Practical Game Performance Analysis Using Intel® Graphics Performance Analyzers

By Sheng Guo, Philipp Gerasimov, and Bonnie Aona
Software and Services Group, Intel Corporation

Abstract

Improve the performance of your games running on Intel® HD Graphics (IHD) platforms using Intel® Graphics Performance Analyzers (Intel® GPA) with the performance analysis methods presented in this article. Intel GPA is a suite of graphics performance optimization tools that enables developers to visualize, isolate and resolve graphics performance issues for Microsoft® DirectX-based games and other graphics applications. In addition to the information presented here, we recommend you review the Intel HD Graphics Quick Reference Guide, and other Intel Graphics-related information at the Intel Integrated Graphics overview page to understand how to use Intel HD Graphics and Intel Integrated Graphics features effectively for your game.

1. Introduction

The performance of video games on integrated graphics has become a more critical issue for game developers due to two trends. First, the latest Intel integrated graphics platforms provide a high-performance feature set; for example, the Intel® 4 Series Express Chipset supports Shader Model 4.0 [SM4.0] and DirectX* 9/10 specifications that meet the requirements of most of the games available today. Second, the price/performance ratio of integrated graphics has spurred rapid adoption of the chipset for the mobile laptop market, increasing its viability as a graphics solution for game players. Laptops represent not only a mobile extension to the gamer’s traditional setting, but also an opportunity to tap into the casual and social game player segment. Optimizing your game’s performance on integrated graphics is critical to improving odds of commercial success for your game as it increases your potential market. This article provides practical advice and examples for optimizing your game.

It’s important to have a goal in mind prior to starting analysis and optimization of your game. For example, you might want to achieve 30 frames per second on a 1280x1024 screen with specific gameplay and graphics settings. Intel GPA can assist you in identifying performance bottlenecks for specific aspects of your game, such as excessive vertex shader use or hardware/driver bandwidth limitations. It is also a visualization tool that represents the positive or negative effect of making a specific code or asset change to your game.

Game developers may also want to analyze and optimize for specific target platforms. The most common mainstream platforms are laptops with the latest Intel HD Graphics chipset, where you have a very large install base due to the volume of systems purchased with these graphics chipsets. On the other hand, hardcore gamers are more likely to buy the highest performing system independent of the cost, so you will want to enable every possible visual effect on those platforms to help increase sales of your game.

Since Intel GPA runs on most applications based on Microsoft DX 9, DX10.0, or DX 10.1, we can help you understand the best optimizations for the DX level supported by your Intel HD Graphics platform. For example, once you’ve optimized the basic scene rendering for your game on a laptop, those changes will carry over to other platforms. You will then be able to determine whether the cost of specific visual effects, such as fog or detailed shadows, is appropriate for your target frame rate.

We recommend that after you use Intel GPA to identify possible improvements appropriate for your target goals and incorporate those changes into your game code, you should verify that your changes achieve the expected performance improvements. To verify this, do two things: first re-run Intel GPA with the new code base to ensure the visual and performance changes show the improvements you expected; second, re-analyze the game with Intel GPA to pinpoint additional “hot spots” for further analysis and optimization.
A common method for reducing bottlenecks, but one that typically has a low success rate, has been changing the load or method for one of the front-end rendering stages to see if that adjustment would have a positive impact on downstream stages' rendering loads, or if the change simply moved the bottleneck to a later stage. This paper discusses more efficient methods for identifying and resolving bottlenecks in games using Intel GPA System Analyzer and Intel GPA Frame Analyzer, and includes examples that demonstrate the tool's use within different stages of the rendering pipeline. For all your game improvement goals, we believe that Intel GPA will become an integral tool for analyzing and optimizing your game.

