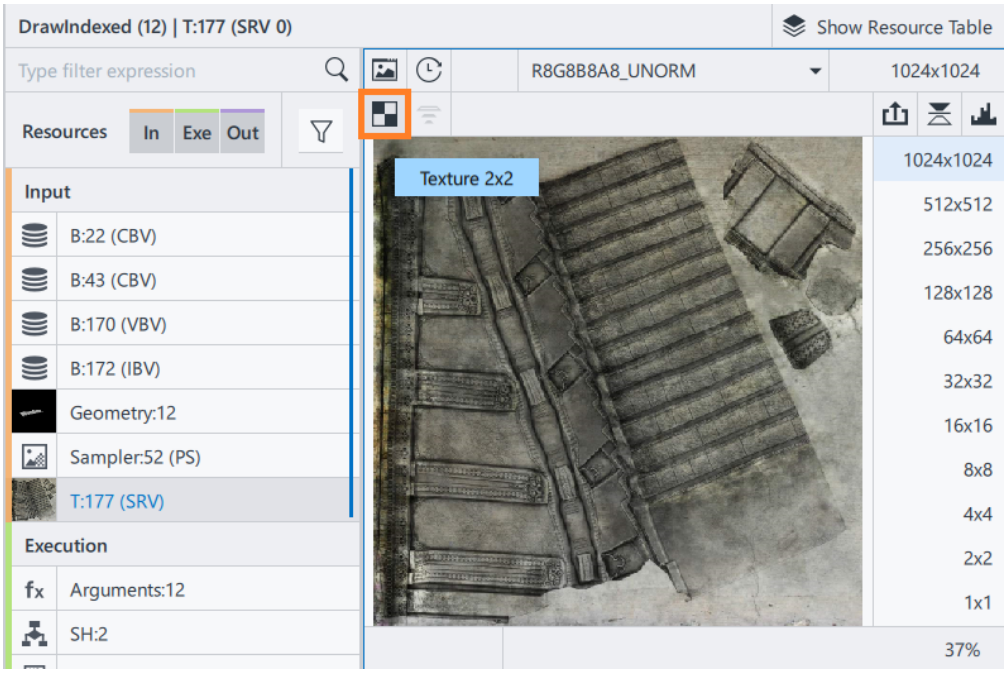
Use to identify which texture is causing the issue when you have multiple textures within a call.

To apply the experiment:

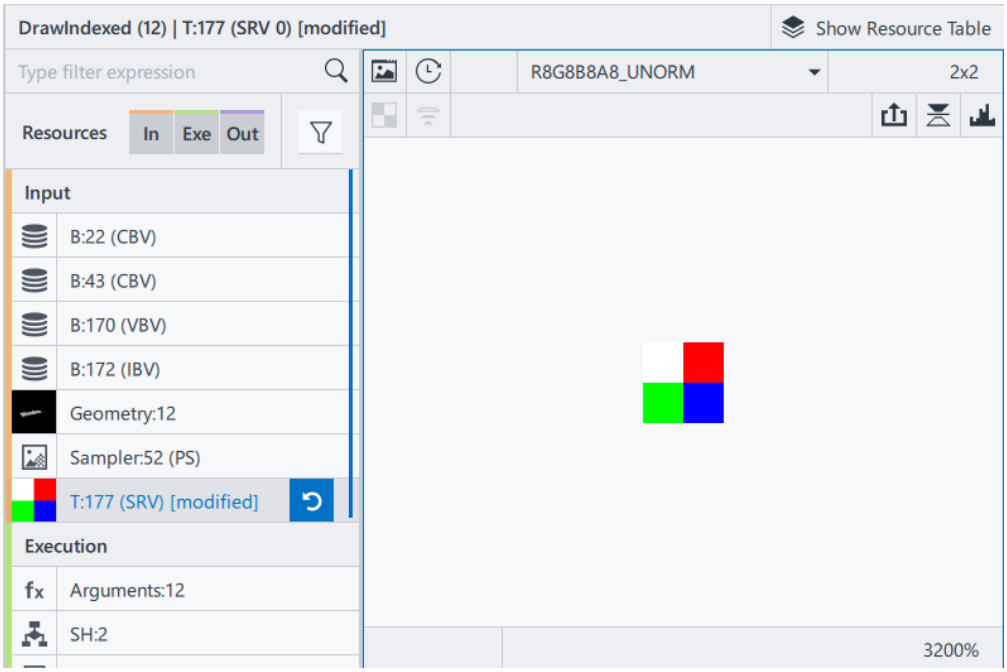
1. Navigate to the texture of interest.
2. Click the



button on the top of the **Resources** pane.



The experiment replaces the texture with a new one of the same type and with the same name, denoted as [modified].



Clamp to Selected MIP Level

This experiment disables all texture mipmaps that are greater than the selected mipmap level.

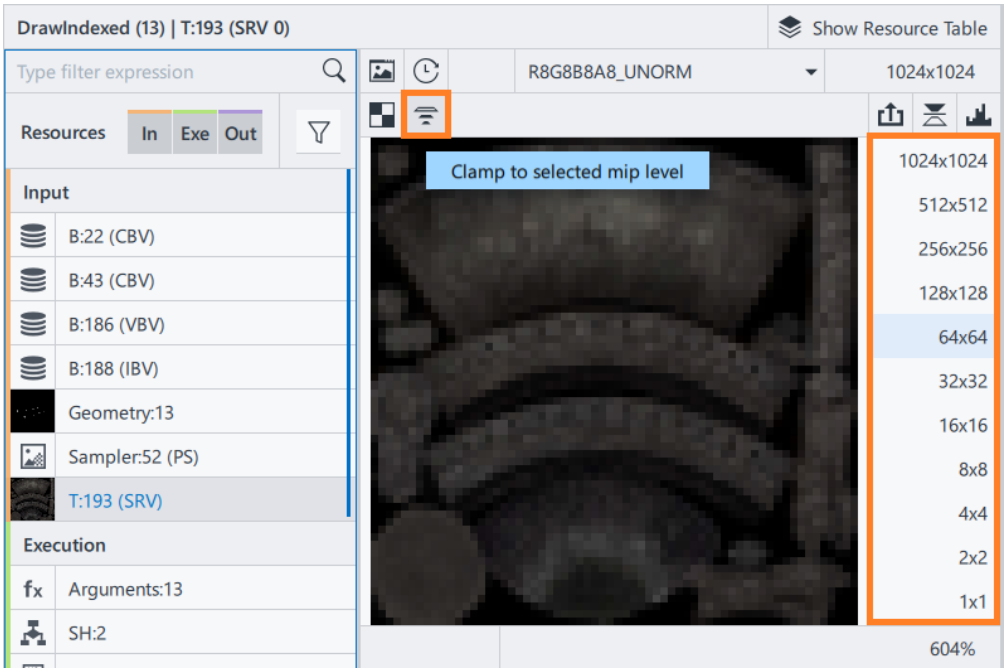
Use to check whether your mipmap algorithm works as expected. Find out whether performance is better when you use less detailed textures and how it impacts the render target.

To apply the experiment:

1. Navigate to the texture of interest.
2. Select the MIP level on the right of the **Resources** pane.
3. Click the



button on the top left of the **Resources** pane.



The experiment replaces the texture with a new one of the same type and with the same name, denoted as [modified].

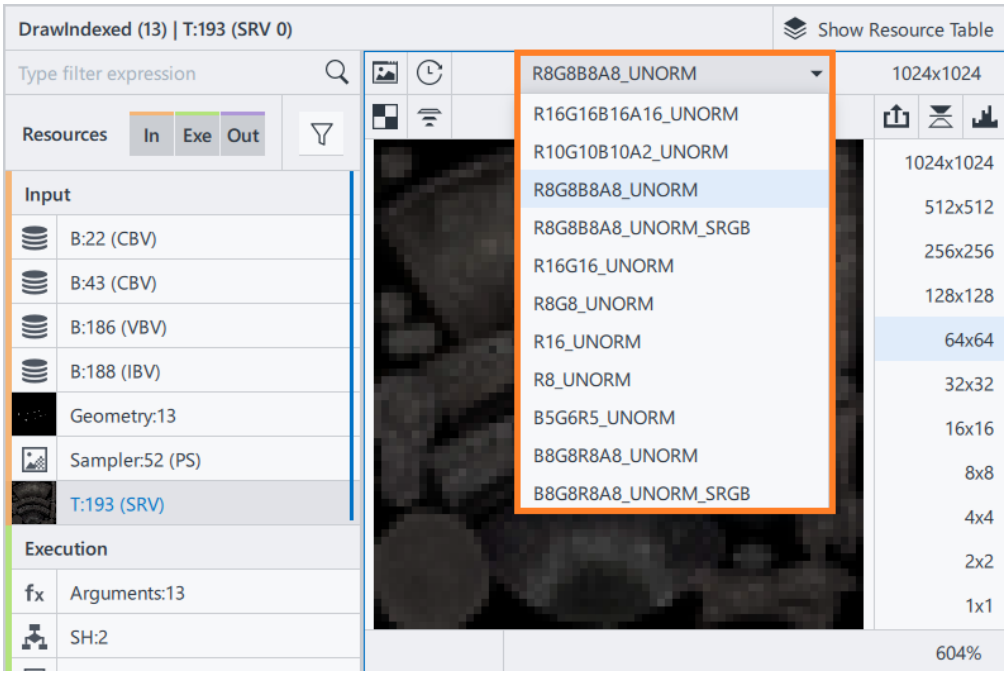
Texture Format

This experiment changes the texture format (DirectX 11 only).

Use to understand how the texture format influences performance.

To apply the experiment:

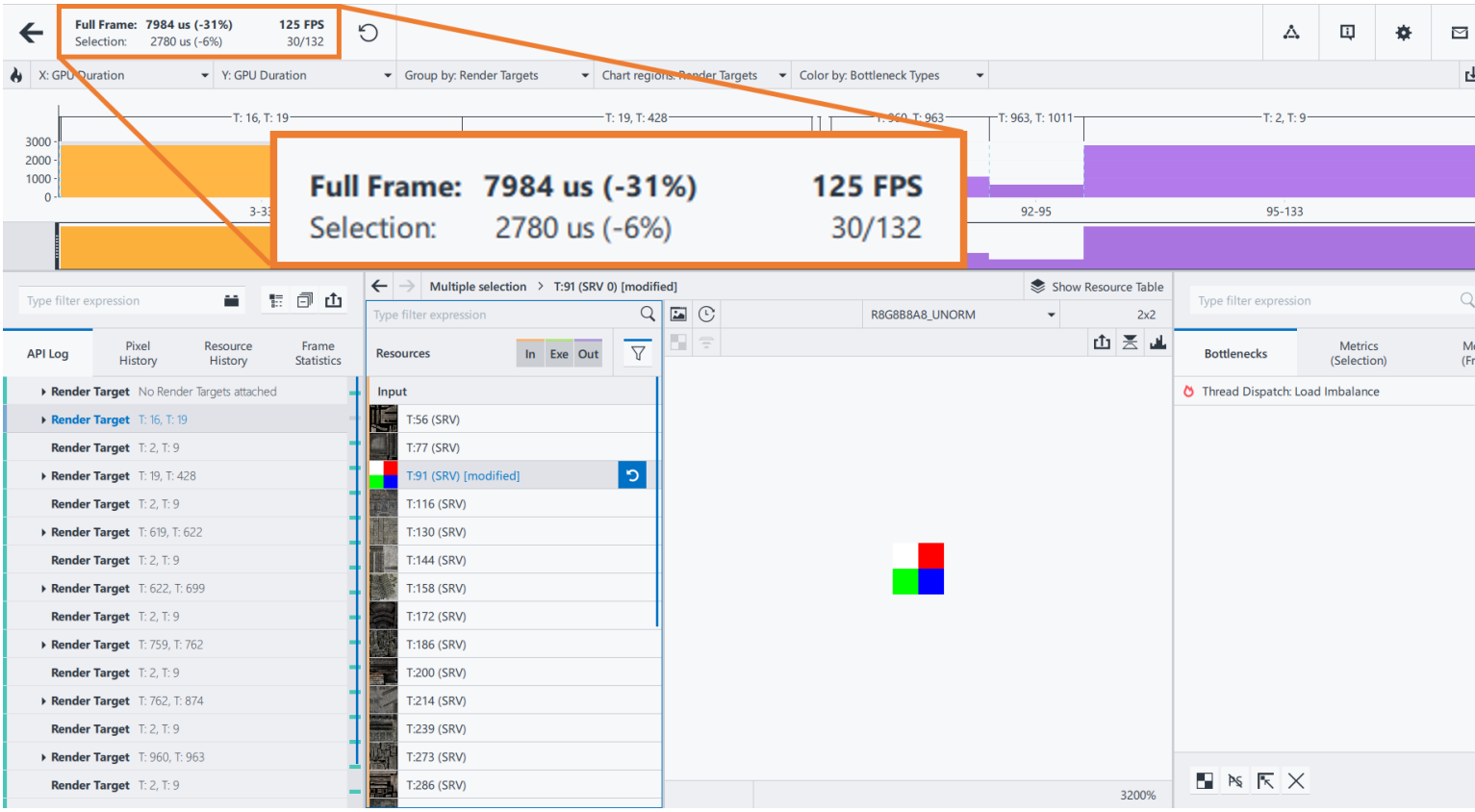
1. Navigate to the texture of interest.
2. Select the format in the drop-down menu on the top of the **Resource Viewer** pane.



Evaluate Performance

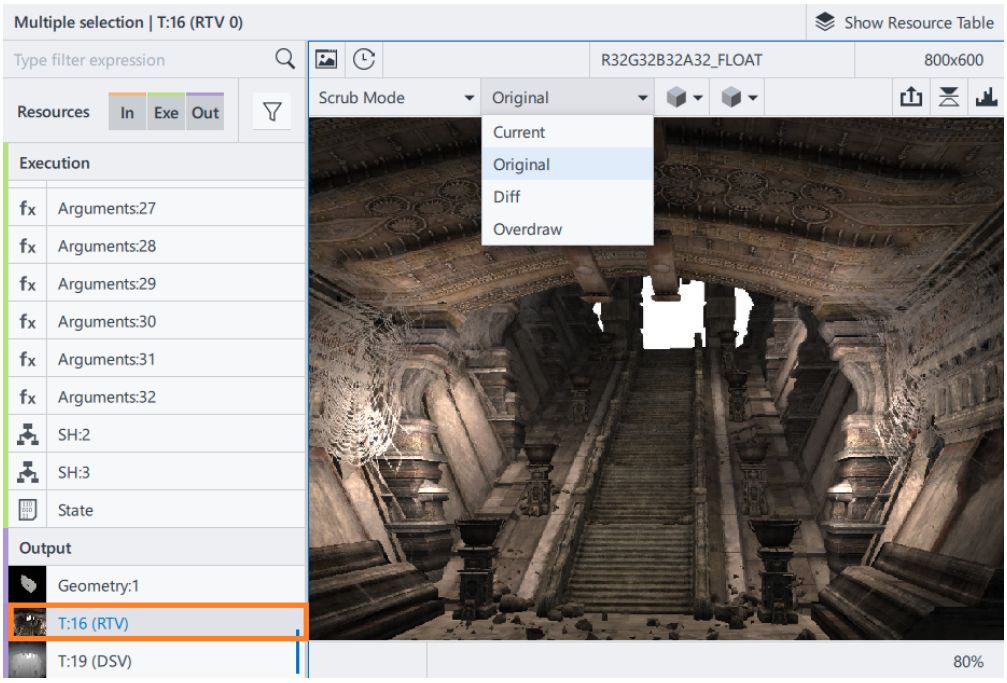
After you apply experiments, metrics change automatically.

See how time spent on selected draw calls changes on the top left of the **Profiling View** window. The difference between current and original states is shown in braces.

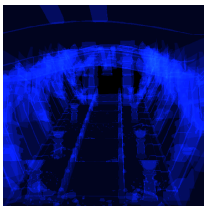


Evaluate Final Picture

To see how the render target changes, open the render target view (RTV) resource from the resource list.



To compare a new render target version and the original one, choose one of the modes from the **Output Texture Visualization Mode** drop-down list:

Current	Original	Diff	Overdraw
Shows the render target with modifications.	Shows the render target without modifications.	Shows the difference between the current and the original mode.	Shows the render target with an overdraw visualization.
			

Save Results

Export changed textures by clicking the



button or pressing **Ctrl+S.** Depending on the texture format, you can choose between PNG, DirectDraw Surface (DDS), or Khronos Texture (KTX) formats.

Use exported images as key points to save your optimization results, compare, and choose the best version.

Optimize your code based on experiment results.

See Also

Analyze GPU-bound Applications with Graphics Frame Analyzer

Optimize Shaders

Visualize Color Distribution

Resource Viewer: Texture

Minimize Overdraw

With the Graphics Frame Analyzer, you can discover API calls that affect specific pixels and minimize overdraw by removing redundant or unimportant calls. For example, lighting effects are typically produced by multiple rendering passes. You can determine whether you can reduce the number of iterations to produce a similar visual effect with higher performance. To detect the sequence of API calls that affects the final color of the pixel:

1. Select the color buffer of the render target from the Resource List. Graphics Frame Analyzer opens the render target image in the Profiling View.
2. Analyze the render target image using different visualization modes available from the drop-down menu:
   - Current (default): shows the render target with modifications. If no experiments are applied, the current state is equal to the original state. In this mode, you can visualize the color distribution through a histogram by clicking the  button.
   - Original: shows the render target without modifications.
   - Diff: shows the difference between the current and original modes.
   - Overdraw: visualizes the hit count for each pixel of the render target image. The brighter the color, the more times the pixel was updated. Hover over the pixel to see the hit count below.
3. In the Overdraw mode, hover over the pixel you would like to analyze. Graphics Frame Analyzer displays the hit count for the selected pixel below the image.
4. Click the pixel you would like to analyze. Graphics Frame Analyzer marks the selected pixel with a crosshair and filters the API calls to display only those clear, draw, blit, or texture image update calls that affected this pixel, with the resulting pixel color shown next to each call.
5. Review the pixel history in the API Log pane. Graphics Frame Analyzer displays the pixel coordinates, pixel components, the number of times each draw call touched the pixel, and rejected draw calls. You can switch back to the color buffer to better understand the place of the selected pixel in the overall scene. To find out the reason for draw call rejection, you can tweak the pipeline state for this draw call. For example, disable depth or stencil tests. For draw calls rejected with scissor test or color mask, the API Log shows the rejection reason automatically.
6. Analyze the calls that affected the pixel to determine how to get the same or similar results at less cost:
   - If you see that two API calls affecting the pixel are of the same type, you may improve performance by removing one of them. To see how much performance you can gain by removing an API call, select it, and check the Disable Event(s) check box in the Experiments pane.
   - Detect API calls used by mistake. Remove such calls by checking the Disable Event(s) check box in the Experiments pane to see the potential performance improvement.

   NOTE You can only perform the Disable Event(s) experiment if metrics data is available for your target system. Otherwise, the Experiments pane is hidden.

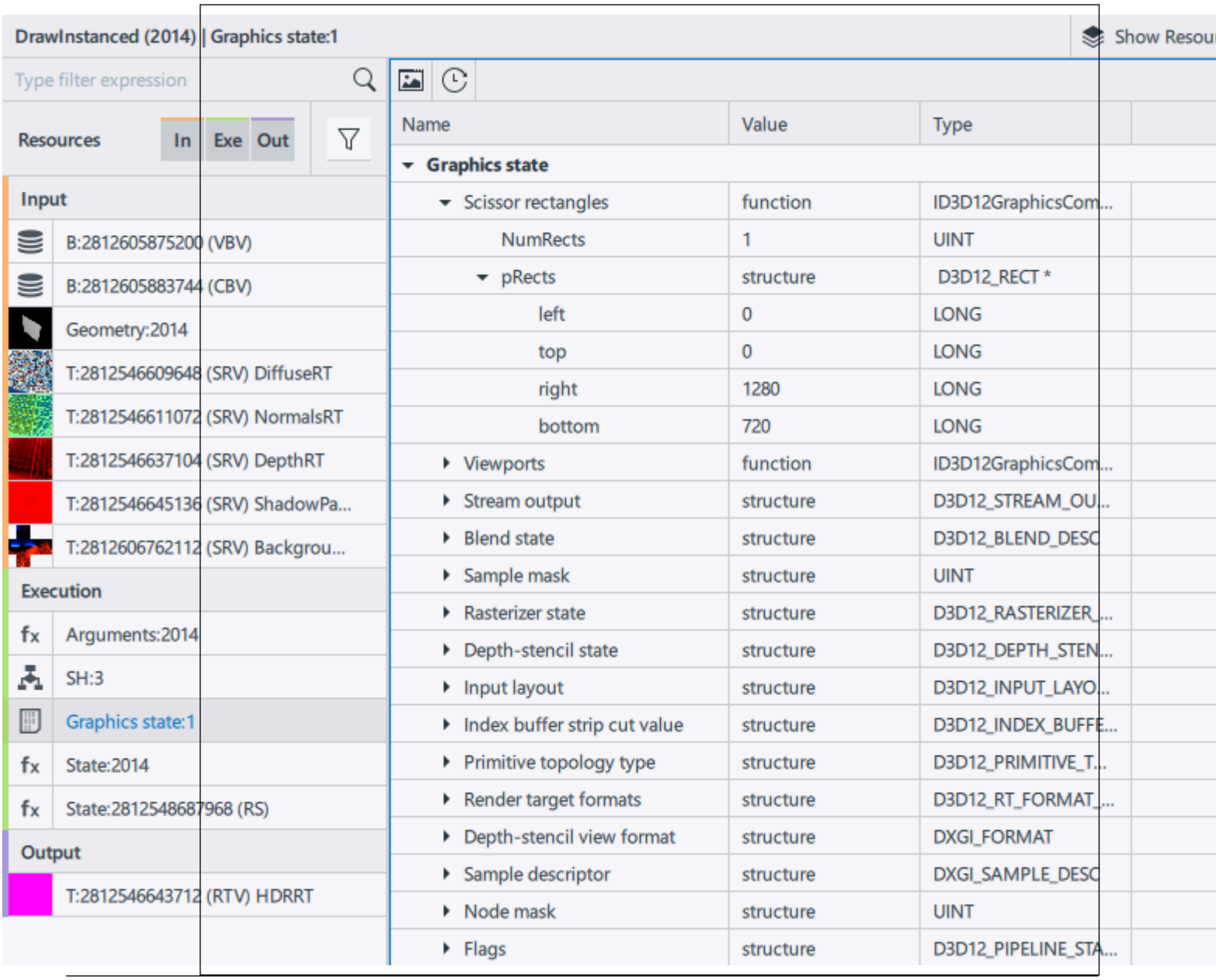
Scenes with many overlapping objects, or objects with semitransparent textures (such as smoke, fog, or water) often cause a high load on the graphics card and therefore lower frame rates. If you see that the overdraw is an issue, you could try culling primitives prior to rendering, altering the draw order (so that Z-buffer tests will reject many primitives prior to rendering them with an expensive pixel shader), or using other level-of-detail optimizations that simplify the complexity of the scene.

Optimize Graphic API States

With Graphics Frame Analyzer you can analyze graphics API states used in the captured frame:

- Graphics pipeline states used by draw calls. You can modify these states directly in the Graphics Frame Analyzer and see how these modifications might improve your application performance.
- Compute pipeline states used by dispatch calls. You can view these states in the Resource Viewer.

For frames opened from stream files, Graphics Frame Analyzer provides a combined state view that unifies both Pipeline State Object (PSO) and non-PSO states:



To experiment with graphics pipeline states:

1. Select a draw call in the Main bar chart. The Resource Viewer is updated to display all resources used by the selected draw call.
2. From the Resource List, choose the state to experiment with.
3. Edit one or more states. Depending on the state type, use one of the following methods:
   - Use ON/OFF toggle buttons to enable/disable state groups.
   - Enter new values to modify numeric scalars, vectors, and bitmask values. If you enter an invalid value, the Graphics Frame Analyzer highlights it in red.

- Click the parameter to toggle between true/false, values or select a new value from the drop-down list with predefined values that conform to the graphics API specification.

Graphics Frame Analyzer recalculates all metrics values to reflect the introduced changes. To revert the change, click the



button that appears in the group title to which the modified states belong. If you want to revert all changes at once, click the



button in the Resource Viewer.

Identify Graphics Elements with Complex Geometry

Geometry analysis can help you both debug rendering issues with the complex objects and optimize your application performance. Using geometry analysis, you can identify and optimize inessential elements that take up a lot of resources. For example, if you find an element in the scene that includes a lot of polygons or complicated artwork but contributes very little to the visual effect of the scene, you can gain significant performance improvements by reducing the level of detail for the geometry of this element.

To analyze geometry for the selected API call:

1. In the Main bar chart or from the API Log pane, select a single API call that results in a rendering issue, or contributes the most to the overall frame rendering time.

   NOTE Graphics Frame Analyzer displays geometry objects only if you select a single API call.

2. Select the geometry thumbnail in the Resource List. The Geometry Viewer opens, displaying the object in the 3D coordinate system. You can assess the object from different angles by pressing the mouse wheel to switch the axes, or rotate the object by clicking and dragging the object with the mouse pointer.
3. Analyze various geometry aspects by switching between the available visualization modes:

   Solid (default) - view the object as a solid model.

   Wireframe - view a wireframe model to inspect the backside of the object and to understand how it is built from triangles.

   Solid Wireframe - view a wireframe model on top of the solid model.

   Normals - view a normal map for your object to analyze how the normal vectors are plotted.
4. Optionally, export geometry data in the Wavefront OBJ format by clicking the

   button.

Visualize Color Distribution

Investigate the texture color data using the color histogram in Graphics Frame Analyzer. Such analysis is useful for debugging and helps you to check whether your shading algorithms work as expected.

Depending on the format type, the pixel values of your texture or render target have different default ranges:

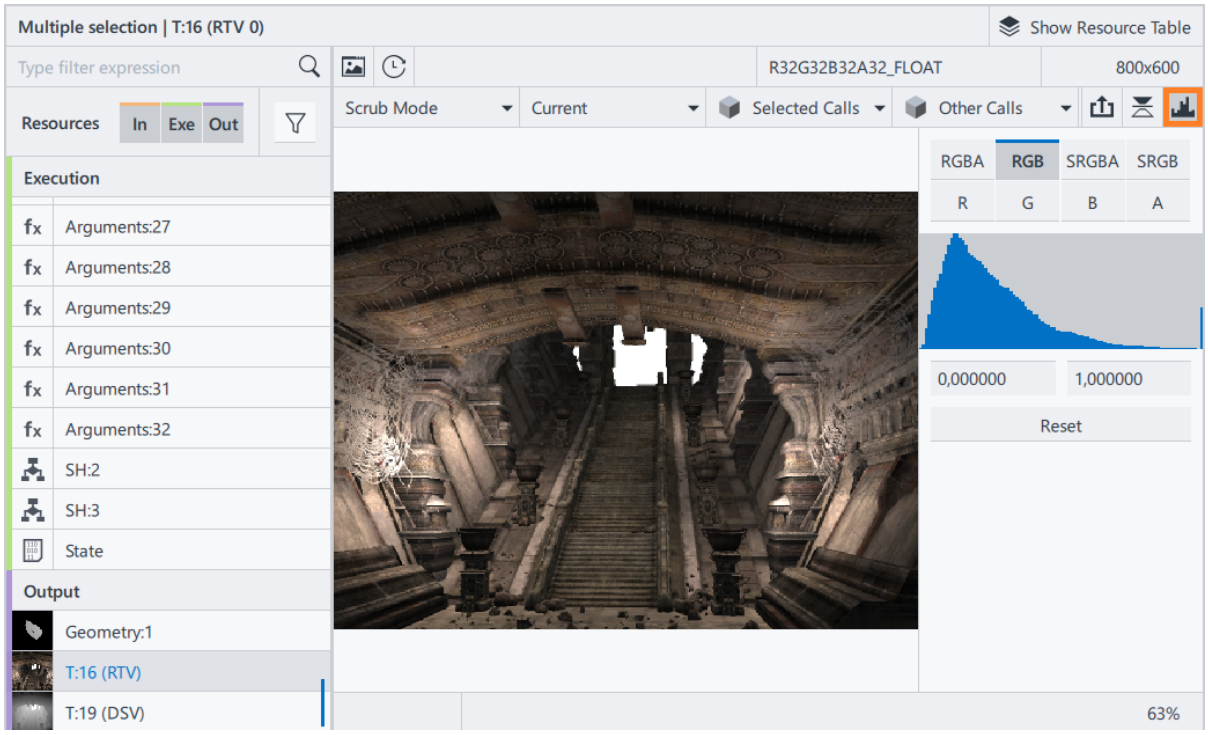
- For floating-point texture formats, the default range is [0, 1].
- For integer formats, the range spans from the minimum to the maximum value.

If the pixel values you are interested in lie within a range much smaller than default, it's hard to see the details of the image with the naked eye. In this case, try the following:

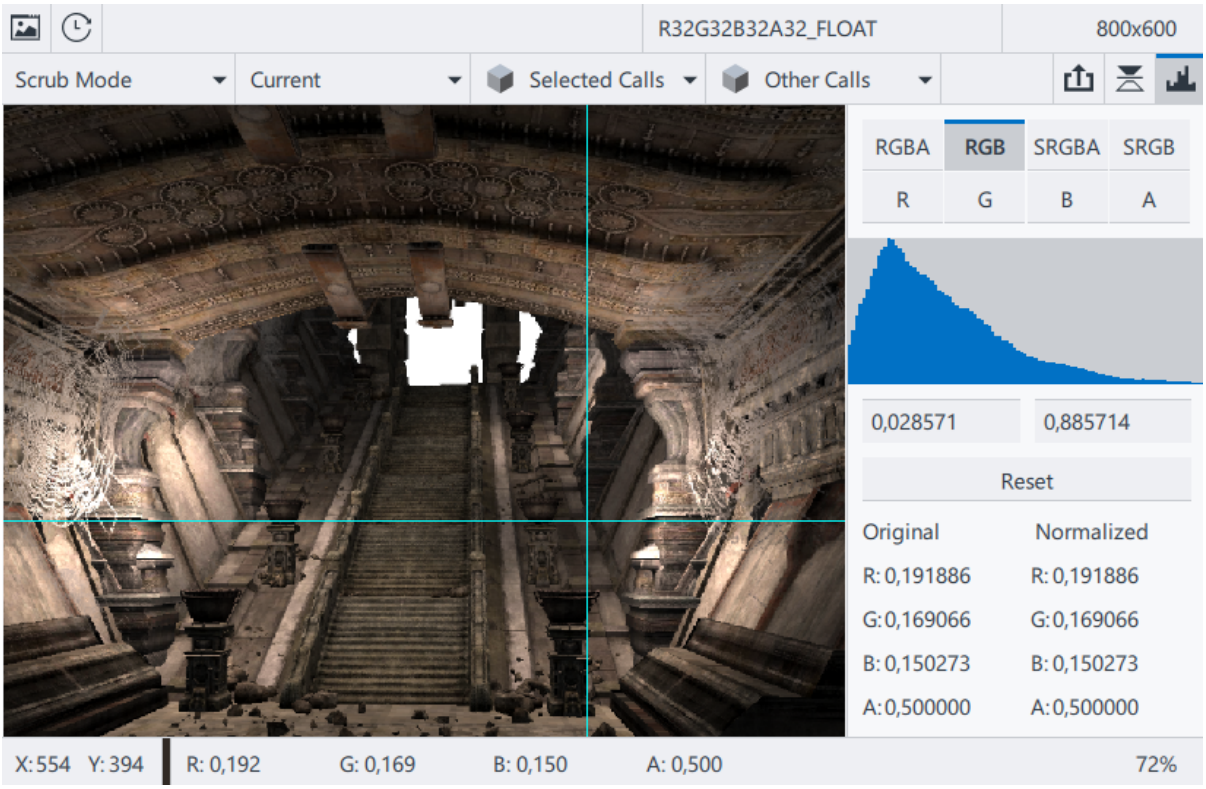
1. Select the texture or render target from the list of resources and click the

   button to open the histogram.

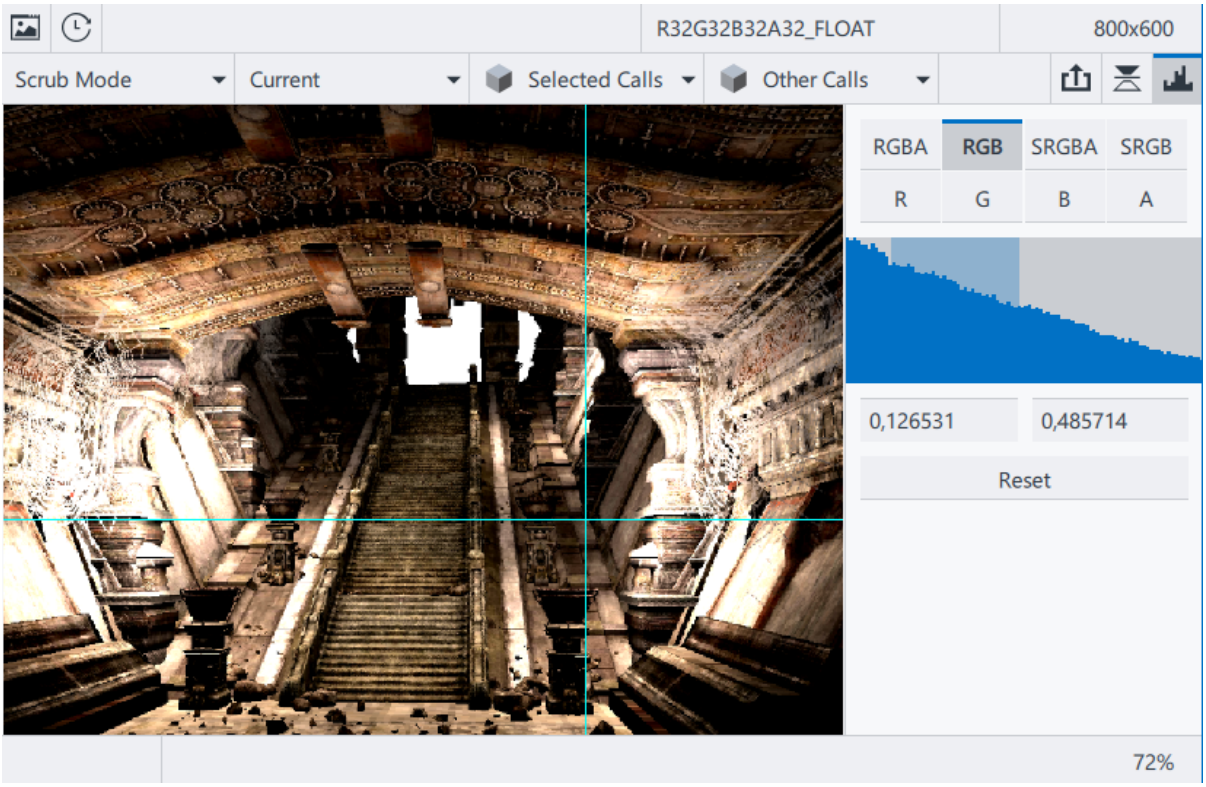


   You can see color intensity values on the X-axis, and the number of pixels with the given intensity on the Y-axis.
2. Analyze color distribution within the frame. Hover over the image to see the pixel values below. On the right, you can also see:
   - **Original** values used to store pixel information in memory - the same as shown below the image, but in a more precise format.
   - **Normalized** values used in shaders.



3. Drill down to the color range you are interested in:

- Select it in the histogram
- Type the threshold pixel values below the histogram



The image updates to show the data for the selected range only.

You can also navigate inside the histogram:

- Zoom in/out of the selected range using the mouse wheel.
- Scroll left/right by clicking and dragging the mouse while holding **Ctrl.**

4. If required for better visualization, toggle between available channels and channel configurations:

   - **RGB** mode includes all main color channels. **RGBA** and **SRGBA** also include alpha channel. **SRGB** and **SRGBA** also include gamma correction.
   - **Red/Green/Blue** color channels show the image information in red/green/blue color plane.

     This is useful to investigate each color channel separately when they contain different data. For example, one of the channels may be responsible for smoothness of rendered objects, and therefore their ability to reflect light.
   - **Alpha** channel is a grayscale conversion of the alpha data of the buffer.

     With alpha channel, you can analyze transparency of rendered objects.

   For non-RGB(A) images, the first available channel is selected. For example, the Depth plane is selected by default for Depth and Depth-Stencil textures.
5. Click the **Reset** button to return the histogram to the full view and select a different range, if needed.

If the pixel values of your floating-point texture/render target lie outside the default range [0, 1], you may not see some parts of the image. Using the histogram, you can detect the actual range, and then drill down to the sub-range of interest:

- Open the histogram for the selected texture/render target.

  Since the RGB mode shows the default color range, some of the data is not displayed.
- Select the color channel with the widest range and click **Reset.**

  The histogram updates to include the entire available range.
- Change the color channel back to the RGB mode.

  Switching color channels does not affect the range selection, so you can now view the full image with the actual range selected.

NOTE When analyzing textures/render targets that use 5-6-5 or similar formats, remember that color channels have different ranges, whish are not updated automatically:

- [0, 32] for the Red channel
- [0, 64] for the Green channel
- [0, 32] for the Blue channel

Since the data from Red, Green, and Blue channels is taken in different proportions for calculating intensity values in RGB mode, peak values in the histogram for RGB mode do not coincide with those for separate color planes.

See Also

Analyze GPU-bound Applications with Graphics Frame Analyzer

Analyze Textures

Resource Viewer: Texture

Quick Start with Profiling Unreal Engine* Games

Intel® GPA allows you to see FPS, shader, texture, and draw call details of the application. With the Graphics Performance Analyzers Plugin (GPAPlugin) for Unreal Engine*, you can capture a multi-frame stream directly while you are in Unreal Editor, and later analyze it with Intel GPA Graphics Frame Analyzer.

Multi-frame stream capturing allows you to analyze problems that are virtually impossible to debug in single frame capture mode. Such problems as intermittent glitches, random hitches, and issues with multi-frame algorithms can be tracked down more easily in Stream (multi-frame) mode because you can toggle between frames in a single capture.

This topic shows you how to start analyzing Unreal Engine* games with Intel GPA tools and the Intel GPA plugin in Unreal Editor.

Understand the Workflow

The common use case is when you know that your application is GPU-bound and you need to look into the details of API calls, shader execution, resources, bottlenecks in the graphics pipeline, or understand how multi-frame algorithms are being executed. In this case, your workflow is the following, and the details for each of these steps are explained below:

1. Enable the Intel GPA plugin in Unreal Editor.
2. With the Intel GPA plugin, capture a multi-frame stream directly from Unreal Editor.
3. Open a stream and analyze it with Graphics Frame Analyzer. In this step, you can track such problems as random hitches and issues with multi-frame algorithms or identify frames that are taking more time than expected.
4. Select a frame, or a set of frames, of interest and continue in-depth analysis with Graphics Frame Analyzer. In this step, you can analyze API calls, graphics resources, and find bottlenecks in the graphics pipeline.

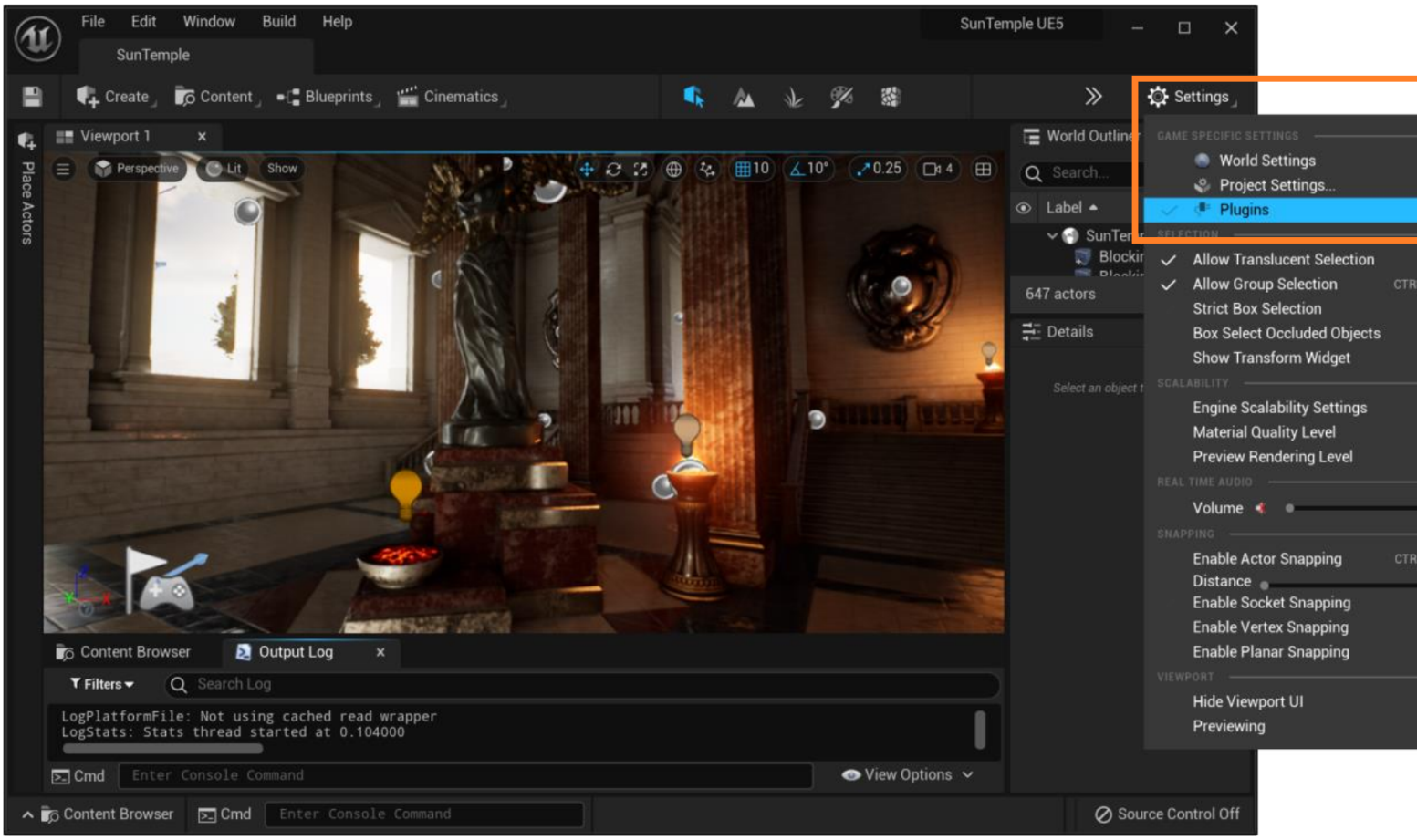
Prerequisites

To be able to capture and analyze multi-frame streams from Unreal Editor:

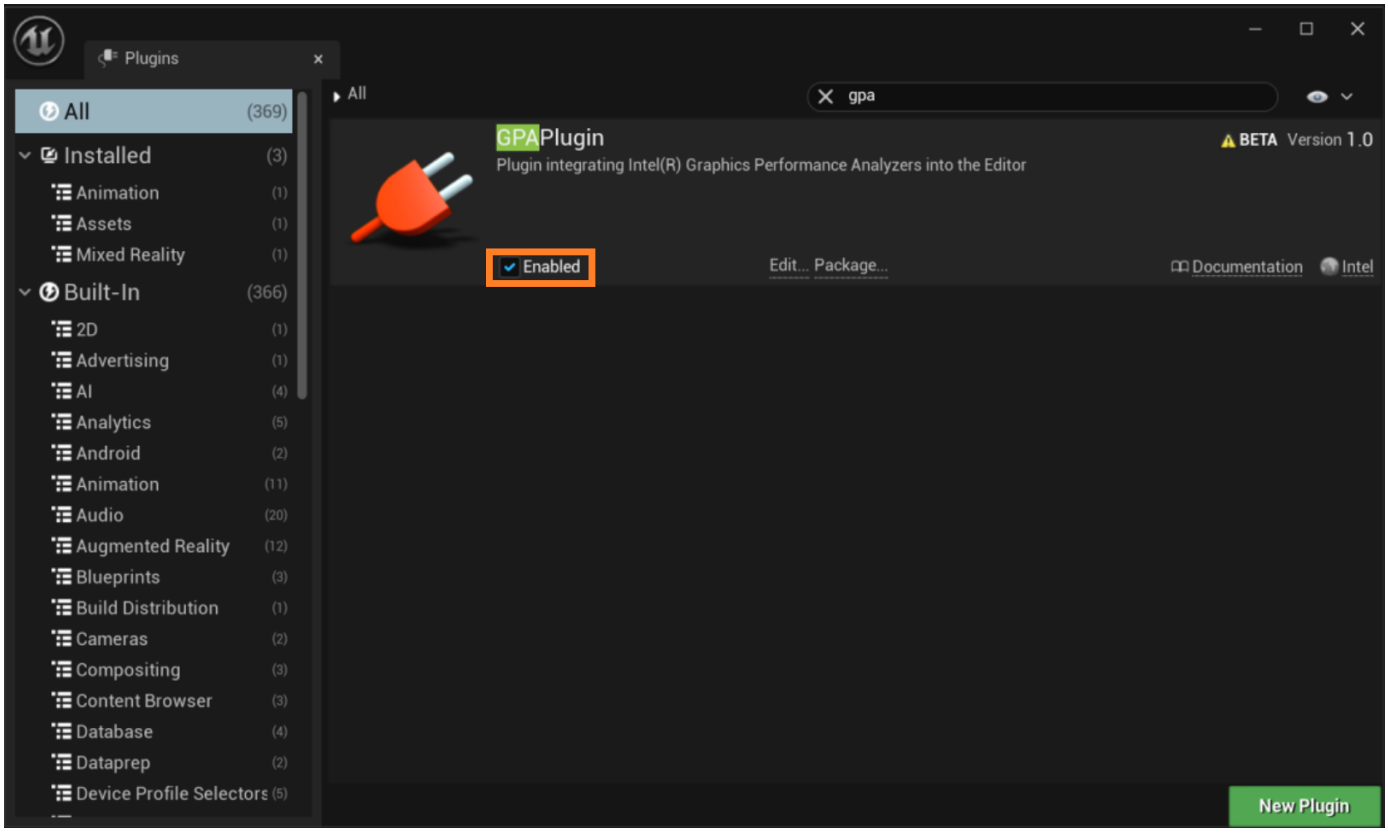
- Install Intel GPA from the download page.
- Download the GPAPlugin plugin from GitHub.

