Profiling OpenGL/OpenGL ES* Frames with Graphics Frame Analyzer

Using Graphics Frame Analyzer, you can analyze performance of your OpenGL/OpenGL ES* application:

- Identify unsupported API extensions
- Locate API errors and RAW hazards in the captured frame
- Find bottlenecks and measure your game performance against a connected target system
- Experiment with render states without changing your game code to quickly find performance opportunities
- Optimize states and shaders within the Graphics Frame Analyzer framework
- Analyze application textures
- Correlate rendering issues with graphics pipeline stages

Identifying Unsupported API Extensions

Pre-requisites:

If you are analyzing an OpenGL ES application, the Android* device to test against must be connected to your host system. This is required to calculate metrics data. Metrics data is only available for devices based on the Intel® Processor Graphics.*

When developing OpenGL ES* applications, you may find that some of your target system do not support certain API extensions. Thus, while your application runs flawlessly on one device, it may crash on another. Graphics Frame Analyzer enables you to check your application compatibility with different systems with little effort. Using this tool, you can capture a frame on one system and then validate it against other systems for which your game is targeted. This enables you to test your application compatibility with various systems and Android* devices that may have different video cards and software installed.

To check whether your system supports all extensions used in the frame:

1. Start the Graphics Frame Analyzer.
2. In the Open Frame Capture window, choose the target system against which you would like to test your frame.
   Graphics Frame Analyzer collects information about all commands the selected system supports and compares them to the API calls used in the captured frames. If the player cannot replay any of the API calls, the frame capture thumbnail receives the ⚠ marker.
3. Select the frame capture with the ⚠ marker.
   The unsupported extensions are listed below the frame image in the Frame preview pane.
4. Open the frame capture to see which function in the frame uses the unsupported extensions.

If Graphics Frame Analyzer shows that your platform does not support certain extensions, using these extensions in the frame can result in rendering issues that otherwise may not occur. For example, extensions that result in visual changes, such as different texture formats or shader extensions, may cause geometry issues. You may need to change your application code before proceeding to further analysis and debugging.

Next Steps

- Create Frame Capture Files with Graphics Frame Analyzer, if you changed your application code.
- Locate API Errors, if you want to proceed with the analysis.

See Also

Graphics Frame Analyzer Window: Open Frame Capture

Opening a Frame Capture with Graphics Frame Analyzer

Pre-requisites:

If you are analyzing an OpenGL ES application, the Android* device to test against must be connected to your host system. This is required to calculate metrics data. Metrics data is only available for devices based on the Intel® Processor Graphics.*

To load a frame capture file with Graphics Frame Analyzer:

1. Launch the Graphics Frame Analyzer on the host system. The **Open Frame Capture File** window opens, displaying all frame captures available on your host system, in alphabetical order.

> **NOTE**
>
> If you need to create new frame capture files, click the **Add** button. All new frames captured while Graphics Frame Analyzer is running appear on top of the list with the NEW marker.

2. Select the frame capture file you want to analyze by clicking one of the frame thumbnails. The currently selected frame capture is highlighted with blue borders and appears in the frame preview field listing high-level details of the frame, such as filename, resolution, OpenGL/OpenGL ES API used, GPU details, and the name of the device on which this frame was captured.

   If your frame uses any extensions unsupported by the selected player, you will see a corresponding warning. You can use Left and Right arrow keys to navigate between the captured frames.

3. Make sure your target system is selected as the player to replay the captured frame.
4. Click the **Open** button to load the frame capture and start profiling.

> **TIP**
>
> If you are opening your frame on a different system, its rendering context may differ from the context of the system where the frame was captured. The difference in rendering contexts may affect performance and metrics data. To understand possible performance impact of the current rendering context, click the button. You can compare the original and current rendering contexts in the pop-up window.

Next Steps

- Locate API Errors and RAW hazards in Your Game
- Find bottlenecks and measure your game performance

See Also

Graphics Frame Analyzer Window: Open Frame Capture

Locating API Errors in the Frame

Graphics Frame Analyzer enables you to quickly locate API errors in the analyzed frame.

To find API errors in the frame:

1. Review the API log in the Profiling view to see whether any function has caused any issues. Graphics Frame Analyzer highlights such functions using the following color scheme:

- orange markers denote warnings
- red markers denote errors

You can review the whole call stack for each draw call by clicking the black triangle ▶ next to the draw call number.

2. Select the marked functions in the API log to review all resources related to these functions in the Resource viewer in the central pane. You can go directly to the highlighted function by clicking the colored marker in the scrollbar.