1.1 Complexity of Performance Analysis

Optimizing game performance is a challenge due to the complexity of the graphics rendering pipeline and the possibility of bottlenecks in multiple rendering phases:

1. Rendering of each game scene can result in different bottlenecks, and various objects within a scene may encounter a completely different set of bottlenecks than in another scene. For example, a scene rendered using multiple, complex textures may experience a bottleneck due to memory access latency, especially when these textures are not cached in graphics processor unit (GPU) memory. Objects rendered using a pixel shader may experience a processing bottleneck for the last batch of pixels from the pixel shader, since earlier fragments are rendered as the geometry is output from the rasterizer. Finally, use of Microsoft DirectX* (DX) locks and graphics processor unit (GPU) queries may transfer bottlenecks from one stage of the rendering pipeline to another.

2. Bottlenecks that occur in the lowest level of the system are more difficult to identify due to the lack of accurate measurement methods at that level; for example, programming the graphics card's video RAM via DirectX or OpenGL*.

3. The graphics processing pipeline (GPPL) renders game scenes and objects in distinct functional stages for 3D geometric rendering:
   a. Application stage is implemented in software. Bottlenecks common in this stage are associated with CPU usage, which can be addressed by using optimizations built into the compiler, implementing parallelism via multi-threading, or offloading some of the graphics-oriented functions to the GPU.
   b. Geometry stage can be implemented in software and/or hardware, performing polynomial functions, vertex operations, transformations, and clipping. Typical bottlenecks in this stage involve processing time required to perform transformations and clipping, which can be addressed by using an appropriate level of detail for each scene, and employing other methods for rejecting trivial primitives that do not contribute to the visual quality of the frame.
   c. Rasterization stage is implemented in software and/or hardware, and might experience fill or interpolation bottlenecks. These bottlenecks can typically be addressed using the methods discussed in section 3b (above).
   d. Pixel shading and texture access stage is implemented in software and/or hardware, and interacts with the rasterization stage. Typical bottlenecks include complex pixel shaders, complex textures, and ray tracing. Improving these bottlenecks depends upon finding ways to reduce the overall complexity of these features while maintaining a visual quality that immerses the user in your game.
   e. Frame buffer stage is the final preparation for display. Implemented in software and/or hardware, typical bottlenecks in this stage occur in fragment re-association and frame buffer memory access for operations such as stencil shadows and alpha blending. Again, you need to identify tradeoffs between rendering time and visual quality. This is not always an easy job but one that you can accomplish using Intel GPA's experiments to make analysis and optimization choices to see how they benefit your game's performance.

1.2 Intel GPA System Analyzer and Intel GPA Frame Analyzer

Intel GPA System Analyzer is a tool that can help identify and isolate issues across four primary hardware categories: CPU, GPU, Bus and Memory (CGBM). Intel GPA Frame Analyzer is an in-depth frame analysis utility useful in exploring issues specific to frame rate and the many aspects of frame drawing complexity.
System Analyzer displays game performance metrics for the CPU and GPU via an interactive, real-time GUI that allows you to select DirectX-level overrides, invoke a simple pixel shader, and null the driver and/or hardware to investigate whether your game is CPU-bound and/or GPU-bound. You can then perform "what-if" experiments to identify the rendering phase(s) where your game's performance bottlenecks are concentrated. If System Analyzer shows your game to be CPU-bound, perform additional fine-tuning of your game code using the optimizations built into the Intel® Professional Edition Compilers, and using Intel performance optimization products that identify opportunities for parallelism, such as Intel® Parallel Advisor or Intel® VTune™ Performance Analyzer.

Frame Analyzer performs analysis at the frame, region, and draw call level. Features include draw call bar chart visualization, scene overview, render target viewer, a rich set of experiments that allow you to see the impact of eliminating portions of the rendering process within your game by using a simple pixel shader, simplified 2x2 textures, a 1x1 scissor rectangle for examining pixel rendering times, and selective texture and shader control.