Enable the Intel GPA Plugin

1. Download the GPAPlugin from the GitHub page and copy the GPAPlugin directory to `UnrealEngine\Engine\Plugins\Developer` in your Unreal Engine installation folder.
2. Run Unreal Editor and open the **Plugins** menu in **Settings.**



3. Look up **GPAPlugin** using the search bar. Enable the plugin using the check box.



4. Follow the prompt and restart Unreal Editor.

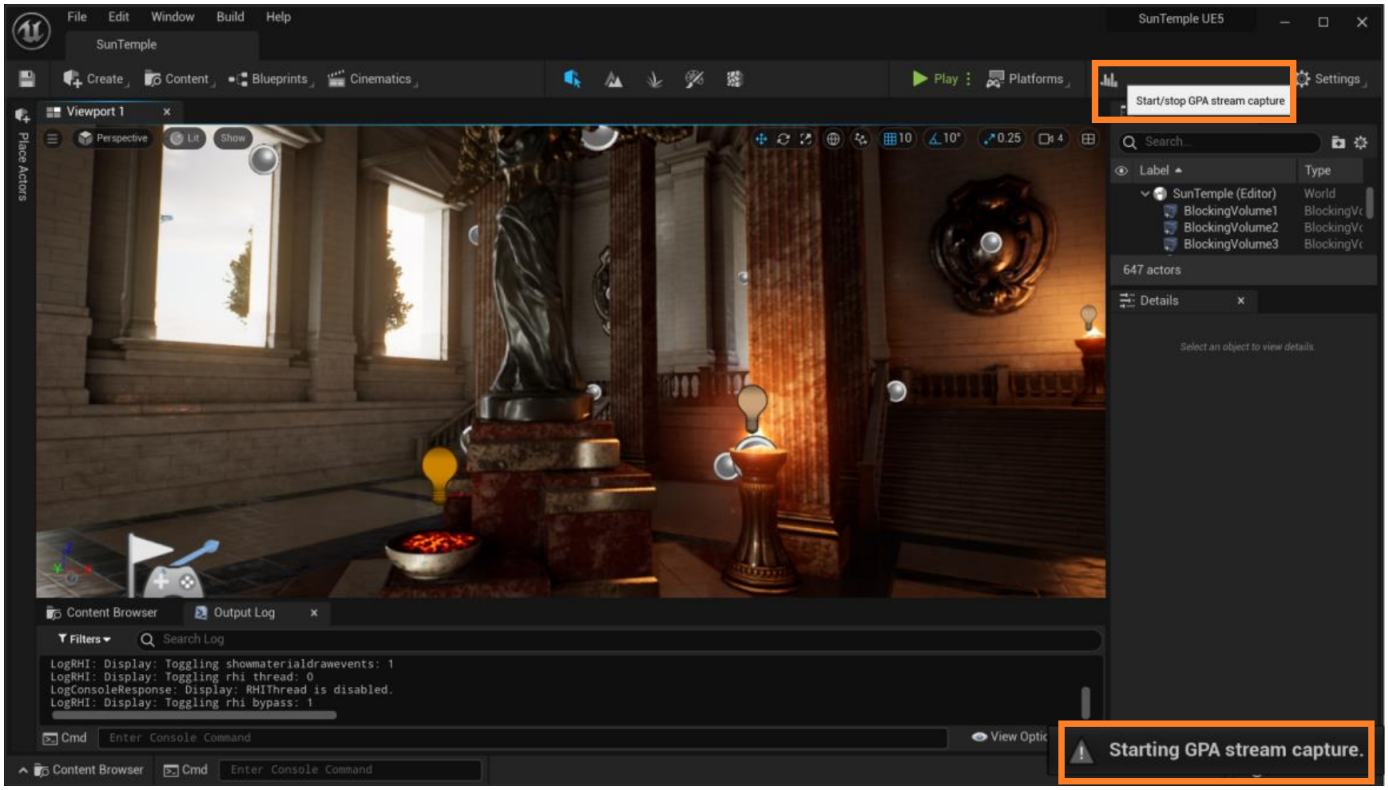
Capture Streams with the Intel GPA Plugin

Choose one of the following ways to capture a multi-frame stream:

- **In the editor:**

  Once you enable the plugin and restart the editor, an additional **Start/stop GPA stream capture** button displays in the toolbar.

  1. To start the capture of a sequence of frames, click the **Start/stop GPA stream capture** button.



A pop-up window in the lower right corner indicates that the capture has started.

2. To stop the capture, click the same button again.

   A message shows that the capture is complete.

- **In the console:**

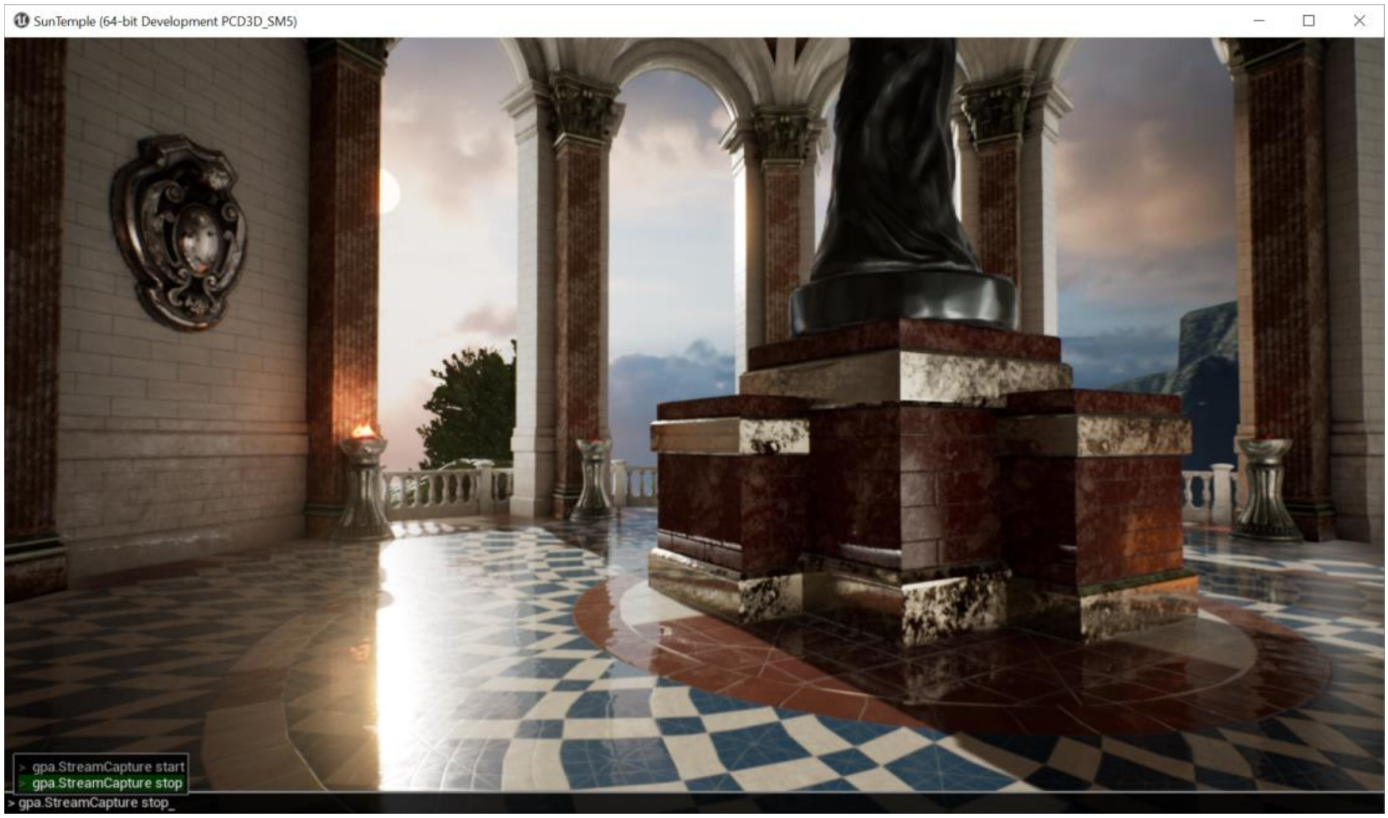
  You can also capture streams when running the game.

  1. To start the capture of a sequence of frames, use the command:

```
gpa.StreamCapture start
```

  2. To stop the capture, use the command:

```
gpa.StreamCapture stop
```



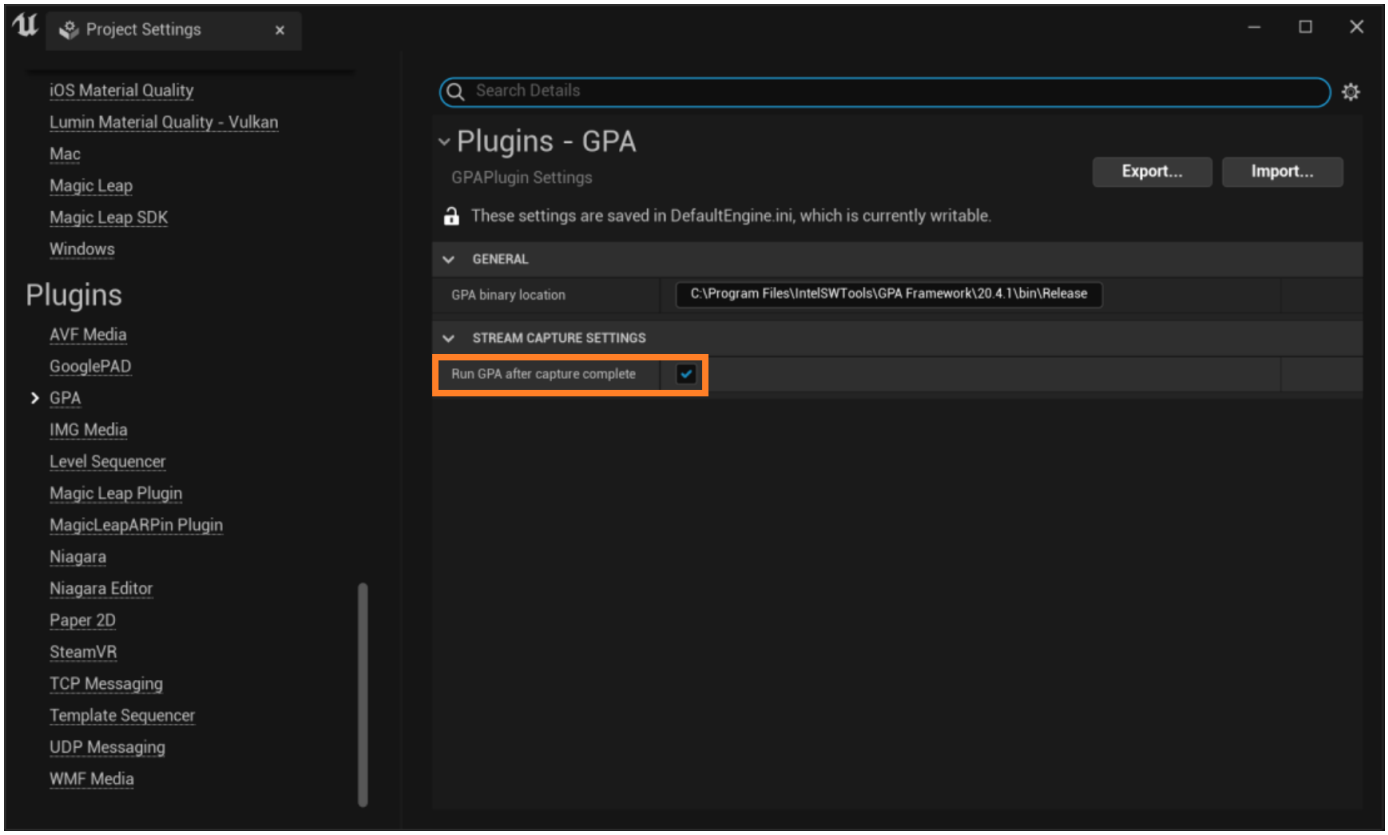
You can create multiple captures, each is stored separately for later analysis.

> **NOTE** Currently only DirectX* 12 captures are supported by the Intel GPA Plugin. Make sure to change the RHI to DX12 in **Settings->Project Settings->Platforms->Windows** and restart Unreal Editor.

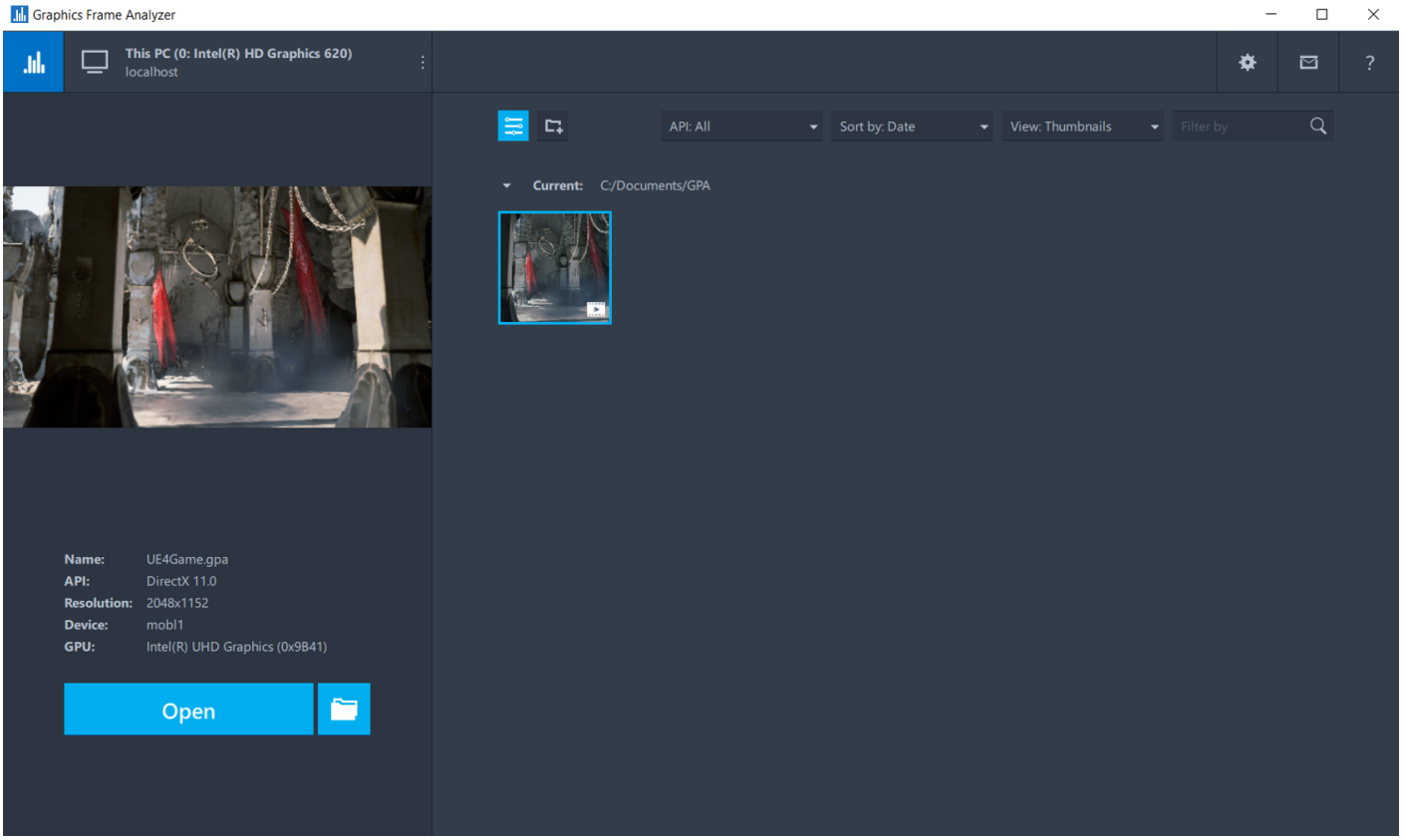
Open Frame Streams in Graphics Frame Analyzer

If you want to start Intel GPA immediately after the capture completes, this can be configured in the plugin settings.

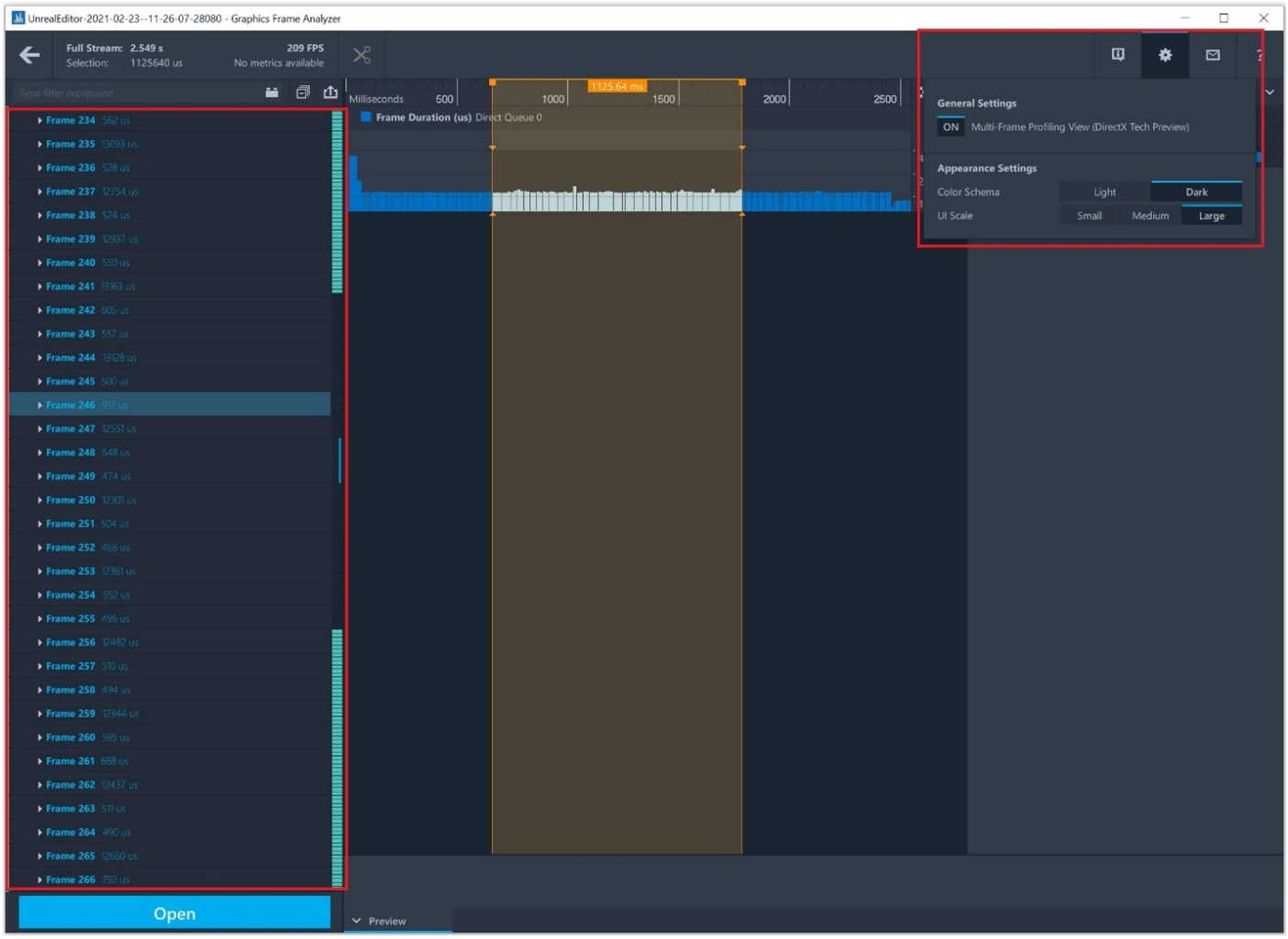
1. Go to **Settings->Project Settings.**
2. Scroll down to the **Plugins** and select **GPA.**
3. Enable the option to run Intel GPA after the capture completes.



In other cases, you can open multi-frame streams through Intel GPA Graphics Frame Analyzer. The picture below shows the case when only one stream has been captured during a session with Unreal Editor. If multiple captures have been completed, each capture is shown as a separate thumbnail. They all are ready for analysis.



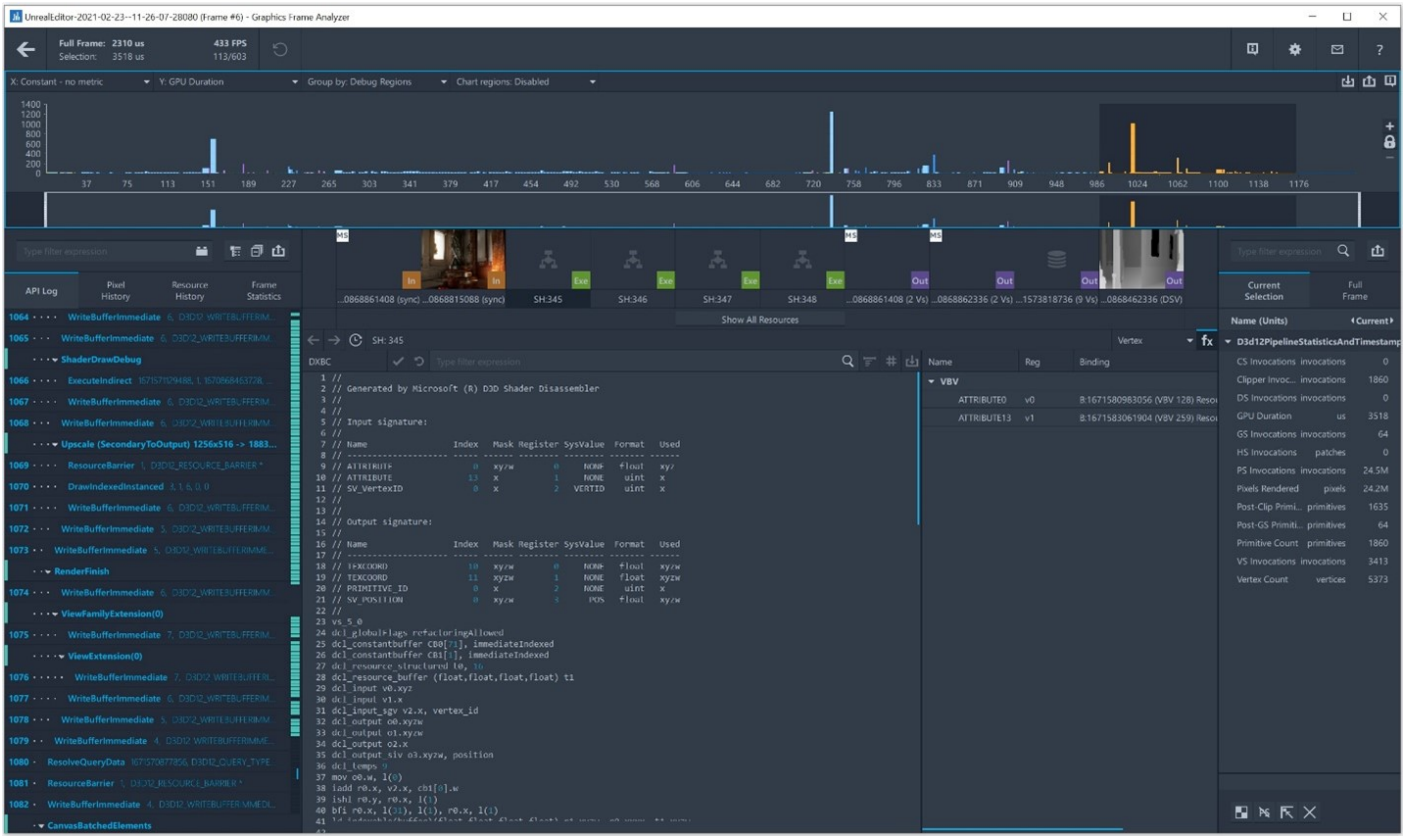
Double-click the stream or choose the stream you want to analyze and click **Open**to open it in Graphics Frame Analyzer. There you will see a list of captured frames.



When analyzing streams with ray tracing enabled, make sure that 'Multi-Frame Profiling View' is turned on in General Settings.

Analyze Frames in Graphics Frame Analyzer

To go one level deeper and analyze an individual frame, double-click a frame or choose the frame you want to analyze and click **Open.** This will open a new view that enables you to analyze each rendering command, along with the associated resources such as shaders, stage input and output textures, and buffers.



To learn more about stream analysis, frame analysis, potential insights, and next steps, refer to Analyze GPU-bound Applications with Graphics Frame Analyzer.

NOTE In some cases, you may need to track application execution on the CPU and GPU to identify whether the bottleneck is on the CPU or GPU side. In this case, before starting in-depth frame analysis, capture a *trace* from your application (a number of seconds of execution) and analyze it with Intel GPA Graphics Trace Analyzer. Note that the Intel GPA plugin for Unreal Engine does not support capturing traces directly from Unreal Editor, so you need to capture a trace with Intel GPA Graphics Monitor using a general procedure.
If you figure out that the app is GPU-bound from your trace analysis, follow the workflow above. Otherwise, you can continue in-depth analysis with Graphics Trace Analyzer: analyze graphics API calls, review user-defined debug markers, thread activity, queued GPU commands and/or perform a high-level analysis of synchronization and parallelism efficiency.

Instrumentation and Tracing Technology API Support

NOTE
The Instrumentation and Tracing Technology API (ITT API) and the Just-in-Time Profiling API (JIT API) are open source components. Visit the GitHub* repository to access source code and contribute.

The Instrumentation and Tracing Technology API (ITT API) provided by the Intel GPA enables your application to generate and control the collection of trace data during its execution.

ITT API has the following features:

- Controls application performance overhead based on the amount of traces that you collect.
- Enables trace collection without recompiling your application.
- Supports applications in C/C++ environments.
- Automatically traces data from Microsoft DirectX* applications.
- Automatically traces data from OpenGL* ES v1, v2, and v3 APIs on Android* OS applications.
- Supports instrumentation for tracing application code.

This chapter describes how to add basic instrumentation to your application, and also includes examples of several advanced instrumentation topics.

User applications/modules linked to the static ITT API library do not have a runtime dependency on a dynamic library. Therefore, they can be executed without Intel GPA or other Intel tools.

ITT API enables you to manually instrument your application for further application performance analysis.

You can use the Intel GPA Trace Analyzer to view ITT trace data. The Intel GPA Trace Analyzer obtains this data from your application through the Intel GPA Monitor.

NOTE
The ITT API is a set of pure C/C++ functions. There are no Java* or .NET* APIs. If you need runtime environment support, you can use a JNI, or C/C++ function call from the managed code. If the collector causes significant overhead or data storage, you can pause the analysis to reduce the overhead.

See Also
Compile and Link with ITT API

Compile and Link with ITT API

Before instrumenting your application, configure your build system to be able to reach the API headers and libraries:

- Add `<gpa_install_dir>\sdk\include` to your `INCLUDE` path
- Add `<gpa_install_dir>\sdk\lib\x86` to your 32-bit `LIBRARIES` path
- Add `<gpa_install_dir>\sdk\lib\x64` to your 64-bit `LIBRARIES` path

The default installation directory on Windows* OS is `C:\Program Files\IntelSWTools\GPA`.

Include the ITT API Header in Your Application

Add the following `#include` statements to every source file that you want to instrument:

```
#include <ittnotify.h>
```

The `ittnotify.h` header contains definitions of ITT API routines and important macros which provide the correct logic of API invocation from a user application.

NOTE
The `ittnotify.h` header defines many functions that are not covered in this document. However, Intel® GPA supports only the functions that are documented herein.

The ITT API is designed to incur almost zero overhead when tracing is disabled. But if you need fully zero overhead, you can compile out all ITT API calls from your application by defining the INTEL_NO_ITTNOTIFY_API macro in your project at compile time, either on the compiler command line, or in your source file, prior to including the `ittnotify.h` file.

Link the `libittnotify.lib` Static Library to Your Application

You need to link the static library, `libittnotify.lib`, to your application. If tracing is enabled, this static library loads the ITT API data collector and forwards ITT API instrumentation data to Intel GPA.If tracing is disabled, the static library ignores ITT API calls, providing nearly zero instrumentation overhead.

NOTE
If you are running Intel® GPA on a system with a legacy version of Microsoft* Visual Studio*, you must have the corresponding security updates installed to link against the dynamic version of the `libittnotify.lib` static library:

Microsoft* Visual Studio* Version	Security Update
Microsoft Visual Studio 2005	KB2538218
Microsoft Visual Studio 2008	KB2538241

If you do not want to update your version of Visual Studio, you must link the `libittnotify.lib` static library to your application.

UNICODE Support

All ITT API functions with string arguments follow the Windows OS UNICODE convention, the actual function names are suffixed with A for the ASCII API and W for the UNICODE API. For example:

```
__itt_domain_createA( const char* );
__itt_domain_createW( const wchar_t* );
```

To facilitate cross-platform code development, the ITT API features the C macro `__itt_domain_create`. If a UNICODE macro is defined when compiling on a Windows OS, the C macro resolves into `__itt_domain_createW`, otherwise it resolves into `__itt_domain_createA`.

Conditional Compilation

For best performance in the release version of your code, use conditional compilation to turn off annotations. Define the macro INTEL_NO_ITTNOTIFY_API before you include `ittnotify.h` during compilation to eliminate all `__itt_*` functions from your code.

You can also remove the static library from the linking stage by defining this macro.

Instrument Your Application

Graphics Trace Analyzer is an instrumentation-based tool. It obtains its data from gpa_trace files, which are generated by Intel® GPA while profiling an application that has been instrumented with Instrumentation and Tracing Technology API (ITT API) calls.

To get the most out of the ITT APIs, you need to add API calls in your code to designate logical tasks. This will help you visualize the relationship between tasks in your code, including when they start and end, relative to other CPU and GPU tasks.

At the highest level a task is a logical group of work executing on a specific thread, and may correspond to any grouping of code within your program that you consider important. You can mark up your code by identifying the beginning and end of each logical execution chunk.

To resolve the majority of performance bottlenecks, the following API calls are enough

- `__itt_domain_create()` creates a domain required in most ITT API calls. You need to define at least one domain.

- `__itt_string_handle_create()` creates string handles for identifying your tasks. String handles are more efficient for identifying traces than strings.
- `__itt_task_begin()` marks the beginning of a task.

Example

The following sample shows how four basic ITT API functions are used in a multi threaded application:

- Domain API
- String Handle API
- Task API
- Thread Naming API

```
#include <windows.h>
#include <ittnotify.h>

// Forward declaration of a thread function.
DWORD WINAPI workerthread(LPVOID);
bool g_done = false;
// Create a domain that is visible globally: we will use it in our example.
__itt_domain* domain = __itt_domain_create("Example.Domain.Global");
// Create string handles which associates with the "main" task.
__itt_string_handle* handle_main = __itt_string_handle_create("main");
__itt_string_handle* handle_createthread = __itt_string_handle_create("CreateThread");
void main(int, char* argv[])
{
// Create a task associated with the "main" routine.
__itt_task_begin(domain, __itt_null, __itt_null, handle_main);
// Now we'll create 4 worker threads
for (int i = 0; i < 4; i++)
{
// We might be curious about the cost of CreateThread. We add tracing to do the measurement.
__itt_task_begin(domain, __itt_null, __itt_null, handle_createthread);
::CreateThread(NULL, 0, workerthread, (LPVOID)i, 0, NULL);
__itt_task_end(domain);
}

// Wait a while,...
::Sleep(5000);
g_done = true;
// Mark the end of the main task
__itt_task_end(domain);
}
// Create string handle for the work task.
__itt_string_handle* handle_work = __itt_string_handle_create("work");
DWORD WINAPI workerthread(LPVOID data)
{
// Set the name of this thread so it shows  up in the UI as something meaningful
char threadname[32];
wsprintf(threadname, "Worker Thread %d", data);
__itt_thread_set_name(threadname);
// Each worker thread does some number of "work" tasks
while(!g_done)
{
__itt_task_begin(domain, __itt_null, __itt_null, handle_work);
::Sleep(150);
__itt_task_end(domain);
}
```

```
return 0;
}
```

See Also

Domain API
String Handle API
Task API

Minimize ITT API Overhead

The ITT API overhead and its impact on the overall application performance depends on the amount of instrumentation code added to the application. When instrumenting an application with ITT API, you should balance between application performance and the amount of performance data that you need to collect, in order to minimize API overhead while collecting sufficient performance data.

Follow these guidelines to achieve good balance between overall performance of the instrumented application and instrumentation detail:

- Instrument only those paths within your application that are important for analysis.
- Create ITT domains and string handles outside the critical paths.
- Filter data collection by different aspects of your application that can be analyzed separately. The overhead for a disabled API call (thus filtering out the associated call) is always less than 10 clock ticks, regardless of the API.

Usage Example: Using Domains and String Handles

The ITT APIs include a subset of functions which create domains and string handles. These functions always return identical handles for the same domain names and strings. This requires these functions to perform string comparisons and table lookups, which can incur serious performance penalties. In addition, the performance of these functions is proportional to the log of the number of created domains or string handles. It is best to create domains and string handles at global scope, or during application startup.

The following code section creates two domains in the global scope. You can use these domains to control the level of detail that is written to the trace file.

```
__itt_domain* basic = __itt_domain_create(L"MyFunction.Basic");
__itt_domain* detailed = __itt_domain_create(L"MyFunction.Detailed");
// Create string handles at global scope.
__itt_string_handle* h_my_funcion = __itt_string_handle_create(L"MyFunction");
void MyFunction(int arg)
{
    __itt_task_begin(basic, __itt_null, __itt_null, h_my_function);
    Foo(arg);
    FooEx();
    __itt_task_end(basic);
}
__itt_string_handle* h_foo = __itt_string_handle_create(L"Foo");
void Foo(int arg)
{
    // Skip tracing detailed data if the detailed domain is disabled.
    __itt_task_begin(detailed, __itt_null, __itt_null, h_foo);
    // Do some work here...
    __itt_task_end(detailed);
}
__itt_string_handle* h_foo_ex = __itt_string_handle_create(L"FooEx");
void FooEx()
{
```

```
      // Skip tracing detailed data if the detailed domain is disabled.
      __itt_task_begin(detailed, __itt_null, __itt_null, h_foo_ex);
      // Do some work here...
      __itt_task_end(detailed);
  }
  // This is my entry point.
  int main(int argc, char** argv)
  {
    if(argc < 2)
        //Disable detailed domain if we do not need tracing from that in this
        //application run
detailed ->flags = 0;
      MyFunction(atoi(argv[1])); }
```

See Also

Instrument Your Application

Clock Domain API

Some applications require the capability to trace events with user-defined timestamps and frequencies different from the timestamps and frequencies generated by a CPU, for example, to instrument events that occur on a GPU. For the purpose you can create a clock domain.

To create a clock domain, use the following primitive:

```
__itt_clock_domain * ITTAPI __itt_clock_domain_create(__itt_get_clock_info_fn fn, void*
fn_data)
```

Parameters of the primitive:

[in]	fn	Pointer to a callback function that retrieves alternative CPU timestamps and frequencies and stores the timestamps and frequencies data in the clock domain structure field `__itt_clock_info`.
[in]	fn_data	Argument of a callback function; can be NULL.

Tasks issued from different clock domains are shown on the same timeline by synchronizing a referenced clock domain base timestamp captured at the moment of clock domain creation and a CPU timestamp captured at the same time.

To recalculate clock domain base timestamps and frequencies, if necessary, for example, when a GPU frequency changes, use the following primitive:

```
__itt_clock_domain_reset()
```

Counter API

A Counter is a user-defined characteristic or metric of hardware or software behavior that can be used to collect information about and correlate with execution breakdown, such as tasks, events, and markers. For example, system on a chip (SoC) development benefits from several counters representing different parts of the SoC to count some hardware characteristics.

Define and create a counter object

Use these primitives:

```
__itt_counter ITTAPI __itt_counter_create_typedA __itt_counter_create_typedA(const char
*name, const char *domain, __itt_metadata_type type)
```

```
__itt_counter ITTAPI __itt_counter_create_typedW __itt_counter_create_typedW(const
wchar_t *name, const wchar_t *domain, __itt_metadata_type type)
```

```
__itt_counter_create_v3(__itt_domain* domain, const char* name,__itt_metadata_type
type);
```

Parameters of the primitives:

[in]	`domain`	Counter domain
[in]	`name`	Counter name
[in]	`type`	Counter type

Use these primitives:

Directly set the counter value

Use:

```
__itt_counter_set_value(__itt_counter id, void *value_ptr);
```

```
__itt_counter_set_value_v3(__itt_counter counter, void *value_ptr);
```

Parameters of the primitive:

[in]	`id`	Counter ID
[in]	`value_ptr`	Counter value

Usage Example

This example creates a counter that measures temperature and memory usage metrics:

```
#include "ittnotify.h"

__itt_counter temperatureCounter = __itt_counter_create("Temperature", "Domain");
__itt_counter memoryUsageCounter = __itt_counter_create("Memory Usage", "Domain");
unsigned __int64 temperature;

while (...)
{
    ...
    temperature = getTemperature();
    __itt_counter_set_value(temperatureCounter, &temperature);

    __itt_counter_inc_delta(memoryUsageCounter, getAllocatedMemSize());
    __itt_counter_dec_delta(memoryUsageCounter, getDeallocatedMemSize());
    ...
}

__itt_counter_destroy(temperatureCounter);
__itt_counter_destroy(memoryUsageCounter);
```

Domain API

A *`domain`* enables tagging trace data for different modules or libraries in a program. Domains are specified by unique character strings, for example `TBB.Internal.Control`.

Each domain is represented by an opaque `__itt_domain` structure, which you can use to tag each of the ITT API calls in your code.

You can selectively enable or disable specific domains in your application, in order to filter the subsets of instrumentation that are collected into the output trace capture file. Upon the first start, the application registers its domains in a configuration file of the Intel® GPA Graphics Monitor. By default, upon the first start of the application all the domains are enabled. To enable or disable domains for subsequent starts of the application, do the following before the start:

1. Open the Graphics Monitor Launcher Screen.
2. Click the **Options** button.
3. Click the **Tracing** tab.
4. Turn on or off the **User Domains** toggle buttons.

The overhead of a disabled domain is a single `if` check.

To create a domain, use the following primitives:

```
__itt_domain *ITTAPI__itt_domain_createA ( const char *name)
```

```
__itt_domain *ITTAPI __itt_domain_createW ( const wchar_t *name)
```

Or use the macro in the following way:

```
__itt_domain *ITTAPI __itt_domain_create (__TEXT("TBB.Internal.Control"))
```

For a domain name, the URI naming style is recommended, for example, `com.my_company.my_application`. The set of domains is expected to be static over the application's execution time, therefore, there is no mechanism to destroy a domain.

Any domain can be accessed by any thread in the process, regardless of which thread created the domain. This call is thread-safe.

Parameters of the primitives:

`[in]`	*name*	Name of domain

Usage Example

```
#include "ittnotify.h"
__itt_domain* pD = __itt_domain_create(__TEXT("My Domain") );
```

See Also

Instrument Your Application

Minimize ITT API Overhead

Marker API

Marker is an instant event on a timeline that can be associated with a particular process, a thread, or specified in a global scope.

To create a marker, use the following primitive:

```
void __itt_marker(const __itt_domain *domain, __itt_id id, __itt_string_handle *name,
__itt_scope scope)
```

Parameters of the primitive:

[in]	`domain`	Marker domain
[in]	`name`	Marker name
[in]	`id`	Optional parameter. Marker ID, or `__itt_nullNOTEMarkers` with different domains cannot have same IDs.
[in]	`scope`	Marker scope: process, thread, and global

Metadata API

Metadata is extra information, generic data which can be attached to a task, a thread, a process, etc. Metadata has a type, name, and value. The value encoding depends on a metadata type. It may contain either string data or a number of integer or floating point values.

To create metadata, use the following primitives:

```
void __itt_metadata_add(const __itt_domain *domain, __itt_id id, __itt_string_handle
*key, __itt_metadata_type type, size_t count, void *data)

void __itt_metadata_str_addA(const __itt_domain *domain, __itt_id id,
__itt_string_handle *key, const char *data, size_t length)

void __itt_metadata_str_addW(const __itt_domain *domain, __itt_id id,
__itt_string_handle *key, const wchar_t *data, size_t length)

void __itt_metadata_add_with_scope(const __itt_domain *domain, __itt_scope scope,
__itt_string_handle *key, __itt_metadata_type type, size_t count, void *data)

void __itt_metadata_str_add_with_scopeA(const __itt_domain *domain, __itt_scope scope,
__itt_string_handle *key, const char *data, size_t length)

void __itt_metadata_str_add_with_scopeW(const __itt_domain *domain, __itt_scope scope,
__itt_string_handle *key, const wchar_t *data, size_t length)
```

The following table defines the parameters used in the Metadata API primitives.

Type	Parameter	Description
[in]	`__itt_domain* domain`	Metadata domain
[in]	`__itt_scope scope`	Metadata scope: task, thread, process, and global. If a scope is undefined, metadata belongs to the last task in the thread.
[in]	`__itt_string_handle* name`	Metadata name
[in]	`__itt_metadata_type type`	Metadata type; used only for numeric metadata
[in]	`size_t count`	Number of numeric metadata items `[in] size_t length`
[in]	`size_t length`	Number of symbols a metadata string

[in]	`void *data`
	`const char *data`
	`const wchar_t *data`
		Actual metadata (array of numerics or string)

Relation API

Relation API binds two named instances, for example, tasks, with a reasonable relation attribute. Relations can be added before or after the actual instances are created and persist independently of the instances.

To logically group a bunch of tasks, you can use different types of relations:

```
void ITTAPI __itt_relation_add(const __itt_domain *domain, __itt_id head,
__itt_relation relation, __itt_id tail)

void ITTAPI __itt_relation_add_ex(const __itt_domain *domain, __itt_clock_domain*
clock_domain, unsigned long long timestamp, __itt_id head, __itt_relation relation,
__itt_id tail)
```

Parameters of the primitives:

Type	Parameter	Description
[in]	`__itt_domain* domain`	Relation domain
[in]	`__itt_relation relation`	User-defined logical relation between two named instances
[in]	`__itt_id head`	Metadata name
[in]	`__itt_id tail`	IDs of two named related instances `size_t` count
[in]	`__itt_clock_domain * clock_domain`	User-defined clock domain

Type	Parameter	Description
[in]	unsigned long long timestamp	User-defined timestamp for the corresponding clock domain

String Handle API

Many API calls require names to identify API objects. String handles are pointers to names. They enable efficient handling of named objects in run time and make collected traces data more compact.

To create and return a handle value that can be associated with a string, use the following primitive:

```
__itt_string_handle *ITTAPI __itt_string_handle_createA ( const char *name)
```

```
__itt_string_handle *ITTAPI __itt_string_handle_createW ( const wchar_t *name)
```

Or use the macro in the following way:

```
__itt_string_handle *ITTAPI __itt_string_handle_create (__TEXT("My Task"))
```

Consecutive calls to `__itt_string_handle_create` with the same name return the same value. The set of string handles is expected to remain static during the application's execution time, therefore, there is no mechanism to destroy a string handle. Any string handle can be accessed by any thread in the process, regardless of which thread created the string handle. This call is thread-safe.

Parameters of the primitive:

[in]	*name*	The input string

See Also

Minimize ITT API Overhead

Thread Naming API

By default, each thread in your application is displayed in the **timeline** track with a default label generated from the process ID and the thread ID, or with the OS thread name. You can use the Thread Naming API in your code to give threads meaningful names.

To set thread name using a char or Unicode string, use the primitive:

```
void __itt_thread_set_nameA (const __itt_char *name)
```

```
void __itt_thread_set_nameW (const wchar_t *name)
```

Or use the macro in the following way:

```
void__itt_thread_set_name(__TEXT("Main"))
```

Parameters of the primitive:

[in]	*name*	The thread name

If the thread name is set multiple times, only the last name is used.

Usage Example

You can use the following thread naming example to give a meaningful name to the thread you wish to focus on and ignore the service thread.

```
DWORD WINAPI service_thread(LPVOID lpArg)

{
  __itt_thread_set_name(__TEXT("My worker thread"));
  // Do thread work here
  return 0;
```

```
}

int main(int argc, char* argv[])
{
  ...
  CreateThread(NULL, 0, service_thread, NULL, 0, NULL);
  CreateThread(NULL, 0, thread_function, NULL, 0, NULL);
  ...
  return 0;
}
```

Task API

A *task* is a logical unit of work performed by a particular thread. Tasks can nest; thus, tasks typically correspond to functions, scopes, or a case block in a switch statement. You can use the Task API to assign tasks to threads.

The Task API does not enable a thread to suspend the current task and switch to a different task (task switching), or move a task to a different thread (task stealing).

A task instance represents a piece of work performed by a particular thread for a period of time. The task is defined by the bracketing of `__itt_task_begin()` and `__itt_task_end()` on the same thread.

Tasks can be of two types: simple or overlapped.

Simple tasks implicitly support a concept of embedded execution. This means that the call __itt_task_end() finalizes the most recent __itt_task_begin() call. For example, the following metacode is a valid sequence, and execution time of "a" tasks incorporates the execution time of "b" tasks:

```
__itt_task_begin(a)
__itt_task_begin(b)
__itt_task_end(b)
__itt_task_end(a)
```

Execution regions of overlapped tasks may intercept. For example, the following metacode is a valid sequence, and a "b" task started after an "a" task can finish upon completion of the "a" task:

```
__itt_task_begin_overlapped(a)
__itt_task_begin_overlapped(b)
__itt_task_end_overlapped(a)
__itt_task_end_overlapped(b)
```

Task API Functions

To create a simple task instance on a thread, use the following functions:

```
void ITTAPI __itt_task_begin (const __itt_domain *domain,__itt_id taskid, __itt_id
parentid, __itt_string_handle *name)
```

```
void ITTAPI __itt_task_begin_fn (const __itt_domain *domain,__itt_id taskid, __itt_id
parentid, void* address)
```

```
void ITTAPI __itt_task_end (const __itt_domain *domain)
```

To create a simple task instance in a different clock domain, use the following functions:

```
void ITTAPI __itt_task_begin_ex(const __itt_domain* domain, __itt_clock_domain*
clock_domain, unsigned long long timestamp, __itt_id taskid, __itt_id parentid,
__itt_string_handle* name)
```

```
void ITTAPI __itt_task_begin_fn_ex(const __itt_domain* domain, __itt_clock_domain*
clock_domain, unsigned long long timestamp, __itt_id taskid, __itt_id parentid, void*
fn)
```

```
void ITTAPI _itt_task_end_ex(const __itt_domain* domain, __itt_clock_domain*
clock_domain, unsigned long long timestamp)
```

To create an overlapped task instance on a thread, use the following functions:

```
void ITTAPI __itt_task_begin_overlapped(const __itt_domain* domain, __itt_id taskid,
__itt_id parentid, __itt_string_handle* name)
```

```
void ITTAPI __itt_task_end_overlapped(const __itt_domain *domain, __itt_id taskid)
```

The argument `taskid` is mandatory for these functions.

To create an overlapped task instance in a different clock domain, use the following functions:

```
void ITTAPI __itt_task_begin_overlapped_ex(const __itt_domain* domain,
__itt_clock_domain* clock_domain, unsigned long long timestamp, __itt_id taskid,
__itt_id parentid, __itt_string_handle* name)
```

```
void ITTAPI __itt_task_end_overlapped_ex(const __itt_domain* domain,
__itt_clock_domain* clock_domain, unsigned long long timestamp, __itt_id taskid)
```

The argument `taskid` is mandatory for these functions.

ITTAPI__itt_task_* Function Parameters

The following table defines the parameters used in the Task API primitives.