TIP

Click the **Show all** toggle button above the API log to view all functions in the frame. You can use Up/Down arrow keys or Page Up/Page Down keys for easier navigation.

Next Step

Find Performance Bottlenecks in OpenGL/OpenGL ES* Frame

See Also

Graphics Frame Analyzer Window: Profiling View

Finding Performance Bottlenecks in OpenGL/OpenGL ES* Frames

Pre-requisites:

If you are analyzing an OpenGL ES application, the Android* device to test against must be connected to your host system. This is required to calculate metrics data. Metrics data is only available for devices based on the Intel® Processor Graphics.*

Using Graphics Frame Analyzer, you can explore a variety of metrics to identify performance bottlenecks in the frame, and analyze performance dependency on different drivers or states of the hardware.

To identify performance bottlenecks in a frame:

1. Open your frame capture with Graphics Frame Analyzer.
2. In the Profiling view, choose the available metrics for X and/or Y axis to visualize specific aspects of performance in the frame.
3. Review the Bar chart to locate performance issues in the frame. You can analyze individual API calls that correspond to separate bars in the chart (default), or group them by render targets using the **Group by Render Targets** toggle button.
   The scrollbar below the chart provides an overview of the entire frame, while the slider reflects the part of the frame currently displayed in the chart. You can stretch/shrink the slider to change the scaling of the 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.
4. Select the bars that contribute the most to the frame time.

   The Metrics pane displays metrics information for the selected API calls/render targets. If multiple metrics are available for your device, Graphics Frame Analyzer groups them by specific hardware blocks they correspond to, such as Shader Execution or Rasterization blocks. The colored markers below the block indicate EU states for this block:

   - Green markers indicate the active execution state
   - Red markers indicate the stalled state caused by waits and holds on external dependencies
   - Gray markers indicate the idle state

IMPORTANT:

The colored markers only provide a high-level picture of the current activity on Intel® Processor Graphics for the selected region of API calls. Red color does not indicate performance bottlenecks directly.

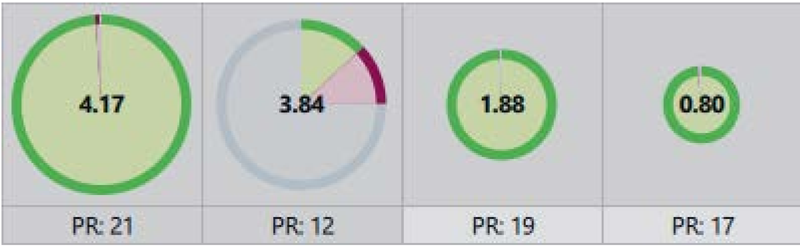
5. Analyze metrics values to identify performance opportunities within the selected region of API calls. The basic analysis methodology is as follows:

    1. Check that the selected region is not memory-bound for GTI bandwidth and/or L3. The main indicators of memory bandwidth issues are multiple red markers in other metrics blocks, as most of them depend on GTI/L3 memory interfaces.
    2. Check the Pixel Back-End block for possible issues. The overall frame performance may be limited by pixel back-end maximum throughput, measured in pixels per clock. This is a typical issue on mobile platforms. If this is the case, try reducing the number of pixels required for output or changing graphics state conditions, as some graphics states may reduce maximum throughput of pixel back-end.
    3. Evaluate Sampler and EU states.

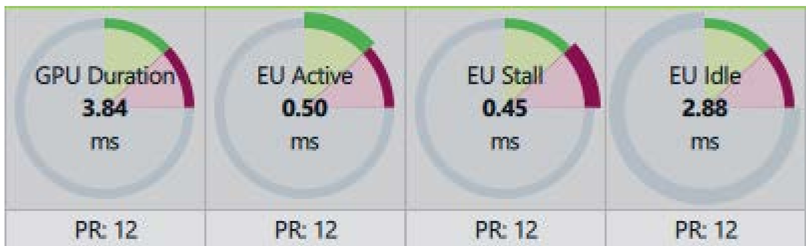
TIP

If you are opening your frame on a different system, its rendering context may differ from the context of the system where the frame was captured. The difference in rendering contexts may affect performance and metrics data. To understand possible performance impact of the current rendering context, click the button. You can compare the original and current rendering contexts in the pop-up window.