Specific Intel GPA features can also be used to debug games. For example, the wireframe override mode of the System Analyzer and Frame Analyzer lets you examine scene objects that overlap each other to see if they are drawn correctly in geometry, and help you identify objects that might be better rendered with a reduced geometry. Frame Analyzer can help you examine the sequence of DirectX calls within each frame and modify the DirectX states on the fly so the effect of the DirectX calls can be seen without making code modifications.

If performance bottlenecks are found in the GPU, Frame Analyzer allows you to drill down within a single graphics frame to pinpoint specific rendering problems in texture bandwidth, pixel shader performance, level-of-detail (LoD) issues, and other bottlenecks within each portion of the rendering pipeline. After each experimental adjustment, you can review the improvement in rendering time and the visual quality of the result, all in real-time.

The rest of this article focuses on practical performance analysis methods for games and other graphics applications running on Intel HD Graphics using the Intel GPA tools.
1.3 Overview of Game Analysis with Intel GPA

Intel GPA System Analyzer

- Identify CGBM Bottlenecks (2.1)
- Analyze Load Distribution (2.2)
  - DirectX Analysis (2.3)

Optimize Game Application using Performance Tools (Intel® Vtune Performance Analyzer or Intel(R) Parallel Advisor, Compilers, Threading Technologies, Microsoft* Perfmon*)

Bottlenecks in System level? Y N

GPU Bound? Y N

Intel GPA Frame Analyzer

- Identify Expensive Draw Call Sets (3.1, 3.2)
- Perform Experiments (3.4)
  - Monitor the Render Target
- Analyze the Primitive Batch Size of Draw Calls (3.3)
- Analyze Draw Order of Scene Objects (3.5)

Re-assess frame rate, rendering target, and visual quality

Done

Rendering bottlenecks or quality problems? Y N

Done
2. System Analysis Methods

To pinpoint game performance bottlenecks, we begin with system profiling.

2.1 CPU, GPU, Bus and Memory (CGBM) Domain Analysis

Game rendering load is distributed across four primary domains: CPU, GPU, Bus and Memory (CGBM). You can use the CGBM methods available in Intel GPA to identify bottlenecks in each domain.

1. **CPU Bottlenecks**: System Analyzer provides a group of performance counters for CPU utilization. The closer the average CPU utilization is to 100%, the more likely there is a CPU bottleneck. Frequency of CPU access, even with a low average CPU utilization, can result in high CPU touch rate, e.g., ~100%, pointing to CPU bottlenecks and an unstable or unbalanced CPU load. Use the Null Driver test described in section 2.2; if the frame rate does not increase, this indicates your game is CPU bound. CPU bottlenecks can be addressed by using optimizations built into the compiler and implementing parallelism via multi-threading. Intel’s optimizing compilers are available in the Intel® C++ Compiler Professional Edition and Intel® Parallel Studio products. Intel® Threading Building Blocks (Intel® TBB) is a library that offers a rich and complete approach to expressing parallelism in a C++ program to take advantage of multi-core processor performance without having to be a threading expert. OpenMP* threading technology is also fully supported by the compilers.

Game developers need analysis tools that enable them to measure performance over time, then quickly and easily zero-in on bottlenecks and other issues that may be hindering performance. To do this effectively, developers must be able to map performance back to their specific game engine environments. The latest Intel GPA 3.0 release includes a tracing infrastructure that makes this contextual analysis possible.

Intel GPA 3.0 ships with an application programming interface (API) that enables developers to instrument their game engines with lines of tracing code. After adding tracing and recompiling the game, a developer can then use the Intel GPA 3.0 Platform View tool to obtain context-specific performance analysis based on the areas of the engine that were instrumented. Mapping performance to the specific game context in this way makes it easier to precisely identify bottlenecks and experiment with changes that might resolve them.

2. **GPU Bottlenecks**: To evaluate the possibility of bottlenecks in GPU processing, use the Null hardware override mode within System Analyzer’s to evaluate the impact of removing all GPU related-workloads. If System Analyzer then shows an increased frame rate for your game, this points to GPU bottlenecks. Sections 2.2, 2.3, and all of section 3 in this paper are devoted to identifying and addressing various types of GPU bottlenecks.