Type	Parameter	Description
[in]	__itt_domain	The domain of the task.
[in]	__itt_id taskid	User-defined ID optional for all task instances, except for overlapped task instances. __itt_null can be used as default for undefined task instances. Task ID is used to define relations between task instances. **NOTE** Tasks with different domains cannot have same IDs.
[in]	__itt_id parentid	Optional parameter. Parent instance ID, to which the task belongs, or __itt_null. **NOTE** Parents with different domains cannot have same IDs.
[in]	__itt_string_handle	The task string handle.
[in]	void* fn	Function address that can be used instead of the name. For example, the function address can be resolved into the function name by using debug symbol information.
[in]	__itt_clock_domain	User-defined clock domain.

Type	Parameter	Description
[in]	unsigned long long timestamp	User-defined timestamp for the corresponding clock domain.

Usage Example

The following code snippet creates a domain and a couple of tasks at global scope.

```
#include "ittnotify.h"

void do_foo(double seconds);

__itt_domain* domain = __itt_domain_create(__TEXT("MyTraces.MyDomain"));
__itt_string_handle* shMyTask = __itt_string_handle_create(__TEXT("My Task");
__itt_string_handle* shMySubtask = __itt_string_handle_create(__TEXT("My SubTask"));

void BeginFrame() {
     __itt_task_begin(domain, __itt_null, __itt_null, shMyTask);
     do_foo(1);
}

void DoWork() {
     __itt_task_begin(domain, __itt_null, __itt_null, shMySubtask);
     do_foo(1);
     __itt_task_end(domain);
}
void EndFrame() {
     do_foo(1);
     __itt_task_end(domain);
}

int main() {
    BeginFrame();
    DoWork();
    EndFrame();
    return 0;
}
```

See Also

Minimize ITT API Overhead

Intel® GPA Reference

This section contains the following subsections:

- Performance Metrics Reference describes the metrics that you can analyze using Intel® GPA:
  - CPU metrics
  - GPU metrics
  - Graphics API metrics
  - Memory metrics
  - Device I/O metrics
- User Interface Reference describes the graphical user interface controls of the Intel® GPA tools:
  - Graphics Monitor

- Heads-Up Display (HUD)
- System Analyzer
- Graphics Frame Analyzer
- Graphics Trace Analyzer

Performance Metrics Reference

This section lists the metrics that you can analyze using Intel® GPA.

Using these metrics within Intel GPA enables you to "dig deep" into exactly where your game is spending its time within the rendering pipeline on the GPU. Intel GPA helps improve the playability of your game, either by increasing the frame rate or enabling you to add new visual effects.

Depending on your host system, you can get access to different groups of metrics:

- CPU metrics
- GPU metrics
- Graphics API metrics
- Memory metrics
- Device I/O metrics

Graphics Frame Analyzer provides full set of GPU metrics.

System Analyzer shows CPU, Memory, Power, Media, Graphics API (DirectX* or OpenGL*), and basic GPU metrics. The same metric set is available for the System Analyzer System View except metrics for a target application.

Heads-Up Display and Graphics Trace Analyzer show the selected subset of metrics available in System Analyzer.

For best results when using these metrics, make sure that you are familiar with both the rendering pipeline of Microsoft* DirectX* and the Intel Integrated Graphics Developer's Guide.

NOTE

- For the list of supported platforms, see Intel GPA Release Notes.
- Some of the metrics described in this chapter may not be available on your particular system configuration.

CPU Metrics

This section describes all CPU metrics accessible from the Intel® GPA.

Metric Name	Description
Aggregated CPU Load	Represents the percentage of time that all cores of the CPU are being utilized by any application on the device. If the **Aggregated CPU Load** metric is high but the **Target App CPU Load** metric is low, check whether another higher priority process on the device is taking time away from your application.
Application Time	Represents the time elapsed from the application start in microseconds.
CPU n Load	Represents the CPU utilization metric of the target system reported per CPU for all processes. There is one metric per each core on the system: CPU 0 Utilization is the core utilization of CPU 0, CPU 1 Utilization is the core utilization of CPU 1, and so on.

Metric Name	Description
CPU n Frequency	Represents the CPU frequency metric of the target system reported per CPU for all processes. There is one metric per each core on the system: CPU 0 Frequency is the core frequency of CPU 0, CPU 1 Frequency is the core frequency of CPU 1, and so on. The CPU Frequency metric is dynamically scaled by the operating system and usually increases as the result of high CPU load. This leads to higher power consumption and lower battery life. Try to optimize the CPU workload of your graphic application while running with a maximum FPS in V-Sync mode to minimize the working CPU frequency.
Target App CPU Load	Represents the aggregated (all cores) CPU utilization of the target application.

Device I/O Metrics

Device Input/Output metrics help you quantify your device's network and disk load.

The table below lists all Device I/O metrics that you can analyze with Intel® GPA.

Metric Name	Description
Disk Read	Indicates the rate at which data is read from internal storage or SD card. The metric includes data read by any application on the device, not just the target application.
Disk Write	Indicates the rate at which data is written to internal storage or SD card. The metric includes data written by any application on the device, not just the target application.
Network RX	Represents the rate at which data is received from the network interfaces. The metric includes data received by any application on the device, not just the target application. The loopback interface is not included.
Network TX	Represents the rate at which data is transmitted by the network interfaces. The metric includes data transmitted by any application on the device, not just the target application. The loopback interface is not included.

GPU Metrics

This section describes all the GPU metrics accessible from the Intel® GPA. The table below provides an overview of all GPU metrics available for Intel GPUs starting from the 3rd Generation Intel Core Processors.

NOTE

- Families of Intel® X^e graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® X^e Graphics.
- For products formerly named Kaby Lake G, see GPU metrics description at https://gpuperfapi.readthedocs.io/en/latest/counters.html.
- For DirectX* 11 targets, metrics are collected for a given application being profiled. For DirectX 12 targets, metrics are collected system-wide, including all running applications. While profiling a DirectX 12 application, it is recommended to stop all other running graphic applications.

Main Metrics

Metric Name	Description
GPU Duration	Represents the total GPU time for the frame, or for the selected event for Graphics Frame Analyzer within that frame. **Examples:** If GPU Duration is 80,000, it means that the GPU spends around 80 milliseconds to render the selected ergs. **Improving Performance:** When using GPU Duration as a metric to help analyze the performance of your game or application, it is important to understand the following: • If this value is too large, examine the underlying components of the rendering pipeline to see if one or more of these areas are too complex, resulting in potential performance bottlenecks. Check: Pixel Shader Duration , Vertex Shader Duration , Geometry Shader Duration metrics. • How effective is the GPU working for the selected ergs? Check: GPU EUs Active , GPU EUs Stalled .
GPU Frequency	Represents the average GPU core frequency during the measurement period. The latest Intel GPUs support the Intel® Turbo Boost Technology 2.0 and can dynamically change frequency depending on CPU and GPU workloads. **Examples:** For Intel® HD Graphics 3000, the GPU Frequency increases to its maximum frequency when a heavy GPU load occurs. **Improving Performance:** Typically the system automatically adjusts the GPU Frequency to optimize total system performance between the CPU and the GPU. When running the HUD, if the GPU frequency is always at its peak value for a particular system configuration, this could indicate that your system is GPU bound; if the GPU frequency is always at the lower end of the range, this could indicate that either you are CPU bound and/or that the GPU is not being fully utilized. When running the Graphics Frame Analyzer, currently this metric does not provide an accurate measure of GPU performance, since the CPU is not being utilized as it would be during the running of your game when the frame was captured. **NOTE** If the Intel graphics device supports multiple GPU frequencies, to minimize variation in metric values the Graphics Frame Analyzer locks the GPU at the maximum frequency available.
Avg GPU Core Frequency, MHz	Represents the average GPU Core Frequency in the measurement.
GPU Core Clocks	Represents the total number of GPU core clocks elapsed during the measurement period.
GPU Busy	Represents the percentage of time when the GPU is busy. **Examples**:

Metric Name	Description
	For GPU-bound workloads, the value of the GPU Busy metric is 100%. A value less than 100% indicates that the GPU is spending time in an idle state, waiting for data from the CPU, in which case your game or application might be CPU-bound. **Improving Performance:** If GPU Busy is consistently less than 100% and you are encountering performance issues, consider threading your game and using the Graphics Trace Analyzer to understand the interaction between the CPU and GPU.
HUD Overhead Time	Represents the Head's-up Display overhead time.
Non-Culled Polygons	Represents the number of polygons processed that were not culled.

GTI Metrics

Main Metric	Description
GTI Write Throughput	Represents the total number of GPU memory bytes written to GTI.
GTI Read Throughput	Represents the total number of GPU memory bytes read from GTI.
DRAM LLC Throughput, bytes	Represents the total number of successful LLC cache lookups done from the GPU.
LLC GPU Accesses, messages	Represents the approximate amount of GPU memory bytes transferred between LLC and DRAM controller. **NOTE** This metric might show incorrect results and will be disabled with the next driver update.
LLC GPU Throughput, bytes	Represents the total number of GPU memory bytes transferred between GPU and LLC.
LLC GPU Hits, messages	Represents the total number of LLC cache lookups done from the GPU (64B reads, 32B writes). **NOTE** This metric might show incorrect results and will be disabled with the next driver update.

EU Array Metrics

Metric Name	Description
EU Idle %	Represents the percentage of time when the GPU execution units (EUs) were idle. An EU is idle when it is neither actively executing shader instructions nor stalled trying to execute shader instructions. **Examples:** • If EU Idle % is 50, it means that the EUs were idle for 50% of the rendering time for selected ergs. • If EU Idle % is 0, it means that the EUs were either active or stalled for the entire duration of the rendering time for the selected erg. **Improving Performance:** If EU Idle % is significantly higher than 0%, this indicates that there are stalls elsewhere in the rendering pipeline.
EU Active %	Represents the percentage of time when the GPU execution units (EUs) were actively executing pixel, geometry, or vertex shader instructions. **Examples**: If EU Active % is 80, it means that the EUs were active 80% of the rendering time for the selected events. **Improving Performance:** If the EUs are not active, it means that they are either stalled waiting for a request to be fulfilled, or idle. You can see how much of the non-active time is caused by stalls by examining the EU Stall % metric. If the total EU busy time (EU Active % + EU Stall %) is significantly lower than 100%, this indicates that there are stalls elsewhere in the rendering pipeline.
EU Stall %	Represents the percentage of time when the GPU execution units (EUs) were stalled. An EU becomes stalled when all of its threads are waiting for results from fixed function units (for example, a pixel shader requests texels from the texture sampler). **Examples**: • If EU Stall % is 50, it means that EUs were stalled for 50% of the rendering time for the selected ergs. • If EU Stall % is 0, it means that there were no stalls in EUs or the stall time is very small. **Improving Performance** If this metric is unexpectedly high, especially when compared with the EU Active % metric, you can analyze where the stalls happen by looking at the **VS EU Stall %| GS EU Stall % | PS EU Stall %** metrics. If any of these metrics show that most of the stall time is in one particular shader, examine your shader code in the Graphics Frame Analyzer to determine why this shader might be causing the EUs to stall.
EU AVG IPC Rate, Number	Represents the average rate of IPC calculated for two FPU pipelines. **NOTE** This metric might show incorrect results and will be disabled with the next driver update.

Metric Name	Description
VS Duration	Represents an approximation of the total GPU time spent executing vertex shader code. **Examples:** • If Vertex Shader Duration is 50,000, it means that GPU spends around 50 milliseconds to execute vertex shaders for selected ergs. • If Vertex Shader Duration is 0, it means that time spent in vertex shaders for selected ergs is very small. **Improving Performance:** If the Vertex Shader Duration time is significant compared to GPU Duration , vertex processing optimizations might be needed. In this situation, optimize the geometry by minimizing the Vertex Count , Primitive Count , and Vertex Shader Invocations Count . If you are using triangle lists, try to convert them to a single triangle strip to minimize the number of vertices sent to pipeline. Also optimize the geometry for VCache (see Vertex Shader Invocations Count metric description).To see whether optimizations are possible, examine your vertex shader code in the Graphics Frame Analyzer. Refer to the Graphics API Performance Guide to find recommendations for vertex shader optimizations.
VS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Vertex Shader instructions. **Examples**: • If VS EU Active % is 50%, half of the overall GPU time was spent actively executing Vertex Shader instructions. • If VS EU Active % is 0%, no Vertex Shader was associated with the selected draw calls, or that the amount of time actively executing Vertex Shader instructions was negligible. **Improving Performance:** • This metric is important if vertex processing seems to be a bottleneck for selected rendering calls. If VS EU Active % accounts for most of the EU active time, then to improve performance you should simplify the vertex shader or simplify and optimize the geometry of your primitives. • If VS EU Active % is significant, you should examine your vertex shader code to find reasons that might be causing stalls. Inspect the shader code in the Graphics Frame Analyzer.
VS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Vertex Shader instructions. **NOTE** This metric does not include the total amount of time stalled in the vertex shader, but only the fraction of the time when vertex shader stalls were causing the entire EU to stall. The entire EU stalls when all of its threads are stalled. **Examples:** • If VS EU Stall % is 50% it means that half of the overall GPU time was spent stalled on Vertex Shader instructions.

Metric Name	Description
	• If VS EU Stall % is 0% it means that no Vertex Shader was associated with selected rendering calls or Vertex Shader threads were not causing EUs stalls. **Improving Performance:** • This metric is important if vertex processing seems to be the bottleneck for selected rendering calls. If VS EU Stall % accounts for most of the EU active time, then to improve performance you may need to simplify the vertex shader or simplify and optimize geometry. • If VS EU Stall % is significant you need to concentrate on vertex shader code to find reasons causing stalls. Inspect the shader code in the Graphics Frame Analyzer.
VS Invocations	Represents the number of vertex shader invocations - the vertex shader is invoked once per vertex. The number of vertex shader invocations depends both on the vertex and primitive counts and the operation of the post-transform vertex cache (VCache). In an optimal situation the GPU fetches already-processed vertices from the cache rather than recalculating this data, which could impact the value of this metric. Therefore, when the VS Invocations and the Vertex Count have similar values, it means that the geometry is not optimized to take advantage of the VCache. **Examples:** The `OptimizedMesh` sample from the Microsoft* DirectX* SDK is a good example to illustrate the Vertex Count and VCache optimizations: • When rendering one un-optimized mesh as a triangle list, the Vertex Count is equal to 141K and the VS Invocations is 112K. • When rendering the same mesh as a triangle list that has been reordered for optimum VCache usage, the Vertex Count is still the same but the VS Invocations number drops to 27K, which is almost four times less. • When rendering the same mesh as a VCache-optimized triangle strip, the Vertex Count drops to 52K and the VS Invocations drops to 25K. **Improving Performance:** To improve vertex processing performance and reduce the number of vertex shader invocations, try to reorder the geometry for optimum VCache usage. The D3DX utility library contains functions that reorder the geometry to improve VCache utilization (ID3DXMesh::Optimize, ID3DXMesh::Optimize, D3DXOptimizeFaces, D3DXOptimizeVertices). **NOTE** • If you render point sprites, the metric is always equal to Vertex Count and Primitive Count (that is, no optimizations are necessary). • The size of the VCache varies for different GPU models, so you may see different metric values when using the same geometry on different hardware.
VS Send Pipe Active %	Represents the percentage of time in which EU send pipeline was actively processing a vertex shader instruction.

Metric Name	Description
VS FPU0 Pipe Active %	Represents the percentage of time in which EU FPU0 pipeline was actively processing a vertex shader instruction.
VS FPU1 Pipe Active %	Represents the percentage of time in which EU FPU1 pipeline was actively processing a vertex shader instruction.
HS Duration	Represents the total amount of time the GPU spent executing hull shader code. **Examples:** • If HS Duration is 50,000 it means that the GPU spent 50 milliseconds executing hull shader code for the selected ergs. • If HS Duration is 0, it means that either the time spent executing hull shader code was negligible, or there was no hull shader in use. The heading in this template is a special field for topic titles, so generally you do not need to edit it. **Improving Performance:** If the HS Duration is larger than you expect, you can examine your hull shader code in the Graphics Frame Analyzer to investigate possible optimizations.
HS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Hull Shader instructions.
HS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Hull Shader instructions. A shader thread will stall when it reaches an instruction that cannot complete until some time-consuming operation is completed. **NOTE** This metrics does not include the total amount of stalled time in the Hull Shader, but only the amount of time when the Hull Shader was causing the entire EU to stall. The EUs in the Intel® HD Graphics are hyperthreaded, which means that each EU can very quickly (within 2 clock cycles) switch from a stalled shader thread to another shader thread. Therefore, it is possible at any given time for a number of shader threads to be stalled on an EU, but for the EU to continue actively executing instructions on another shader thread. The entire EU is considered to be stalled only when all of its threads are stalled. **Improving Performance:** If a large amount of stall time seems to be occurring in a particular shader, then you should examine that shader to see whether you can reduce or eliminate some of the stalls. Short shaders might normally stall for a majority of their execution time, since in such situations instruction or data fetch (texels, constants) latency cannot be 'hidden'. If a large stall time occurs in longer shaders, it usually indicates inefficient shader execution and possible optimization opportunities. Inspect the shader code that was executed for a given draw call and experiment with optimizations in the Graphics Frame Analyzer.
HS Invocations	Represents the number of Hull Shader invocations. The Hull Shader is invoked once per patch. **Examples:**

Metric Name	Description
	The SimpleBezier11 sample from the Microsoft* DirectX* SDK is a good example to understand Hull Shaders. This sample renders a Mobius strip comprised of four patches with 64 control points per patch. Execution of this sample will result in an HS Invocations value of four. **Improving Performance:** The Hull Shader is not usually a performance bottleneck, but it can definitely cause performance issues further down the rendering pipeline. If the Hull Shader specifies large tessellation factors, or as the HS Invocations value increases, it will result in more work for the fixed function tessellator as well as an increased number of DS Invocations and GS Invocations .
DS Duration	Represents the total amount of time the GPU spent executing domain shader code. **Examples:** • If DS Duration is 50,000 it means that the GPU spent 50 milliseconds executing domain shader code for the selected ergs. • If DS Duration is 0, it means that either the time spent executing domain shader code was negligible, or there was no domain shader in use. **Improving Performance:** If DS Duration is larger than you expect, you can examine your domain shader code in the Graphics Frame Analyzer to investigate possible optimizations.
DS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Domain Shader instructions.
DS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Domain Shader instructions. A shader thread will stall when it reaches an instruction that cannot complete until some time-consuming operation is completed. **NOTE** This metrics does not include the total amount of stalled time in the Domain Shader, but only the amount of time when the Domain Shader was causing the entire EU to stall. The EUs in the Intel® HD Graphics are hyperthreaded, which means that each EU can very quickly (within 2 clock cycles) switch from a stalled shader thread to another shader thread. Therefore, it is possible at any given time for a number of shader threads to be stalled on an EU, but for the EU to continue actively executing instructions on another shader thread. The entire EU is considered to be stalled only when all of its threads are stalled. **Improving Performance** If a large amount of stall time seems to be occurring in a particular shader, then you should examine that shader to see whether you can reduce or eliminate some of the stalls. Short shaders might normally stall for a majority of their execution time, since in such situations instruction or data fetch (texels, constants) latency cannot be 'hidden'. If a large stall time occurs in longer shaders, it usually indicates inefficient shader execution and possible optimization opportunities. You can inspect the shader code that was executed for a given draw call and experiment with optimizations in Graphics Frame Analyzer.

Metric Name	Description
DS Invocations	Represents the number of Domain Shader invocations. The Domain Shader is invoked once per fixed function tessellator output point. **Examples:** The SimpleBezier11 sample from the Microsoft* DirectX* SDK is a good example to understand Domain Shaders. This sample renders a Mobius strip comprised of 4 patches with 64 control points per patch.Increasing the Patch Divisions slider increases the tessellation factors of the Hull Shader which results in and increased number of inputs into the Domain Shader. When the Patch Divisions slider is set to 4.0, the DS Invocations value will be 192. When the Patch Divisions slider is set to 5.0, the DS Invocations value will be 320. **Improving Performance:** The purpose of a Domain Shader is to calculate the vertex positions for subdivided points output by the fixed function tessellator. The best way to improve performance is to minimize the number of DS Invocations . This can be done by decreasing the amount of tessellation performed by either decreasing the number Hull Shader Invocations or decreasing the tessellation factors in the Hull Shader.
GS Duration	Represents the approximate total GPU time spent executing geometry shader code. **Examples:** • If GS Duration is 50,000, it means that GPU spends around 50 milliseconds to execute geometry shaders for selected ergs. • If GS Duration is 0, it means that time spent in geometry shaders for selected ergs is very small or no geometry shaders were associated with selected ergs. **Improving Performance:** If you are encountering performance issues and the GS Duration time is more than 20% to 40% of the total GPU Duration , geometry shader code optimizations may be needed.Examine geometry shader code in the Graphics Frame Analyzer to see if optimizations are possible.Refer to the Graphics API Performance Guide for recommendations on how to optimize the geometry shader.
GS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Geometry Shader instructions. **Examples:** • If GS EU Active % is 50% it means that half of the overall GPU time was spent actively executing geometry shader instructions. • If GS EU Active % is 0% it means that no geometry shader was associated with the selected draw calls, or that the amount of time actively executing geometry shader instructions was negligible. **Improving Performance:** • This metric is important if geometry shader seems to be the bottleneck for selected rendering calls. If GS EU Active % accounts for most of the EU active time, then to improve performance you may need to simplify the geometry shader or simplify and optimize the geometry of the scene. • If GS EU Active % is more than a nominal amount, you may need to examine your geometry shader code to find reasons for what might be causing these stalls. Inspect the shader code using the Graphics Frame Analyzer.

Metric Name	Description
GS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Geometry Shader instructions. **NOTE** This metric does not include the total amount of stalled time in the geometry shader but only the fraction of time when the geometry shader stalls were causing the entire EU to stall. The entire EU stalls when all of its threads are stalled. **Examples:** • If GS EU Stall % is 50%, it means that half of the overall GPU time was spent stalled on Geometry Shader instructions. • If GS EU Stall % is 0%, it means that no Geometry Shader was associated with selected rendering calls or Geometry Shader threads were not causing EUs stalls. **Improving Performance:** • This metric will be important if you think that geometry shader looks like the bottleneck for selected rendering calls. If GS EU Stall % accounts for most of the EU active time, then to improve performance you may need to simplify the geometry shader or simplify and optimize geometry. • If GS EU Stall % is more than a nominal amount, you may need to examine your geometry shader code to find reasons for what might be causing these stalls. Inspect the shader code using the Graphics Frame Analyzer.
GS Invocations	Represents the number of geometry shader invocations. The value is 0 if no geometry shader is associated with the rendering call. **NOTE** See Microsoft* DirectX* SDK for a description of the shader invocation count. **Examples:** If GS Invocations is 1000 it means that the geometry shader was invoked for 1000 primitives. **Improving Performance:** The only way to minimize the number of geometry shader invocations is to minimize the number of input primitives. The impact on rendering performance of reducing the invocation count is highly dependent upon your specific game or application.
Post-GS Primitives	Represents the number of primitives that flowed out of the geometry shader (GS), if enabled, to the clipper. This metric is important if a geometry shader was associated with the selected rendering calls, and even more important if the number of primitives spawned by geometry shader code is dynamic. **NOTE** If the GS was not enabled for the selected rendering calls, the metric returns a value of 0. **Examples:**

Metric Name	Description
	If Post-GS Primitives is 1000 and Primitive Count is 100, it means that 1000 primitives were constructed in the geometry shader from the original 100. **Improving Performance:** Analyze the geometry shader code using Graphics Frame Analyzer.
PS Duration	Represents an approximation of the total GPU time spent executing pixel shader code. **Examples:** • If Pixel Shader Duration is 50,000 it means that GPU spends around 50 milliseconds to execute pixel shaders for the selected ergs. • If Pixel Shader Duration is 0 it means that time spent in pixel shaders for selected ergs is very small. **Improving Performance:** Examine the Pixel Shader Duration time versus the GPU Duration ; when Pixel Shader Duration is high you may improve overall rendering performance by optimizing your pixel shader code.Refer to the Graphics API Performance Guide to find advice for pixel shader optimizations.
PS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Pixel Shader instructions. **Examples:** • IfPS EU Active %is 50% it means that half of the overall GPU time was spent actively executing Pixel Shader instructions. • If PS EU Active % is 0% it means that no Pixel Shader was associated with the selected draw calls, or that the amount of time actively executing Pixel Shader instructions was negligible. **Improving Performance:** • This metric is important if pixel shading seems to be the bottleneck for selected rendering calls. • If PS EU Active % accounts for most of the EU active time, then to improve performance you may need to simplify the pixel shader. • If PS EU Active % is larger than you would expect and you are encountering slow rendering times, you should examine the pixel shader code for potential reasons why these stalls may be occurring.
PS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Pixel Shader instructions. **NOTE** This metric does not show total amount of stalled time in the pixel shader, but only the fraction of time when pixel shader stalls caused the entire EU to stall. The entire EU stalls when all of its threads are stalled. **Examples:** • If PS EU Stall % is 50% it means that half of the overall GPU time was spent stalled on Pixel Shader instructions.

Metric Name	Description
	• If PS EU Stall % is 0% it means that no Pixel Shader was associated with selected rendering calls or Pixel Shader threads were not causing EUs stalls. **Improving Performance:** • This metric is important if pixel shading seems to be the bottleneck for selected rendering calls. If PS EU Stall % accounts for most the EU active time, then to improve performance you may need to simplify the pixel shader. • If PS EU Stall % is larger than you expect and you are encountering slow rendering times, you need to concentrate on pixel shader code to find reasons for these stalls.
PS Invocations	Represents the number of pixel shader invocations. The pixel shader is invoked once per pixel. **Examples:** If you render a quad with 8x8 pixels size that is located entirely within the viewing frustum, the Pixel Shader Invocation Count is 64. **Improving Performance:** Usually PS Invocations workloads are one of the most expensive in the rendering pipeline due to the processing time required within the pixel shader. Therefore, keeping the number of invocations as low as possible will likely improve your rendering performance. **NOTE** For Intel© microarchitecture code name Ivy Bridge and Bay Trail, this metric includes pixels rejected by Early-Depth test, even though the pixel shader was not actually invoked for these pixels.
PS Send Pipeline Active %	Represents the percentage of time in which EU send pipeline was actively processing a pixel shader instruction.
PS FPU0 Pipe Active %	Represents the percentage of time in which EU FPU0 pipeline was actively processing a pixel shader instruction.
PS FPU1 Pipe Active %	Represents the percentage of time in which EU FPU1 pipeline was actively processing a pixel shader instruction.
EU FPU0 Pipe Active %	Represents the percentage of time during which the EU FPU0 pipeline was actively processing.
EU FPU1 Pipe Active %	Represents the percentage of time during which the EU FPU1 pipeline was actively processing.
EU Both FPU Pipes Active %	Represents the percentage of time in which both EU FPU pipelines were actively processing.
EU Send Pipe Active %	Represents the percentage of time during which the EU Send pipeline was actively processing.
CS Duration	Represents the total amount of time the GPU spent executing compute shader code. **Examples:**

Metric Name	Description
	• If CS Duration is 50,000 it means that the GPU spent 50 milliseconds executing compute shader code for the selected ergs. • If CS Duration is 0 it means that either the time spent executing compute shader code was negligible, or there was no compute shader in use. **Improving Performance:** If CS Duration is larger than you expect, you can examine your compute shader code in the Graphics Frame Analyzer to investigate possible optimizations.
CS EU Active %	Represents the percentage of overall GPU time that the EUs were actively executing Compute Shader instructions. **Examples:** • If CS EU Active % is 0%, it means that no compute shader was associated with the selected draw calls, or that the amount of time actively executing compute shader instructions was negligible.
CS EU Stall %	Represents the percentage of overall GPU time that the EUs were stalled in Compute Shader instructions. A shader thread will stall when it reaches an instruction that cannot complete until some time-consuming operation is completed. **Examples:** • If CS EU Stall % is 0%, it means that no Compute shader was associated with the selected draw calls, or that the amount of time stalled on Compute shader instructions was negligible. **NOTE** This metric does not include the total amount of stalled time in the Compute Shader, but only the amount of time when the Compute Shader was causing the entire EU to stall. The EUs in the Intel© HD Graphics are hyperthreaded, which means that each EU can very quickly (within 2 clock cycles) switch from a stalled shader thread to another shader thread. Therefore, it is possible at any given time for a number of shader threads to be stalled on an EU, but for the EU to continue actively executing instructions on another shader thread. The entire EU is considered to be stalled only when all of its threads are stalled. **Improving Performance:** If a large amount of stall time seems to be occurring in a particular shader, then you should examine that shader to see whether you can reduce or eliminate some of the stalls.Short shaders might normally stall for a majority of their execution time, since in such situations instruction or data fetch (texels, constants) latency cannot be 'hidden'. If a large stall time occurs in longer shaders, it usually indicates inefficient shader execution and possible optimization opportunities. Inspect the shader code that was executed for a given draw call and experiment with optimizations in Graphics Frame Analyzer.
CS Invocations	Represents the number of compute shader invocations. The Compute Shader is invoked once per thread per thread group. The number of threads per thread group is defined by the Compute Shader's numthreads attribute (numthreads(tX, tY, tZ)).

Metric Name	Description
	The number of thread groups executed is determined by the parameters to the Dispatch call (Dispatch(gX, gY, gZ)). CS Invocations is equal to (gX*gY*gZ)*(tX*tY*tZ). **Examples:** • If the numthreads attribute is numthreads(4, 4, 1) and Dispatch is called as Dispatch(16, 16, 16), the CS Invocations value will be equal to (16*16*16)*(4*4*1) = 65536.

Sampler Metrics

Metric Name	Description
Sampler Busy %	Represents the percentage of time the texture sampler was busy handling texel fetch requests (that is, was either active or stalled). **NOTE** This metric is unreliable when protected HD media content is being played back on a system with Intel© HD Graphics 5000/ 4600 / 4400 / 4200, Intel© Iris© graphics 5100, or Intel© Iris© Pro graphics 5200 configuration. **Examples:** • If Sampler Busy % is 50, it means that texture sampler was active 50% of the rendering time for the selected ergs. • If Sampler Busy % is 0, it means that texture sampler was not used or the time during which it was active is very small. **Improving Performance:** When Sampler Busy % is running this might lead to execution unit stalls, especially if texture fetch latency does not occur in parallel with mathematical instructions (as the shader compiler attempts to optimize shader code to cover such latencies). Examine the EU Stall % metric to see the amount of EUs stalls. If the percentage is high and the Sampler Busy % is close to 100%, most likely you have a texturing bottleneck. Try the **2x2 textures** experiment in the Experiments pane in the Graphics Frame Analyzer to see if this is the case.
Sampler Texels, texels	Represents the number of texels returned from the texture sampler. **NOTE** This metric is unreliable when protected HD media content is being played back on a system with Intel© HD Graphics 5000/ 4600 / 4400 / 4200, Intel© Iris© graphics 5100, or Intel© Iris© Pro graphics 5200 configuration. **Examples:** If Sampler Texels, texels is 1000, it means that 1000 texels were delivered to the execution units (EUs) from the texture sampler. **Improving Performance:**

Metric Name	Description
	A high number of texels fetched from textures leads to a higher texture bandwidth and a higher number of texture sampler unit stalls, which might cause a high number of EU stalls caused by shaders awaiting texels from the sampler unit.Note that this metric could indicate that the shader stalls while fetching texture data inside branching logic. For example, if the shader fetches texture samples only inside an if() block in the code, this metric can help you understand how often the shader takes the branch. **NOTE** This metric is accurate only to four texels, and generally is slightly larger than the actual number of texels used. This is because the texture sampler returns data in 2x2 texel quads. When sampling along angular edges, this inaccuracy becomes more pronounced.
Sampler Cache Misses, messages	Represents the number of bytes of texture data read from memory by the GPU due to texture cache misses when rendering this frame. Note that the Texture Sampler reads data from memory in 64-byte blocks, so this metric can be used to calculate the number of texture cache misses as follows: **Examples:** • If Sampler Cache Misses, messages is 64000, it means the Texture Sampler missed the cache 1000 times and needed to read 64000 bytes of memory. • If Sampler Cache Misses, messages is 0, it means that no texture data was read from memory for the selected ergs. **NOTE** This metric is unreliable when protected HD media content is being played back on a system with Intel© HD Graphics 5000/ 4600 / 4400 / 4200, Intel© Iris© graphics 5100, or Intel© Iris© Pro graphics 5200 configuration. **Improving Performance:** Usually a higher value for this metric leads to a higher percentage of Texture Sampler stalls. Therefore, utilize techniques that minimize the number of texture reads, such as shown in the "Improving Performance" section of the Sampler Stalled metric.
Sampler Bottleneck %	Represents the percentage of time that the texture sampler is a bottleneck. The sampler is stalling Execution Units (EUs) due to a full input FIFO and starving EUs due to a lack of results. **NOTE** This metric is unreliable when protected HD media content is being played back on a system with Intel© HD Graphics 5000/ 4600 / 4400 / 4200, Intel© Iris© graphics 5100, or Intel© Iris© Pro graphics 5200 configuration. **Examples:**

Metric Name	Description
	If Sampler Bottleneck % is 90, then the texture sampler is a bottleneck (stalling some EUs and/or causing other EUs to idle) 90% of the time.Improving PerformanceThe following techniques may improve the texture sampler performance: • Reducing the size of textures, by using a lower resolution or lower color precision (such as RGBA4444 instead of RGBA8888) • Using texture compression to reduce the amount of memory to transfer textures • Using mipmapping, so that smaller textures (mipmaps) can be used • Reducing the number of textures in the scene • Using a different filtering algorithm For example, anisotropic filtering is more expensive to compute than a simpler algorithm, such as bilinear filtering. To help minimize overhead in this area, capture a typical frame while the game is running, use this frame as input to the Graphics Frame Analyzer, and try one or more of the following techniques: • the 2x2 Textures experiment in the Experiments tab to see if textures are a bottleneck • the Texture tab to see the texture size, format, and mip level **NOTE** This metric might show incorrect results and will be disabled with the next driver update.
Sampler Stalled	Represents the percentage of time the texture sampler was stalled. The texture sampler is stalled when its output queue is full, which can occur when it returns texture requests faster than the EUs can process them. When the texture sampler is stalled, it cannot process new requests. **Examples:** • If Sampler Stalled is 50%, it means that half of the time when texture sampler was busy it was waiting for space to open up in its output queue. • If Sampler Stalled is 0%, it means that texture sampler never stalled. **Improving Performance:** • Reduce the number of texture fetches in the shader code. • Reduce texture size and texture filtering setting under the Texture tab in the Graphics Frame Analyzer to see if this helps improve performance without adversely affecting image quality. • Minimize anisotropic filtering, because it requires a high number of additional texel fetches and is therefore "expensive" to use. • Modify the texture fetching pattern in the shader code to optimize texture cache utilization. To inspect shader code, see the Shaders tab in the Graphics Frame Analyzer.

3D Pipe Metrics

Metric Name	Description
Early Hi-Depth Test Fails, pixels	Represents the total number of pixels dropped on the early hierarchical depth test.

Metric Name	Description
Early Depth Test Fails, pixels	Represents the number of pixels that failed the early depth/stencil tests.
Clipper Invocations	Represents the number of primitives processed by the Clipper. **Examples:** • If you render 100 triangles and clipping is enabled, the Clipper Invocations is 100. • If you render 100 triangles and clipping is disabled, the Clipper Invocations is 0. **Improving Performance:** In most cases you do not have to care about the clipper performance on Intel® HD Graphics 2000/3000 GPUs because these graphic processors utilize a fast clipping algorithm implemented in silicon. For more information on enabling/disabling hardware clipping read the Microsoft* DirectX* SDK documentation.
Post-Clip Primitives	Represents the number of primitives that flowed out of the clipper. The metric includes original primitives that passed the trivial clipping test (trivial accept), and new primitives that were created by the clipper as a result of the clipping operation. **Examples:** • If you render 100 triangles and clipping is enabled and all the triangles are trivially accepted, the Post-Clip Primitives is 100. • If you render 100 triangles and clipping is enabled and all the triangles are trivially rejected, the Post-Clip Primitives is 0. • If you render 100 triangles and clipping is enabled and one or more triangles are partially located within the viewing frustum, the Post-Clip Primitive count returns a value which could be more or less than 100 depending on the number of triangles that were clipped. If the value is significantly higher than 100 it means that many triangles where partially clipped, and the clipper created additional triangles. **Improving Performance:** In most cases you do not have to care about the clipper performance on Intel® HD Graphics 2000/3000 GPUs because these graphic processors implement an efficient clipping algorithm in silicon. For more information on enabling/disabling hardware clipping read the Microsoft* DirectX* SDK documentation .
Samples Killed in PS, pixels	Represents the total number of samples or pixels dropped in pixel shaders.
Primitive Count	Represents the number of primitives sent to the 3D hardware. **NOTE** For Microsoft* DirectX* 9: the Primitive Count metric matches the PrimitiveCount parameter in the rendering calls. **Examples:**

<table>
<tr><th>Metric Name</th><th>Description</th></tr>
<tr><td></td><td>

If you render 100 points, the IA stage assembles 100 point primitives and the Primitive Count is 100.
If you render two triangles as a triangle list, the IA stage assembles two triangles and the Primitive Count is two.
If you render two triangles as a triangle strip, the IA assembles two triangles and the Primitive Count is two.

Improving Performance:
If geometry/vertex processing becomes a bottleneck, try to reduce number of primitives sent to GPU for each frame by:

Simplifying your rendering geometry; for example, show small geometry details using bump maps instead of triangles, use lower detail models for far away objects, or use textures with multiple mip maps.
Optimizing your scene through various culling techniques; for example, use Binary Space Partitioning (BSP), Portal rendering, or Octrees.

</td></tr>
<tr><td>Vertex Count</td><td>
Represents the number of vertices sent to the 3D hardware pipeline during the D3D Input Assembler (IA) stage. The number of vertices depends on the primitive type and the number of primitives. The following formulas are used:
<table>
<tr><th>Primitive Type</th><th>Vertex Count</th></tr>
<tr><td>Point list</td><td>Number of Primitives</td></tr>
<tr><td>Triangle list</td><td>Number of Primitives *3</td></tr>
<tr><td>Triangle strip</td><td>Number of Primitives +2</td></tr>
<tr><td>Line list</td><td>Number of Primitives *2</td></tr>
<tr><td>Line strip</td><td>Number of Primitives +1</td></tr>
</table>
NOTE

For Microsoft* DirectX* 9 : When rendering indexed primitives the Vertex Count metric does not match the <code>NumVertices</code> parameter in the <code>::DrawIndexedPrimitive</code> , <code>::DrawIndexedPrimitiveUP</code> functions because the Input Assembler counts shared vertices multiple times.
For Microsoft* DirectX* 10and later : The Vertex Count metric does not include vertices created during the geometry shader stage.

Examples:

If you render 100 points, the IA stage assembles 100 point primitives with 100 vertices total, and the Vertex Count is 100.
If you render two triangles as a triangle list, the IA stage assembles two triangles with six vertices total, and the Vertex Count is six.
If you render two triangles as a triangle strip, the IA stage assembles two triangles with four vertices total, and the Vertex Count is four.

Improving Performance:
To minimize the number of vertices sent to the pipeline and thereby improve vertex processing performance, use graphics primitives that minimize the amount of data being sent to and processed by the GPU, such as using single triangle strips.
</td></tr>
</table>

Metric Name	Description
Samples Blended, pixels	Represents the total number of samplers or pixels written to all render targets.
Samples Written	Represents the number of pixels/samples written to render targets. The graphics driver 9.17.10 introduces a new notion of deferred clears. For the sake of optimization, the driver decides whether to defer the actual rendering of clear calls in case subsequent clear and draw calls make it unnecessary. As a result, when clear calls are deferred, the Graphics Frame Analyzer shows their GPU Duration and Samples Written as zero. If later it turns out that a clear call needs to be drawn, the work associated with that clear call gets included in the duration of the erg that was being drawn when this clear call was deferred, not necessarily a clear call. This means that in the Graphics Frame Analyzer metrics associated with a clear call accurately reflect the real work associated with that erg.
Alpha Test Fails	Represents the number of pixels that failed the alpha test and are ignored (not written to the surface). **Examples:** If Alpha Test Fails is 5000, then 5000 pixels failed the alpha test and were not written to the surface.
Pixels Rendered	Represents the number of pixels that passed the depth-test (both Z-buffer and Stencil if enabled). If the depth-test was disabled, Pixels Rendered counts all the pixels that passed through from the previous pipeline stage. **NOTE** Pixels that passed the depth-test might not necessarily appear in the render target, which could occur if the color buffer write mask is set to 0. **Examples:** • If you render a quad with 8x8 pixels, located entirely within the viewing frustum and all the pixels passed depth test, Pixels Rendered is 64. • If you render a quad with 8x8 pixels, located entirely within the viewing frustum and half of the pixels are rejected by depth tests or other stages of the graphics pipeline, Pixels Rendered is 32. **Improving Performance:** A high number of rendered pixels results in a high number of pixel shader executions, which requires more rendering time. To keep the number of rendered pixels as low as possible, optimize the rendering order to maximize Early-Z benefit or use a Z-only pass if possible. To find areas with high depth complexity, use the Overdraw option in the Graphics Frame Analyzer.

Graphics API Metrics

This section describes all Graphics API metrics accessible from the Intel® GPA.

Main Metrics

Metric Name	Description
Draw Calls	Represents the number of draw calls issued from your application per frame. When the Draw Calls metric is high, see if you can optimize your application's geometry to improve vertex batching into fewer draw calls. (Windows* only) Represents the number of draws submitted to the D3D Runtime.
Frames per Second	Represents the instantaneous frame rate normalized to seconds (inverted frame time).
Frame Number	Represents the number of the current CPU frame.
Frame Time	Represents the instantaneous frame time in microseconds.

DirectX* Metrics

Metric Name	Description
RT Changes	Represents the number of calls per frame: • Microsoft DirectX 9: The aggregated number of `IDirect3DDevice9::SetRenderTarget` calls per frame. • Microsoft DirectX 9Ex: The aggregated number of `IDirect3DDevice9Ex::SetRenderTarget` calls per frame.
Shader Creations	Represents the number of calls per frame: • Microsoft DirectX 9: The aggregated number of `IDirect3DDevice9::CreateVertexShader/CreatePixelShader` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::CreateVertexShader/CreatePixelShader` calls per frame. • Microsoft DirectX 10: The aggregated number of `ID3D10Device::CreateVertexShader/CreateGeometryShader/ CreateGeometryShaderWithStreamOutput/CreatePixelShader` calls per frame. • Microsoft DirectX 11: The aggregated number of `ID3D11Device::CreateVertexShader/CreateGeometryShader/ CreateGeometryShaderWithStreamOutput/CreatePixelShader/ CreateHullShader/CreateDomainShader` calls per frame.
State Block Applies	Represents the number of `IDirect3DStateBlock9::Apply` calls per frame.
State Block Captures	Represents the number of `IDirect3DStateBlock9::Capture` calls per frame.
State Changes	Represents the number of state changes set to the D3D Runtime.
Texture Creations	Represents the number of calls per frame: • Microsoft DirectX 9: The aggregated number of `IDirect3DDevice9::CreateTexture/CreateCubeTexture/ CreateVolumeTexture` calls per frame.

Metric Name	Description
	• Microsoft DirectX 9Ex: The aggregated number of `IDirect3DDevice9Ex::CreateTexture/CreateCubeTexture/CreateVolumeTexture` calls per frame. • Microsoft DirectX 10: The aggregated number of `ID3D10Device::CreateTexture1D/2D/3D` calls per frame. • Microsoft DirectX 11: The aggregated number of `ID3D11Device::CreateTexture1D/2D/3D` calls per frame.
Buffer Creations	Represents the following: • Microsoft DirectX 10: The number of `ID3D10Device::CreateBuffer` calls per frame. • Microsoft DirectX 11: The number of `ID3D11Device::CreateBuffer` calls per frame.
Buffer Maps	Represents the following: • Microsoft DirectX 10: The number of `ID3D10Buffer::Map` calls per frame. • Microsoft DirectX 11: The number of `ID3D11DeviceContext::Map` calls that receive ID3D11Buffer as a resource parameter per frame.
Clears	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::Clear` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::Clear` calls per frame.
Color Fills	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::ColorFill` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::ColorFill` calls per frame.
IB Creations	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::CreateIndexBuffer` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::CreateIndexBuffer` calls per frame.
IB Lock Time	Represents the amount of time in microseconds spent in IB locks. This is the amount of time actually inside the lock routines only.
IB Locks	Represents the number of index buffer (IB) locks called.
Locks	Represents the number of vertex buffer, index buffer, surface, and volume locks called per frame.
Lock Time	Represents the total amount of time in microseconds spent in vertex buffer, index buffer, surface, and volume locks. This is the amount of time actually inside the lock routines only.
Maps	Represents the number of texture and buffer maps.

Metric Name	Description
Resource Copies	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Device::CopyResource` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::CopyResource` calls per frame.
Resource Creations	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9 CreateXXX` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex CreateXXX` calls per frame. • Microsoft DirectX 10 : The aggregated number of `ID3D10Device::CreateXXX` calls per frame. • Microsoft DirectX 11 : The aggregated number of `ID3D11Device::CreateXXX` calls per frame. `CreateXXX` means all Create calls; that is, the aggregation for DX Create Texture per Frame, DX Create Shader per Frame, DX Create Surface per Frame, etc.
RT Clears	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10RenderTargetView::Clear` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::ClearRenderT` calls per frame.
RT Data Gets	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::GetRenderTargetData` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::GetRenderTargetData` calls per frame.
Stretch Rects	Represents the number of calls per frame. • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::StretchRect` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::StretchRect` calls per frame.
Surface Creations	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::CreateOffscreenPlainSurface/ CreateDepthStencilSurface/CreateRenderTarget` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::CreateOffscreenPlainSurface/ CreateDepthStencilSurface/CreateRenderTarget` calls per frame.
Surface Lock Time	Represents the total amount of time in microseconds spent in DX surface locks per frame.
Surface Locks	Represents the number of DirectX surface resource locks per frame.

Metric Name	Description
Surface Updates	Represents the number of calls per frame: • Microsoft DirectX 9: The aggregated number of `IDirect3DDevice9::UpdateSurface` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::UpdateSurface` calls per frame.
Subresource Copies	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Device::CopySubresourceRegion` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::CopySubresourceRegion` calls per frame.
Subresource Update	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Device::UpdateSubresource` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::UpdateSubresource` calls per frame.
Texture1D Maps	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Texture1D::Map` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::Map` calls that receive. `ID3D11Texture1D` as a resource parameter per frame.
Texture2D Maps	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Texture2D::Map` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::Map` calls that receive. `ID3D11Texture2D` as a resource parameter per frame.
Texture3D Maps	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10Texture3D::Map` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::Map` calls that receive. `ID3D11Texture3D` as a resource parameter per frame.
VB Creations	Represents the number of calls per frame: • Microsoft DirectX 9 : The aggregated number of `IDirect3DDevice9::CreateVertexBuffer` calls per frame. • Microsoft DirectX 9Ex : The aggregated number of `IDirect3DDevice9Ex::CreateVertexBuffer` calls per frame.
VB Lock Time	Represents the amount of time in microseconds spent within all VB lock APIs. This is the amount of time actually inside the lock routines only.
VB Locks	Represents the number of vertex buffer (VB) locks called.
Volume Lock Time	Represents the total amount of time in microseconds spent in DX volume locks per frame.