6. If your target platform supports GPU Duration, EU Active, and EU Stall GPU metrics, Graphics Frame Analyzer visualizes GPU duration for each program used by the selected API calls. The full circle of the pie chart represents GPU Duration of all the API calls where the program is used. For multiple API call selections that use more than one program, the size of the pie chart correlates with the GPU duration values displayed on the pie chart. Inner sectors of the pie chart represent the GPU time distribution between the EU Active (shades of green), EU Stall (shades of purple) and EU Idle (grey color):

    Hover over the pie chart to get details on EU Active, EU Stall, and EU Idle timings for each program. Each state receives a highlight outside the pie chart, and you can see the corresponding metric value, in ms:

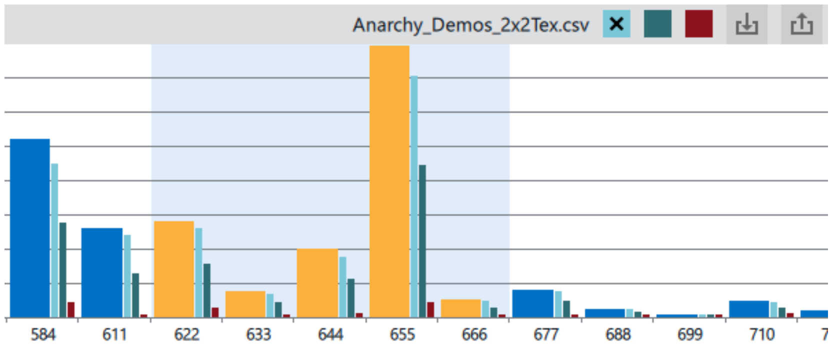
The origin of some bottlenecks may be hard to troubleshoot. For example, a tiny API call may turn out to be a bottleneck because of latency issues. However, in most cases, these steps should be sufficient to identify performance issues in your frame.

To analyze your frame's performance dependency on different drivers or states of the hardware:

1. Open the frame in Graphics Frame Analyzer in the Profiling view.
2. Click the Export button to save the metrics data in a CSV file.
3. Modify your device settings and reopen the frame file.

   Graphics Frame Analyzer calculates the new metrics data for your frame.
4. Import one or more saved CSV files into the bar chart using the Import button , or simply drag and drop them from the file explorer onto the bar chart.

   The bar chart updates to show the imported metric values as thin colored bars, side-by-side with the current data represented by thick bars:

NOTE:

You can only import metrics data that was exported for the same frame file, with the same set of metrics. You cannot compare metrics data collected on different platforms with different sets of metrics available.

---

5. Compare metric values in the chart to understand performance impact of the hardware settings. If you imported more than one metrics snapshots, hover over the rectangles above the chart to view the *.csv filenames. To remove the imported metrics data from the chart, click the corresponding rectangle.

Next Steps

- Experiment with Render States
- Optimize States and Shaders
- Analyze Application Textures

See Also

Graphics Frame Analyzer Window: Profiling View

Experimenting with Render States

Pre-requisites:

If you are analyzing an OpenGL ES application, the Android* device to test against must be connected to your host system. This is required to calculate metrics data. Metrics data is only available for devices based on the Intel® Processor Graphics.*

You can run experiments to see the effect that changing render states may have in terms of visual impact and performance time of the frame. These experiments simplify the stages of the graphics pipeline that often require a significant percentage of the total number of GPU cycles. The modification of these render states happens within the graphics driver itself, so you can see the results of these experiments without changing your application code.

Graphics Frame Analyzer provides the following experiments:

- 2x2 Textures
- 1x1 Scissor Rect
- Simple Fragment Shader - only available for OpenGL* ES 2.0 or higher and OpenGL 3.2 or higher (Core Profile)
- Disable Erg(s)

Use these experiments to isolate potential performance bottlenecks in your application. If a certain experiment significantly improves the performance of your application, you should examine what your application is doing at this pipeline stage that might cause the slowdown.

NOTE
You can only perform experiments if metrics data is available for your target system. Otherwise, the Experiments pane is hidden.

To run experiments:

1. Select one or more ergs in the Bar chart that represent a potential bottleneck.
2. Toggle the supported experiment between "on" and "off". You can run multiple experiments at the same time on the selected ergs.
   The Render Target in the Output section of the Resource bar update to reflect the visual changes caused by enabling/disabling the experiments. Graphics Frame Analyzer also recalculates the metrics data and displays the updated results in the Overview tab.

2x2 Textures

Use the **Texture 2x2** experiment to identify potential performance bottlenecks caused by texture maps used in your application. All textures for a scene are replaced with 2x2 texture containing four different colors.

If using this experiment significantly improves the frame rate, this is an indicator that the application is bound to memory bandwidth, or there is an excessive amount of sampling in fragment shaders.