3. **Bus bandwidth bottlenecks**: Since video memory for integrated graphics resides primarily in system memory, the bus bandwidth between system memory and the GPU may be a source of bottlenecks. Use the System Memory Overall Bandwidth metric within System Analyzer to view the current bandwidth as a percentage of the maximum bandwidth for the platform. According to current device driver technology, the actual maximum of the bus bandwidth that a platform can maintain is 65%~70% of its theoretical peak. Bus bandwidth bottlenecks cannot be directly resolved by the game developer, but the techniques discussed in this paper for reducing GPU load can improve the usage of the available bus bandwidth.

The peak double data rate bandwidth is dependent upon your particular system configuration, so consult the detailed specifications for your system at [http://www.intel.com/Products/Desktop/Chipsets/](http://www.intel.com/Products/Desktop/Chipsets/) to understand the peak bandwidth and whether you are fully utilizing this bandwidth within your game. Essentially, systems built with processors that have higher Front Side Bus (FSB) frequency, higher processor multiplier values, and more channels will result in faster rendering. Please note that Intel® Core™ microarchitecture works with Intel® QuickPath Interconnect instead of FSB, so operates at much higher frequencies and is simultaneously bidirectional. Intel QuickPath Interconnect should improve performance of multipass rendering used in most of today’s games.

4. **Memory Bottlenecks**: Use the Microsoft* Perfmon* tool to identify processes that may be causing bottlenecks
due to increased memory paging. Review your code to ensure you are releasing memory resources as soon as possible, increasing the reuse of previously allocated memory, and reducing the number and complexity of your textures wherever possible (see Experiments 3.4.2, 3.4.3, 3.4.4 and 3.4.5 in the Experiments section of this paper).

2.2 Load distribution analysis

Use the state override modes within System Analyzer to investigate load distribution for the rendering pipeline phases (Figure 1).

1. Null Driver Mode: Use this override to remove both graphics driver and the GPU loads to evaluate the impact of the application’s CPU utilization on the entire frame rate. This mode also lets you see the theoretical maximum performance of the game on your system by removing the work of the driver and graphics card without modifying any other configurations.

2. Null Hardware Mode: Use this override to remove the GPU load to see if the game is GPU-bound vs. CPU-bound. Significant improvement in frame rate using this mode indicates your game is GPU-bound. Section 2.3 and all of Section 3 in this paper help you address GPU-bound games. Comparing the results with the game’s previous mode will let you see whether the driver or API could be a bottleneck.

3. 1X1 Scissor Rect: On Intel HD Graphics-based systems, this override reduces pixel processing to one pixel per draw call by discarding all pixels after the pixel shader has run but before the pixel values are written to the render target. Significant improvement in frame rate in this mode indicates pixel fill-rate is a bottleneck. Several experiments described in section 3 help you address pixel processing bottlenecks.

![Figure 1: Load Distribution of Rendering Process](image)
System Analyzer measures frame rate and frame time in determining whether slowdowns and bottlenecks are associated with the CPU, GPU, or Microsoft DirectX® (DX) runtime operations. By enabling each override mode in System Analyzer to see the resulting frame times, you can calculate the four time spans (T1, T2, T3, T4) in a load distribution chart. Mark your target frame time in the load distribution chart, and compare it with T1/T2/T3/T4 to diagnose whether you can achieve the target performance by just optimizing the load on the graphics card, for example, by selecting a different shader. If not, you may need to optimize the application level code. Then begin your bottleneck analysis and conduct the various experiments we describe, re-checking your game's frame rate and visual quality after each experiment or set of related experiments.

From the System Analyzer load distribution chart, you can draw several initial conclusions discussed below. Please note the order of discussion is T4, T3, T2, T1 because it follows the usual analysis order for optimizing performance of the graphics pipeline. Looking for the bottlenecks from the back-end to front-end of graphics pipeline is the recommended approach for analysis and optimization of your game.