Metric Name	Description
Volume Locks	Represents the number of DirectX volume resource locks per frame.
Z/Stencil Clears	Represents the number of calls per frame: • Microsoft DirectX 10 : The number of `ID3D10DepthStencilView::Clear` calls per frame. • Microsoft DirectX 11 : The number of `ID3D11DeviceContext::ClearDepthStencilView` calls per frame.

OpenGL* Metrics

Metric Name	Description
Buffer Creations	Represents the number of OpenGL ES buffers that are allocated from your application per frame. **Improving Performance:** When the Buffer Creations metric is high, see if you can change your application to allocate required buffers once at application startup.
Indexed Draw Calls	Represents the number of indexed draw calls being issued from your application per frame. **Improving Performance:** • When the Indexed Draw Calls metric is high, see if you can optimize your application's geometry to better batch vertices into fewer draw calls, specifically those that use `glDrawElements` or its modifications. • When the Indexed Draw Calls metric is significantly lower than the Draw Calls metric see if you can optimize your application to use indexed geometry passed to the `glDrawElements` API or its modifications, rather than `glDrawArrays` and its modifications. Additionally, check the Vertex Count and Indexed Vertex Count metrics to see how much data is being drawn without indexing to estimate the impact of such optimization.
Vertex Count	Represents the number of vertices passed by your application to OpenGL draw calls per frame. This totals the counts passed to the draw calls. **Improving Performance:** • When the Vertex Count metric is higher than expected and fragment processing is not the bottleneck, check whether you can improve culling, or whether you can use LOD or other techniques to reduce geometric complexity. • When the Vertex Count metric is low, try to increase geometric complexity as an alternative to expensive operations such as Alpha Test that might slow down the pipeline.
Indexed Vertex Count	Represents the number of indexed vertices passed by your application to indexed draw calls per frame. **Improving Performance:**

Metric Name	Description
	When the Indexed Vertex Count metric is significantly lower than the Vertex Count metric, see if you can optimize your application to use indexed geometry passed to the `glDrawElements` API or its modifications, rather than `glDrawArrays` and its modifications.
RT Clears	Represents the number of Render Target clears per frame.
UseProgram Calls	Represents the number of calls to use OpenGL ES shader programs from your application per frame. **Improving Performance:** When the UseProgram Calls metric is high, see if you can improve batching to draw all the objects that use a particular program before binding the next one. Combining several shaders into an uber-shader with static flow control can also reduce this metric and its associated cost, but does not always result in a performance improvement.
Error Gets	Represents the number of times that the `glGetError` method is called from your application per frame. **Improving Performance:** When the **Error Gets** metric is high, see if you can optimize your application to avoid checking for error conditions during its render path.
BindBuffer Calls	Represents the number of calls to bind OpenGL* ES buffers from your application per frame. **Improving Performance:** When the **BindBuffer Calls** metric is high, see if you can improve batching to draw all instances of the same geometry without switching the bound buffer. Additional performance gains might come from using one buffer for several pieces of geometry and just drawing the appropriate piece where needed.
BindTexture Calls	Represents the number of calls to bind OpenGL ES textures from your application per frame. **Improving Performance:** When the **BindTexture Calls** metric is high, see if you can improve batching to draw all the objects that use a particular texture before binding the next one. If possible, a further step to improve performance would be to use texture atlasing rather than multiple smaller textures.
RT Changes	Represents the number of Render Target changes per frame.

Memory Metrics

Memory metrics help you understand how much memory is utilized by the target application.

The table below lists all Memory metrics that you can analyze with Intel® GPA.

Metric Name	Description
App Resident Memory	Represents the resident set size (RSS) of the target application. RSS is the portion of the application's memory that is currently held in RAM.

Metric Name	Description
Available Memory	Represents the memory on the platform that is available for use by the target application. This includes cached memory and I/O buffers, which are made available by the kernel when needed.

User Interface Reference

This section describes the graphical user interface controls of the Intel® GPA tools:

- Graphics Monitor
- Heads-Up Display (HUD)
- System Analyzer
- Graphics Frame Analyzer
- Graphics Trace Analyzer

Graphics Monitor UI Reference

This section includes detailed reference on the Intel® GPA Graphics Monitor user interface elements.

Graphics Monitor has the following graphical user interface controls:

- Context Menu
- Graphics Monitor configuration window
- Control Panel
- Desktop Applications Tab
- Universal Windows Applications Tab
- Options Screen

Graphics Monitor Context Menu

Use the context menu to launch any of the Intel® Graphics Performance Analyzers.

To access the **Context Menu**, run the Graphics Monitor application and right-click the 

Graphics Monitor icon on the taskbar notification area.

The context menu provides the following options:

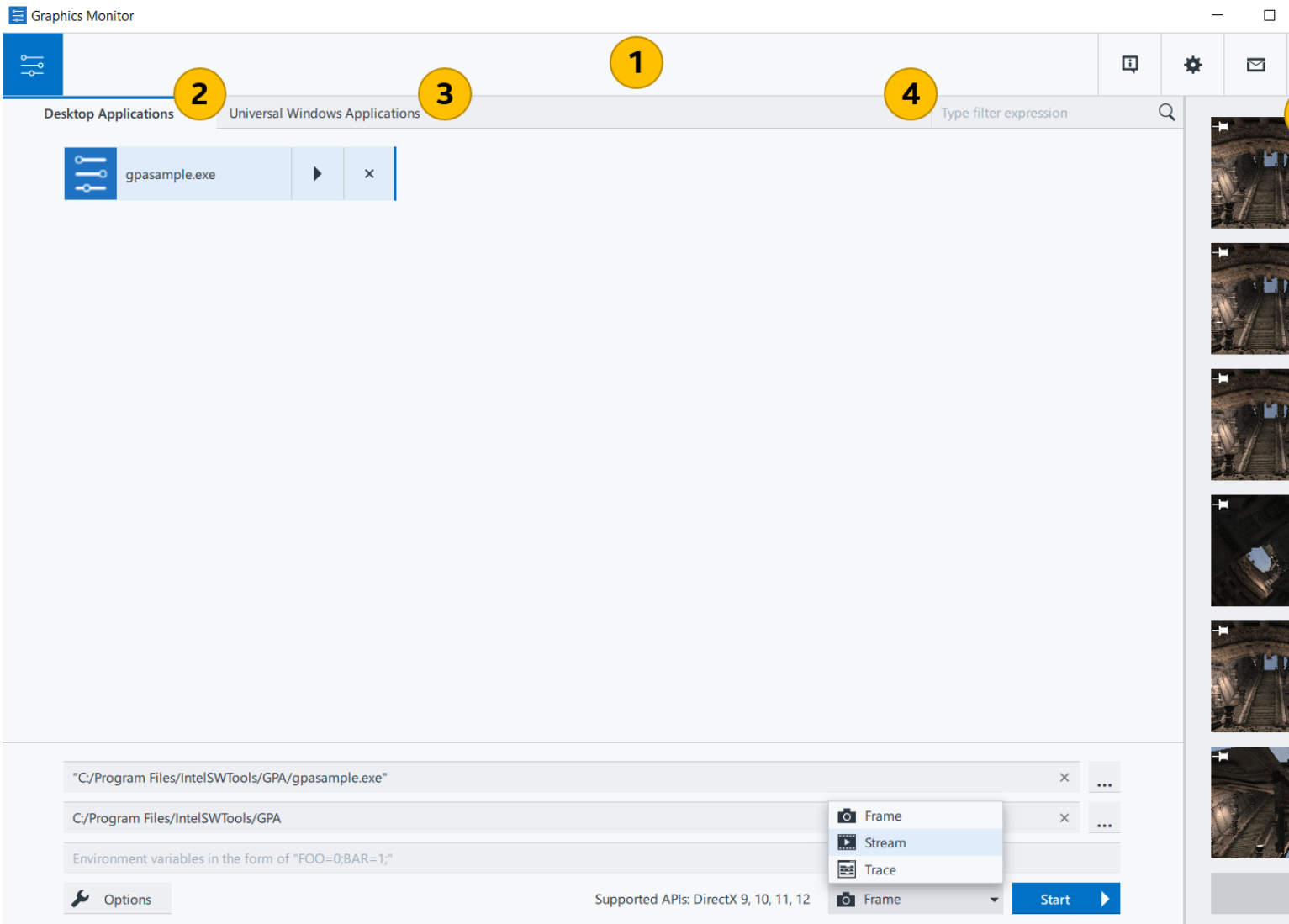
- **Graphics Monitor** opens a Graphics Monitor configuration window that enables you to manually start and analyze an application, manage application lists, and set analysis options.
- **Graphics Frame Analyzer** launches Graphics Frame Analyzer.
- **Graphics Trace Analyzer** launches Graphics Trace Analyzer.
- **System Analyzer** launches the System Analyzer.
- **Exit** closes the Graphics Monitor.

Graphics Monitor Configuration Window

Use the Graphics Monitor configuration window to specify and run a graphics application for analysis with Intel® GPA.

To access the Graphics Monitor configuration window, right-click the 

Graphics Monitor icon in the taskbar notification area, and select **Graphics Monitor.**



1 **Control panel**

Enables you to review your system data, configure settings, change the color schema, identify network IP address(es) of the local system, send feedback, and read the Intel® GPA user guide.

2 **Desktop Applications** tab

Enables you to choose and start a desktop application for analysis.

3 **Universal Windows Applications** tab

Enables you to choose and start a universal Windows* application for analysis.

4 **Type Filter Expression** field

Enables you to find the required application in the list. Saves previous search keywords.

5 **Open File** dialog box

Enables you to open captured frames, streams, and traces.

Capture files thumbnails are marked with the following icons:

- Frame:



- Stream:

- Trace: 
- Invalid file: 


Application Startup Mode drop-down menu at the bottom of the window enables you to choose between the following startup modes:

- **Frame:** choose this mode to monitor real-time metrics and capture a frame. This mode supports DirectX* 9, 10, and 11 applications.
- **Frame [DX12]**: choose this mode to monitor real-time metrics and capture a frame for DirectX 12 applications. This mode supports DirectX 12 Ultimate features, like Mesh Shading and Ray Tracing.
- **Trace:** choose this mode to monitor real-time metrics and capture a trace.
- **Stream:** choose this mode to capture a sequence of frames. This can be useful for isolating sporadic performance problems that only affect a subset of frames.

See Also

Launch Application for Analysis

Graphics Monitor Control Panel

Use the Graphics Monitor control panel to configure general and appearance settings, and specify a list of remote systems for analysis.

The panel includes the following controls:

- 

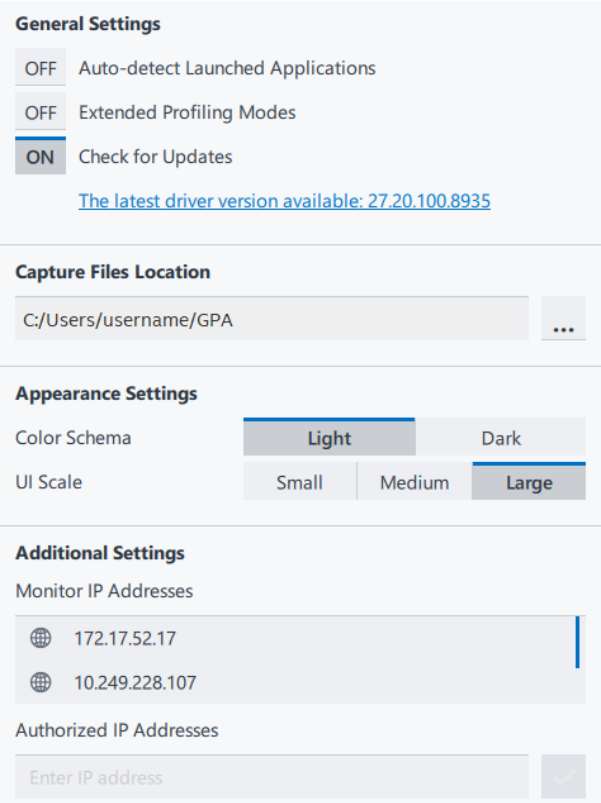
  System Information control provides information about the Intel® GPA version installed on the system and extended system information. You can copy the system information to the clipboard or save it to a `.txt` file.
- 

  Settings control provides access to the **Settings** panel.
- 

  Send Feedback control opens a form to send feedback to the Intel® GPA technical support.
- 

  Help control provides access to the Intel® GPA user guide (this document).

Settings Panel



- **General Settings**:
  - (Windows* only) Turn ON the **Auto-detect Launched Applications** toggle button to automatically start analyzing DirectX* applications that were not launched through the Graphics Monitor. For example, if they were launched through the Windows* Explorer or a console command. If this button is on, the command line of any graphics application is added to the **Application list** in the **Desktop Applications Tab**. All child processes started from graphics applications are also automatically added to the **Application list**. Auto-detection of launched applications does not apply to universal Windows* applications .
  - Turn ON the **Extended Profiling Modes** toggle button to enable the selection of a specific graphics API in the Graphics Monitor launch mode drop-down menu. Selecting one of these launch mode entries enables you to only capture performance data from a specific graphics API. Use this feature for more accurate performance of Intel® GPA. It is most recommended to turn ON **Extended Profiling Modes** toggle button, if your project uses multiple APIs.
  - Turn OFF the **Check for Updates** toggle button to disable update notifications. By default, Intel GPA notifies you when a newer version of the product or Intel® Graphics Driver is available.

    NOTE
    If an update is available, an exclamation mark in a red circle appears on the **Settings** button:

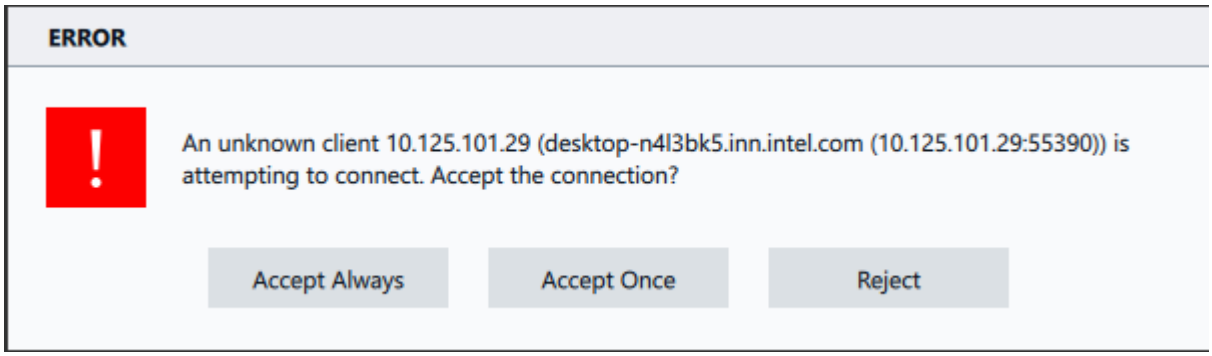
    .

- Use the **Capture Files Location** option to set a folder to save captured frames, traces or streams. The folder is defined as **Current** when you open **Frame Capture** or **Trace Capture** windows.
- **Appearance Settings**:

- Use the **Color Schema** setting to change the color schema of the Graphics Monitor interface: **Light** (default) or **Dark**.
- Use the **UI Scale** setting to change the size of the Graphics Monitor window and its elements to **Small**, **Medium**, or **Large** (default).

- **Additional Settings**:
  - Use the **Monitor IP Addresses** field to identify network IP address(es) of the local system.
  - Use the **Authorized IP addresses** field to view and change the list of target systems authorized to connect to your host system. In this field, you can view and change the list of analysis systems authorized to connect to your target system automatically:
    - To authorize a new analysis system, type in a new IP address, and then click the check mark button.
    - To forget an analysis system, select its IP address in the list, and then click the X button.

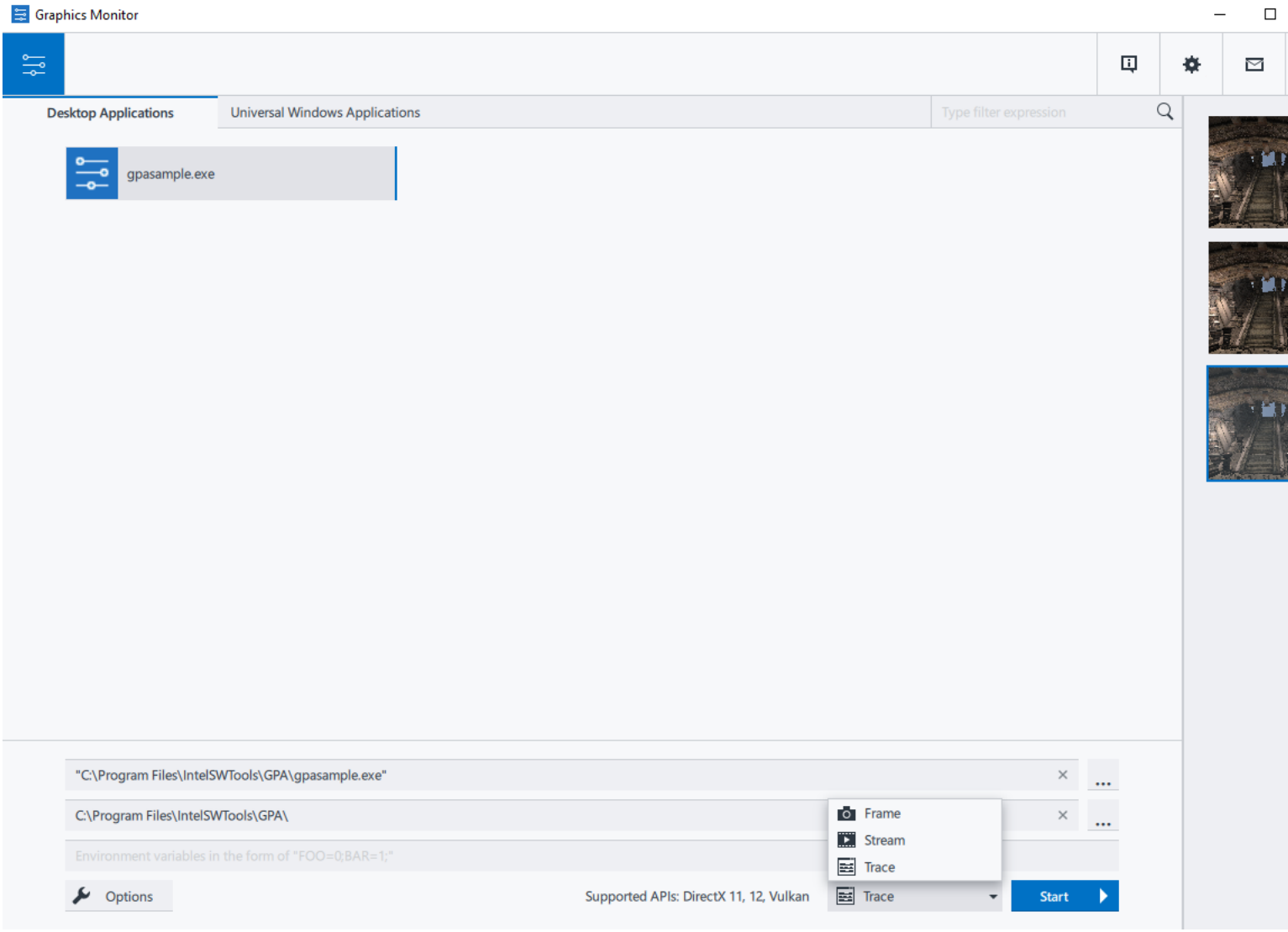
NOTE
When a remote host system tries to connect to the local system, a notification message displays asking you to accept the connection.



Graphics Monitor Window: Desktop Applications Tab

*Use the **Desktop Applications** tab to specify a desktop app, configure and launch graphics performance analysis with Intel© GPA.*

To access the configuration, click the **Desktop Applications** tab on the Graphics Monitor Configuration Window.



Configure the following analysis settings:

- **Applications list** shows a list of recently run applications. Each application is represented by its icon, name, and/or command-line arguments. You can remove an application from the list by hovering the cursor over the application name and clicking the X button. By default, for Windows* OS host this field has a DirectX* 11 sample application that is supplied with Intel® GPA.
- **Application Executable with Command Line Arguments** field shows the path and file name of the application you want to analyze; it also allows adding command-line options for the analyzed application.

> **NOTE**
> If a command-line argument is specified and a path to an executable contains spaces, put the path in quotation marks.

- **Working Directory** field shows the directory in which the application is to be run. This is useful for games that use a launcher or mission packs located in a different directory than the game files.
- Use the **Environment Variables** field to set environment variables.
- Use the **Options** button to open the Graphics Monitor Window: Options Dialog Box.
- Use the **Application Startup Mode** drop-down list to choose one of the following modes to launch an application:

- **Frame**: Choose this mode to capture a frame for a DirectX (Windows*) or OpenGL* (Ubuntu*) application
- **Stream**: Choose this mode to capture a multi-frame stream for a DirectX 11/12 (Windows*) or Vulkan* application
- **Trace**: Choose this mode to capture a trace for a DirectX, Vulkan* (Windows*) or OpenGL* (Ubuntu*) application

- Click the **Start** button to launch the selected application.

See Also

Start a Desktop Application with Graphics Monitor

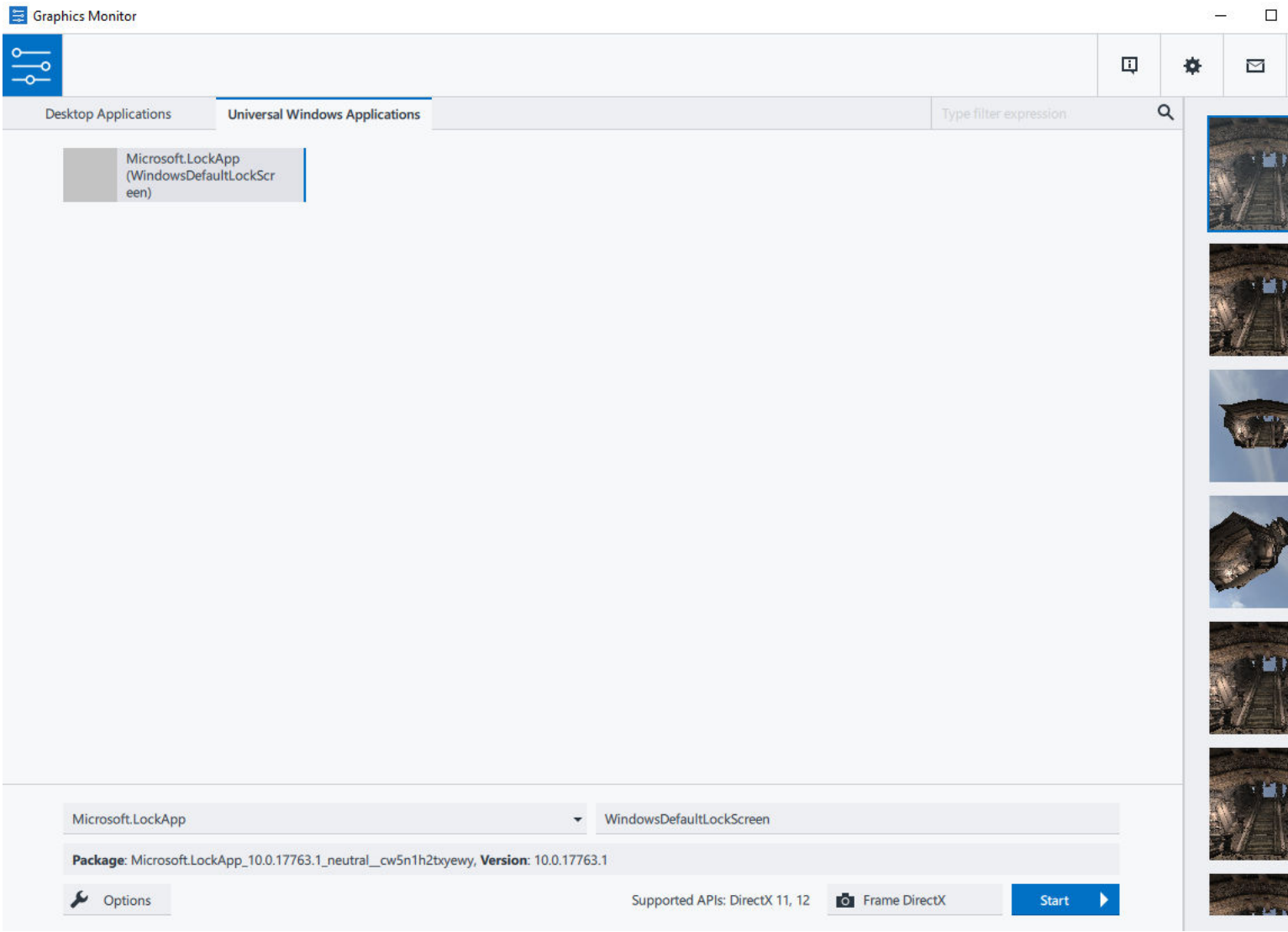
Analyze Desktop Graphic Applications

Graphics Monitor Options Window

Graphics Monitor Window: Universal Windows Applications Tab (Windows* only)

*Use the **Universal Windows Applications** tab to specify a universal Windows* app, configure and launch graphics performance analysis with Intel© GPA.*

To access the analysis configuration, click the **Universal Windows Applications** tab on the Graphics Monitor configuration window:



Configure the following analysis settings:

- **Applications list** shows a list of recently run applications. Each application is introduced by its icon, name, and ID (in round brackets). You can remove an application from the list by hovering the cursor over the application name and clicking the X button.
- Use the **Select application name** drop-down list to select the name of the application you want to analyze. You can search the required application by entering its name.
- Use the **Select application ID** drop-down list to view and select the application ID from the package. This drop-down list is active when there are several IDs in one package. Usually a package contains only one application ID.
- **Package and version information** field shows the package name and version of the application you selected for analysis.
- Use the **Options** button to open the Graphics Monitor Window: Options Dialog Box.
- Click the **Start** button to launch the selected application.

See Also

Graphics Monitor Options Window

Graphics Monitor Window: Options Dialog Box

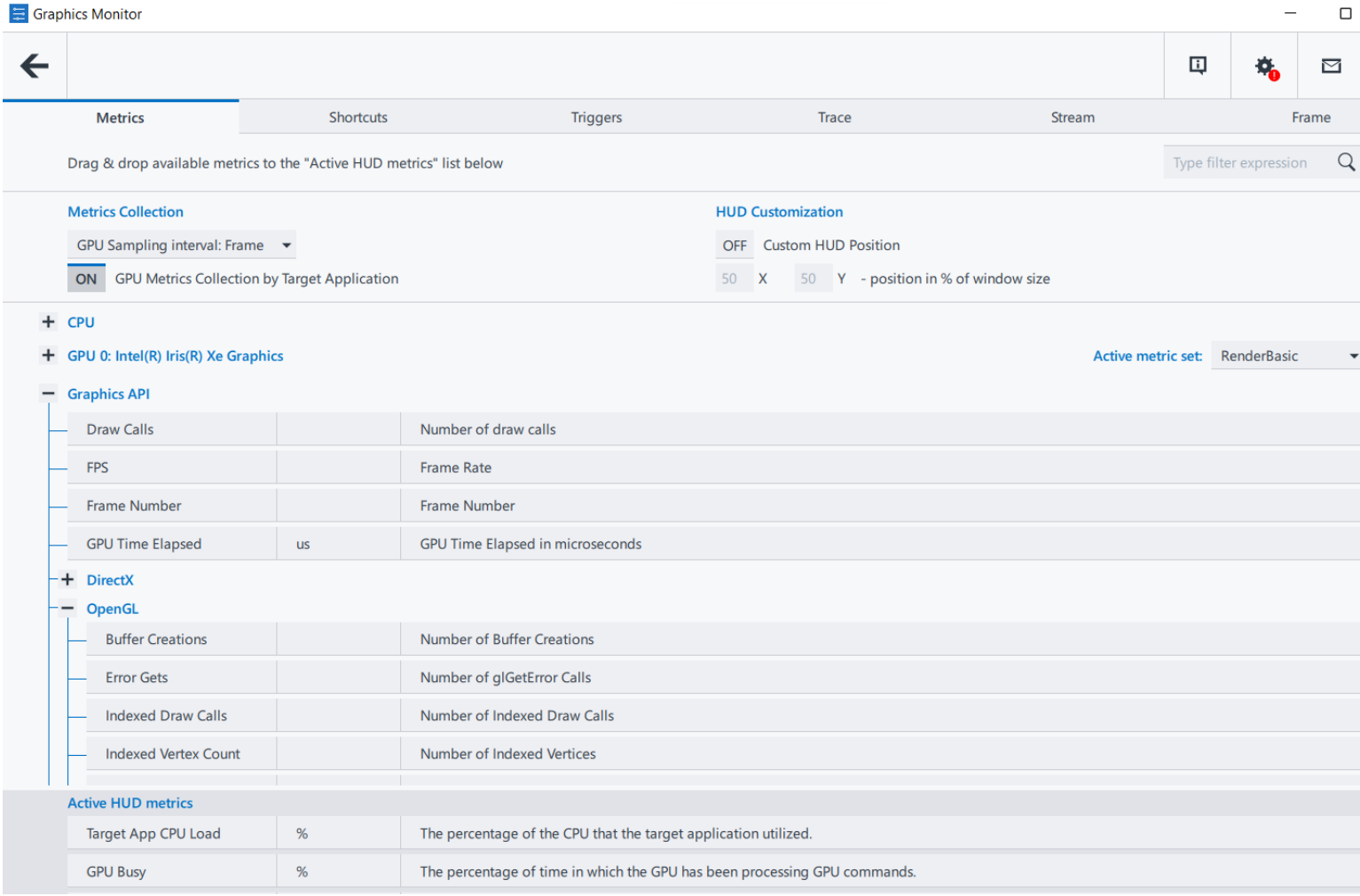
*Use the **Options** dialog box to configure trace, stream, and frame collection settings and specify GPU metrics for analysis.*

To access the **Options** dialog box, click the **Options** button on the **Graphics Monitor** window. The **Options** dialog box comprises the following tabs:

- Metrics: choose metrics to be displayed in the Heads-Up Display (HUD) in the target application, customize the HUD position, and configure metrics collection.
- Shortcuts: view and change the default keyboard shortcuts.
- Triggers: enable and configure a trigger.
- Trace: set trace collection settings.
- Stream: set stream collection settings.
- Frame: set frame delimiters.

NOTE Settings in the **Metrics** tab and the **Trigger** tab are not applicable to applications running in the Steam mode.

Metrics Tab



Use the **Metrics** tab to do the following:

- Configure the GPU sampling interval

- Enable/disable GPU metrics collection by the target application
- Customize the System Analyzer Heads-Up Display (HUD) position
- Configure the HUD metrics
- Select the active metric set from a list for each Intel® Graphics adapter

The **Metrics** tab comprises two lists:

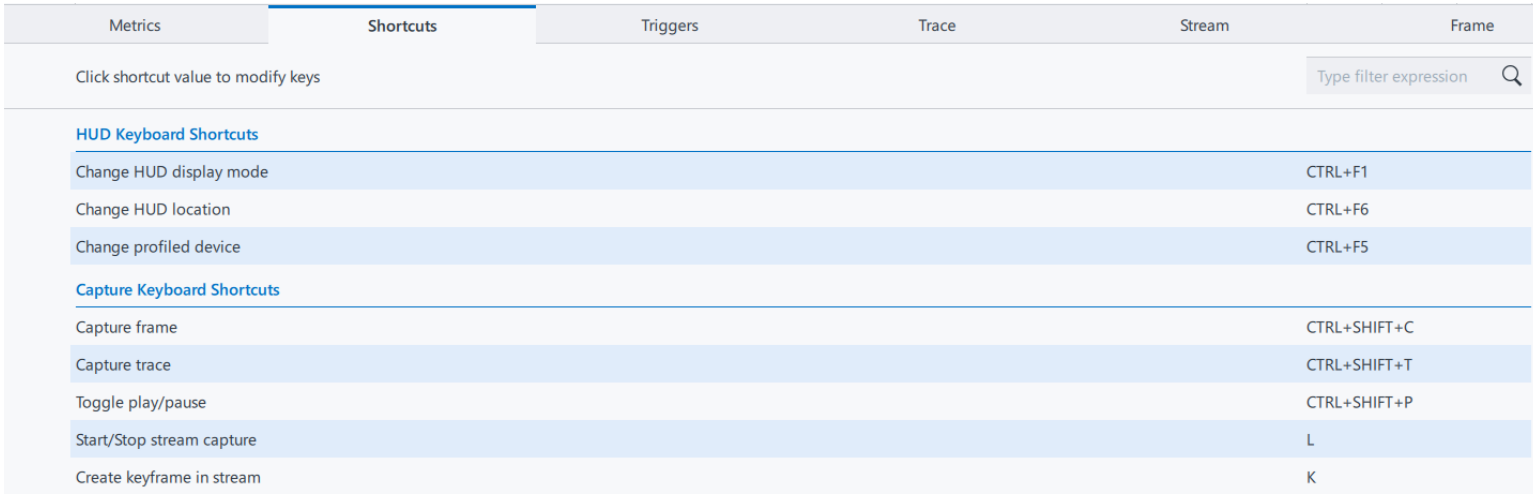
- List of all available HUD metrics
- List of active HUD metrics

To configure the HUD metrics for the target application, drag and drop a metric from the list of available metrics to the list of active HUD metrics. The maximum number of active HUD metrics is four. To restore the default list of active HUD metrics, click the



button. To change the displayed metrics, remove a metric from the **Active HUD metrics** list by dragging and dropping this metric back to the list of available HUD metrics, or reorder the **Active HUD metrics** list by dragging and dropping a desired metric over another active metric. Use the search bar at the top of the tab to find relevant metrics. To select the active metric set, use the **Active metric set** drop-down menu in the GPU section of the metric tree.

Shortcuts Tab



The **Shortcuts** tab enables you to substitute a default keyboard shortcut with a custom value . To change a shortcut, click the shortcut value, and then click the buttons combination that you want to use as a new shortcut: start with a combination of modifier keys (ALT, CTRL, SHIFT) and end with an alpha-numeric character key. To restore default shortcut values, click the



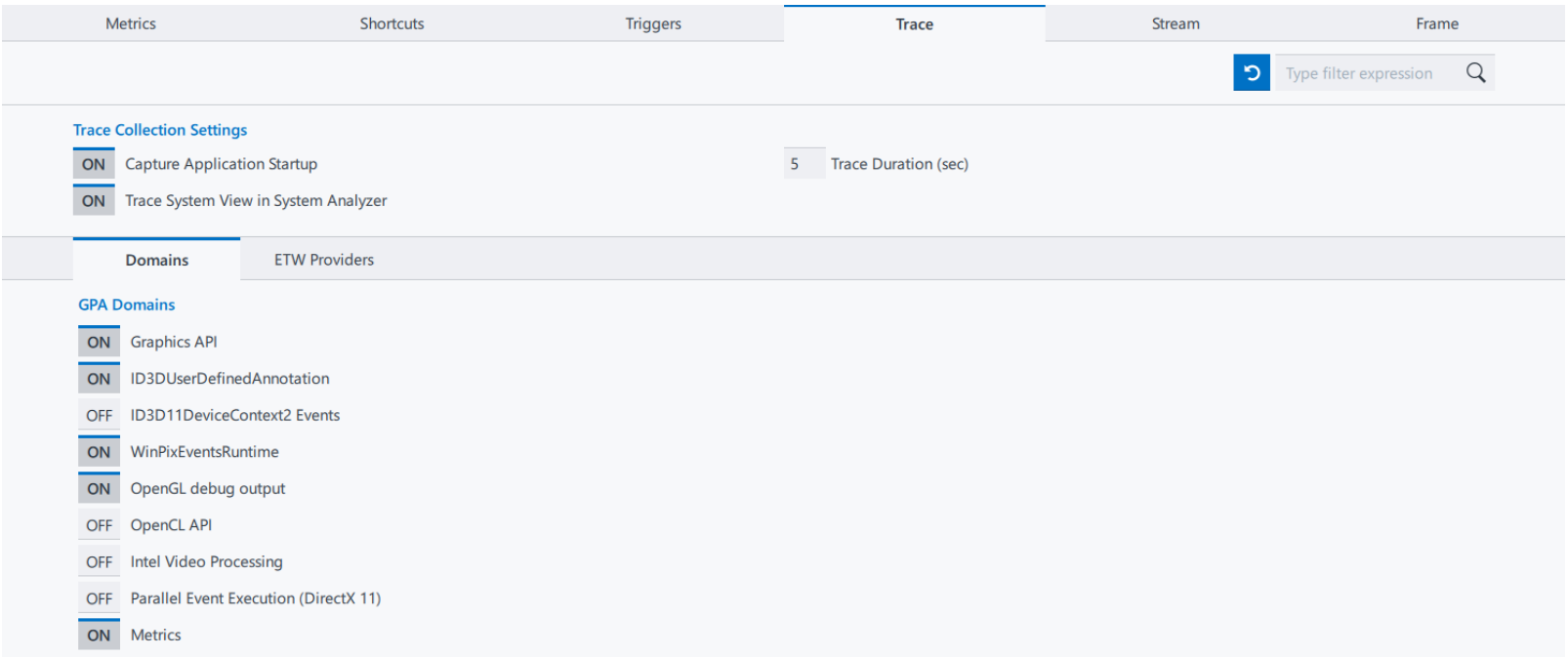
button. Use the search field at the top of the tab to find relevant shortcuts.

Triggers Tab



The **Triggers** tab enables you to set a custom condition for frame, trace, or system trace capturing. For instructions on setting up a trigger, see Set Up a Trigger.

Trace Tab



In the **Trace** tab you can modify tracing options:

- Enable capturing a trace upon the application startup.
- Enable Trace System View in System Analyzer.
- Change the duration of the application run to be captured in a single trace file. By default, the trace duration is five seconds.
- Select predefined Intel GPA domains to collect data about. For example, enable capturing of events to show parallel execution data in Graphics Trace Analyzer for DirectX* 11 applications.

NOTE
ID3DUserDefinedAnnotation, ID3D11DeviceContext2 Events, and WinPixEventsRuntime apply to the debug annotation API. It is not recommended to use these domains in the code together and enable these domains simultaneously in the **Trace** tab. The default settings represent the common use case.

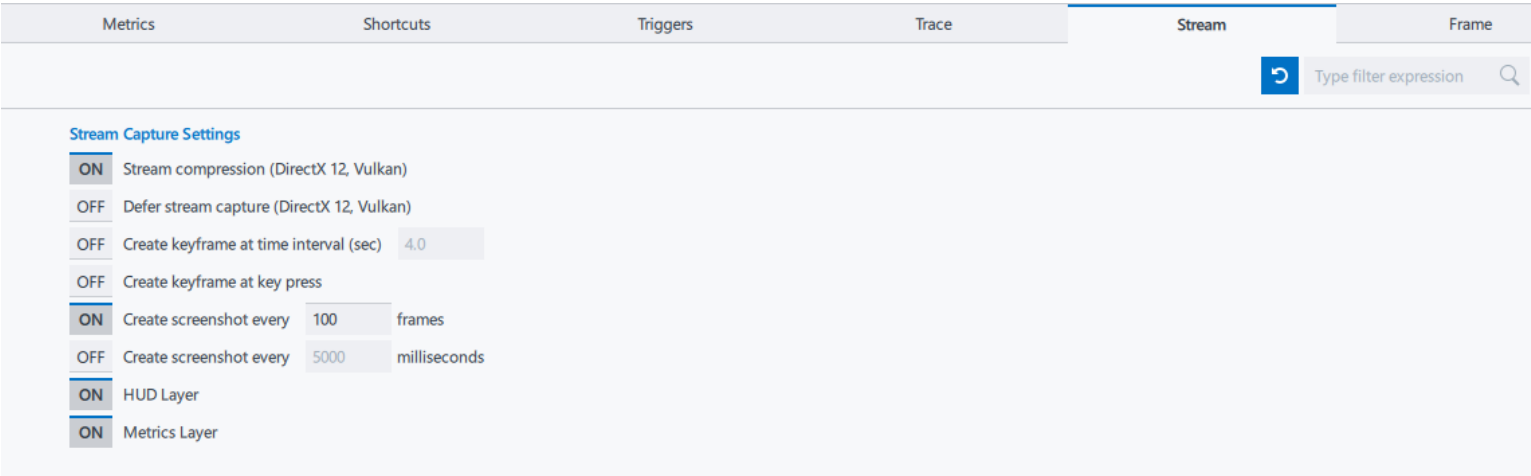
- Select **User ITT Domains** to collect data about. The list of user domains is gathered from the analyzed applications defining them with Instrumentation and Tracing Technology API (ITT API) and is updated upon exit from the application.
- (Windows* only) Click **ETW Providers** to add custom Event Tracing for Windows event providers.

To restore the default tracing parameters, click the



button. Use the search field at the top of the tab to find relevant domains.

Stream Tab



Use the **Stream** tab to configure the following stream capture settings:

- Enable stream compression to reduce the captured stream size by up to 30%. This option is also useful if you store streams for such tasks as regression testing. (DirectX* 12, Vulkan* specific).

  NOTE
  When capturing streams from workloads with a high CPU load, enabling the compression may result in significant FPS reduction/performance drop since the compression adds an additional CPU overhead. In such cases, the reduced stream size results from a smaller number of captured frames per second.

- Enable deferred stream capturing. Streams are captured upon pressing **L** . (DirectX* 12, Vulkan* specific)
- Create a keyframe at a specified time interval.
- Create a keyframe upon pressing a key.
- Enable/disable screenshots creation in a specified sequence of frames.
- Enable/disable screenshots creation in a specified time interval.

  NOTE
  You can disable screenshots creation to overcome possible overheads caused by this function.

- Enable/disable a HUD layer in the running application window.
- Enable/disable a Metrics layer in the running application window.

Frame Tab



Use the **Frame** tab to set frame delimiters for OpenGL and DirectX 12 applications. *Frame delimiter* is an API call acting as a frame border. Set frame delimiters to enable Intel GPA analysis for applications that do not use default frame delimiters.

- OpenGL Frame Delimiters:
  - SwapBuffers
  - SetCurrentContext
  - glBlitFramebuffer
  - glClear
  - glFinish
  - glFlush
- DirectX 12 frame delimiters:
  - Present
  - ExecuteCommandLists

Use the **Frame compression** option (DirectX 12 only) to reduce the captured frame size by up to 30%.

See Also

Configure a Metric Set

Set Frame Delimiters

Set Up a Trigger

Set Up Tracing Options

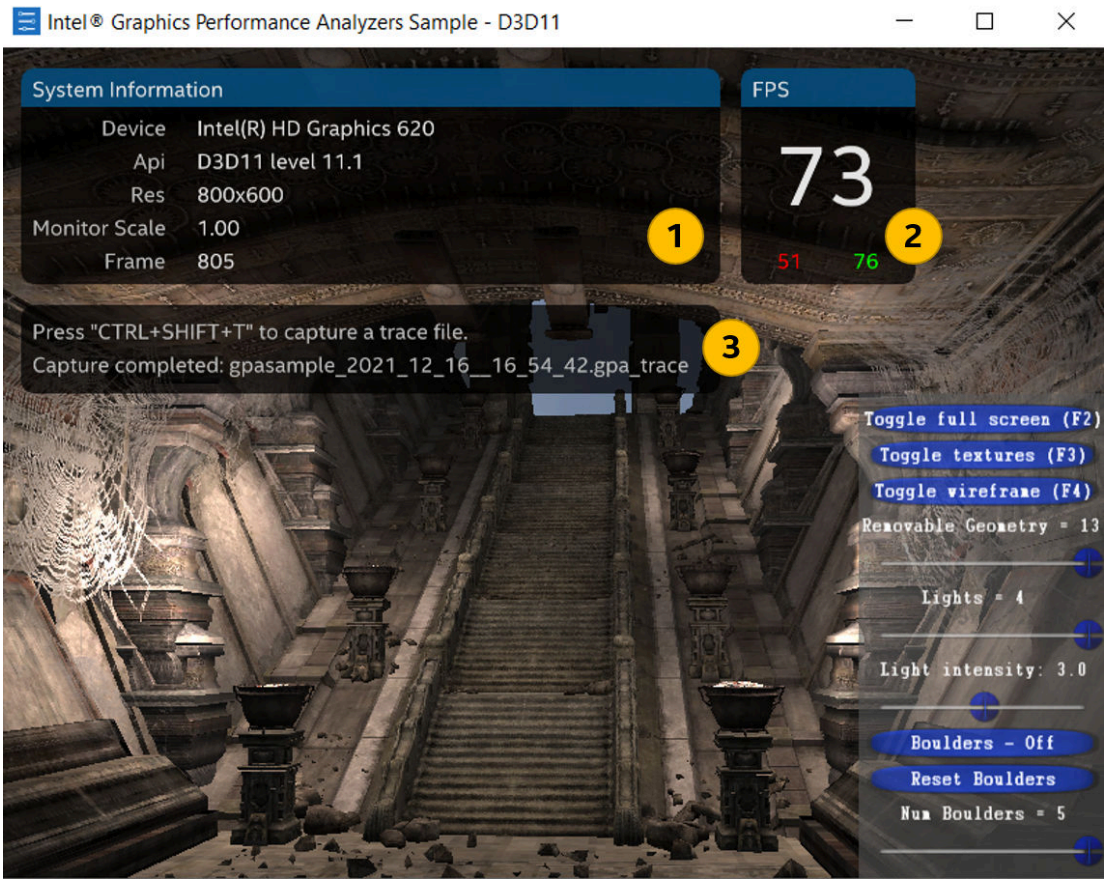
Customize Keyboard Shortcuts

Heads-Up Display (HUD) UI Reference

This section provides reference on the UI elements of the Heads-Up Display (HUD), which is an overlay provided at the top left of the screen when you launch the application.

HUD displays system and application statistics on application performance using the GPU and CPU performance metrics selected in the Graphics Monitor **Options** dialog box.

When you create trace or frame capture files, HUD displays the following information:



1 **System Information** pane

System Information includes the following information:

- **Device** - device name
- **Api** - Microsoft DirectX* version and device driver type
- **Res** - screen resolution
- **Monitor Scale**
- **Frame** - index number of the current frame

2 **FPS** pane

FPS (Frames Per Second) - frame rate, with lowest value shown in red (51) and the highest shown in green (76); the instantaneous value (73) is calculated for one second.