1x1 Scissor Rect

The **1x1 Scissor Rect** experiment 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. In this case, proceed to analyzing the geometry of the frame, or experiment with shader code.

However, the effect of this override mode is driver/graphics hardware vendor specific and depends upon whether scissoring on your device occurs prior to or after the pixel shader stage.

Simple Fragment Shader

The **Simple Fragment Shader** experiment replaces every fragment shader with the fragment shader that writes a constant color to the render target.

If the frame rate shows a large increase with this experiment enabled, this is an indicator that the performance slowdown is caused by fragment shader computation and/or texture sampling.

Disable Erg(s)

Use this experiment to keep the selected ergs from being rendered. Use this option to test scene efficiency.

See Also

Graphics Frame Analyzer Window: Profiling View

Identifying Graphics Elements with Complex Geometry

Geometry analysis can help you both debug rendering issues with the complex objects and optimize your application performance. To analyze geometry for the selected API call:

1. Select a single API call in the bar chart or from the API log 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 viewer. The **Geometry** pane 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 back side of the object and understand who it is built from triangles.

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

   Normal - 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.

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.

See Also

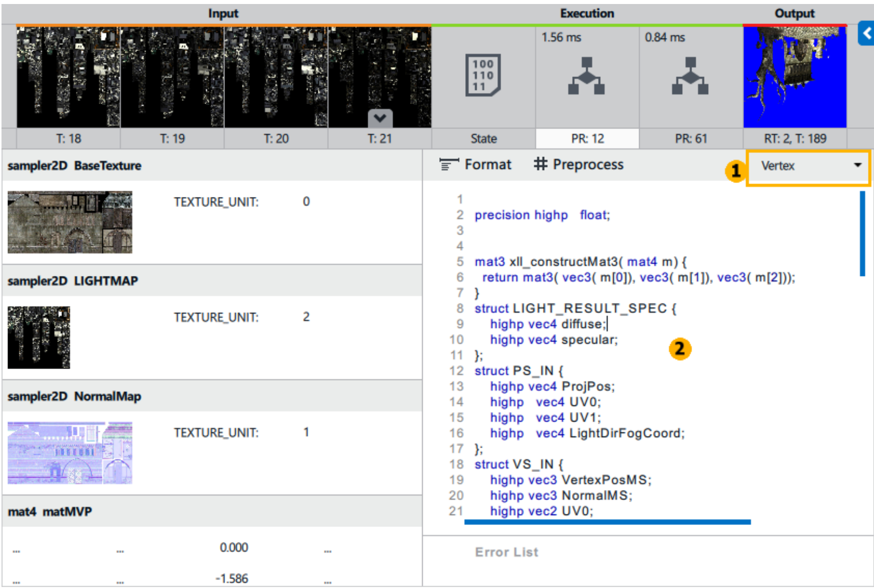
Graphics Frame Analyzer Window: Profiling View

Optimizing States and Shaders

If the Metrics panel shows that shaders cause a performance bottleneck for the selected draw calls, you can change the shader code inside Graphics Frame Analyzer to check the performance effect without recompiling your code.

To experiment with shader code:

1. From the Resource viewer, choose the program used in the analyzed draw calls.
2. In the drop down menu 1, select the type of shader you would like to analyze. Graphics Frame Analyzer supports vertex, fragment, compute, geometry, and tesselation shaders. The shader code opens for editing in the pane 2 below:



3. Edit the shader code to recompile the shader on the fly. If you introduced any errors, you can see the corresponding message in the **Error List** below the shader code pane.
4. If the code looks fine, click the **Apply** button to save the changes.
   Graphics Frame Analyzer recalculates all metrics and displays new data in the Overview panel and in the Bar Chart.

TIP

When you click the **Apply** button, Graphics Frame Analyzer save all the shaders. This enables you to write your own code and replace the whole shader to experiment with binary programs that you cannot modify.

---

5. If you want to undo your edits, click the **Revert** button. The original shaders are restored.

To experiment with states:

1. From the Resource viewer, choose the state thumbnail to experiment with.
2. 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, 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 OpenGL/OpenGL* ES 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.