3 **Messages** pane

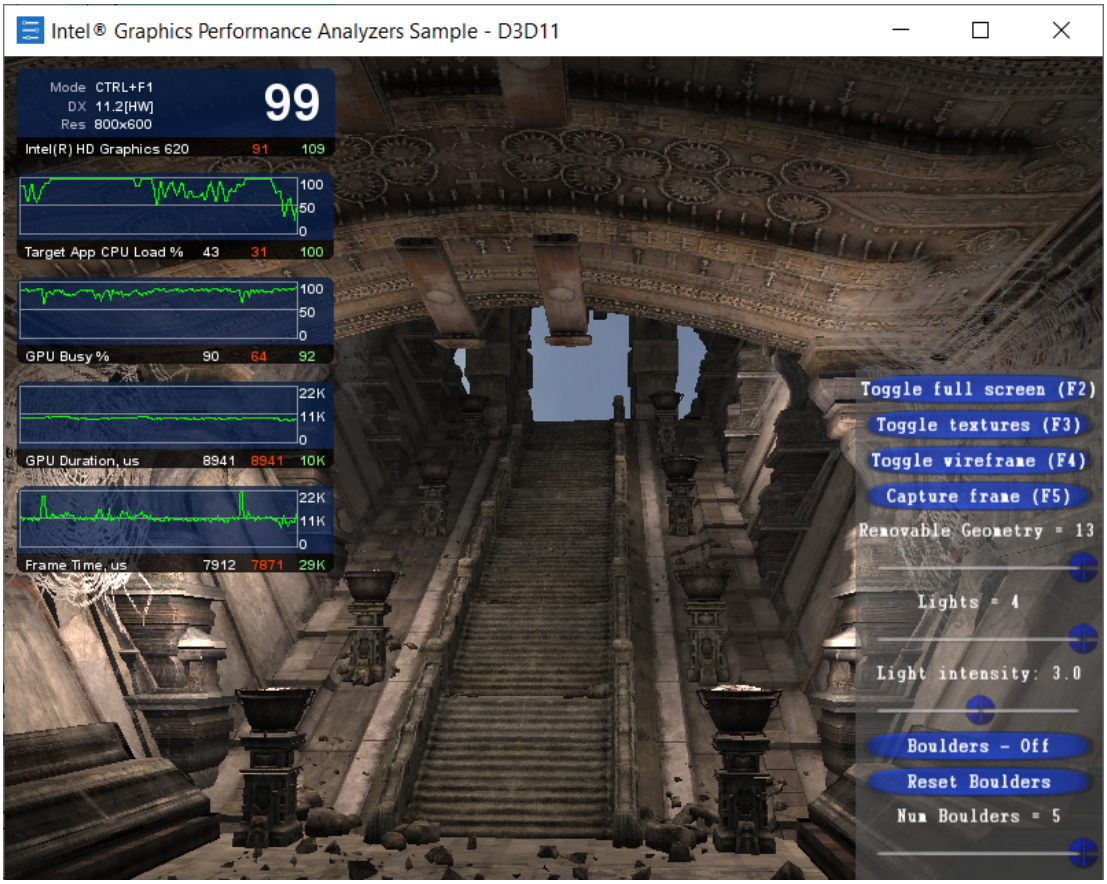
Status messages are displayed when you create trace or stream capture files. Possible messages are the following:

- "Stream recording: ON"
- Name of the captured file on disk
- Error messages (if any)

On the **Messages** pane, you can also see the tips on keyboard shortcuts. With the HUD default keyboard shortcuts you can capture frame and trace files, pause and resume the application, and switch between HUD modes.

Metrics Charts

When you create frame capture files, HUD displays four metrics charts.



For Intel® HD Graphics 3000/2000 and Intel® HD Graphics GPUs, the Heads-Up Display provides several kinds of data:

- CPU data
- DirectX* version
- GPU data:
  - Execution Units
  - Geometry Shader Input-Assembler
  - Main
  - Memory
  - Output-Merger
  - Pixel Shader
  - Rasterizer
  - Texture Sampler
  - Vertex Shader

For non-Intel GPUs and Intel GPUs prior to Intel® HD Graphics, the HUD displays the following data:

- CPU
- DirectX* version

Metrics are displayed in the form of sparklines and include the following data:

- Metrics name
- Instantaneous values (white)
- Minimum metrics values (red)
- Maximum metrics values (green)

NOTE
The minimum and maximum values reflect the min/max during the last seven seconds.

Limitations

The HUD may make available only the default set of metrics that it provides on non-Intel GPUs, even when using Intel® HD Graphics, in the following cases:

- On non-Intel GPUs and on older Intel GPUs such as Intel® GMA 4500/X4500/X4500HD/4500MHD, only the default set of metrics is available
- The application is using the Microsoft DirectX* 11 API, but is forcing it to use feature level 9. This is also known as DX11-on-9 mode.
- The application is using the WARP rasterization (Windows* Advanced Rasterization Platform).

HUD Default Keyboard-Shortcut Assignments

Some games might be using a particular keyboard-shortcut combination. If a default keyboard shortcut does not appear to be working with your game, you can assign new keyboard shortcuts. For information on how to assign a different capture keyboard shortcut, refer to Customize Keyboard Shortcuts.

With the keyboard shortcuts you can capture frame and trace files, pause and resume the application, and switch between HUD modes.

NOTE
Some games and applications trap all keystrokes, so keyboard shortcuts within the Intel® GPA do not work. In this case, to create frame/trace capture files you can:

- Use **Capture Frame/Capture Trace** buttons of the System Analyzer.
- Use **Triggers** tab of the **Options** dialog box in the Graphics Monitor to automatically create frame/trace capture files when certain conditions occur, for example when FPS is 20.

Frame Capture	Ctrl+Shift+C
Toggle Play/Pause	Ctrl+Shift+P
Trace Capture	Ctrl+Shift+T
Change Display Mode	Ctrl+F1
Change Profiled DX Device (DirectX* 11 only)	Ctrl+F5
Change HUD Location	Ctrl+F6

See Also

Launch Application

Customize Keyboard Shortcuts

System Analyzer UI Reference

Use System Analyzer tool of the Intel® GPA to monitor your application performance metrics in real time to determine whether the application is CPU- or GPU-bound.

The System Analyzer tool allows you to:

- Collect various metrics when running your application, or profile your system in the System View mode
- Analyze the results
- Capture data for further analysis
- Export metrics to a CSV file
- Pause the metrics charts and zoom in and out to get a better view of the metrics



The window is divided into these main areas:

1

Main toolbar

2

Metrics Control pane

3

Metrics Charts pane

Main Toolbar

Export Metrics to CSV Files button	Save metric values to a CSV file to post-process the analysis of your application.
Capture Frame button	Capture and save a frame instantly.

	NOTE This button is available only when you select **Frame Capture** mode in the Graphics Monitor window and connect System Analyzer to a single host. You can also capture frames using the dedicated button on the **Metrics Control** pane.
 **Capture Trace** button	Capture and save a trace instantly. **NOTE** This button is available only when you select **Trace Capture** mode in the Graphics Monitor window and connect System Analyzer to a single host. You can also capture traces using the dedicated button on the **Metrics Control** pane.
 **Messages** pane	See messages and non-critical errors. **NOTE** If more than one message is open, use the **Up** or **Down** arrows to navigate between messages.
Enter hostname or IP box	Connect to a specified host system. Enter a hostname or IP address, and then click the  button. You can also get a drop-down menu with a history of recent IP addresses or hostnames by clicking the button.
 **System Information** box	Get various information about the system running System Analyzer, including the operating system version, hardware information, and system firmware.

 **Settings** button	Open the **Settings** pane, which contains the following options:
	• With **Pause mode caching duration (minutes)** option, set the time frame within which metric values will be preserved in **Pause** mode, before the oldest values are discarded.
	• Restore the **Layout**, **Visible Tracks**, and **Track Colors** to default.
	• Change the color schema of the System Analyzer interface:
	  • Light color schema (default)
	  • Dark color schema
	• Adjust the user interface scale:
	  • Small
	  • Medium
	  • Large

 **Send feedback** button	Send feedback to the Intel® GPA technical support.
 **Help** button	Get information on navigation controls and opens the Intel® GPA online documentation.

Metrics Control Pane

The **Metrics Control** pane displays a group of hardware, driver, and application metrics used for the analysis of your system or a specific graphics application. For different target systems and workloads, there might be different metrics available. All metrics are subdivided into logical groups as described in Performance Metrics Reference.The **Metrics Control** pane allows you to:

- Select available metrics from the list, as well as switch between metric sets
- Instantly capture a trace
- Instantly capture a frame
- Select the process to profile

 **Type filter expression** field	Filter metrics by name. Save previous search keywords.
 **Hide inactive metrics** button	Hide all metrics that are not available under the currently selected metric set.
Collapse all groups button	Close all unfolded groups to the list of connected hosts.
Expand all groups button	Unfold all groups.
 **Capture Trace** button	Capture a trace file for the connected host and save the captured trace to the folder specified in the Settings pane of the Graphics Monitor Control panel. **NOTE** This button is available only when you select **Trace Capture** mode in the Graphics Monitor window.
 **Select Process** button	Select an available application for analysis or switch to System View mode.
 **Select Metric Set** button	Switch between different metric sets. This option is only available on systems with Intel® Graphics hardware.
 **Capture Frame** button	Capture a frame for the connected host and save the captured frame to the folder specified in the Settings pane of the Graphics Monitor Control panel. **NOTE** This button is available only when you select **Frame Capture** mode in the Graphics Monitor window.
 **Color** boxes	Open the color palette to customize individual track colors.

Metrics Charts Pane

The **Metric Charts** pane can accommodate an unlimited number of charts that visualize the metrics collected by the System Analyzer.

You can customize the System Analyzer **Metric Charts** pane as follows:

- Display any combination of metrics and instantly switch between different counter sets.
- Combine multiple metrics with the same measurement unit on one chart by dragging the charts onto each other. For example, the EU Active (%) and EU Stall (%) metrics can be combined since these metrics are represented as percentage. To do this, grab the metric by the tab

  and drag it onto another chart, or drag a metric from the **Metrics Control** pane and drop it onto an existing chart. To remove a track from the combined chart, click the

  Close button by the metric name.
- Change the appearance of each chart by switching between

  Bar Chart and

  Line Chart modes. If multiple metrics are combined on one chart in **Bar Chart** mode, their values are added to each other.
- Zoom in and out on the charts to get a closer look, using **Pause** mode.

Once you customize the chart area, System Analyzer saves your settings and restores them next time you run an analysis.



The pane is divided into two main areas:

1

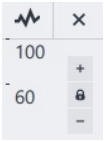
Chart Details panes

Each chart displays a graph for the selected metrics over time. The **Chart Details** pane of each chart provides the following metric data:

- Metric name and units
- Minimum value for the visible chart area ("L")
- Current value
- Maximum value for the visible chart area ("H")

2

Metric Charts and Controls

Timeline ruler	Correlate all the displayed metrics with the time line. Depending on the zoom-in level, the time intervals are displayed in seconds, milliseconds, microseconds, on nanoseconds.
 /  **Real-time Mode/Pause Mode** toggle	Pause the metric charts at any point. When Pause mode is enabled, the System Analyzer shows metric values for the preceding period specified by the Pause mode caching duration option of the Settings pane. In the Pause mode, metric collection continues in the background, and you can zoom in on the charts to get a closer look. Click the button again to resume metric display in real time.
 **Full Zoom Out** button	Zoom out to get the full view of the metric charts. This button is only available in Pause mode.
 **Individual chart controls**	Hover over any chart to show additional chart controls for each chart: • Bar Chart mode / Line Chart mode toggle • Close chart • Chart scale controls

See Also

Performance Metrics Reference

Monitor Real Time Performance with System Analyzer

Graphics Frame Analyzer UI Reference

This section describes the graphical user interface controls of Graphics Frame Analyzer:

Graphics Frame Analyzer Window: Open Frame Capture

Graphics Frame Analyzer Window: Multiframe View

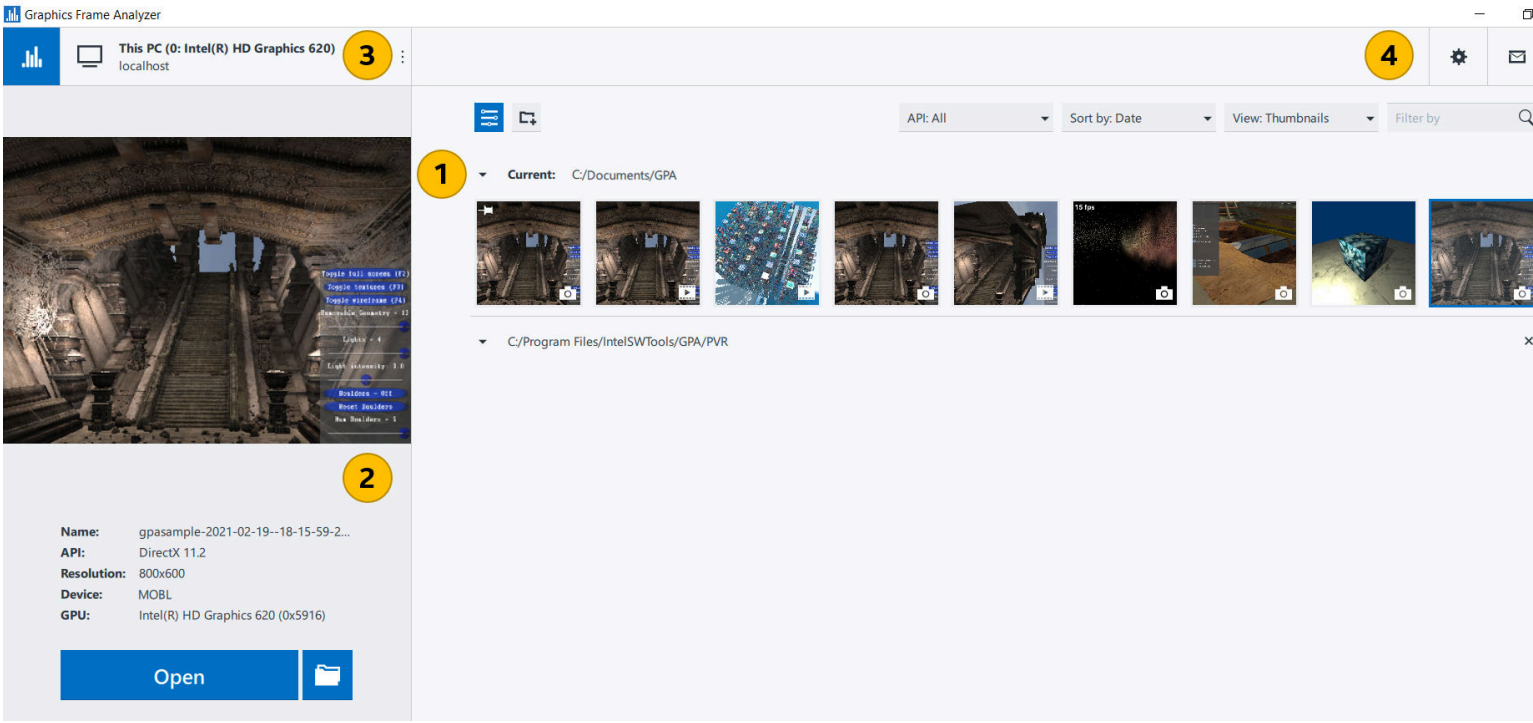
Graphics Frame Analyzer Window: Profiling View

Graphics Frame Analyzer Window: Dependency Viewer

Graphics Frame Analyzer Window: Open Frame Capture

To access this window, right-click the Graphics Monitor icon  in the taskbar notification area and select **Graphics Frame Analyzer**. You can also run **Graphics Frame Analyzer** from the Windows* Start menu or Apps view.

Use this window to open previously captured frame files and streams for profiling. You can preview all the frames and streams available on your host system:



1

Frame Capture Thumbnails pane

Select the capture you want to profile. By default, this field lists all frame and stream captures located in your Intel® GPA home directory `%USERPROFILE%\Documents\GPA`. You can sort and filter the files by the API type or date using the corresponding buttons or using the **Filter by** field. If all the capture files on your system are of the same type, the API type tabs are hidden.

The currently selected file capture is highlighted with a blue border and appears in the preview pane.

Capture file thumbnails are marked with the following icons:

- Frame 
- Stream 
- Invalid file 

To delete capture files, hover the mouse pointer over a capture file thumbnail and click the **Close** button in the top right-hand corner of the thumbnail. Confirm your choice by clicking **Yes** on the thumbnail.

You can pin favorite capture files to the top of the list by clicking the **Pin** button in the top left-hand corner of the thumbnail.

2	**Preview** pane Preview the selected file capture. This field shows the buffer image and the following file capture properties: • File name • API used • Screen resolution • Name of the device on which the file was captured • GPU details If you are trying to open a stream captured on a different GPU or using a different graphics driver version, you will see the warning that it can be a cause of unpredictable playback failure. To rename the selected capture file, you can click its filename and enter a new value. Click **Open** to open the selected capture file in the **Multiframe View** window for streams or in the **Profiling View** window for frames. To open the storage directory of the selected file capture, click the  button.
3	**Choose Target System** button • Select the target system or GPU against which to test the capture files by clicking this button and selecting the required system or GPU from the list. By default, Graphics Frame Analyzer uses your local host system (this PC). • To add a remote target system to the list, click this button and enter the IP address of the target system in the field on the bottom of the list. Optionally, you can set an alias for the remote system in the **Enter device name** field.
4	**Message** pane See the Intel® GPA messages or notifications, if there are any.

 **Run Graphics Monitor** button	Click to capture new frames and streams with Graphics Monitor.
 **Add Capture Files Location** button	Click to open additional folder with capture files. Capture files from the opened folder are listed under the **Current folder**. To change the **Current folder**, change the **Capture Files Location** in the Graphics Monitor **Settings**.
Open button	Open the currently selected capture file to start profiling.
 **Show in Folder** button	Open the storage directory of the selected capture file.
 **Settings** button	Manage Graphics Frame Analyzer settings:

	• Enable/disable checking for the newest available version of Graphics Frame Analyzer. **NOTE** During this check, Graphics Frame Analyzer collects information about your host system. No personally identifiable information is collected. If a newer version of the product is available, an exclamation mark in a red circle appears on the **Settings** button. To download the update, click the **Settings** button and select the available version from the context menu. • Change color schema and user interface scale
 **Send feedback** button	Open a form to send feedback to the Intel® GPA technical support.
 **Help** button	Open Intel® GPA online documentation.

See Also

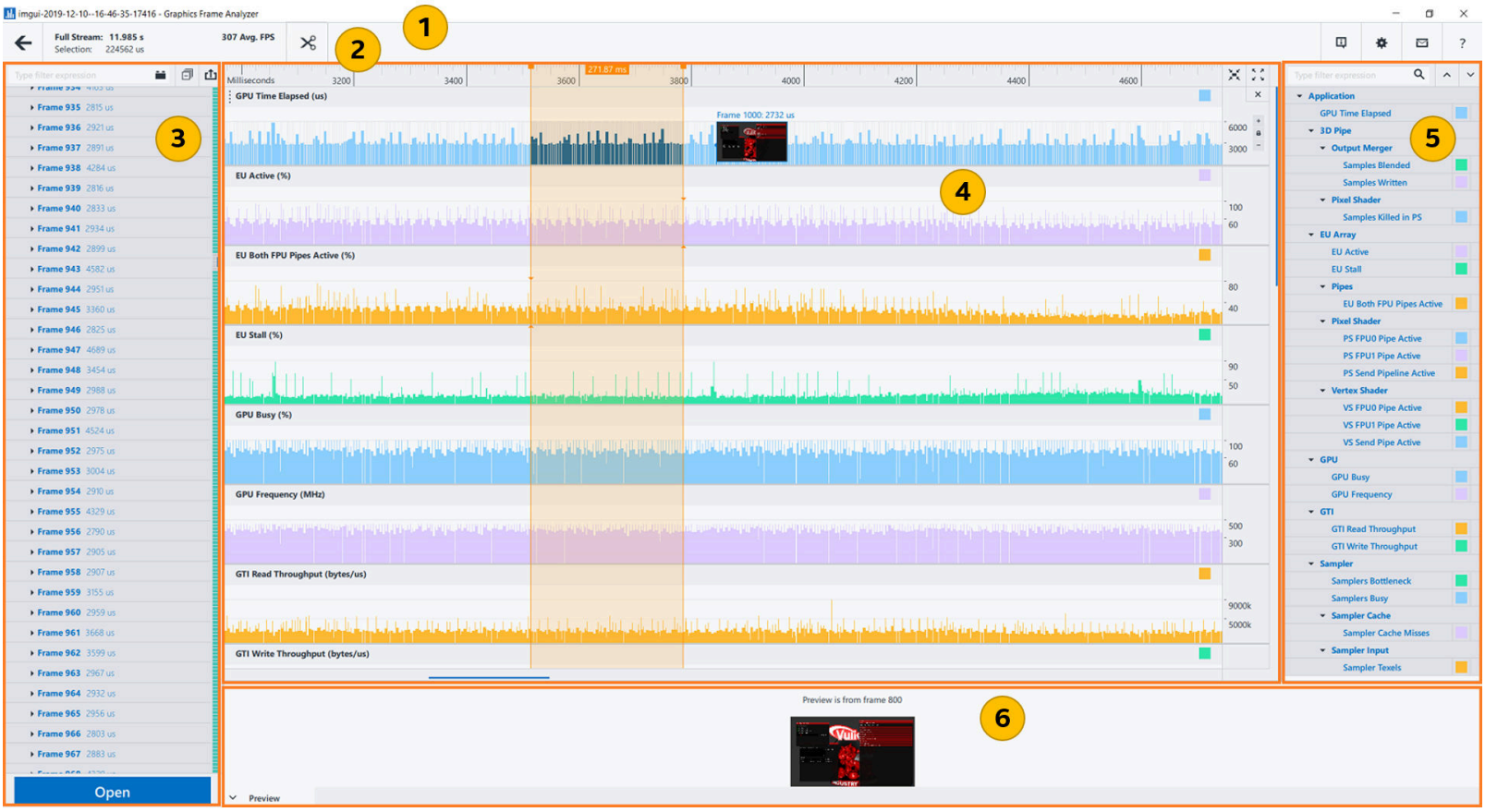
Analyze GPU-bound Applications with Graphics Frame Analyzer

Graphics Frame Analyzer Window: Multiframe View

To access the **Multiframe View** window, select a stream for analysis in the **Open Frame Capture** window, and then click **Open**, or double-click on a stream preview.

NOTE

- Multiframe View is available for Vulkan, DirectX 11 (preview) and DirectX 12 applications.
- **Stream** capturing is not supported on Linux* OS.



1	**Stream Title** bar Displays the opened stream file name. The file name consists of the application name, capture date and time in the following format: YYYY_MM_DD_HH_MM_SS.
2	**Main toolbar** Displays the opened stream data, notifications, shows software and hardware information, provides access to general and appearance settings, restores the default view, contains **Feedback** and **Intel® GPA User Guide** buttons.
3	**Stream Content** pane Shows frames within the opened stream.
4	**Stream Viewer** pane Visualizes stream metrics data, zooms to the selection and zooms out.
5	**Metrics** pane Shows metrics collected for the opened stream - you can choose which ones to show on the Stream Viewer pane.
6	**Preview** pane Displays screenshots captured for the opened stream. To disable or enable screenshot capturing, go to the Graphics Monitor Options**Stream** tab.

NOTE "No preview is available for current selection" message may appear for the first frames, if you turn on the **Create screenshot every X frame/ millisecond** toggle buttons in Graphics Monitor **Options**.

Main Toolbar

 **Back** button	Return to the Open Frame Capture window.
Stream Overview pane	Get statistics for the opened stream: • **Full Stream:** Duration of the full stream, in seconds. • **Selection:** Sum of durations of the selected frames, in microseconds. • **FPS:** Frame Rate for a selected range of frames, in frames per second (# of frames / **Selection**, in seconds) • **% busy:** The relation of the sum of the selected frames duration to the total selected range time. Indicates the actual GPU load.
 **Trim Stream** button	Cut the selected frames from the stream and save them into a new stream. This can be useful when a long stream only contains a few frames of interest. Trimming such a stream can save both disk space and processing time. Enable the **Multi-Frame Profiling View** (DirectX Tech Preview) toggle in **Settings** to enable the **Trim Stream** button.
Message pane	See messages and non-critical errors, if there are any. **NOTE** If more than one message is open, use the **Up** or **Down** arrows to navigate between messages.
 **Settings** button	• **Multi-Frame Profiling View (DirectX 11 Tech Preview)** toggle. Enables Multi-Frame Profiling View as a preview feature for DirectX 11 applications. For DirectX 12 applications, Multi-Frame Profiling View is always enabled by default.

	NOTEMulti-Frame Profiling View for DirectX 11 is a tech preview feature and may not work as expected. Please, share your feedback using the **Send feedback** button found in all Intel® GPA tools.

	• Change the color scheme of the Graphics Frame Analyzer interface:
	  • Light color scheme (default)
	  • Dark color scheme
	• Change the size of the Graphics Frame Analyzer screen and the screen elements:
	  • Small
	  • Medium
	  • Large (default)

 **Send feedback** button	Send feedback to the Intel® GPA technical support.
 **Help** button	Open the Intel® GPA online documentation.

Stream Content Pane

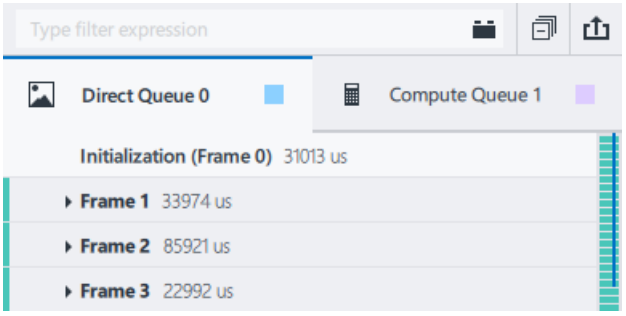
Type Filter Expression field	Filter frames by number or duration. Save previous search keywords. You can use the **Type Filter Expression** field to find specific frames by running Graphics Frame Analyzer plugins.
 **Graphics Frame Analyzer plugins** button	Open the list of the available Graphics Frame Analyzer plugins, which help you to find specific frames. For example, **Frame Search Assistant** plugin searches for a frame of the specified duration.
 **Collapse All Groups** button	Close all the groups of events and commands for each frame in the **Frame List** tab, if available.
 **Export Frame Log** button	Save the frame log in CSV format. You can also save the Frame log by pressing **Ctrl+S**.
Frame List tab	Get the list of frames of one rendering context/ queue in the CPU execution order. The number of **Frame List** tabs depends on the number of rendering contexts/queues in the opened stream. To profile an individual frame, double-click the required frame; or click the required frame, and then click **Open**.

NOTE The first frame in a stream cannot be opened for profiling.

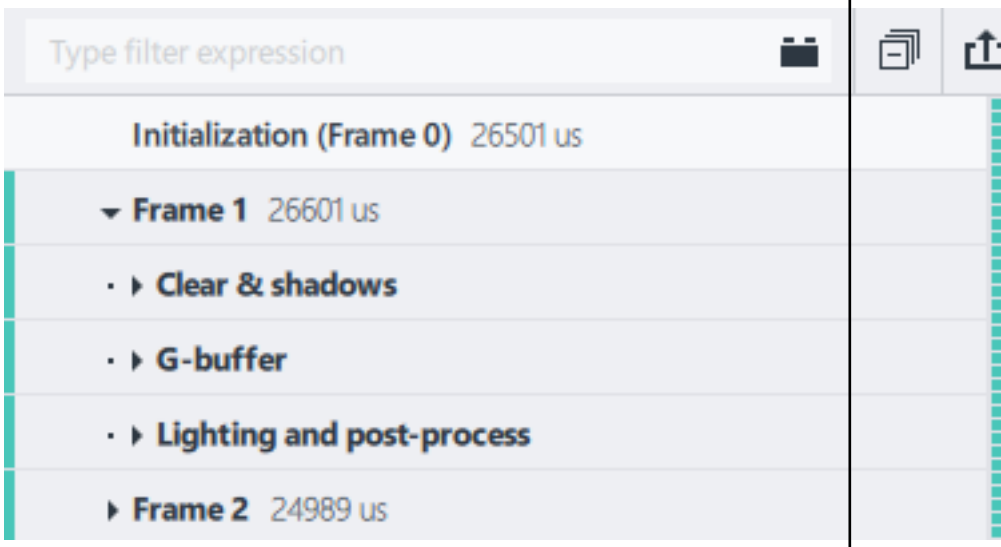
For Vulkan-based application streams, the **Multiframe View** shows subframes:

- Queue Submit calls. Submit calls can be selected and opened in the **Profiling View**.
- Queue Present calls. Present calls can be selected.

For applications based on Microsoft DirectX* 12 that use several command queues, the frame list shows each queue in a different tab:



For DirectX 12 applications with command queues that are instrumented with events, each frame can be expanded to view individual events and their details:

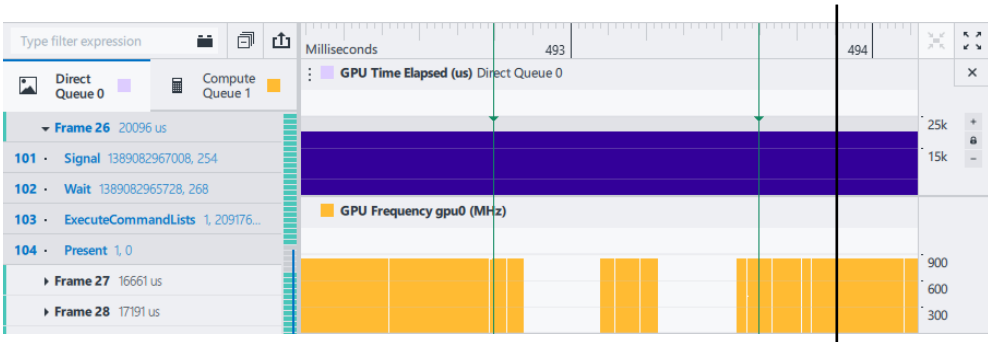


Open button

Open the selected frame in the Profiling View window. For Vulkan-based applications, you can select several frames and subframes.

Stream Viewer Pane

Time line ruler	Correlate all the displayed frames with the time line. The range of the time line corresponds to the length of the stream in the current zoom level. Depending on the zoom-in level, the time intervals are displayed in seconds, milliseconds, microseconds, or nanoseconds.
 **Zoom to Selection** button	Zoom into the time range of selected frames.
 **Restore zoom level** button	Restore the default zoom level.
Context/Queue bar chart	Visualize individual frames, their contribution to the overall stream time in the CPU execution order, and keyframes. Keyframe is a saved state of an application, from which you can play back the captured stream. • Select frames in the **Frame List** tab by clicking on one frame or pressing CTRL and clicking on several frames. • Select frames in the **Stream Viewer** pane by clicking the bars on the **GPU Time Elapsed** track, or by pressing the left mouse button and dragging the mouse horizontally on any track. Bars for selected frames are highlighted on the **Context/Queue** bar chart. If you only select a part of a frame, the corresponding bar is hatched. Keyframes are shaded grey. By default, streams are visualized on the **GPU Time Elapsed** track. You can visualize other GPU metrics by selecting a sampling interval other than **Frame** in the **Metrics** tab of the Graphics Monitor **Options** before you start an application on **Stream** mode. When you set a sampling interval other than **Frame**, metrics are collected system-wide with the chosen frequency. For DirectX 12, calls from the queues are also shown on the bar chart, represented as markers with downward-pointing triangles. Each marker represents a point in time when a certain call was made. All markers extend downwards, which enables you to correlate calls to the overall frame and GPU metrics, such as **GPU Busy (%)**.



You can also click on each call from the **Frame List** tab to highlight the corresponding frame and marker on the bar chart.

Individual chart controls

Change the vertical scaling of the bar chart by increasing or decreasing the relative height of the bars using the

or

buttons respectively. By default, all bars are displayed in their original height.

To enable or disable the vertical auto-scaling, click the



/



toggle button. By default, the vertical auto-scaling is enabled.

Metrics Pane

Type Filter Expression field

Filter stream file tracks by name or parameters. Save previous search keywords.

Collapse All Groups button

Close all unfolded track groups.

Expand All Groups button

Unfold all track groups.

Metrics list

See the list of metrics collected for the captured stream file. Click tracks from the **Metrics** to add them to the **Stream Viewer**.

NOTE To change track color or event color in the **Stream Viewer**, use the **Track Color Modification** button (the colored square to the right of the metric).

See Also

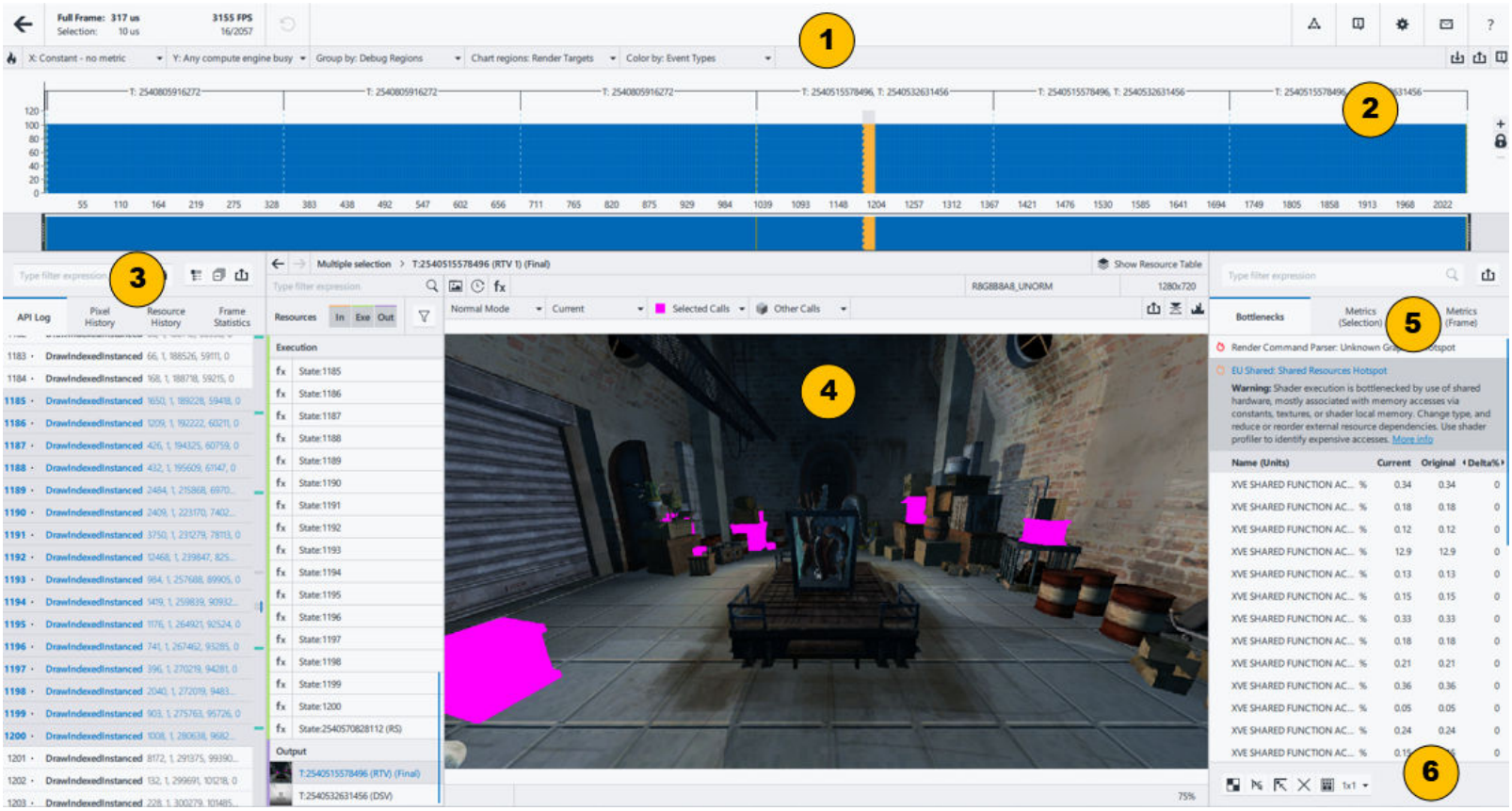
Analyze GPU-bound Applications with Graphics Frame Analyzer

Graphics Frame Analyzer Window: Profiling View

*Use **Profiling View** to identify potential performance bottlenecks and possible improvements in your application by performance quick 3D pipeline experiments.*

You can access the **Profiling View** window in several ways:

- Select the frame for analysis in the **Open Frame Capture** window and click **Open**
- Select a frame in the **Open Frame Capture** window and double-click on the frame preview
- Select a frame or a frame sequence in **Multiframe View** and click **Open**



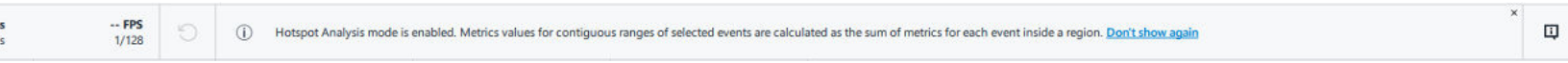
1	Main Toolbar Displays the opened frame data, notifications, opens the Intel® GPA Online Help, restores the default view, and changes the Intel® GPA color scheme.
2	Visualization pane Displays the sequence of captured events in a graphical format based on GPU metrics.
3	API Log pane Displays the list of all graphics API functions used in the frame in the GPU execution order and parameters of each function.

4	Resources pane Displays all resources used by the selected graphics API functions.
5	Metrics pane Displays metrics information for the selected graphics API functions.
6	Experiments pane Discovers potential performance bottlenecks in your application by modifying render states of the graphics API.

See Also

Analyze GPU-bound Applications with Graphics Frame Analyzer

Main Toolbar



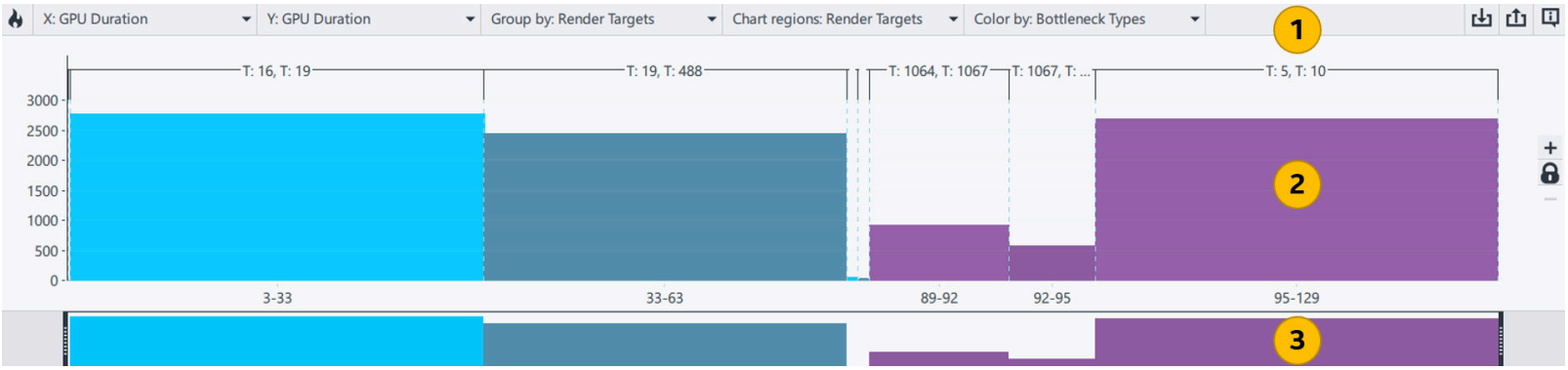
 **Back** button	Return to the **Open Frame Capture** window in case of a frame, or to the **Multiframe View** in case of a stream. You can use the Backspace key instead.

Frame Overview pane	Get the following statistical information for the analyzed frame:
	• Performance statistics, if available for your target system:
	  • Duration of the full frame, in microseconds (us).
	  • Duration of the selected events, in microseconds (us).
	  • Theoretical frame rate for the whole frame, in frames per second (FPS), based on the full frame duration.
	  • Time difference between the original and the modified frame. This value is shown during experiments with render states or in case of frame resources modifications.
	• The total number of events in the frame, and the number of events currently selected in the **Main bar chart** and **API Log** pane.

 **Revert All Modifications** button	Turns off all performed experiments.
Message pane	Receive messages and non-critical errors. **NOTE** If more than one message is open, use the **Up** or **Down** arrows to navigate between messages.
 **Frame Info** button	Get the following properties of the analyzed frame: • Intel® GPA version • Filename • App name • API used • Frame resolution • Name of the system on which the frame was captured • Hardware and software of the system on which the frame was captured

	• Hardware and software of the system on which the frame was played back
Settings button	General Settings • Import new Python plugins or reload existing into the Graphics Frame Analyzer. Appearance Settings • Restore the default user interface layout • Change the color scheme of the Graphics Frame Analyzer interface • Change the size of the Graphics Frame Analyzer screen and the screen elements
Send feedback button	Send feedback to the Intel® GPA technical support.
Help button	Open the Intel® GPA online documentation.

Visualization Pane



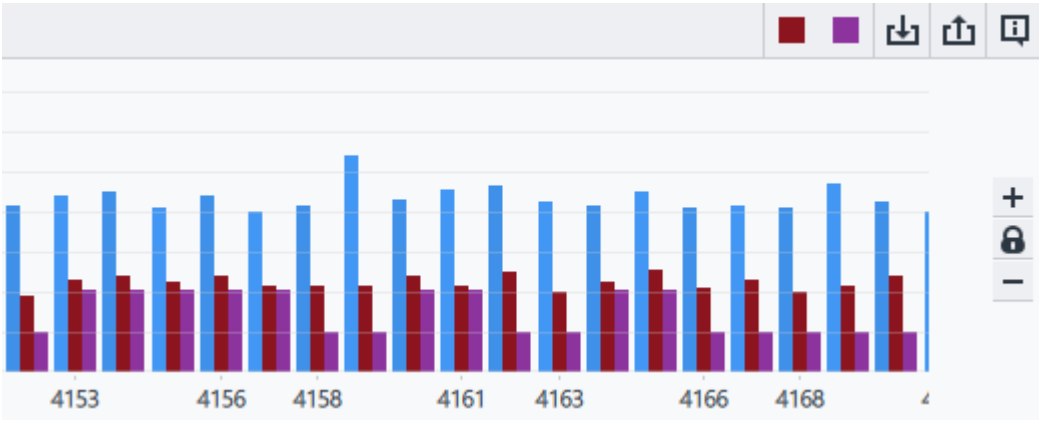
1	Visualization Settings Toolbar Enables you to adjust the manner of displaying events in the **Main** bar chart and **API Log** pane.
2	Main Bar Chart Visualizes performance in the frame or selected events over time, if metrics are available for the target system.
3	Frame Navigation Bar Chart Enables you to navigate along the frame and select events that are shown on the **Main** bar chart.

Visualization Settings Toolbar

Use the **Visualization Settings** toolbar to configure the **Main** bar chart mode:

Advanced Profiling Mode button	Enables a hotspot analysis that discovers performance bottlenecks in individual events and allows you to group the events by the causes of bottlenecks and by pipeline states. This button becomes active, if the system supports the full list of Intel performance counters. When you click the **Advanced Profiling Mode** button, data representation changes in all Graphics Frame Analyzer components:

	• The bar charts shows grouped calls, with the longest group on the left side of the bar charts. • The **API Log** tab provides a short description of the discovered bottleneck and of the pipeline state. • The **3D Pipeline** tab of the **Metrics** pane provides detailed descriptions of discovered bottlenecks and potential ways to fix them. To see the description, click on blocks with bottlenecks that are underlined by red or yellow. In the **Advanced Profiling** mode, calls can be grouped by: • **Bottleneck/State** (default) - calls are simultaneously grouped by two parameters: bottleneck description and pipeline state. • **Bottleneck** - calls are first grouped by bottleneck descriptions, followed by pipeline state. • **State** - calls are first grouped by pipeline state, followed by bottleneck description. **NOTE** If the **Advanced Profiling** mode is enabled, the **Full Frame** parameter in the **Frame Statistics** pane shows the aggregate duration of events, ignoring parallel execution.
X drop-down button	Enables you to choose the available metrics for the X axis to visualize specific aspects of performance in the frame.
Y drop-down button	Enables you to choose the available metrics for the Y axis to visualize specific aspects of performance in the frame.
Group by drop-down button	Groups the events visualized on the bar charts by: • **Custom Regions** - enables you to group and name regions in the **API Log** pane. To learn how to create custom regions, see the **API Log** pane description. • **Draw Calls** - no grouping, each bar corresponds to a single event • **Render Targets** - groups of events combined by bound render targets • **Shader Sets** - groups of events combined by bound shaders (applies to DirectX* 11, DirectX 12, and Vulkan frames) • **Command Lists** - groups of events combined by command lists and bundles (applies to DirectX 12 and Vulkan frames) • **Pipeline States** - groups of events combined by bound pipeline state objects (applies to DirectX 12 and Vulkan frames)
Chart Regions drop-down button	Enables a markup on the top of the **Main** bar chart to highlight the same groups that are available in the **Group By** drop-down list.
Color By drop-down button	Colors bars on the **Main** bar chart according to the types of events or bottlenecks. The button is available only if the system supports the full list of Intel performance counters.
 **Import** button	Imports the saved metrics data onto the chart for comparison with the current metrics values. You can only import metrics data for the same frame file with the same set of metrics selected. If you import one or more CSV files into the bar chart, the X-axis is reset to a constant metric.
 **Export** button	Saves metrics data in the CSV format. • You can import and export metrics data in CSV format to compare performance results for the same frame measured with different drivers or hardware settings:



- You cannot compare metrics data collected on different platforms with different sets of metrics.

Hover over the



rectangles above the chart to view the *.csv file names and associate the color code in the bar chart with the imported files. To remove the imported metrics data from the chart, click the corresponding rectangle.

Show Color Legend button

Explains the color legend used in the **Main** bar chart for events and bottlenecks. Legend for bottlenecks is available only if the system supports the full list of Intel performance counters. By default, the bars on the **Main** bar chart are colored by event types.

Main Bar Chart

The **Main** bar chart enables you to review individual events or groups of events and their contribution to the overall frame time.

An event is any graphics API function that produces a GPU activity, such as "draw" calls or "clear" calls. Once you load the capture file, events are visualized as vertical bars, which can be of one of the following types:

- Draw Call (colored Dark Blue)
- Dispatch Call (colored Purple)
- Copy Call (colored Cyan)
- Clear Call (colored Green)
- Sync Call (colored Yellow) - DirectX 12 and Vulkan frames
- Render Pass Call (colored Magenta) - DirectX 12 and Vulkan frames
- Raytracing Call (colored Pink) - DirectX 12 frames

NOTE If you filter the API calls in the **API Log**, the excluded calls are displayed in gray.

Events are shown in the GPU execution order starting from the left. By default, the height of an event bar indicates the duration of the event execution on the GPU. GPU time is measured in microseconds.

You can change the vertical scaling of the bar chart by increasing or decreasing the relative height of the bars using the

or

buttons respectively. Alternatively, you can hold **Ctrl** and scroll up or down to re-scale the bars.

To enable or disable the vertical auto-scaling, click the

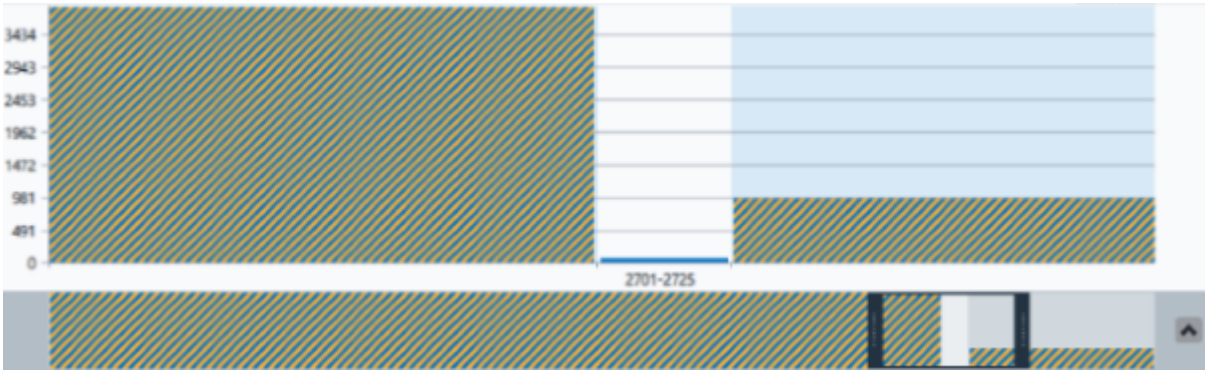
/

toggle button. By default, the vertical auto-scaling is enabled.

You can identify all events using a specific texture, buffer, shader, or render target selected in **Resource List**. You can see a colored stripe below each event using the selected resource. The color of the stripe depends on the group of the resource: orange for **Input**, green for **Execution**, and purple for **Output** resources. Half-transparent colored stripes indicate that the resource is used by some of the events in the render target.

To analyze metrics data and resources for specific parts of the frame, select a single event or a group of events in the bar chart. The selected bars are highlighted in orange. You can use the Ctrl+Z/Ctrl+Y keyboard shortcuts to return to the previously selected event and back.

If you select one or more events in the chart and group them by any criterion, for instance, by render targets, this partial selection within the render target is marked with orange hatching:



Frame Navigation Bar Chart

Use the **Frame Navigation** bar chart to navigate along the frame and to zoom into particular events in the **Main** bar chart.

To change the horizontal scaling of the **Main** bar chart, stretch or shrink the **Frame Navigation** bar chart scrollbar by dragging its boarders. The **Frame Navigation** bar chart provides an overview of the entire frame, while the scrollbar reflects the part of the frame currently displayed in the **Main** bar chart. If the X-axis represents a non-constant metric, you can double-click the slider to toggle between the full frame view and the currently selected part.

See Also

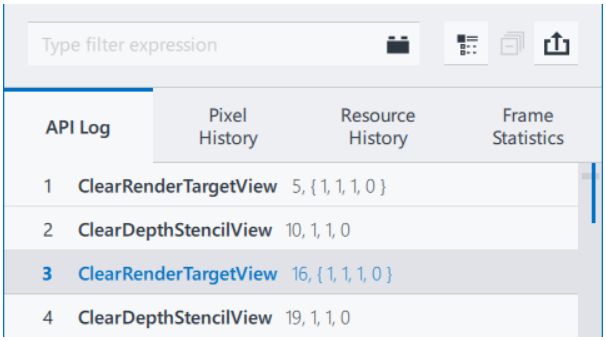
Use Advanced Profiling Mode

API Log Pane

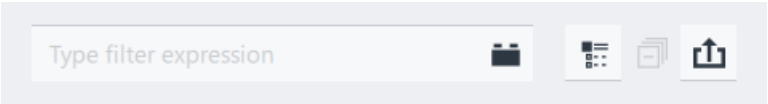
The **API Log** pane consists of the following items:

- **API Log** toolbar
- **API Log** tab
- **Pixel History** tab
- **Resource History** tab

- **Frame Statistics** tab



API Log Toolbar



Use the **API Log** toolbar to manage the **API Log** pane:

- **Type Filter Expression** field filters graphics API functions by name or parameters. Saves previous search keywords. You can use the **Type Filter Expression** field to find specific events by running

  Graphics Frame Analyzer plugins.
- 

  Full API Log button displays in the **API Log** tab all functions used in the frame, in the call order.

  NOTE By default, the **API Log** tab shows only graphics API functions that produce any GPU activity, such as draw calls or clear calls. These graphics API functions are also called "events".
- 

  Collapse All Groups button closes all the groups in the **API Log** tab, if available.
- 

  ExportAPI Log button saves the API log in CSV format. You can also save the API log by pressing Ctrl+S.

API Log Tab

Use the **API Log** tab to do the following:

- Review the list of all functions, in the GPU execution order
- Review parameters passed to each function
- Create custom regions

NOTE

- To facilitate debugging, you can specify object labels for various resources in your application, such as textures, buffers, shaders, by using special functions of graphics APIs.
- Graphics Frame Analyzer displays object labels in the **API Log** pane and **Resources** pane.

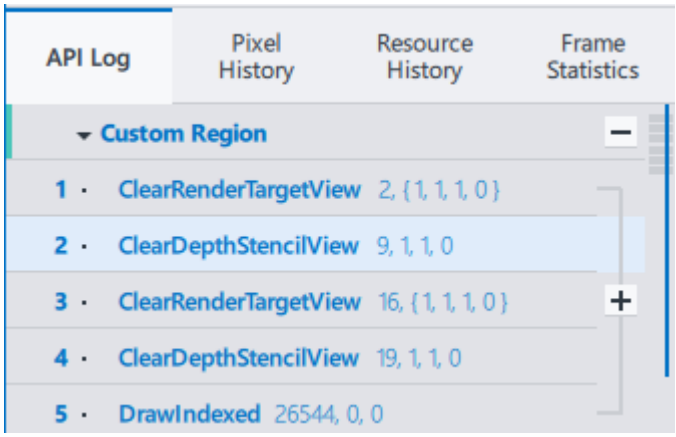
When you select an event in the **Main** bar chart, Graphics Frame Analyzer highlights the corresponding functions in the **API Log** tabs. For easier navigation, Graphics Frame Analyzer marks the functions that require your attention with colored bars. You can click the bar to navigate directly to the marked function:

- Green bars denote user-defined markers inserted in the application code or regions with the same render target or shader sets. Grouping can be selected using the **Group By** drop-down button on the **Visualization Settings** toolbar.
- (OpenGL-specific) Orange bars denote functions that caused warnings.
- (OpenGL-specific) Red bars denote functions that caused errors.

> **NOTE** When you select a function in the list, Graphics Frame Analyzer updates all other panes in **Profiling View** to reflect the state of the frame for the function call. You can use **Up/Down** arrow keys or **Page Up/Page Down** keys to navigate between the displayed functions.

To create a custom region, do the following:

1. In the Graphics Frame Analyzer **Main** toolbar, select **Custom Regions** from the **Group By** drop-down list.
2. In the **API Log** pane, select an event or a group of sequential events, and click **+**.



3. Enter a name for the created custom region.

   To delete the created custom region, click **-**. To add a nested list inside the created custom region, click **+**.

Pixel History Tab

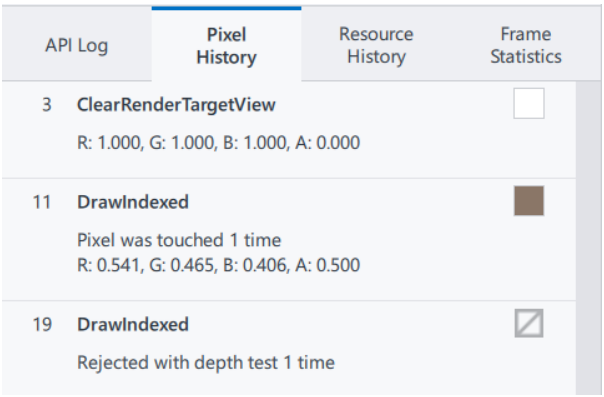
The **Pixel History** tab shows only the events that contribute into the pixel specified in **Resources** pane.

Pixel history analysis is supported for DX11 Frame/Stream on Windows, DX12 Frame (Legacy) on Windows, and OpenGL Frame on Ubuntu.

To analyze pixel history, select the render target in **Resource List** and click on the pixel you want to analyze in **Resource Viewer**. Graphics Frame Analyzer marks the selected pixel with a crosshair and filters the API calls to display only those events that affected this pixel. The colored boxes next to each call show the color written into frame buffer after the API call execution. If the call has been rejected at one of the pipeline stages, the



box is displayed next to the call.



You can use **Up/Down** arrow keys or **Page Up/Page Down** keys to navigate between the displayed functions.

Resource History Tab

The **Resource History** tab shows only the events that use the resource selected from **Resource List**. To enable the resource history, you can:

- Click **Show All Resources** button in the **Resources** pane, double-click any resource in the **Resource List**, and click the

  button
- Choose any resource from the **Select Resource** drop-down list in the **Resource History** tab

You can use **Up/Down** arrow keys or **Page Up/Page Down** keys to navigate between the displayed functions.

Frame Statistics Tab

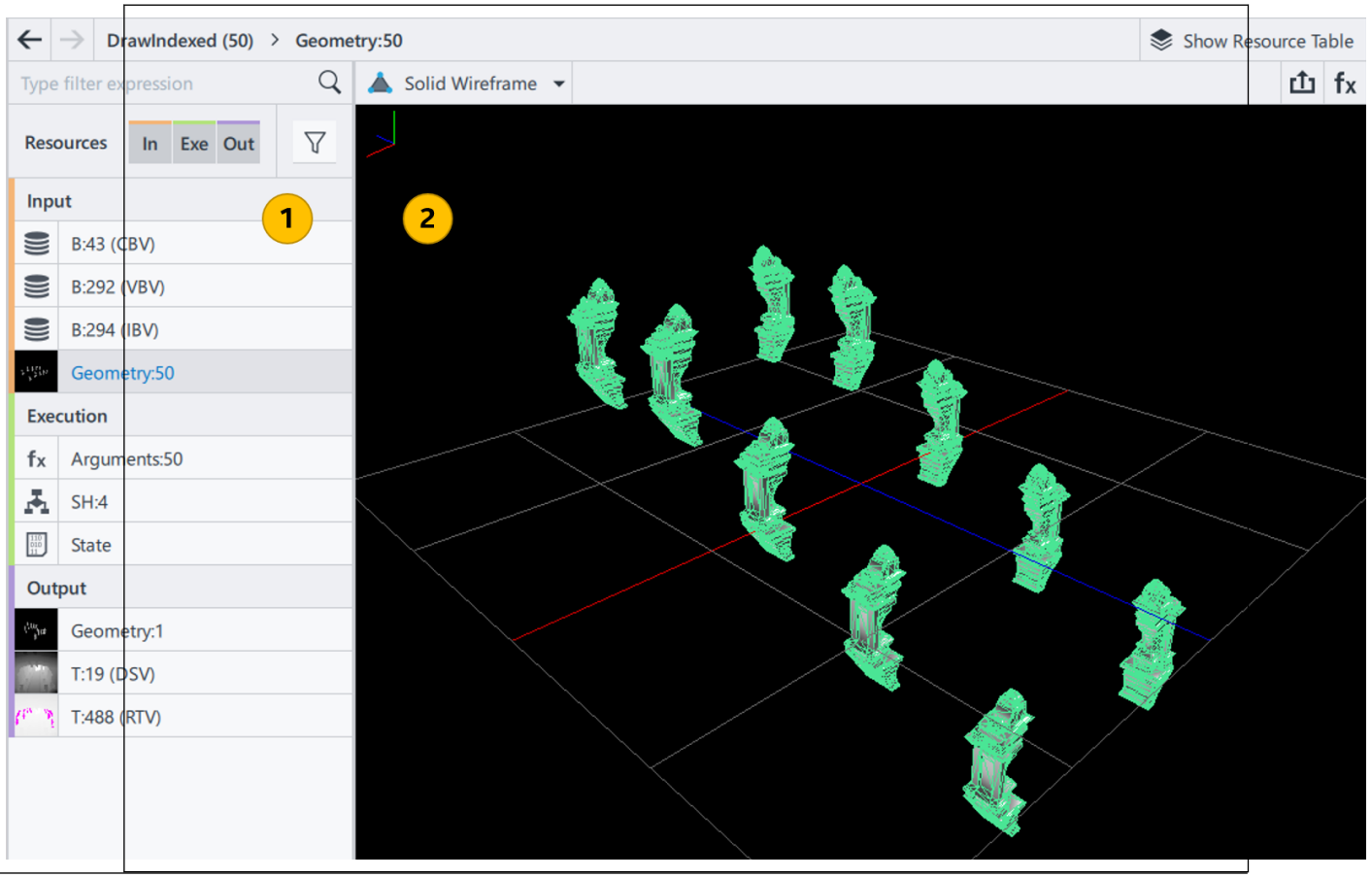
The **Frame Statistics** tab shows the number of calls of each type in the frame.

You can use **Up/Down** arrow keys or **Page Up/Page Down** keys to navigate between the displayed functions.

See Also

Graphics Frame Analyzer Plugins

Resources Pane



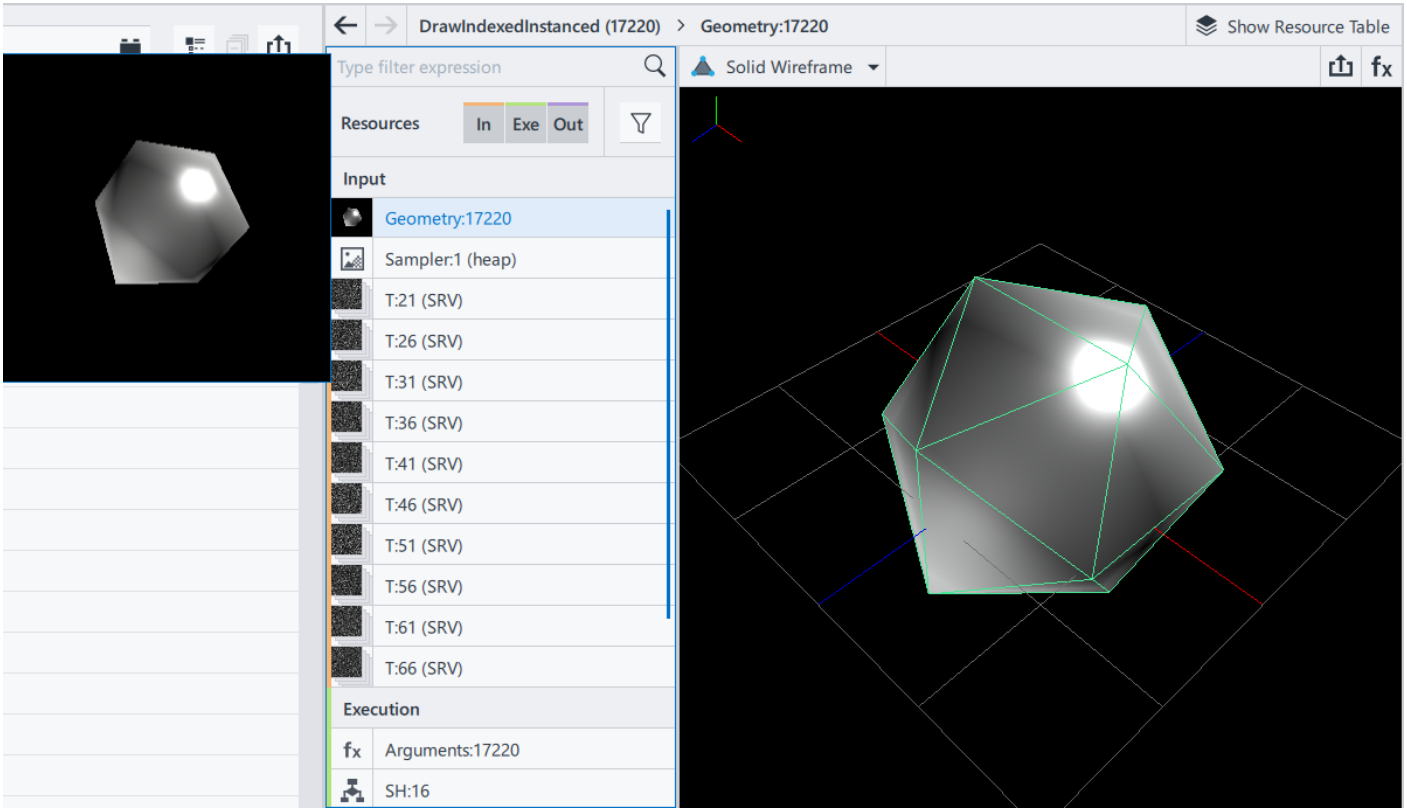
1. Resource List

   Displays all resources used by the selected graphics API functions.

2. Resource Viewer

   Displays details about the resource selected in **Resource List**.

Resource List



All resources are grouped by **Input**, **Execution**, and **Output** stages.

By default, Frame Analyzer shows all resources attributed to the current selection. Use the



Filter menu to filter resources by type:

- Geometry
- Textures
- Shaders
- Data buffers
- Graphics/Compute Pipeline States
- Samplers
- API call arguments

To filter resources by name, type the desired name or parameter into the **Type filter expression** field.

Some resources support previews. To see the preview, hover over the resource thumbnail in the **Resource List**.

Use



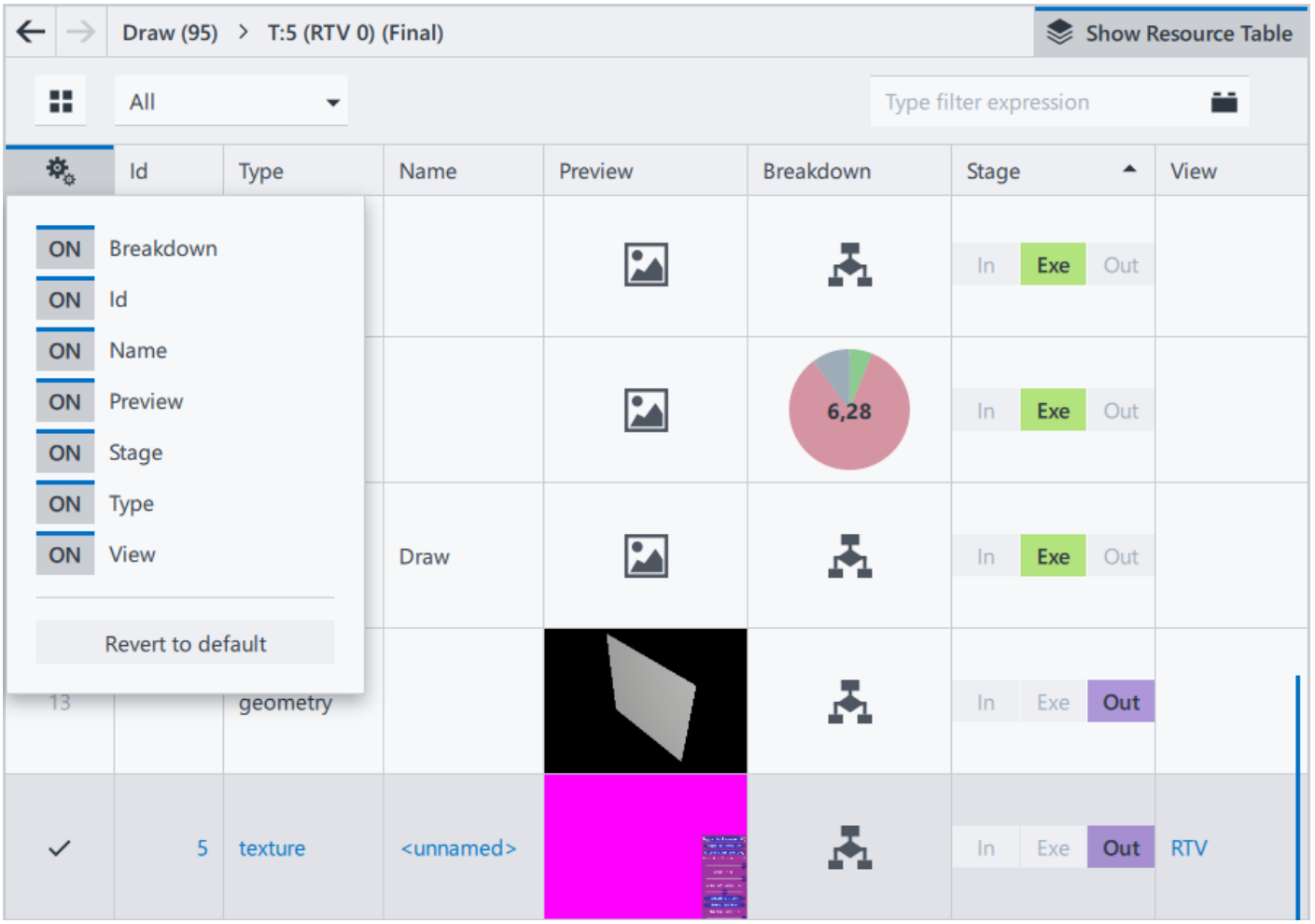
Back and



Next buttons to navigate back and forth between recently selected resources and events.

For textures, shaders, or geometry, you can choose the following view types:

- **Table View**



To see resources in a tabular view, click the



Show Resource Table button.

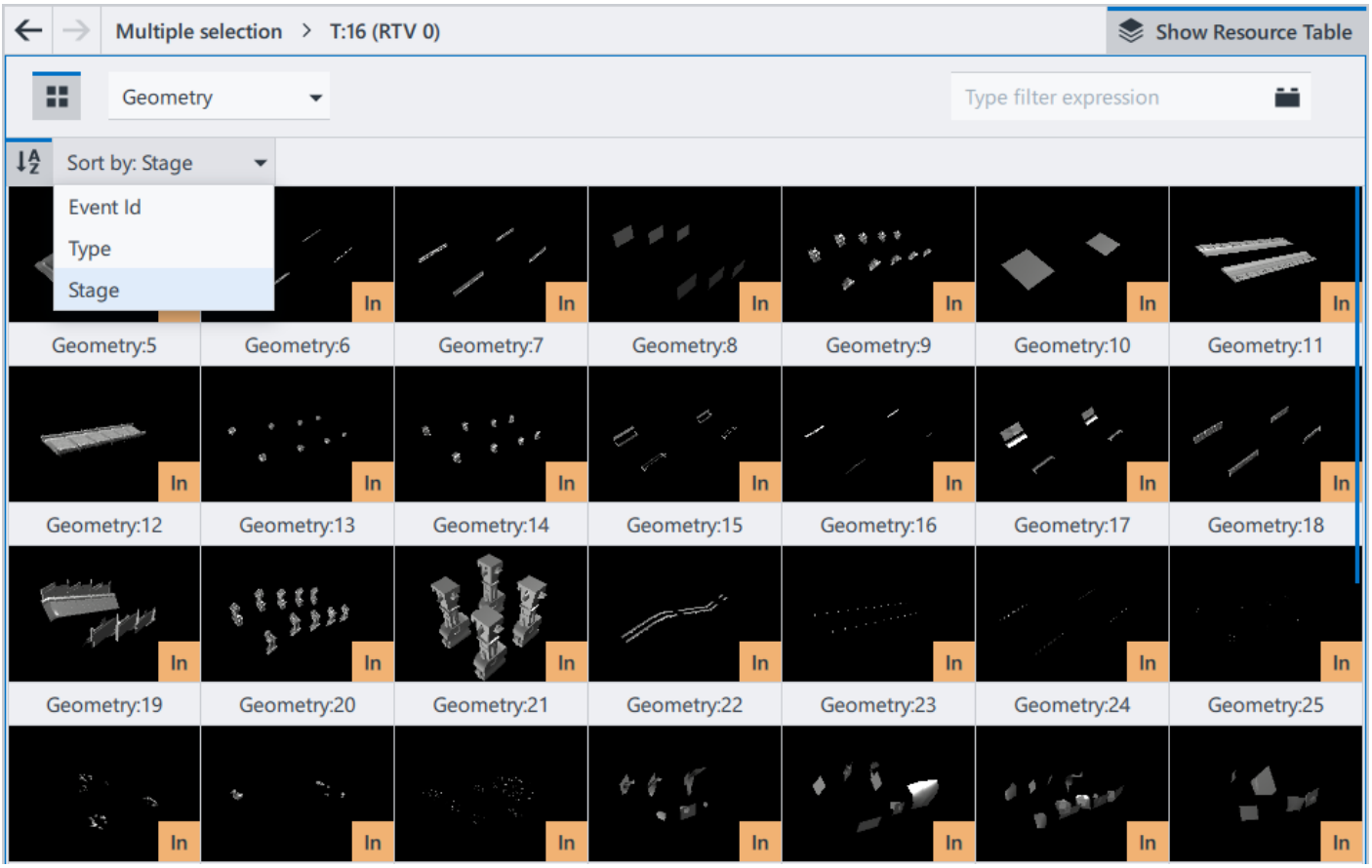
To configure table columns, click the



button.

To select a resource and open it in the **Resource Viewer**, click the row with the required resource.

- **Tile View**



Tile view shows only resource thumbnails. To open tile view, click the



button. Resources can be sorted by resource type, name, ID, or stage.

To select a resource and open it in the **Resource Viewer**, click the row with the required resource.

Resource Notations

Resource names, except texture and buffer names, are abbreviated and shown in the following format:

Resource List and **Resource Viewer**

<resource type>:<resource ID> [<user-defined name>]

Texture names and buffer names are shown in the following format:

<resource type>:<resource ID> (<view type><view ID>)

Resources Names for DirectX API

Name Component	Description
<resource type>	Can be B (buffer) or T (texture).
<resource ID>	Any positive integer. 0 is an exception used to identify a synthetic DirectX 12 root constant resource.
<view type>	Depends on the resource type. Texture • RTV - render target view

Name Component	Description

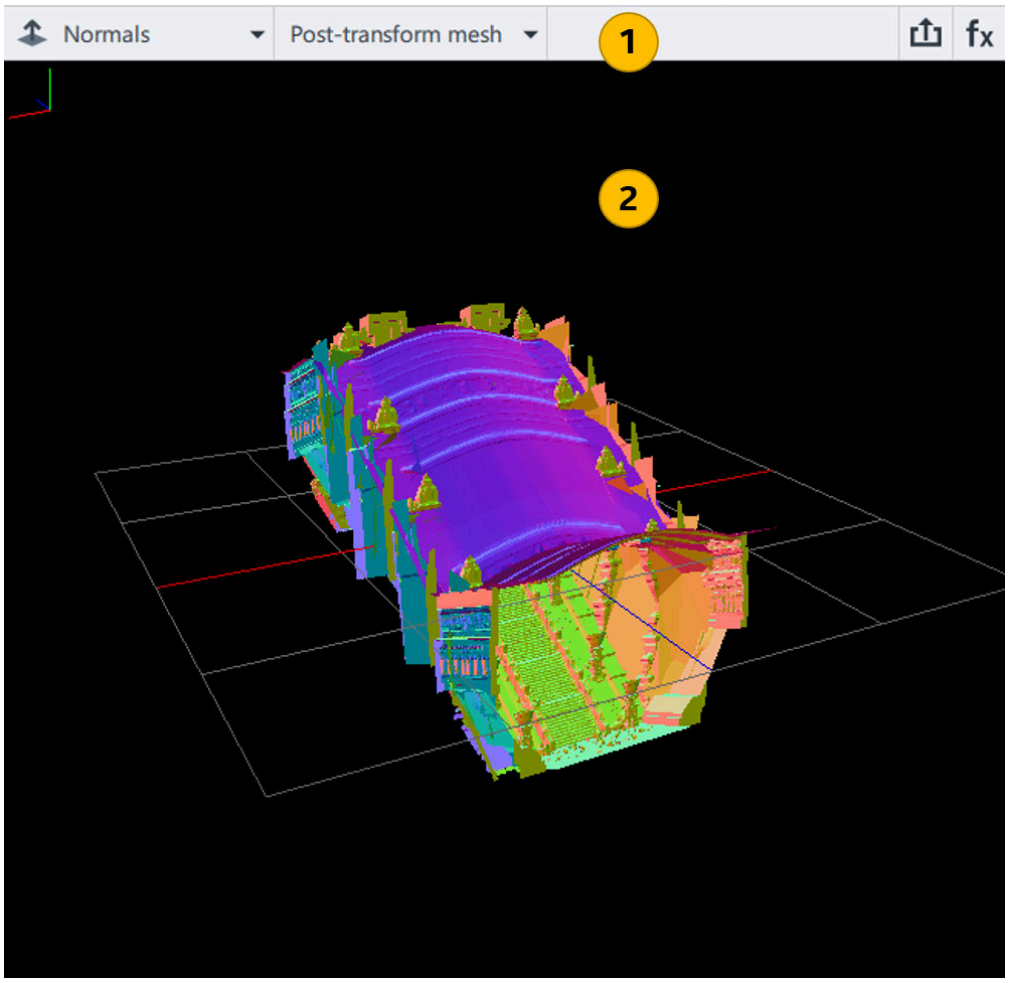
	• DSV - depth-stencil view
	• UAV - unordered access view
	• SRV - shader resource view
	• genmips - subresources generated automatcally by GenerateMips call
	• copy - subresources used in copy calls
	
	Buffer
	
	• VBV - vertex buffer view
	• copy - subresources used in copy calls
	• args - draw and dispatch indirect arguments
	• IBV - index buffer view
	• count - query results buffer
	• UAV - unordered access view
	• SRV - shader resource view
	• CBV - constant buffer view
	• Shader binding table view types:
	  • raygen - shader binding table for ray generation shaders
	  • hit - shader binding table for hit shaders
	  • miss - shader binding table for miss shaders

<view ID>	Any non-negative integer number. Continuous numbering is used, that is a view ID value does not depend on a view type.

Resource Viewer

Resource Viewer displays details of the resource selected in **Resource List**. **Resource Viewer** layout depends on the resource type.

Resource Viewer: Geometry



1

Geometry Viewer panel

Visualization Mode drop-down list - switches between view modes to analyze various geometry aspects of the selected geometry object. There are the following visualization modes available:

Solid (default) - view the object as a solid model.

Wireframe - view a wireframe model to inspect the backside of the object and to understand how it is built from triangles.

Solid Wireframe - view a wireframe model on top of the solid model.

Normals - view a normal map for your object to analyze how the normal vectors are plotted.

Output Geometry View Mode drop-down list is available for output geometry. It toggles between the **Post-transform mesh** mode and **Screen space view** mode.

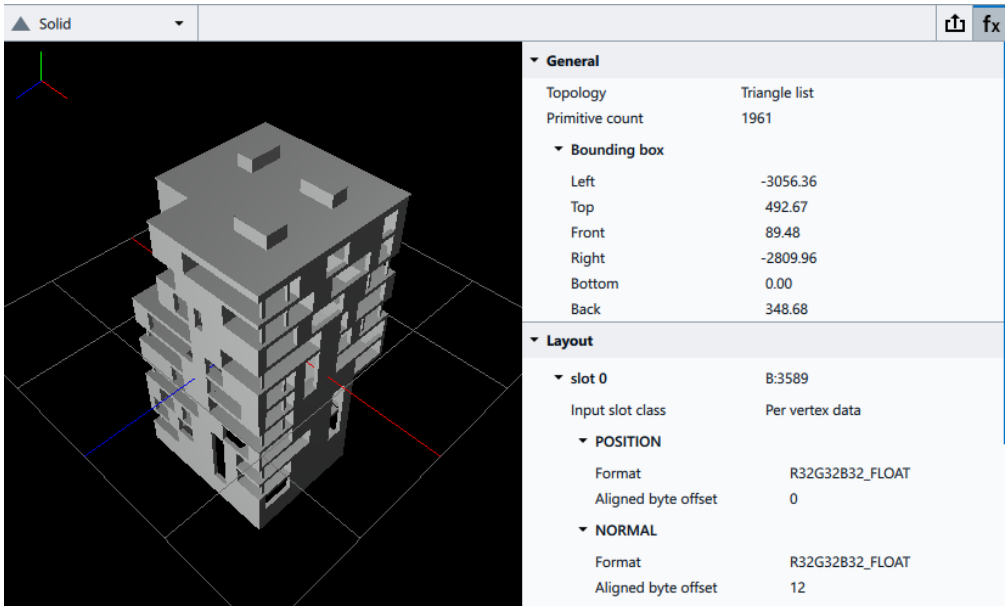
- **Post-transform mesh** mode displays an object with transformed vertexes.
- **Screen space view** mode displays a projected object to a 2D screen space.



Save Geometry to File button exports geometry data in the Wavefront OBJ format.



Geometry Layout button (specific for DirectX 11, DirectX 12, and Vulcan) displays geometry details. The **Geometry Layout** pane contains general information, such as topology and bounding box, as well as the attached vertex buffers and their layout:



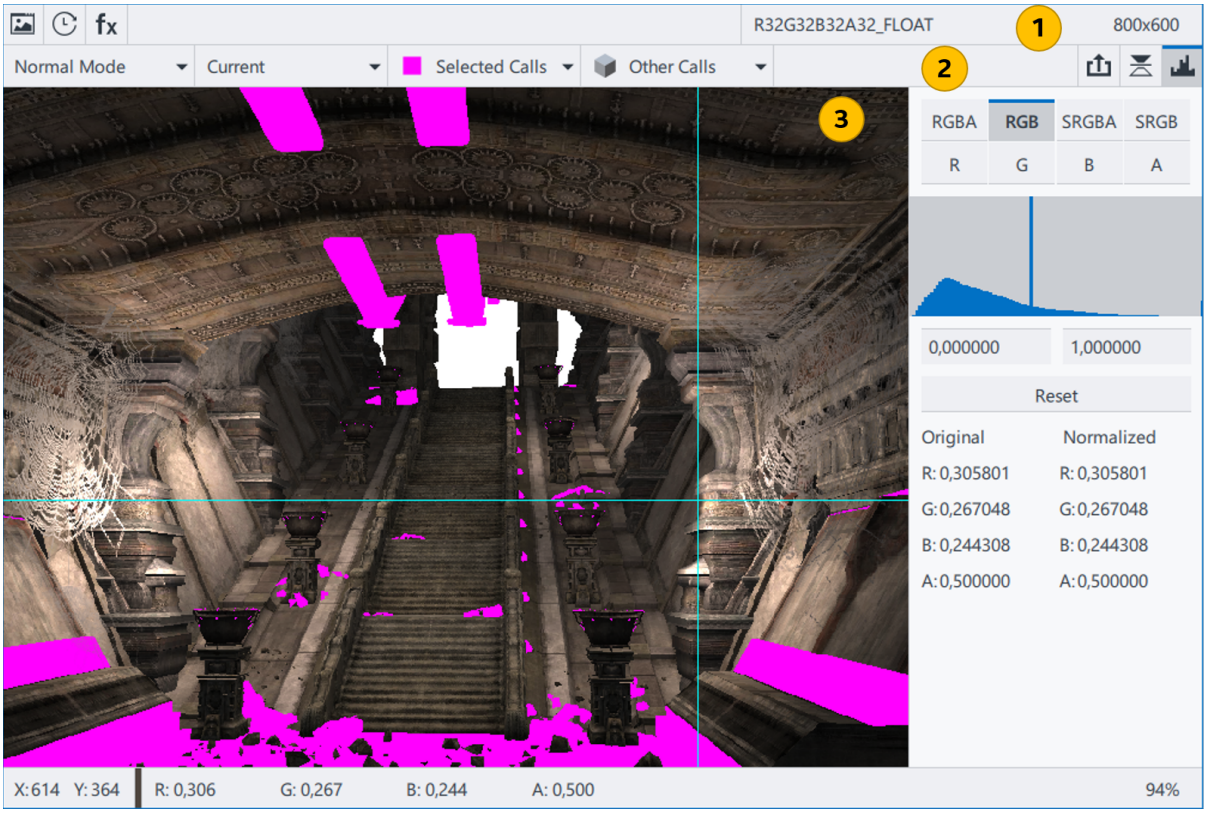
2 **Geometry Viewer** window

Shows a mesh for the selected call.

NOTE For DirectX 11 and 12, output geometry can be selected for several calls within one render target region.

You can rotate the object in any direction by clicking and holding the mouse pointer. To invert normals by 180 degrees, right-click the geometry object. To flip vertical and horizontal planes, middle-click the geometry object.

Resource Viewer: Texture



1 **Texture Viewer** pane



Open in Separate Window button - displays the selected in a separate window.



Resource History button - shows in the API Log pane only the graphics API functions that use the selected resource.



Resource Creation Parameters button - shows the descriptor with available parameters for DirectX resources.

Texture Info field - shows the resource type and internal ID; view type and ID; and debug name, if available.

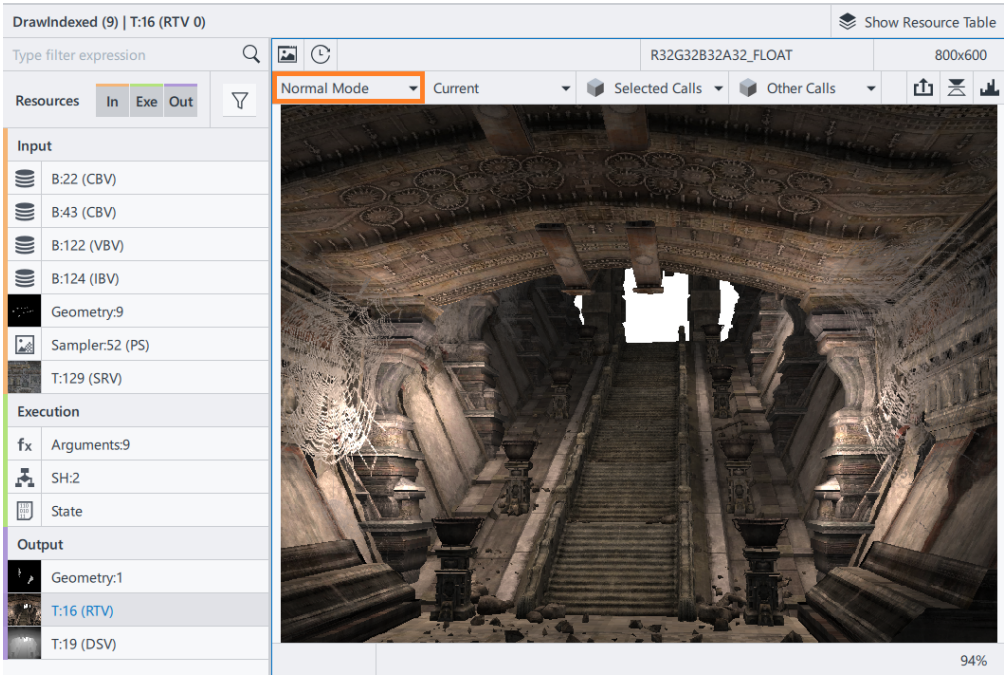
Format field - shows the original texture format.

Dimensions field - shows the resource dimensions, such as width, height, depth - or layer count and sample count for multisample textures.

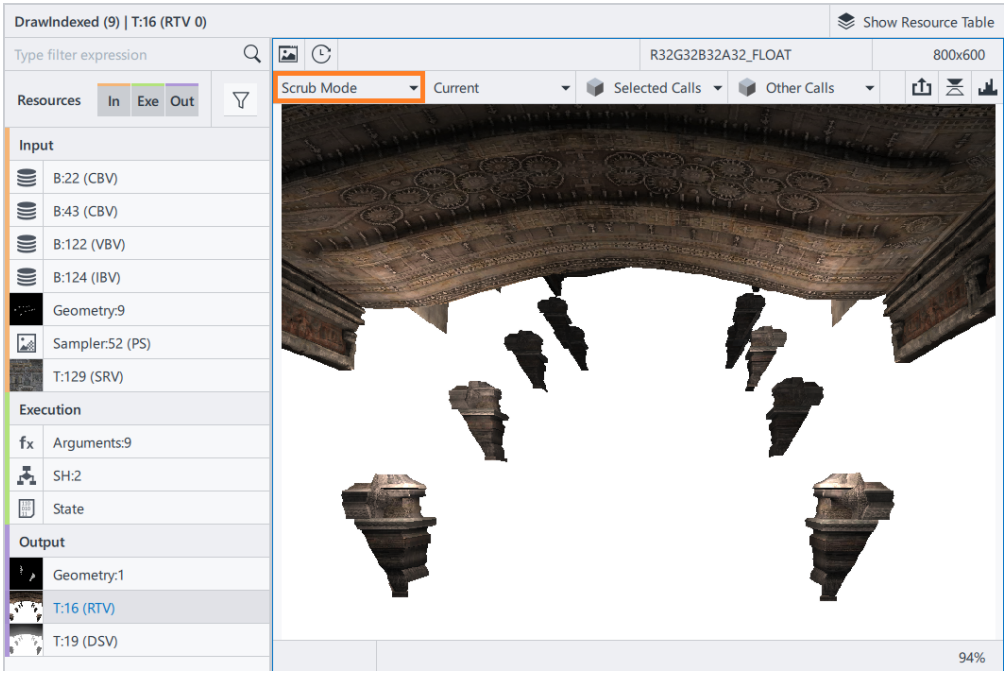
2 **Texture Viewer** toolbar

Render Target State drop-down list toggles between the following modes of the selected render target:

- **Normal Mode** shows the render target state for the last event, which writes to this render target before reading the render target or before the frame end.



- **Scrub Mode** shows the render target state for the last selected event.



Output Texture Visualization Mode drop-down list switches between the following viewing modes of the selected output texture:

- **Current** shows the render target with modifications.
- **Original** shows the render target without modifications.
- **Diff** shows the difference between the current and original modes.
- **Overdraw** shows the render target with a blue scale overdraw visualization. The brighter the color, the more times the pixel was updated. Hover over the pixel to see the hit count below.

Selected Calls drop-down list switches between the following ways to render the selected calls:

- **Original** renders all selected events with no modifications.
- **Highlight** sets all selected events to a pink solid color fill mode.
- **Wireframe**(only for DirectX 11, DirectX 12, and Vulcan) shows all selected events in a pink wireframe mode.
- **Hide** does not show all selected events.

Other Calls drop-down list switches between the following options to show non-selected calls:

- **Original** renders all non-selected events with no modifications.
- **Highlight** sets all non-selected events to a blue solid color fill mode.
- **Wireframe**(only for DirectX 11, DirectX 12, and Vulcan) shows all non-selected events in a blue wireframe mode.
- **Hide** does not show all non-selected events.

NOTE **Selected Calls** and **Other Calls** viewing options apply only to the draw calls that render into the selected render target.



Save Texture to File button - saves a texture image or render target in the Direct Draw Surface (DDS) format, Khronos Texture (KTX) format, or PNG format.



Flip Image button - turns a texture image or render target upside down.



Histogram button - opens a color histogram of a texture or render target.

3 **Texture Viewer** window

Shows a texture image or render target.

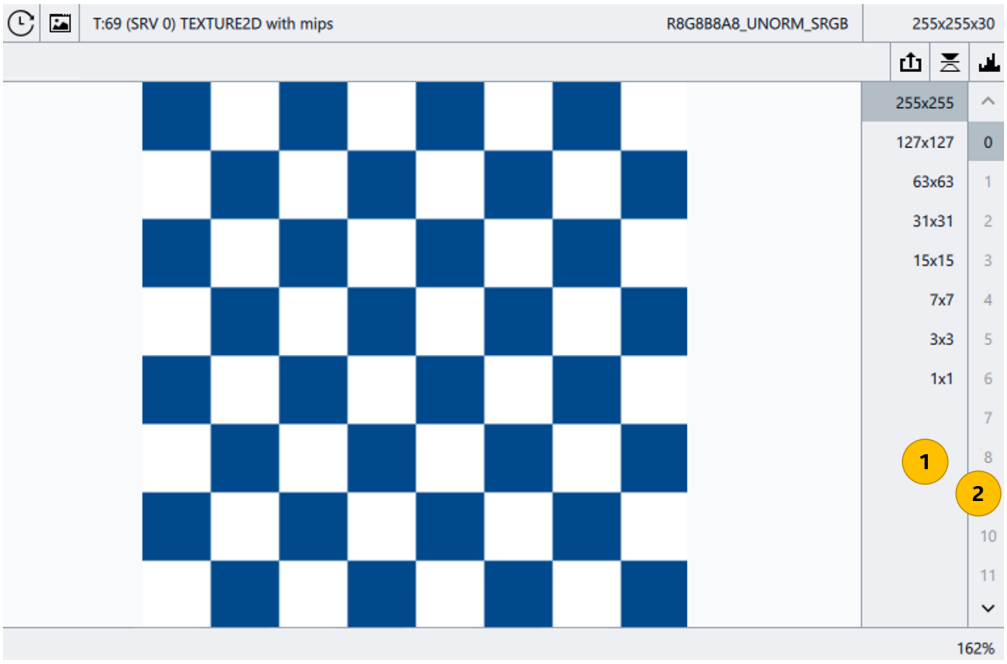
For textures with mipmaps, 3D textures, and texture arrays, you can select a particular subresource in the

1

MIP-level selector or the

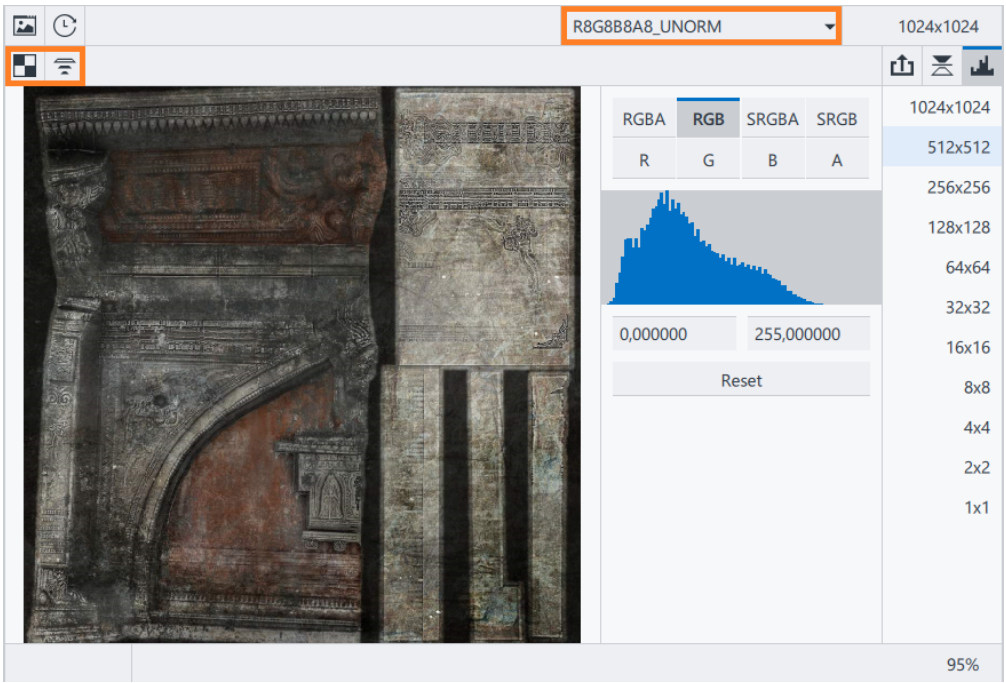
2

Layer selector:



Grayed-out values in the **MIP-level selector** or in the **Layer selector** mean that the corresponding subresources are not defined in the selected resource view.

You can experiment with SRV textures using the following elements:



- 

  Texture 2x2 button - replaces the opened texture with a simple 2x2 pixel texture for the selected events. The experiment creates a new texture with the same name and of the same type, and denoted as [modified].

- 

Clamp to Selected MIP Level button - disables all mipmaps of the opened textures that are greater than the current mipmap level of the texture for the selected events. The experiment creates a new texture with the same name and of the same type, and denoted as [modified].