See Also

Graphics Frame Analyzer Window: Profiling View (OpenGL/OpenGL ES* Workloads)

Analyzing Application Textures

Graphics Frame Analyzer enables you to review all textures for the selected ergs and determine whether this aspect of your application can be optimized. Using high-resolution textures, non-compressed formats, or multiple textures within a frame can negatively affect your application performance. To speed up the rendering, you can try to:

- reduce the size of textures
- reduce the number of textures used in the scene by using one texture for a set of objects
- reduce the texture filtering setting
- reduce the number of texture fetches in the shaders

To review a texture used in the frame:

1. In the Bar chart, select the ergs that use the texture you would like to analyze. To preview all textures used in the frame, you can select all the ergs.
2. Select the texture thumbnail in the Resource viewer.

   Graphics Frame Analyzer displays the texture preview, alongside all the texture parameters. At the same time, all ergs that are using this texture get highlighted with an orange marker in the Bar chart.

   For 3D textures and image textures, you can also select the image layer from the Layer input control.

You can also experiment with storage parameters of a texture to determine whether specifying different parameters during texture creation can reduce memory bandwidth and/or sampler utilization.

NOTE

You cannot modify the storage parameters in the following cases:

- The texture is used in the frame as a render target or an image texture.
- The texture is a cube map or a cube map array.

To experiment with the selected texture:

1. Modify storage parameters of a texture:
   - Reduce each of the texture dimensions.
   - Change the sample count in the range between 1 and the maximum number of samples supported by the platform.
     This option is only available for OpenGL desktop applications.

   CAUTION

   Changing the sample count may break rendering since the original texture is replaced with a multisampled one. Multisample textures also require changes to shaders sampling from those textures, as multisampled textures have a dedicated sampler type and texel fetch functions. These changes are not applied automatically and you have to make them manually for each draw call that uses the original texture as input.

   ---

2. Click the button to apply the changes.
   A new texture is created with the specified parameters and the contents of the original texture is scaled to the new size. This texture replaces the original one across the entire frame.

You do not have to rebuild the application and re-capture the scene to see the effect of this experiment: render targets and the metric values get updated automatically.

You can export the texture image by clicking the button or pressing Ctrl+S. You can choose between DirectDraw Surface (DDS), Khronos Texture (KTX), or PNG formats.

See Also

Graphics Frame Analyzer Window: Profiling View

Minimizing Overdraw

With the Graphics Frame Analyzer, you can discover which API calls influence specific pixels to determine redundant or unimportant calls and minimize overdraw. 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. In the Profiling view, select the render target from the Resource viewer.
2. Click a pixel you would like to analyze. Graphics Frame Analyzer marks the selected pixel with a crosshair and filters the API call to display only those clear, draw, blit, or texture image update calls that affected this pixel.
3. Review the pixel history in the API log. You can see the pixel coordinates, pixel components, the number of times each draw call touched the pixel, and rejected draw calls.

TIP

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.

4. Analyze the calls that affected the pixel to determine what you can do 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 Erg(s)** check box in the Experiments tab.
- Detect API calls used by mistake. Remove such calls by checking the **Disable Erg(s)** check box in the Experiments tab to see the potential performance improvement.

NOTE

You can only perform the **Disable Erg(s)** experiment if metrics data is available for your target system. Otherwise, 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.

See Also

Graphics Frame Analyzer Window: Profiling View

Correlating Rendering Issues with Graphics Pipeline Stages

If you locate a function that causes a rendering issue, you need to understand the origin of the issue and find possible solutions. Using Graphics Frame Analyzer, you can identify the exact graphics pipeline stage at which the issue occurred and perform in-depth rendering analysis:

1. Double-click the function that causes an issue to drill down to the graphics pipeline view.
2. Review the thumbnail images that illustrate the effect of the selected function at each rendering stage. Once you see an unexpected result, review the detailed data about the rendering at this stage to understand the root cause of the issue.

The table below provides some tips about typical rendering issues and possible solutions.

Symptom	**Suggested Solution**
An object has disappeared from the screen at the Transform & Lighting rendering stage.	Check the data in the world view and/or projection matrices provided by Graphics Frame Analyzer at this stage. Abnormally big numbers may cause the issue.
The texture of the object is broken.	Check the Transform & Lighting rendering stage. You may have wrong values in the texture matrix.
Vertex Shader thumbnail shows unexpected results.	Check the detailed data provided for the Vertex Shader rendering stage: • The shader code may have an error. • A uniform variable may have a wrong value.
Rasterization stage shows signs of Z-fighting issues for the selected draw call.	Try to enable polygon offset to avoid this issue.
An object has disappeared from the screen at the Pixel Processing stage.	Check depth, stencil, and scissor test parameters at the Render Target of Frame Buffer stages. Incorrect configuration at one of these stages may cause this issue.

See Also

Graphics Frame Analyzer Window: Graphics Pipeline View