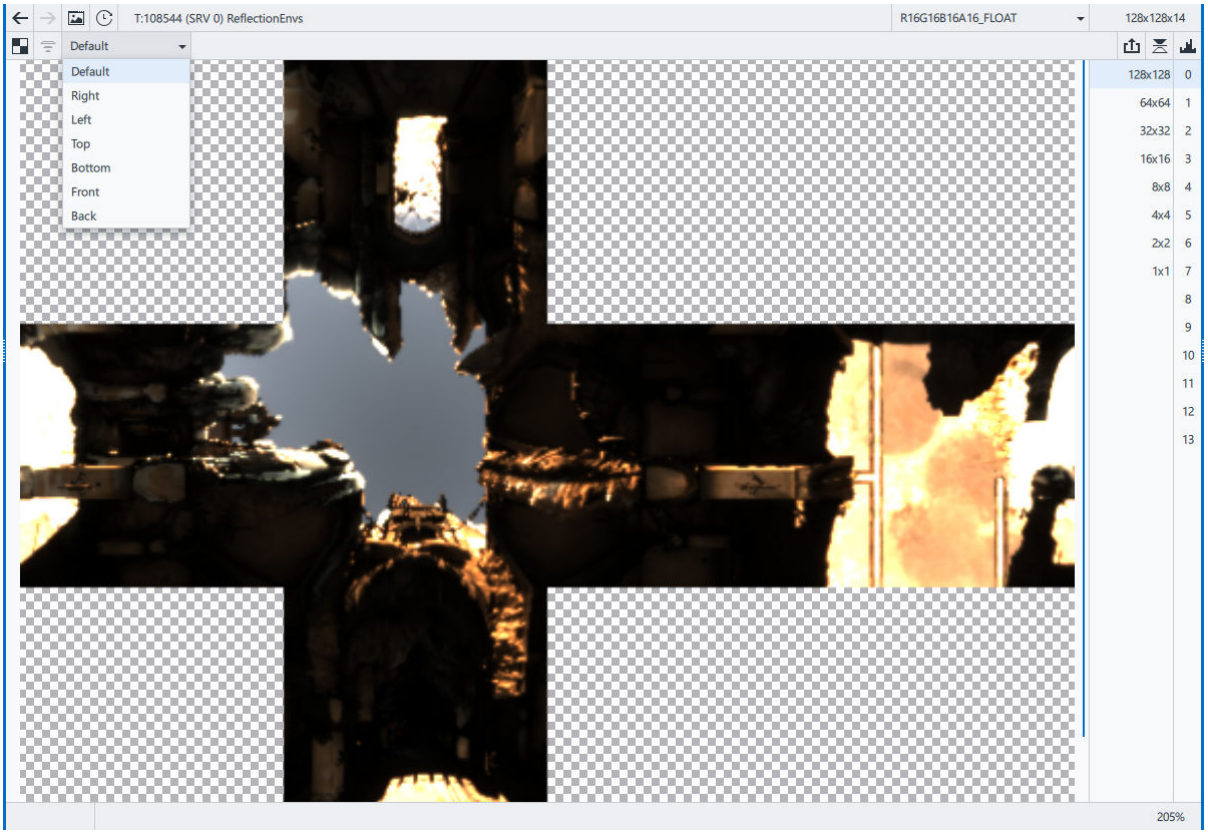
- **Format** drop-down list - shows the original texture format and enables texture format replacement (DirectX 11 only).

To disable the applied experiments, click the



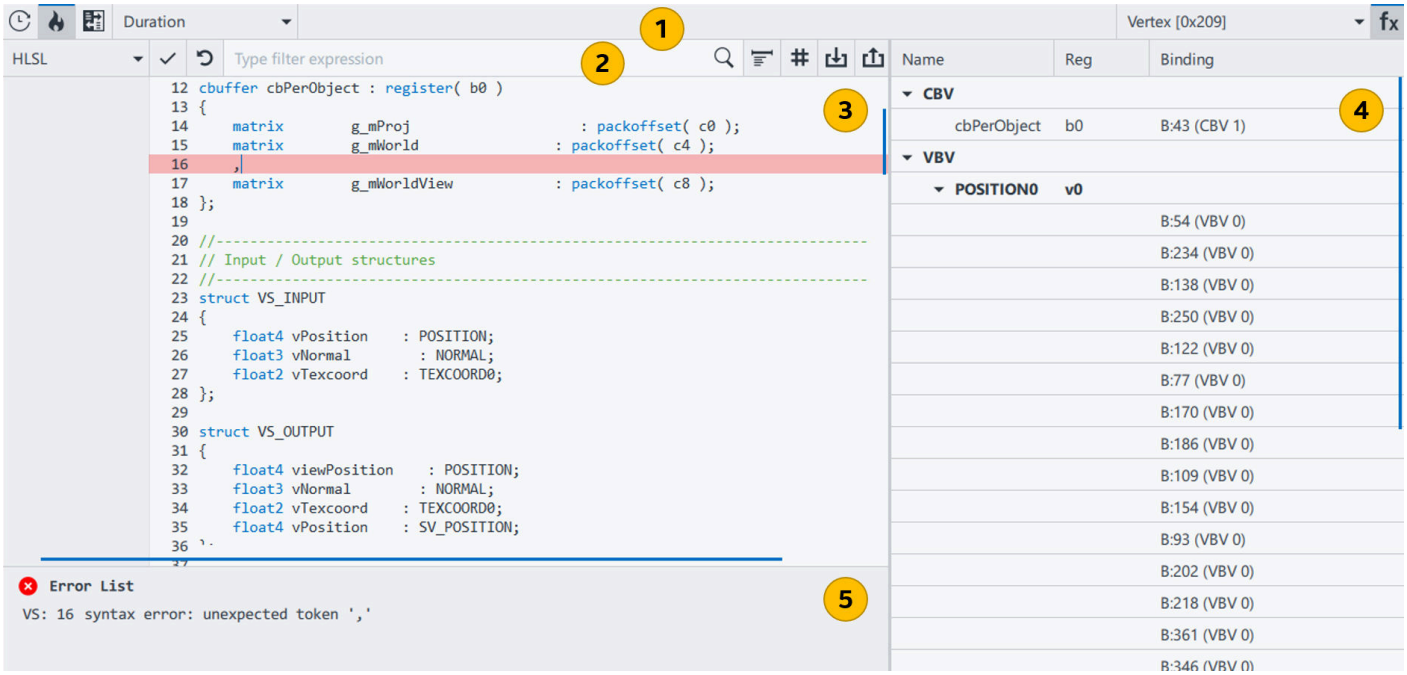
button on the texture thumbnail in **Resource List**.

For cube map textures, you can open one face of a cube map by selecting the desired face from the drop-down list. The **Default** option is a cube map layout.



NOTE Cube map layout for a MIP-level, where a face size is greater than 4096px, is unavailable.

Resource Viewer: Shader



1 **Shader Viewer** pane



Resource History button shows in the **API Log** pane only the draw calls that use the selected shader set.



Shader Profiler button - switch to Shader Profiling mode.



Show source-assembly mapping button - enable side-by-side view of shader and assembly code.

Shader Type drop-down list - shows the type of the opened shader.

2 **Shader Viewer** toolbar

Shader Code drop-down list - shows different code representations of the opened shader. Supported **Shader Code** types:

- DirectX: HLSL; DXBC/DXIL; ISA
- Vulkan: GLSL; HLSL; SPIR-V
- OpenGL:GLSL

For DirectX frames, you can choose Compiler Flags from the **Shader Code** list to see what options are used to compile the source code.



Apply button - compiles a modified shader source code and applies it to the selected draw calls or dispatch calls.



Revert button - discards all changes made to the opened shader.

Type Filter Expression field - filters shader code values.



Format button - applies default formatting to the opened shader code.



Preprocess button - preprocesses the opened shader code by applying all defines from the list of definitions.



Load source code button - replaces the current shader source code with the imported code.



Export shader profile to CSV button - saves the shader code currently displayed in the Shader Editor, in the CSV format.



Bindings button - displays a list of resources and constants that are used in the opened shader.

3 **Shader Editor**

Enables you to view a shader code, change it and apply changes to the selected draw calls or dispatch calls.

4 **Shader Resource List**

Lists bound resources and constants used in the opened shader.

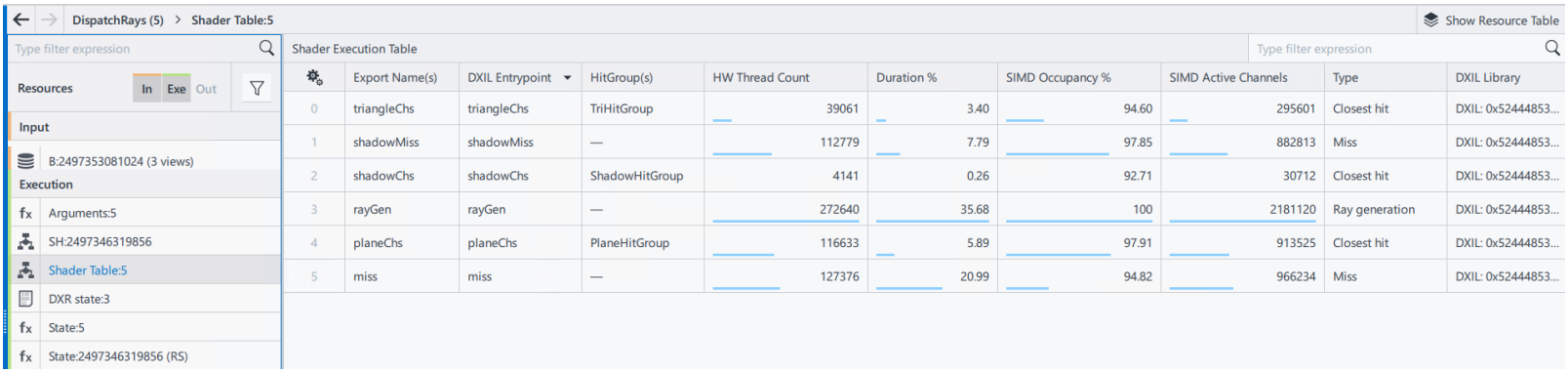
5 **Notification** pane

Displays errors and warnings that relate to the selected shader code.

Resource Viewer: Shader Table

Shader Table is a tool for deep analysis of ray tracing shaders. The table visualizes all shaders that can be potentially executed as a result of a `DispatchRays` call. Shader Table also contains performance metrics for each shader.

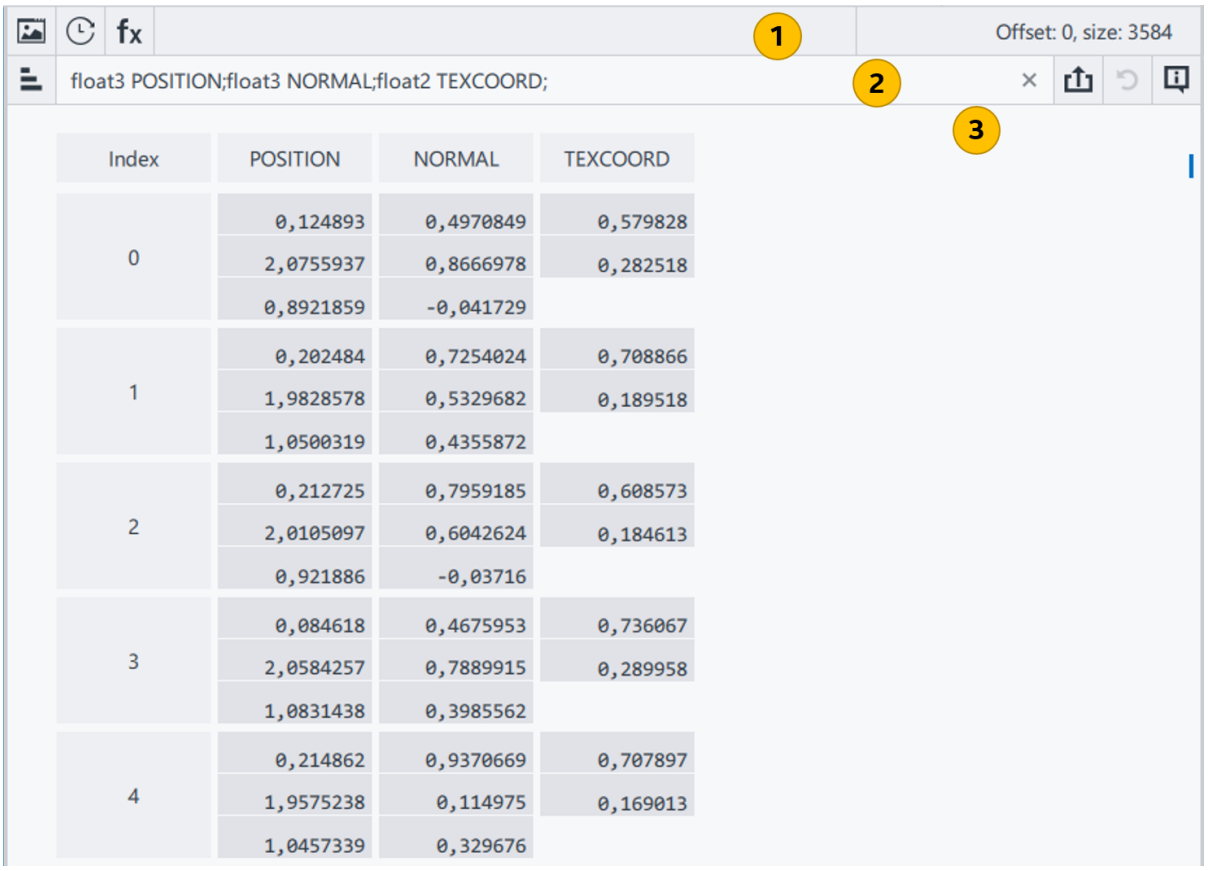
NOTE Shader Table is available for DirectX* Raytracing applications starting with Intel® Arc™ high-performance GPUs code named Alchemist, and previously known as DG2.



When you select a `DispatchRays` call in Graphics Frame Analyzer, Shader Table becomes available in the Resource List as a separate item. Click the Shader Table resource to view detailed information about shaders:

- **Export Name(s)** - unique shader names, which are exported from the raytracing pipeline state object and used to generate the shader identifier.
- **DXIL Entrypoint** - shader name from the HLSL source code.
- **Hit Group(s)** - names of hit groups where the shader is used. Only *closest hit,any hit,* and *intersection* shaders can be included in a hit group. Cells in the Hit Group(s) column are always empty for *ray generation,miss,* and *callable* shaders.
- **HW Thread Count** - number of hardware threads executing instances of a shader. If the number is zero, the shader was bound to the pipeline but not executed. Maximum number of shader instances in a hardware thread depends on SIMD width (8, 16, or 32).
- **Duration %** - relative duration of a shader, where 100% is the sum of durations for all shaders.
- **SIMD Occupancy %** - number of SIMD lanes executing a shader, in relation to the total number available across all threads. If SIMD occupancy is low, the hardware scheduler could not aggregate different shader instances in one thread. Developer and Optimization Guide describes how to improve SIMD occupancy.
- **SIMD Active Channels** - the number of rays processed by the shader. The value of SIMD Active Channels metric equals to HW Thread Count * SIMD width * SIMD occupancy / 100%.
- **Type** - shader type as defined in the DXIL library.
- **DXIL Library** - identifier of the DXIL library where a shader is declared. You can use the identifier to find a proper shader code from the Resource List.

Resource Viewer: Buffer



1. **Buffer Viewer** pane

   Show Resource in Separate Window button - opens **Buffer Viewer** in a separate window.

   Resource History button - shows in the **API Log** pane only the graphics API functions that use the selected buffer.

   Resource Creation Parameters button - shows the descriptor with available parameters for DirectX resources.

   Offset and Size field - shows the offset value and size of the opened buffer block in the original buffer object.

2. **Buffer Viewer** toolbar

   View toggle button - toggles between the list view and table view of the **Buffer Viewer**.

   Buffer Layout field - enables you to view and edit the opened buffer layout. The layout is written in the buffer markup language.

Export buffer to CSV button - exports the contents off the buffer in .csv file format. Your are prompted to specify the file name and the location.



Restore Default Layout button - reverts any buffer layout modifications made by the user and restores the default shader-defined or pipeline state-defined buffer layout or a default format (float4 or xuint4), if the buffer layout was not received from shader.



Info button - opens the layout editor help with the markup language help.

3 **Buffer Viewer** window

Shows the opened buffer details in a table or list format.

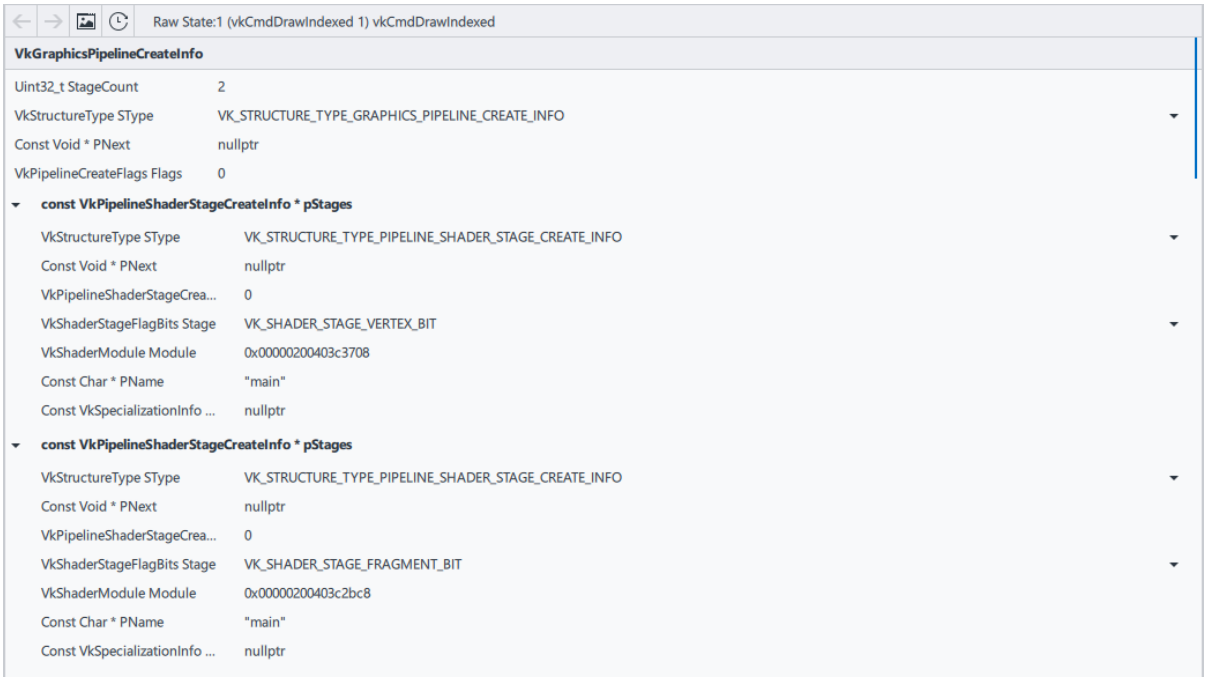
Resource Viewer: Graphics/Compute Pipeline State

The **Graphics/Compute Pipeline State Viewer** shows settings for fixed function blocks of the graphics/compute pipeline. The settings can be changed.

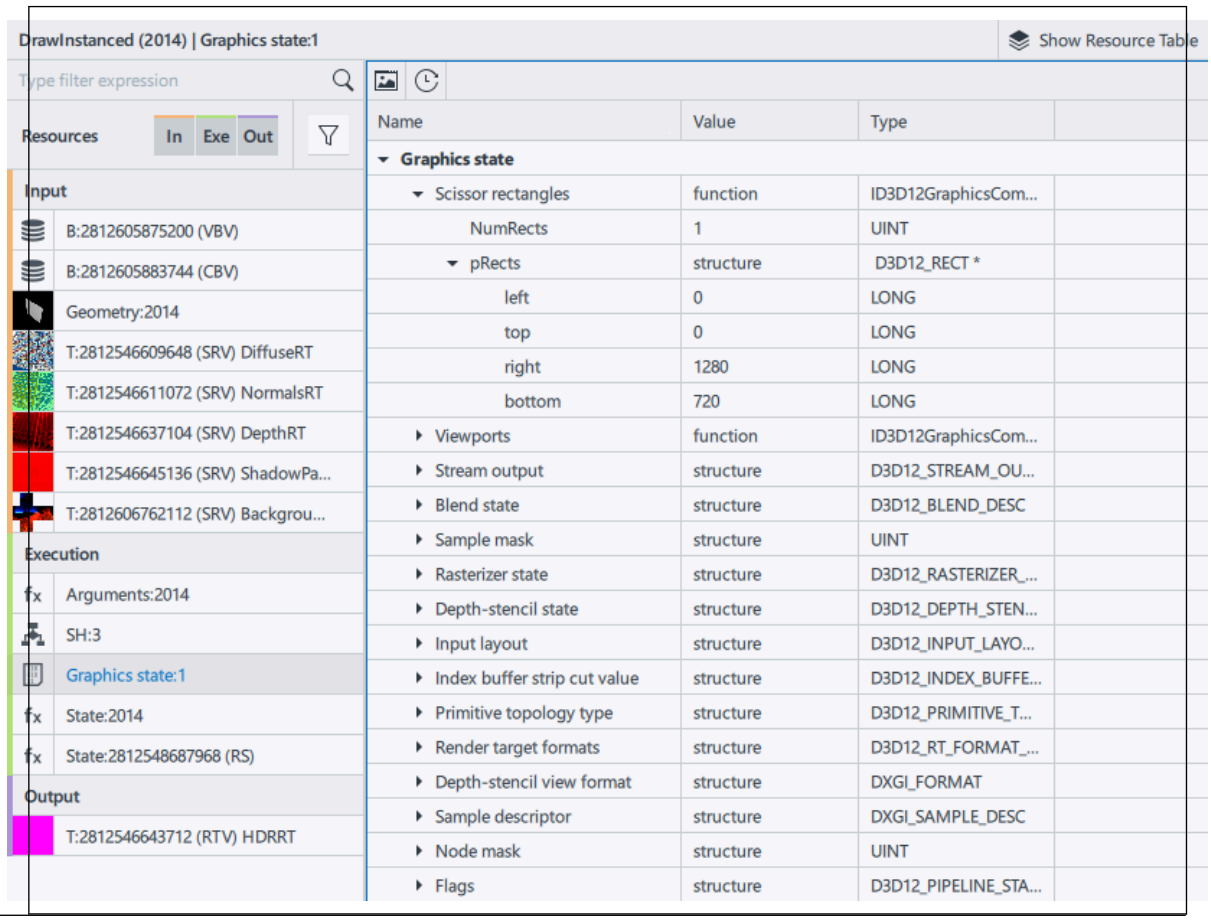
For DirectX and OpenGL applications, **State Viewer** shows API-agnostic data. The data can be modified for several draw calls simultaneously:

DEPTH		ON
WRITE MASK	D3D11_DEPTH_WRITE_MASK_ALL	
FUNC	D3D11_COMPARISON_LESS	
STENCIL		OFF
DEPTH-STENCIL OPERATIONS		
FRONT STENCIL FAIL OP	D3D11_STENCIL_OP_KEEP	
FRONT STENCIL DEPTH FAIL...	D3D11_STENCIL_OP_KEEP	
FRONT STENCIL PASS OP	D3D11_STENCIL_OP_KEEP	
FRONT STENCIL FUNC	D3D11_COMPARISON_ALWAYS	
BACK STENCIL FAIL OP	D3D11_STENCIL_OP_KEEP	
BACK STENCIL DEPTH FAIL OP	D3D11_STENCIL_OP_KEEP	
BACK STENCIL PASS OP	D3D11_STENCIL_OP_KEEP	
BACK STENCIL FUNC	D3D11_COMPARISON_ALWAYS	
RASTERIZER		
FILL MODE	D3D11_FILL_SOLID	

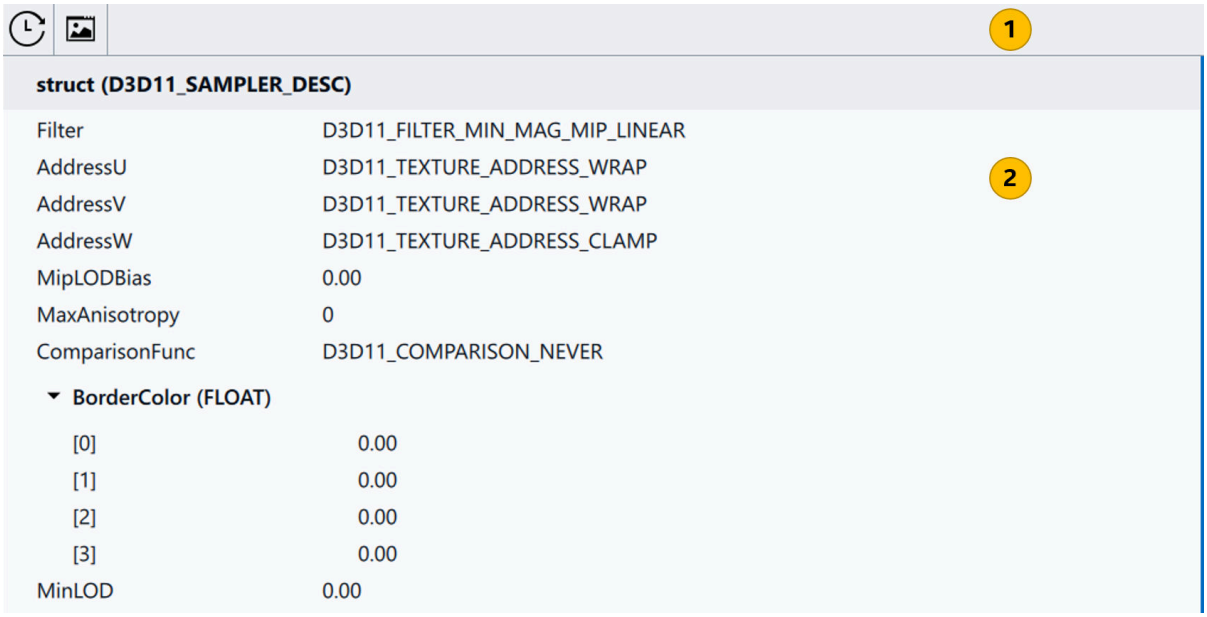
For Vulkan, State Viewer shows Vulkan-specific data only for one draw call:



For frames opened from stream files, Graphics Frame Analyzer provides a combined state view that unifies both Pipeline State Object (PSO) and non-PSO states:



Resource Viewer: Samplers



1. **Sampler Viewer** pane

   Resource History button - shows in the **API Log** pane only the graphics API functions that use the selected sampler.

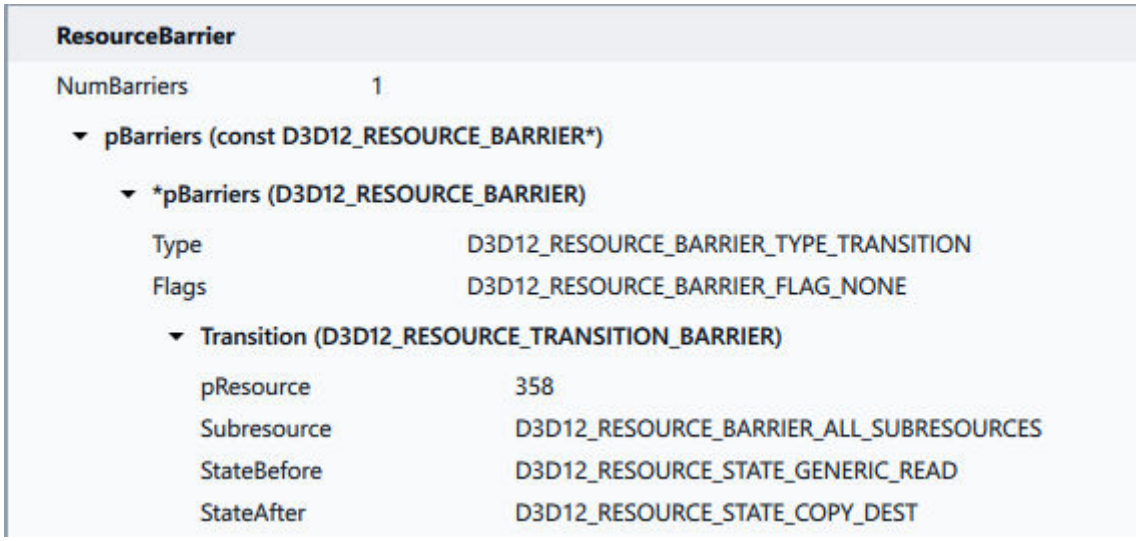
   Show Resource in Separate Window button - opens **Sampler Viewer** in a separate window.

2. **Sampler Viewer** window

   Shows read-only settings for sampling textures.

Resource Viewer: API Call Arguments

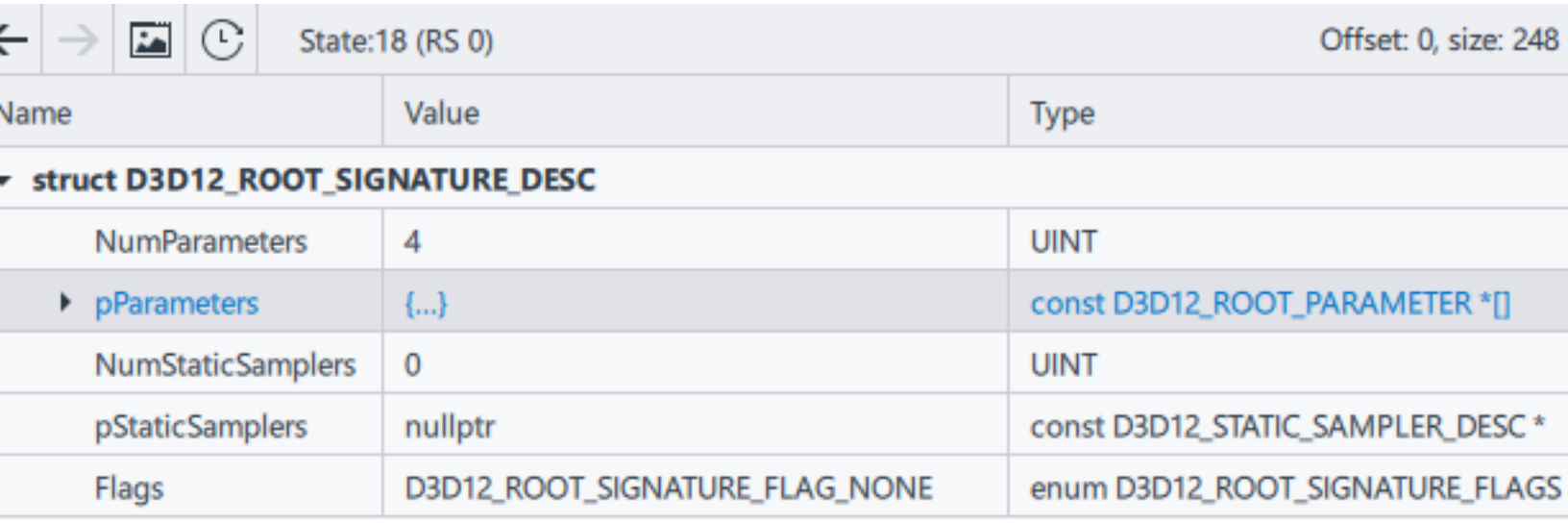
The **API Call Arguments Viewer** lists arguments of the selected graphics API function. The viewer shows data only for a single function.



- For the DirectX 11 and DirectX 12 frames, you can change arguments to change input geometry, clear color or the number of computing threads.
- You can change arguments for the following calls:
  - ClearRenderTargetView
  - ClearDepthStencilView
  - ClearUnorderedAccessViewFloat
  - Dispatch calls
  - For DirectX 11, all draw calls except DrawIndexedInstancedIndirect and DrawInstancedIndirect
  - For DirectX 12, DrawIndexedInstanced and DrawInstanced

Resource Viewer: Root Signatures and Descriptors (DirectX 12 and Vulkan only)

This resource view enables you to explore the internals of DirectX 12 Root Signatures and Vulkan Descriptor Sets. This can be useful for debugging sporadic rendering issues, like a wrong texture being used for an object.



See Also

Optimize Shaders

Analyze Textures

Minimize Overdraw

Optimize Graphic API States

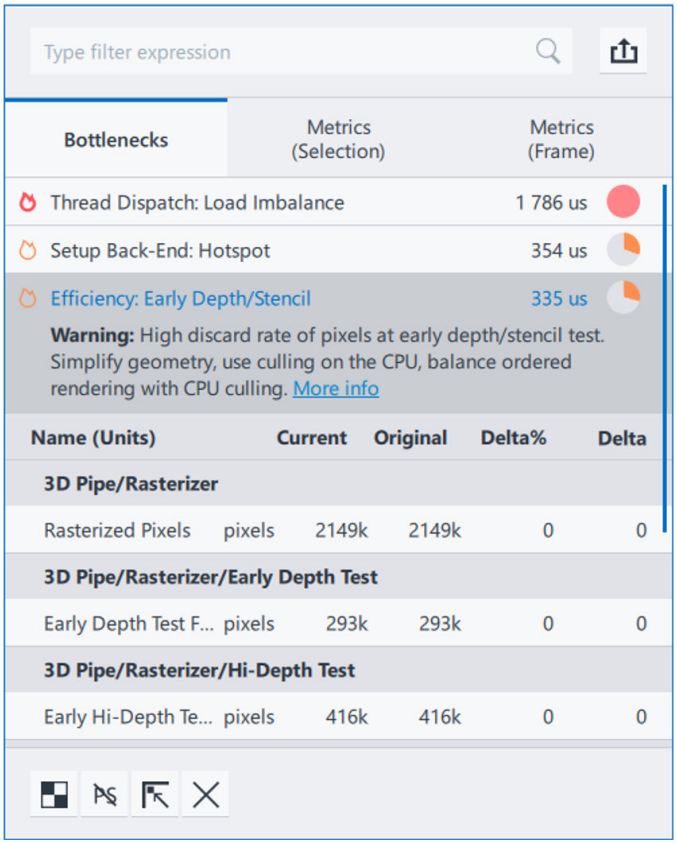
Identify Graphics Elements with Complex Geometry

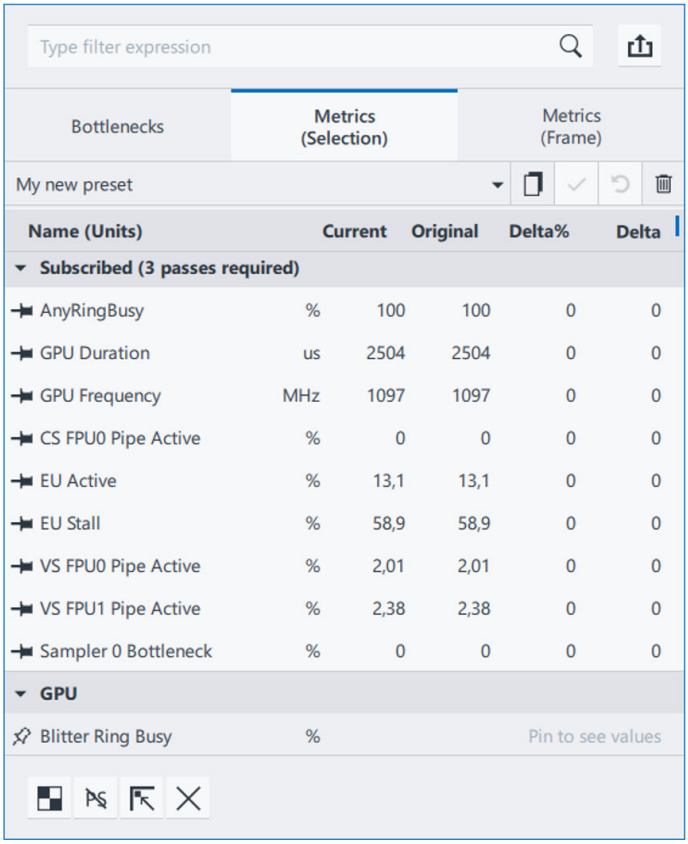
Visualize Color Distribution

Metrics Pane
*In the **Metrics** pane, you can analyze metrics information for the selected events. If nothing is selected, Graphics Frame Analyzer displays the total values for the entire frame. If metrics are unavailable for your platform, the **Metrics** pane is hidden.*

In the Metrics pane, you can analyze metrics information for the selected events. When nothing is selected, Graphics Frame Analyzer displays total values for the entire frame. If metrics are unavailable for your platform, the Metrics pane is hidden.

When you experiment with the frame, metrics update immediately to reflect the introduced changes.





To switch between different views, click the corresponding tabs on the Metrics pane:

- **Bottlenecks** tab - groups all metrics by hardware blocks that are used for geometry rendering or computing processes, and bottlenecks related to these blocks.

  To see the bottleneck cause, hints, and related metrics, click the corresponding bottleneck.

  Bottlenecks are marked with a colored hot spot icon on the left:

- 

  indicates a primary bottleneck.
- 

  indicates a secondary, or potential, bottleneck.

  On the right of the bottleneck name, you can see:

  - Total duration of all API calls with this bottleneck, in microseconds.
  - Pie chart showing the percentage of time spent on API calls that have this bottleneck in relation to the time spent on all API calls with primary (red) or secondary (orange) bottlenecks.
- **Metrics (Selection)** tab - displays metrics for the current selection in a table format.

  Values are available only for pinned, or subscribed, metrics in order to reduce duration of metrics calculation. The more passes are required for the Subscribed group, the longer it takes to request metrics.
- **Metrics (Frame)** tab - displays all metrics for the full frame in a table format.

  Metric values in this tab do not depend on selected events. Values are available only for pinned, or subscribed, metrics in order to reduce duration of metrics calculation. The more passes are required for the Subscribed group, the longer it takes to request metrics.

You can perform the following actions with metrics in Metrics (Selection) and Metrics (Frame) tabs:

- Filter metrics by typing the required metric name into the Type Filter Expression field.
- Add the most frequently used metrics to the Subscribed groups, or presets, by pinning required metrics in the metrics list. Subscribed group is created when at least one metric is pinned.

  By default, there is one *Default preset*, which you can modify.

  To create a new preset:

  1. Click the

     Clone to new preset button to copy a current state of the selected preset.
  2. Type a preset name in the **Type new preset name** field and click

     .
  3. Hover over metrics,

     pin necessary metrics and

     unpin ones that you do not need in this preset.

     You can

     Revert changes to the previous applied version if necessary.

NOTE Your changes are stored even if you do not apply them, and restart Graphics Frame Analyzer.

  4. Click the

     Apply changes button when the preset is ready.
  5. You can



Delete a preset when you no longer need it.

Experiments Pane

In the Experiments pane, you can isolate potential performance bottlenecks in your application by modifying render states of Graphics API.



NOTE You can only perform experiments if metrics data is available for your target system; otherwise, the **Experiments** pane is hidden from view.

Icon	Description	DX11 Frame/ Stream (Windows)	DX12 Frame (Legacy) Available in Extended Profile Mode (Windows)	DX12 Frame/ Stream (Windows)	Vulkan Frame/ Stream (Windows)	OpenGL Frame (Ubuntu)
	2x2 Textures Identifies potential performance bottlenecks caused by texture maps used in your application. All sampling textures for a scene are replaced with 2x2 texture containing four different colors.	Yes	Yes	Yes	Yes	Yes
	Simple Pixel/ Fragment Shader Replaces every pixel/ fragment shader with the pixel/ fragment shader that writes a constant color to all render targets.	Yes	Yes	Yes	Yes	Yes
	1x1 Scissor Rect Bypasses pixel processing from the rendering pipeline: if the frame rate does not increase when this experiment is enabled, then a complex geometry or vertex shader is a bottleneck.	Yes	Yes	Yes	No	Yes

Disable Event(s) Keeps the selected events from being rendered. This experiment can help you test scene efficiency.	Yes	Yes	Yes	Yes	Yes
Shading Rate Adjusts the shading rate for the selected resource, which can help identify objects that could be rendered with less quality without noticeable visual impact.	No	Yes	Yes	No	No

For details, see Run Render State Experiments with Frame Analyzer.

See Also

Run Render State Experiments with Frame Analyzer

Graphics Frame Analyzer Window: Dependency Viewer

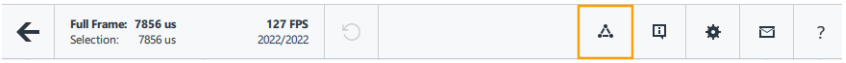
Dependency Viewer is a tool for exploring the dependencies in frames captured with Graphics Frame Analyzer. The tool is available for DirectX* 11 and DirectX* 12.

To open the Dependency Viewer, do the following:

1. Open a frame in the Profiling View window of the Graphics Frame Analyzer.
2. Click the



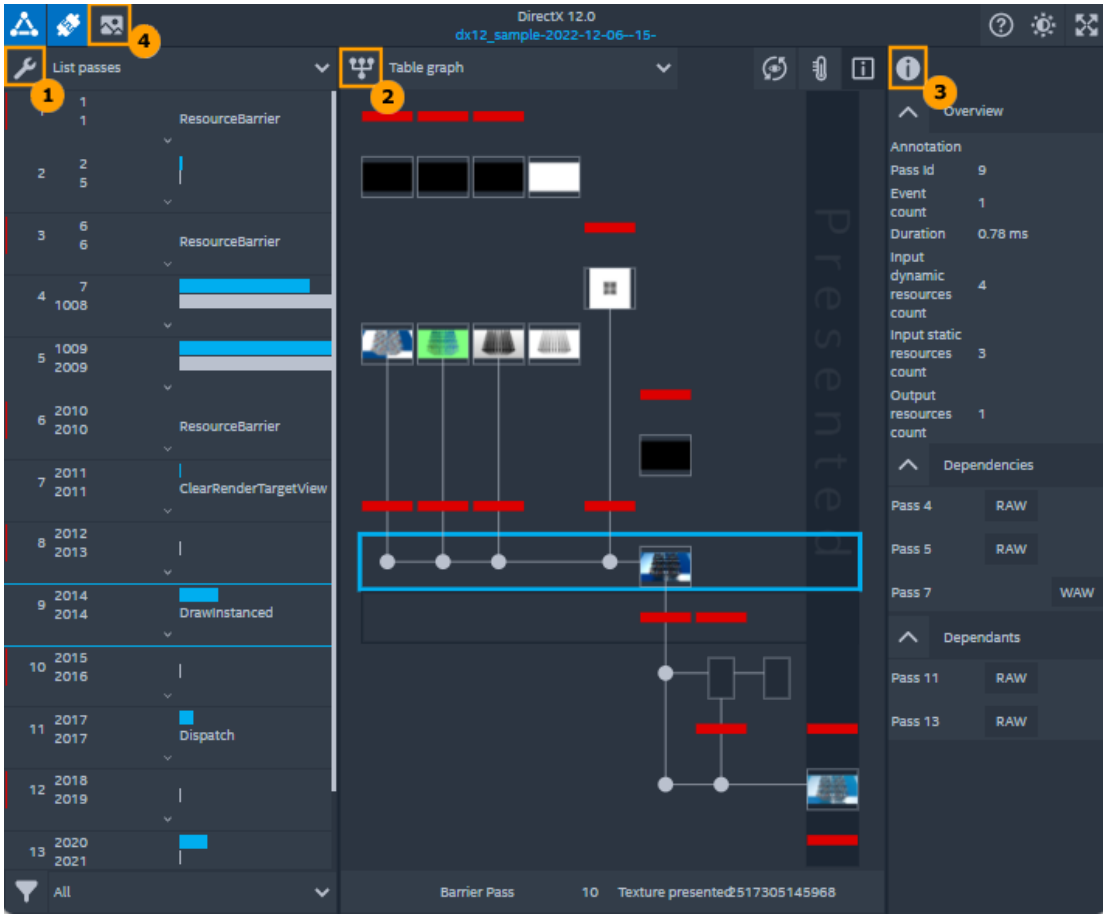
Dependency Viewer icon on the main toolbar at the top of the page.



The Dependency Viewer opens in a browser window.

NOTE After working with Dependency Viewer, you must close the browser window before interacting with Graphics Frame Analyzer.

The Dependency Viewer window consists of resizable panels, each of which could be configured to display one of the following views:



1. Tools view
2. Graphs view
3. Inspector view or

   Walkers view
4. **Preview Thumbnails** view

You can switch the desired view for each panel by clicking on the view icon (

1

) in the top left corner of each panel.

Additionally, you can enable resource content previews by clicking the **Preview Thumbnails** button (

2

). However, keep in mind that for large frames extraction of all render targets may take a considerable amount of time.

Tools View

The

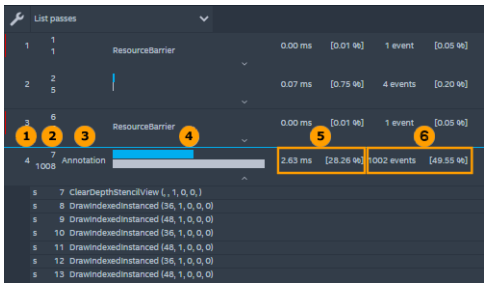
Tools view shows the structure of a frame in a table format. You can choose between the **List passes** or **List resources** mode via the drop-down menu:



List Passes

The passes list shows the sequence of render passes in the order they are executed in the frame. A *render pass* is a sequential list of graphic events writing to the same set of resources. A *graphic event* is an API call that generates a GPU command such as Draw, Dispatch, Copy, and so on.

Each entry on the passes list consists of the following:



- 1

  Pass ID that matches the Graphics Frame Analyzer IDs.
- 2

  The two IDs of the first and last graphics events in the render pass that match the IDs in Graphics Frame Analyzer.
- 3

  User-defined annotation that can be specified in the **Inspector** view.
- 4

  Blue and gray horizontal bars illustrate the duration of the render passes and *graphic event* count, respectively. Bars are normalized relative to the longest render pass in the frame. In case of single-operation passes, the operation pass name is shown instead of a gray bar.
- 5

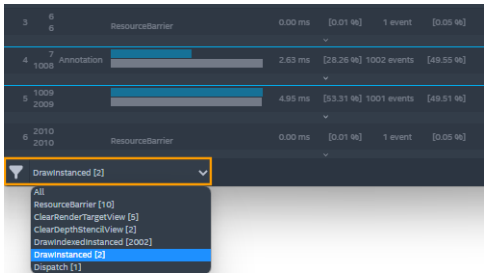
  Duration of the render pass in milliseconds and percentage of the frame duration.
- 6

  Count of graphic events in the render pass as well as the percentage of total events in the frame.

> **NOTE** Render pass is a sequential list of graphic events writing to the same set of resources. A graphic event is any API call which generates a GPU command, such as Draw, Dispatch, Copy, etc.

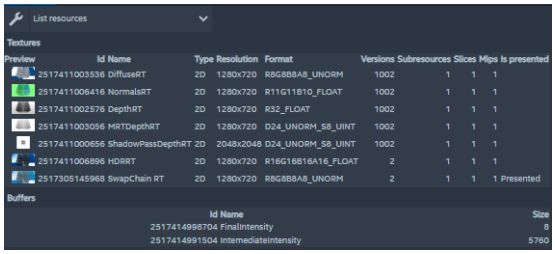
Use the **Expand** button to see the list of graphic events in the render pass. All events that perform reads from CPU-produced resources are marked with **s** on the left. Additionally, **Resource Barriers** are marked with a red bar on the left.

Use the filter field at the bottom of this view to highlight passes containing operations of a given type:



List Resources

This tool shows basic information about each resource generated in the frame. There are two separate lists of textures and buffers.



The tool represents each texture by a row containing the following information:

- Thumbnail of the resource on the last *graphics event* that modifies it. Thumbnails are not collected by default and must be enabled via the

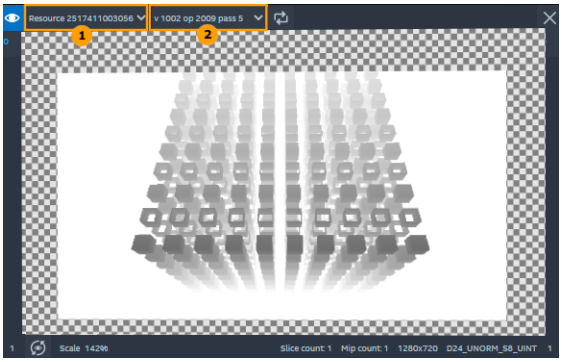
  Preview Thumbnails button at the top of the Dependency Viewer.
- ID that matches the Graphics Frame Analyzer resource IDs.
- Debug name, if available in the captured frame. This value can be manually specified or overridden in the

  Inspector view.
- Texture type
- Resolution
- Format
- Number of sub-resources, slices, and mips
- Flag for resources presented on the screen

> **NOTE** Resource version is a snapshot of the resource on each graphic event (such as Draw, Dispatch, Copy, etc.) which modifies its state.

Each buffer row contains an ID, a name, and a size in bytes.

Double-click a texture row to open the **Resource Preview** that enables you to switch between textures and their versions or compare two versions of the same resource:



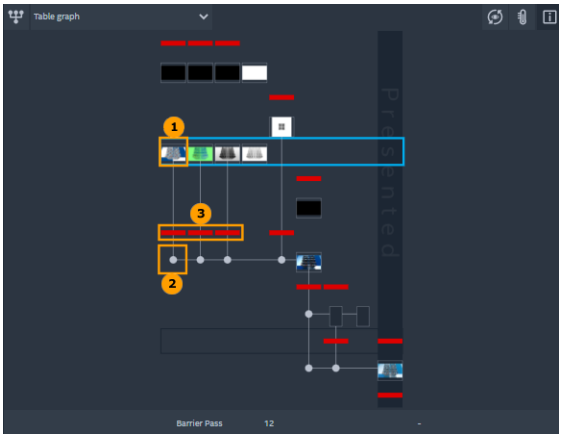
Graphs View

The

Graphs view shows render pass dependencies: pass reads from the resources produced by other passes. This view could operate in either a Table graph or a Focused graph mode.

Table Graph

The **Table graph** helps you visually analyze the frame structure:



- Rows represent rendering passes; columns represent resources. If a pass N writes to resource M, a graph node representing the state of the resource at the end of the render pass is placed in row N, column M.
- 1

  Each node shows a brief resource description:

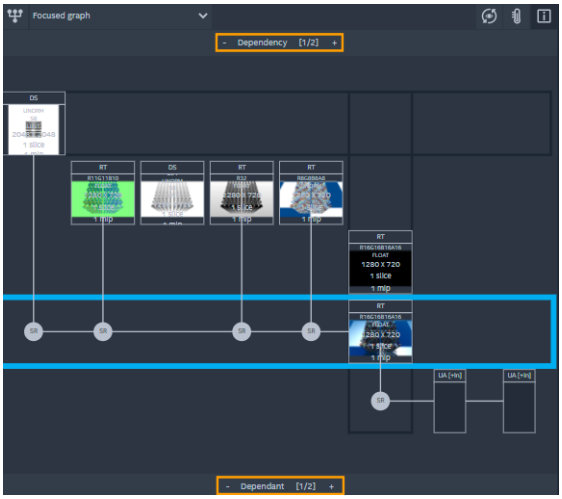
  - **View type**: Render Target (RT), Unordered Access (UA), Depth Stencil (DS).
  - For textures (can be toggled on or off via the **Texture Information** button): **Format**, **Dimension**, **Number of slices**, **Number of mips**, and **Preview** (if enabled via the **Preview Thumbnails** button at the top of the Dependency Viewer).
- 2

  If a pass reads from a resource, a dot is placed underneath it in the same column with the input view type: Shader Resource (SR), Constant Buffer (CB)
- 3

  Barriers are represented with a red bar and a letter representing its type, for example, **T** represents a transition. Ideally for optimal performance, a barrier should be right before the resource usage.

Focused Graph

Because graphs for the entire frame may be very complex, the **Focused graph** shows data similar to the **Table graph** but limits the depth of dependencies before and after the selected render pass. You can alter the dependency depth using the controls at the top and at the bottom:



Inspector View

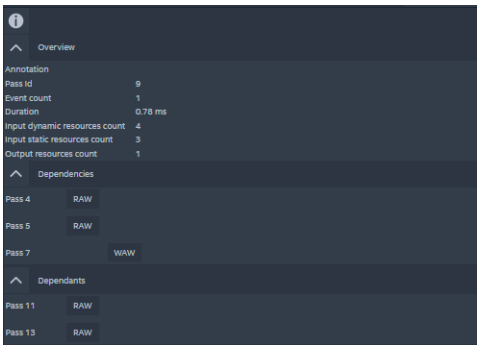
The

Inspector view displays additional information for the selected render pass, resource, or the buffer.

Render Pass Inspection

If you select a render pass in the

Tools view, the following information is available:



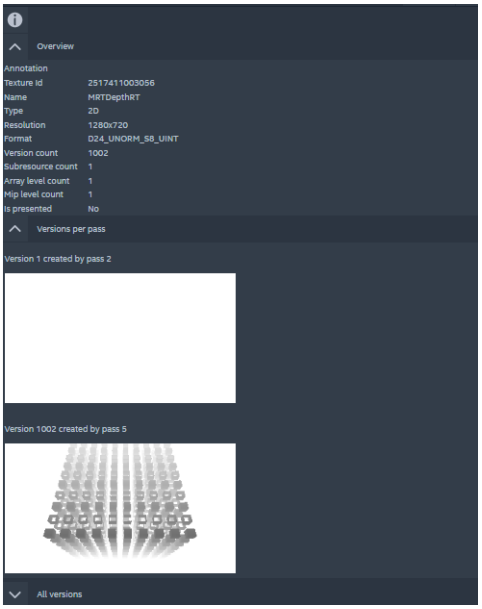
- **Overview**:
  - **Annotation**: An annotation for the selected render pass that will be displayed in **Tools** view. Click and type a required annotation here.
  - **Pass Id**: Render pass ID as it is listed in **Tools** view.
  - **Event count**: Number of graphic events in the selected render pass.
  - **Duration**: Duration of the render pass in milliseconds.
  - **Input dynamic resources**: Number of input resources with the content produced by another render pass.
  - **Input static resources**: Number of input resources with the static content.
  - **Output resources count**: Number of output resources for the selected render pass.

- **Dependencies**: Number of preceding render passes with IDs from which the selected render pass uses dynamic resources. For each render pass, the list specifies the dependency type: read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW).
- **Dependents**: Number of render passes with IDs which depend on the selected render pass. For each render pass, the list specifies the dependency type: read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW).

Resource Inspection

If you select a texture is selected in the

Tools view, the following information is available:



- **Overview**:
  - **Annotation**: An annotation for the selected texture that will be displayed in **Tools** view. Click and type a required annotation here.
  - **Texture Id**: Texture ID as it is listed in the **Tools** view.
  - **Type**, **Resolution**, **Format**, **Subresource count**, **Array level count**, **Mip level count**: Texture creation parameters.
  - **Name**: Texture name from the original application.
  - **Version count**: Number of modifications of the texture in the frame.
  - **Is presented**: True if this texture was presented on the screen.
- **Versions per pass**: First and last versions of the texture in the frame. You can double-click a version to inspect it further in the **Texture Preview** view.
- **All versions**: All versions of the texture in the frame. You can double-click a version to inspect it further in **Texture Preview** view.

Buffer Inspection

If you select the buffer in the

Tools view, then its overview is displayed:

- **Annotation**: An annotation for the selected buffer that will be displayed in **Tools** view. Click and type a required annotation here.
- **Buffer Id**: Buffer ID as it is listed in **Tools** view.
- **Name**: Buffer name from the original application.

- **Size**: Buffer size in bytes.

Walkers View

The

Walkers view automatically identifies and displays potential performance issues caused by redundant work, redundant allocated resources, and barriers that have been placed too early.



NOTE
The **Walkers** view may produce false positives for the following reasons:

- Dependency Viewer takes only a single frame into account; cross-frame dependencies are not discovered. Multi-frame support is planned for a later release.
- Some depth buffers might be marked as unused because they are not read in subsequent passes but still do their job.

See Also

Graphics Frame Analyzer UI Reference

Analyze GPU-bound Applications with Graphics Frame Analyzer

Graphics Trace Analyzer UI Reference

This section includes detailed reference on the Intel® GPA Trace Analyzer user interface elements.

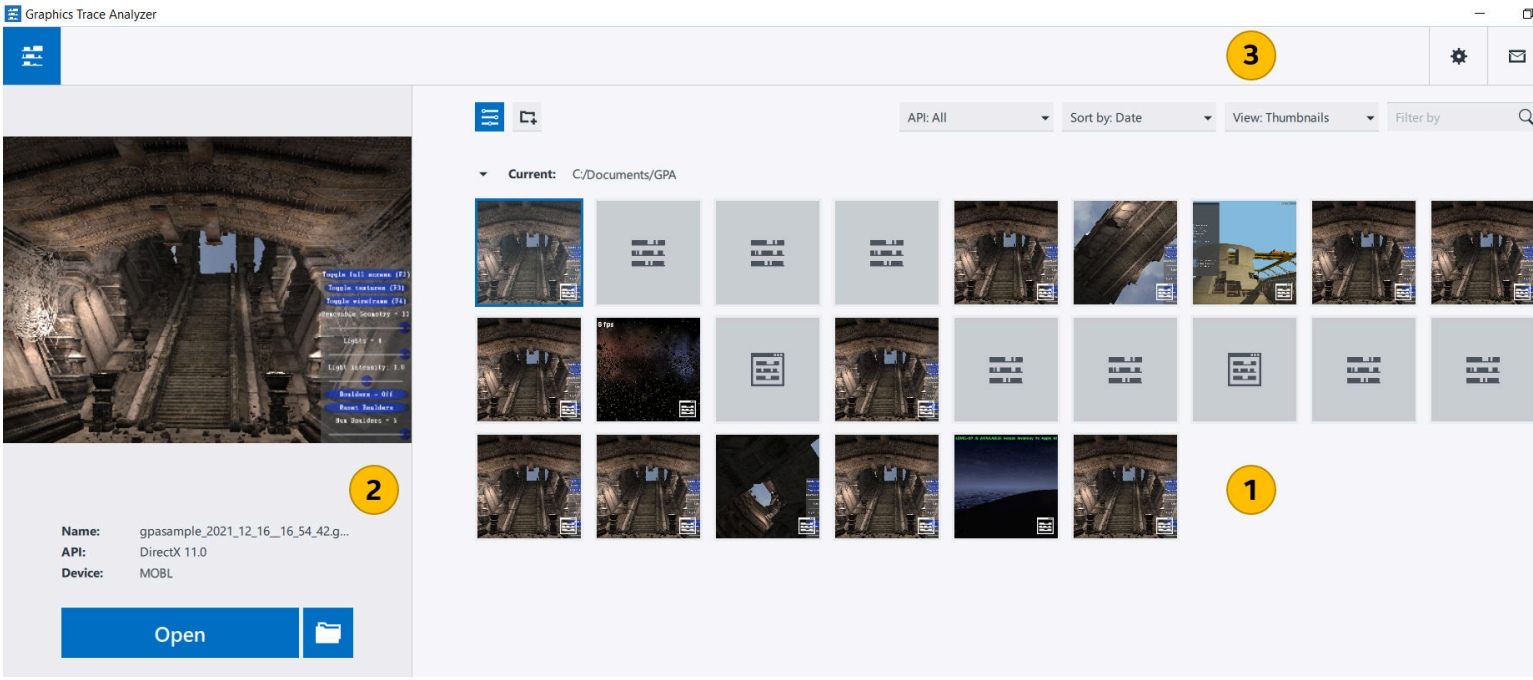
Graphics Trace Analyzer Window: Open Trace Capture

Graphics Trace Analyzer Window: Platform View

Graphics Trace Analyzer Window: Open Trace Capture

*Use the **Open Trace Capture** window to select and open your trace file.*

To access the **Open Trace Capture** window, in the Intel® GPA context menu, select **Graphics Trace Analyzer**. On Windows* OS, if you opted to enable context menu entries during installation, you can also open trace files using the Windows context menu.



1 **Trace Capture Thumbnails** pane

Select the trace capture you want to debug. By default, this field lists all trace captures located in your Intel GPA home directory: `%USERPROFILE%\Documents\GPA` (Windows* OS) or `~/GPA` (Ubuntu*). You can sort the trace captures by name or date with the **Sort by** button. If there are no trace captures on your system, these toggle buttons are hidden. The currently selected trace capture is highlighted with blue borders and appears in the trace preview pane.

Trace file thumbnails are marked with the



icon. If the captured trace data is invalid, the thumbnail is marked with the



icon.

To delete captured traces, hover the mouse pointer over a trace thumbnail and click the **Close** button in the top right-hand corner of the thumbnail. Confirm your choice by clicking **Yes** on the thumbnail.

2 **Preview** pane

Preview the selected trace captures. This field shows the buffer image and the following trace capture properties:

- File name
- API used
- Name of the device on which the trace was captured

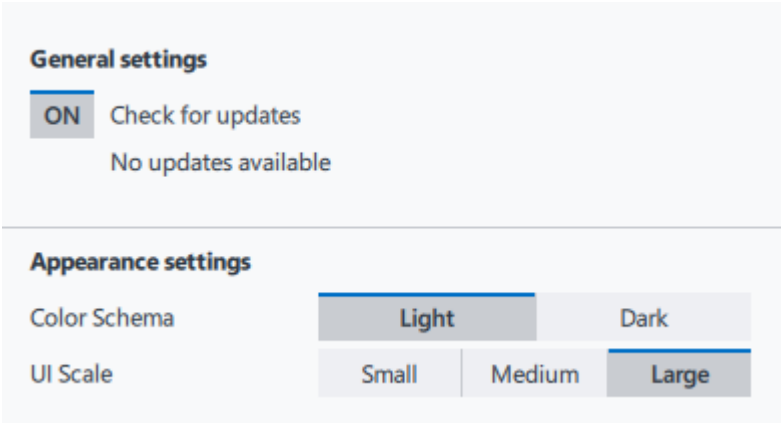
To rename the selected trace capture, you can click its filename and enter a new value. Click **Open**, to open the selected trace in the **Platform View**. Click the



Show in Folder button to open the storage directory of the selected trace.

3 **Message** pane

View the Intel GPA messages or notifications.

Type filter expression field	Filter the displayed trace capture files by typing in a part of the trace file name.
 **Run Graphics Monitor** button	Click to capture new traces with Graphics Monitor.
 **Settings** button	Manage Graphics Trace Analyzer settings:  • Use the **General Settings** toggle button to enable/disable checking for the newest available version of Graphics Trace Analyzer. **NOTE** During this check, Graphics Trace Analyzer collects information about your host system. No personally identifiable information is collected. If a newer version of the product is available, the **Settings** button is marked with the exclamation mark. To download the update, click the **Settings** button and select the available version from the context menu. • Use the **Color schema** setting to change the palette of the Graphics Trace Analyzer interface from Light (default) to Dark. • Use the **UI Scale** setting to change the size of the Graphics Trace Analyzer screen and the screen elements. Large size is default.
 **Help** button	Open Intel GPA online documentation.

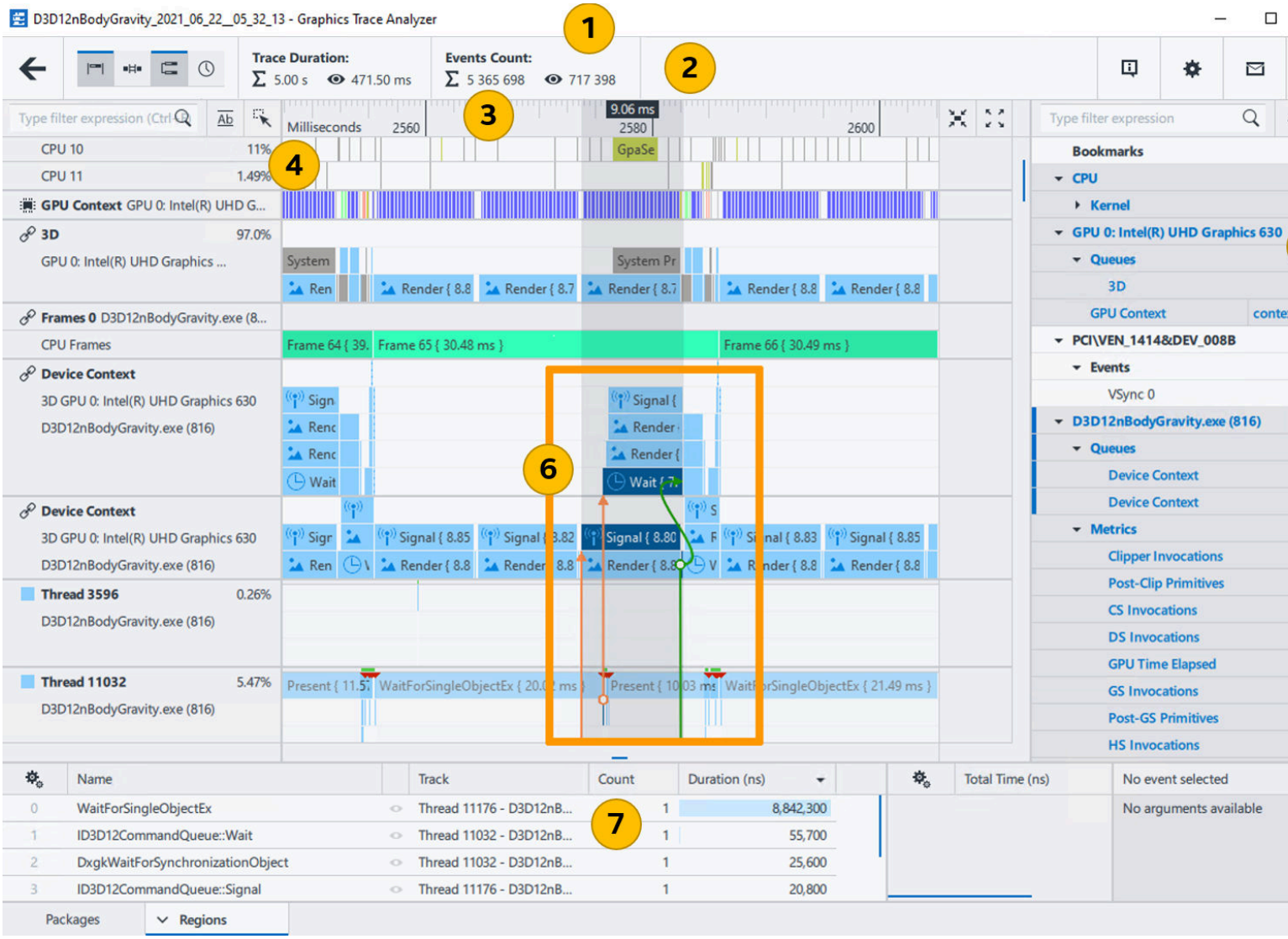
See Also

Identify Issues in Graphics Application Execution with Trace Analyzer

Graphics Trace Analyzer Window: Platform View

*Use the **Platform View** window to analyze concurrent execution of all your application threads, as well as correlate CPU and GPU activity.*

To access **Platform View** window, select a trace for analysis in the Open Trace Capture window, and then click **Open**, or double-click the trace preview. The **Platform View** window graphically represents a distribution of performance data over time.



1 **Trace title** bar

Displays the opened trace file name. The file name consists of the application name, capture date and time in the following format: YYYY_MM_DD_HH_MM_SS.

2 Main toolbar

Displays the opened trace data, notifications, opens the Intel® GPA User Guide, restores the default view, and changes the Intel® GPA color scheme.

3 Time Line Viewer toolbar

Filters tracks, shows event duration, zooms the selection in and out.

4 Time Line Viewer pane

Visualizes the opened trace content.

5 Trace Content pane

Lists the opened trace content.

6 **Synchronization Flow Graph**

The arrows reveal event dependencies on the time line. Identify inefficient events synchronization leading to incomplete GPU load. To see the graphs, click one of the objects. Red arrows demonstrate parent-child relationships between events. Green arrows connect

synchronization objects in UMD Driver Queues (Device Context Pane) and CPU threads. They also show synchronization points between objects. If you want to see several dependencies at a time, press and hold CTRL and pick objects by clicking.

NOTE This feature is only available on Windows* hosts.

7 Selection Statistics pane

Shows detailed information about the selection in the **Trace Content** pane.

NOTE The pane is only visible when any event or group of events is selected, unless the **Aggregated selection summary** is enabled in the **Settings** pane.

Main Toolbar

 **Back** button	Returns to the **Open Trace Capture** window.

Trace Overview pane	Provides the following statistical information for the analyzed trace:
	• **Events count**:
	  • Total number of events in the trace
	  • Number of events in the visible area of the **Time Line Viewer**
	• **Selected count**:
	  • Total number of selected events in the trace
	  • Number of selected events in the visible area of the **Time Line Viewer**

View Options pane	Provides controls for trace display customization: •  **Highlight selection** - highlight selected events and their relative position on to other events the **Trace Content** pane using vertical bars. •  **Cross-Track dependency selection** - enable/disable automatic highlighting of all related events from multiple tracks, as opposed to only highlighting the selected event. • **Inactive thread overlay** - enable/disable highlighting inactive thread zones in CPU Thread tracks. •  **Track utilization duration in time units/percents** button - switch between percents and time units in track utilization duration.
Message pane	Displays messages and non-critical errors.

	NOTE If more than one message is open, use the Up or Down arrows to navigate between messages.
 **Trace Info** button	Provides the following properties of the analyzed trace: • Intel® GPA version • Name of the system on which the trace was captured • Hardware and software of the system on which the trace was captured

 **Settings** button	• Enables the following:
	  • Highlight selection - highlights the selected events
	  • Cross-track dependency - simultaneous selection of dependent events
	  • Bottom-up queues - changes order of GPU/CPU queues visualization
	  • Fully-qualified function names - shows the full function name along with the name of the interface the function belongs to
	  • Aggregated selection summary - pins the **Selection Statistics** pane so that it remains open after deselecting events
	• Adjusts UI scale
	• Restores the default layout and track colors
	• Changes the color scheme of the **Platform View** interface

 **Send feedback** button	Opens a form to send feedback to the Intel® GPA technical support.
 **Help** button	Displays information on navigation controls and opens the Intel® GPA online documentation.

Time Line Viewer Toolbar

The **Time Line Viewer** toolbar enables you to search and filter events, assess event position and duration on the time line, and adjust the manner of displaying the trace in the **Time Line Viewer** pane. The toolbar includes the following options:

- **Type Filter Expression** field searches and filters events based on lexicographical comparison.
- 

  Match Whole Word button enables searching and filtering events by whole names.
- 

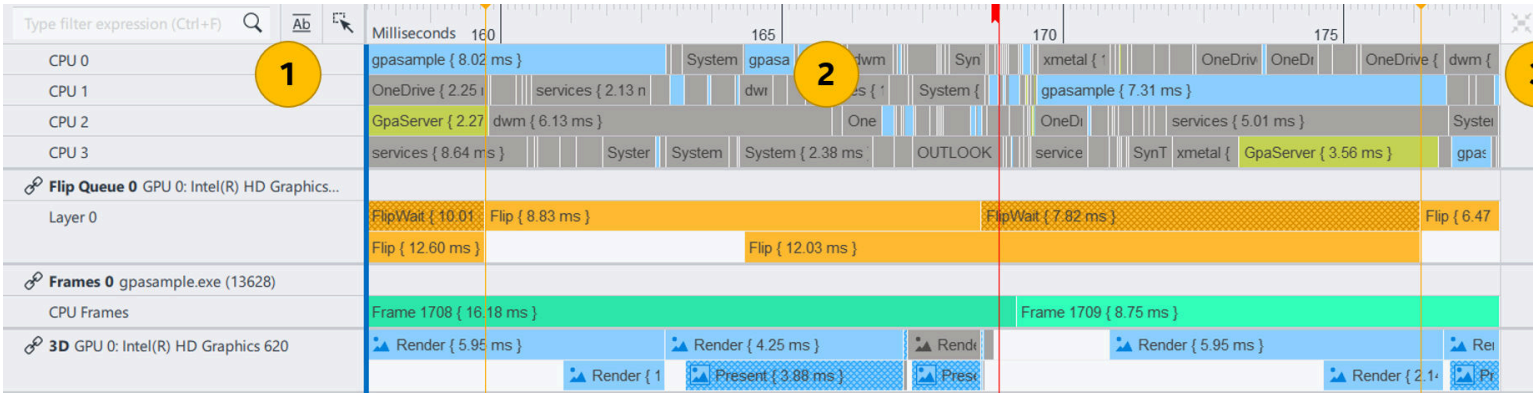
  Select Filtered Tasks button selects filtered events in the **Time Line Viewer** pane.
- Time line ruler correlates all the displayed events with the time line. The range of the time line corresponds to the length of the trace in the current zoom level. Depending on the zoom-in level, the time intervals are displayed in seconds, milliseconds, microseconds, on nanoseconds. The default trace length is 5s.

NOTE To mark a specific timestamp, you can create a bookmark on the time line ruler by double-clicking the necessary point on the time ruler.

- **Zoom to Selection** button zooms into the time range of selected events.
- **Full Zoom Out** button restores the default zoom level.

Time Line Viewer Pane

Use the Time Line Viewer pane to view and configure time line tracks of the traced processes. Each track comprises the following areas:



1

Track Description consists of a track name and name of the process or platform, which the track belongs to, as well as the percentage or time when the device was utilized for the trace area visible in the current zoom level.

2

Track Data is graphic representation of performance events and other data, such as metrics and queue packages.

3

Track Toolbar closes tracks and shows metric ranges.

- **CPU tracks**

  Visualize aggregated CPU activity reflecting thread execution on CPU cores. Use to analyze thread execution order, execution duration, and distribution between CPU cores. Blocks of the same color represent the same thread. If you zoom into the trace, you can see the process name the thread belongs to and thread execution duration.
- **GPU queues**

  Show GPU queues for all active video adapters generating graphics content. GPU queues can be of different types depending on the application. The Time Line Viewer pane visualizes GPU utilization over time: execution of the DMA packages on GPU. DMA package color corresponds to the color of the thread, from which the DMA package was submitted. All DMA packages have names visible on mouse hover. Additionally, DMA packages, which are essential for analysis, are marked with different icons depending on their type:

  - 

    Render package. Render package with a present call is crosshatched.
  - 

    Signal package
  - 

Wait package

- 

  Paging package

Selecting any of these packages shows an arrow that reveals calls in CPU threads related to that package. For example, you can trace the origin of a Render package from the CPU thread, to the User-Mode Driver, and up to the hardware queue.

- **Flip queues**

  Shows flip queues for all active video adapters. Flip queue reflects the relationship between the application present calls, present packages of GPU/CPU queues, and Vertical Synchronization (VSync) event of the monitor. Flip queue package consists of two blocks: solid-color and crosshatched. Solid-color block shows the time when Desktop Window Manager (DWM) is generating the content that needs to be displayed. The crosshatched block shows the time during which the content waits for the moment to be displayed (VSync).

  NOTE For applications utilizing layered flip queues, multiple layers can be displayed as sub-tracks of the general Flip Queue track.

- **CPU queues**

  Shows CPU queues for all processes generating graphics content. CPU queue represents ordered command packages to be executed, but not yet submitted to GPU. CPU queues can be of different types depending on the application. CPU queue package color corresponds to the color of the thread, from which the CPU queue package was submitted. All CPU packages have names visible on mouse hover. Additionally, CPU packages, which are essential for analysis, are marked with different icons depending on their type:

  - 

    Render package. Render package with a present call is hatched for packages named Present Token and crosshatched for packages named Present Render.
  - 

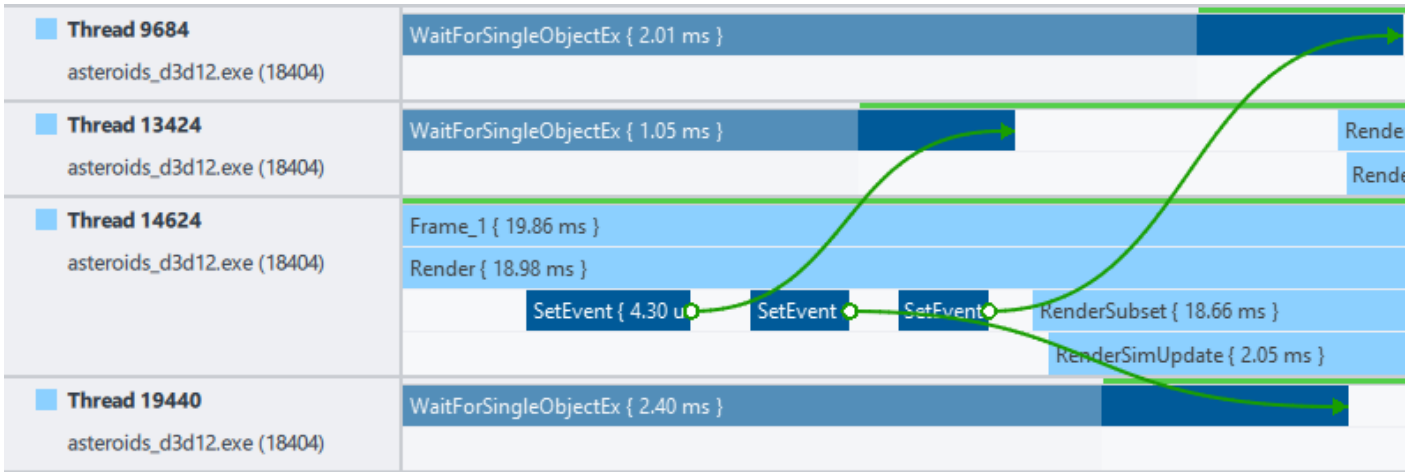
    Signal package
  - 

    Wait package
  - 

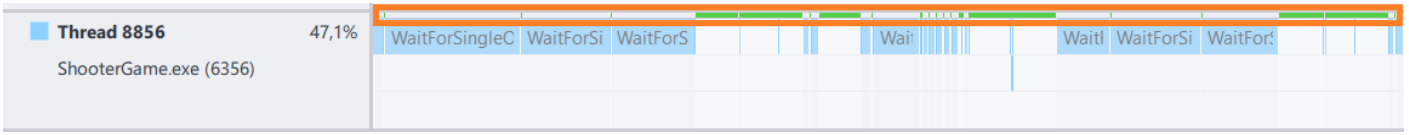
    Paging package

  CPU synchronization is represented as `WaitForSingleObject`, `WaitForMultipleObjects`, `SetEvent`, and `ReleaseSemaphore` function calls on the time line track of the threads. You can use these captured events to profile different synchronization issues.

  Click on any of these events to visualize dependencies between synchronization events in the form of arrows. For example, an arrow pointing from a `SetEvent` call in one thread to a `WaitForSingleObject` call in a different thread indicates that a thread was unblocked by a `SetEvent` call in another thread.



On CPU queues tracks, areas where a thread was active are highlighted as green bars. Areas where a thread was idle are shown as gray bars.



A thread is considered active and is highlighted accordingly if this thread was actively executed on a CPU core at this time.

NOTE To see OpenCL™ API calls on CPU tracks, enable OpenCL domain in **Options**.

- **Events**

  Shows events as markers that have a timestamp, but do not have duration. Markers scope can be global or process-defined. Markers are visualized as colored triangles. Global markers are placed on the Time Line ruler; process-defined markers are shown on executed threads.
- **Metrics**

  Shows any GPU/CPU metrics, which you enabled in System Analyzer or System Analyzer HUD.
- **Regions**

  Shows regions, which are logical application execution blocks, defined with Instrumentation and Tracing Technology API (ITT API) or graphics debug API. By default, block color corresponds to the color of the time line track. Each block has a name and duration.
- **Threads** track

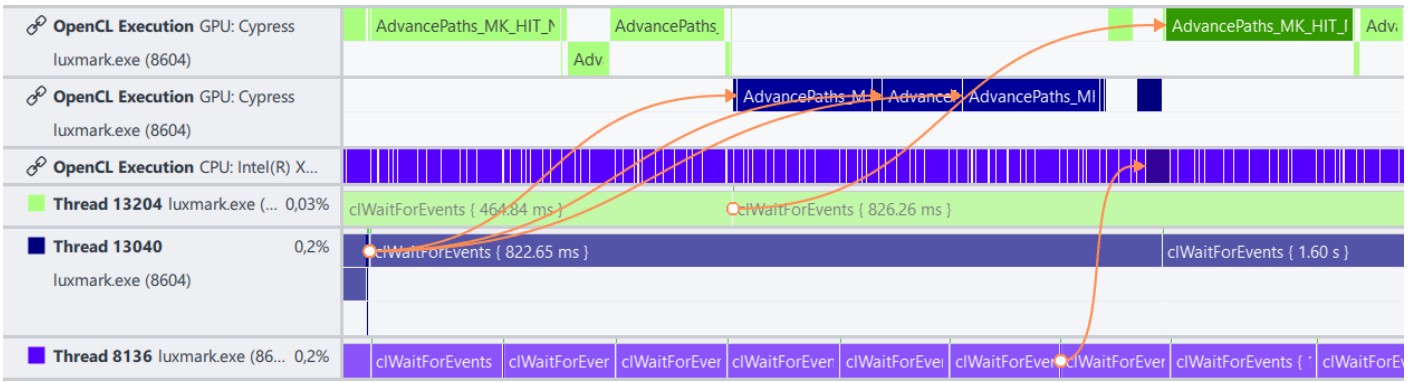
  Shows executed threads of profiled processes. Use this data to analyze your application performance and behavior based on ITT API and system events. Each colored block represents a logical block of application execution marked up by the user or generated by the system. Each block has a name and duration, and can have nested blocks. By default, block color corresponds to the color of the time line track.
- **Parallel Execution** track

  Parallel Execution track visualizes how the driver parallelizes execution of submitted render events.



- **OpenCL Execution** tracks

  OpenCL Execution tracks visualize execution of OpenCL kernels on a GPU or a CPU. To see the dependency between the tasks of submitting and executing a particular kernel, click any OpenCL packet or OpenCL API call on the CPU Thread track.

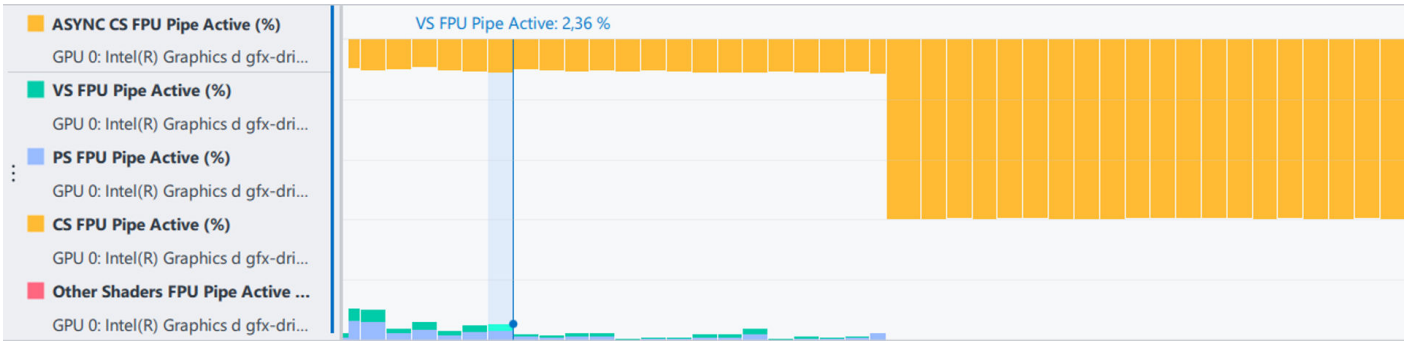


  The data is useful to spot synchronization issues or understand whether there is a problem in OpenCL code if you use different APIs. For example, if OpenCL Execution track is fully loaded, you can detect a problematic kernel for detailed profiling with Intel® VTune™ Profiler.

  To see OpenCL Execution tracks, enable OpenCL domain in **Options.**

- **Shader Breakdown** tracks (Windows only)

  Shader Breakdown tracks visualize workload distribution among shaders and asynchronous compute work for floating-point unit and extended math pipes.



  Shader Breakdown track contains two types of bars:

  - Bars on the top show the percentage of time when the Execution Unit (EU) was performing asynchronous compute work from the compute queue.
  - Stacked bars show the percentage of time per shader when the EU was executing shader instructions from the main graphics queue.

The track contains the data for the following types of shaders:

- Compute shader (CS)
- Vertex shader (VS)
- Pixel shader (PS)
- Other shaders, for example, mesh or ray tracing shaders

Hover over the bar to see the percentage above the track.

To see Shader Breakdown tracks, enable *AsyncCompute* metric set and select any GPU sampling interval except *Frame* in **Options.**

NOTE

- From the 9th to the11th generation Intel Core™ processor family, only shader workload distribution data is available in Shader Breakdown tracks. Asynchronous compute data is available starting with the 12th generation Intel Core™ processor family.
- Families of Intel® X^{e} graphics products starting with Intel® Arc™ Alchemist (formerly DG2) and newer generations feature GPU architecture terminology that shifts from legacy terms. For more information on the terminology changes and to understand their mapping with legacy content, see GPU Architecture Terminology for Intel® X^{e} Graphics.

Trace Content Pane

Type Filter Expression field	Filters trace file tracks by name or parameters. Saves previous search keywords.
Collapse All Groups button	Closes all unfolded track groups.
Expand All Groups button	Unfolds all track groups.
Track list	Includes aggregated information on trace file content in the following order: • **Bookmarks** enables you to mark any timestamp on the Time Line ruler. You can change the name and color of a Bookmarks Group. To add a bookmark: 1. Select the required Bookmarks Group to enable a bullet selector. 2. Click on a desired timestamp on the Time Line ruler. To delete a bookmark, do the following: 1. Select the required Bookmarks Group to highlight it and make the corresponding bookmarks visible. 2. Click on a bookmark to be deleted. 3. Press **Delete** keyboard button.

NOTE On the Time Line ruler, you can see only the bookmarks of the Bookmarks Groups highlighted in the Trace Content pane. You can edit only a Bookmarks Group that has a bullet selector.

- **Aggregated CPU activity** (CPU context switches)
- **Active GPU adapters** (GPU frames; GPU queues; Flip queues; VSync events)
- **Target instrumented process** for which the opened trace was captured
- **Other processes** in the system contributed to the trace

NOTE

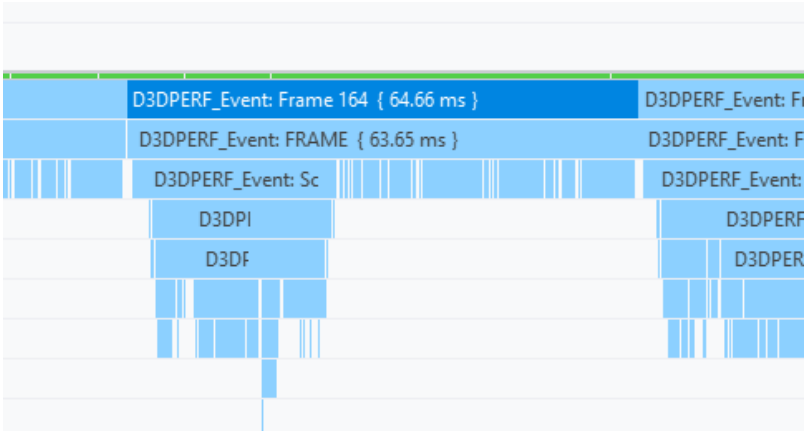
- Target instrumented processes and other processes contain CPU queues, executed threads, system events, regions, and metrics.
- To change track color or event color in the Trace Viewer, use the **Track Color Modification** button or **Marker Color Modification** button, respectively.
- Track from the Track list can be dragged and dropped to the Trace Viewer.

Selection Statistics Pane

Use the **Selection Statistics** pane to view summary information for the selected objects in the Trace Viewer. You can select objects in the Trace Viewer in several ways:

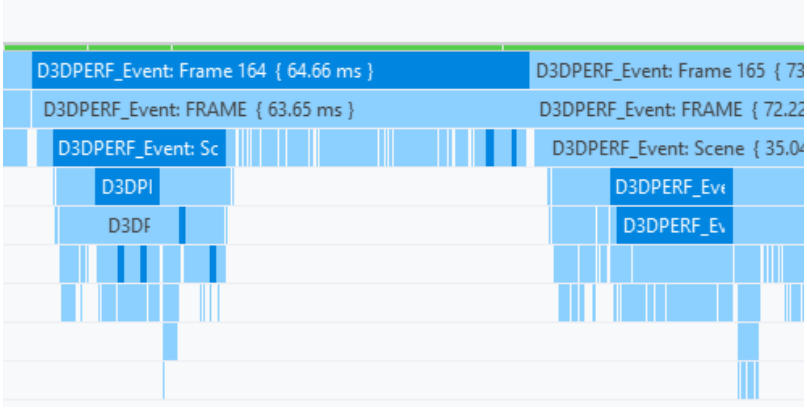
- Single Selection

  Left-click any object in the track.



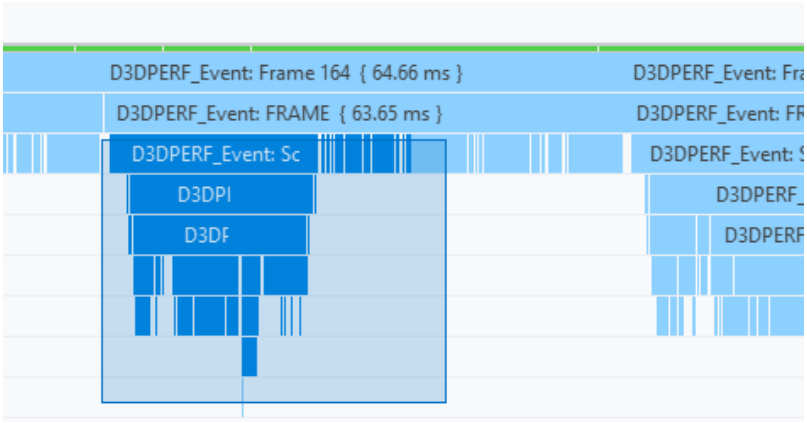
- Group Selection

  Left-click any objects in tracks while keeping the **CTRL** keyboard button pressed down.



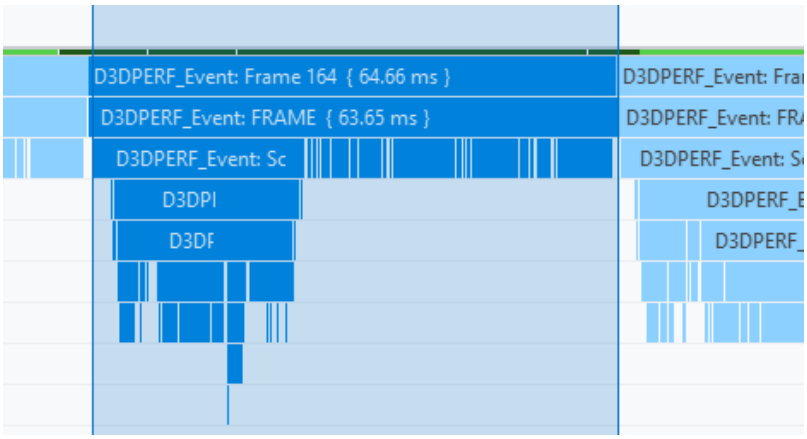
- Rectangular Selection

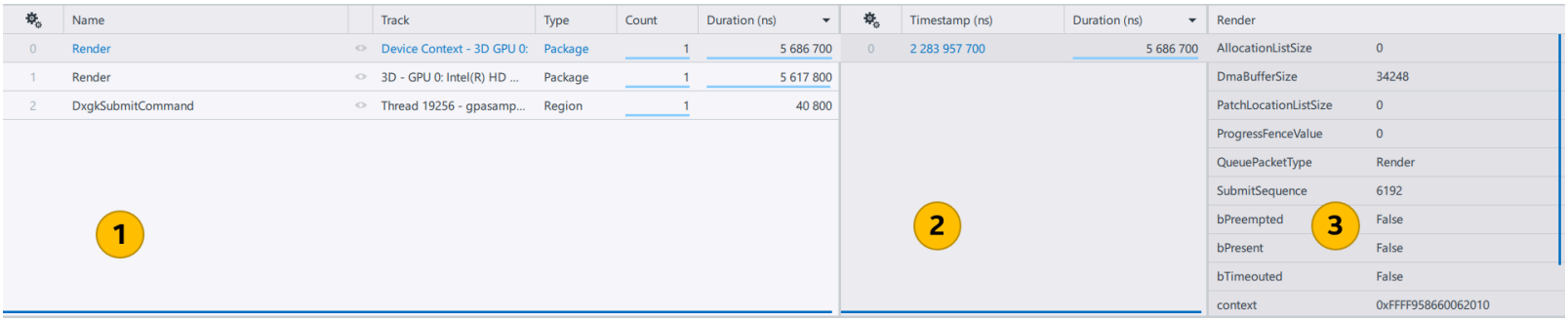
  Press the left mouse button and drag the mouse in any desired direction to select the required region.



- Range Selection

  Press the right mouse button and drag the mouse horizontally to select events in all tracks within the selected range. You can change the selection by dragging the right or left boundary. The selected range includes all the intercepted tasks.





1 **Aggregated Summary table**

All the selected events are grouped by event types:

- Packages
- Regions
- Markers
- Metrics

For all the selected events you can view the following aggregated information:

- Name - event name.
- Track Visibility Status:
  -

    Track is visible on working area - track is opened in the Trace Viewer.
  - 

    Track is not visible on working area - track is not opened in the Trace Viewer. You can open the track by clicking the

    button.
- Track - track name.
- Count - number of events with the same name in one track.
- Inclusive Time (ns) - total event duration or sum of total event durations, if the count value exceeds one.
- Exclusive Time (ns) - total event duration or sum of total event durations, excluding duration of child events, if any.
- Exclusive % - relation of Exclusive Time to Inclusive Time.

2 **Itemized Summary table**

For each selected item in the **Aggregated Summary** table you can view the list of events with the following information:

- Timestamp (ns) - relative time passed from the beginning of the trace before the task execution started.
- Inclusive Time (ns) - total event duration.
- Exclusive Time (ns) - total event duration, excluding duration of child events, if any.
- Exclusive % - relation of Exclusive Time to Inclusive Time.

3 **Performance Event Details table**

Shows arguments for the event selected in the **Itemized Summary** table. Available for GPU queue packages, CPU queue packets, and VSync events.

To customize the displayed columns, click the



button and enable/disable the required column.

See Also

Set Up Tracing options

Identify Issues in Graphics Application Execution with Trace Analyzer

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.