- **T4:** When Intel HD Graphics Post-Scissor Process time is greater than zero, this indicates bottlenecks in pixel processing. See Section 3.4 for several experiments that address pixel processing bottlenecks.

- **T3:** When Intel HD Graphics Pre-Scissor Process time is high, this indicates bottlenecks in vertex processing. The goal is to see Pre-Scissor Process time close to zero. Vertex processing bottlenecks can be addressed by reducing the mesh LoD complexity for object meshes, using vertex shaders with simpler transform and lighting (T&L) algorithms, and applying an occlusion culling query to eliminate hidden draws.

- **T2:** When Intel HD Graphics driver time occupies a significant proportion of the entire frame time, investigate potential inefficiencies in DirectX API calls and whether the application is copying large amounts of data unnecessarily to the GPU via the graphics driver. See Sections 2.3, 3.1 and 3.3 for methods to address bottlenecks related to DirectX calls.

- **T1:** Evaluate the upper limit of your game's frame rate for the application level. When the frame rate is lower than frame rates for other games of the same type, consider optimizing the application code, including DirectX middleware, by identifying opportunities for parallelism. Use Intel® Parallel Advisor or Intel® Vtune Performance Analyzer to analyze the hotspots for implementing parallelism, then use Intel TBB or another threading technology to parallelize your application code.

### 2.3 DirectX analysis

System Analyzer provides a group of performance counters on DirectX (Figure 2) that calculate the time consumed in DirectX calls to analyze their impact on subsequent phases of the rendering pipeline:

- DirectX draw calls generate performance overhead, and can significantly increase the time the application spends in the driver, especially when each DirectX call contains a relatively small number of primitives. Compare the value of the **DX Draw Calls per Frame** counter for your game with the average value for similar games.

- Frequent switching of DirectX states leads to severe performance overhead. Use the counters **DX State Changes per Frame** and **DX Draw Calls per Frame** to calculate the number of DX state switches per draw call, and compare with similar games, then use the methods discussed in Sections 3.1 and 3.3 in this paper to identify reductions in DirectX calls.

- DirectX locks force synchronization where the GPU waits for the CPU. Use the counters **DX Locks per Frame** and **DX Lock Percent of Frame Time** to identify the impact of DX locks in your code, and identify any DX locks that can be eliminated.
System Analyzer can help you identify frames where hitching occurs in your game, i.e., a slight freeze or stutter every few seconds of game play while moving or occasionally while the figures are standing still. If you notice hitching, look at graphs that have frequent spikes to pinpoint the frames where hitching occurs. Hitching can be addressed by reducing the number of textures your game is using, turning off anti-aliasing, and by ensuring the latest driver for the video card is being used. You might need to look at the graphs under different sampling settings to see when the hitching or jumping happens.

![DirectX Counter Sampling](image)

**Figure 2: DirectX Counter Sampling**

3. **Single Frame Analysis Methods**

Use the following methods to analyze the cause of a slow frame rate for any single frame.

3.1 **Identify expensive draw calls**

Analysis of draw calls is a key to improving game performance since draws are relatively expensive operations that account for a significant portion of a game’s GPU time. Use Frame Analyzer to sort all draw calls by GPU Duration, then investigate the most expensive draw calls (Figure 3); that is, examine the calls that account for the highest percentage of the entire frame time. Keep in mind that although a particular draw call may be consuming most of the GPU time, it may utilize only a small percentage of the entire frame time; thus, optimizing a single expensive draw call may not result in a noticeably improved frame rate.

Instead, evaluate a batch of draw calls with the same characteristics, e.g., calls using the same textures or the same pixel shaders, and calls that render the same kind of objects (terrain, vegetation, etc.). Frame Analyzer identifies batches that occupy a significant proportion of the entire frame time, which you can evaluate for optimization.
opportunities. On the Shader Tab within Frame Analyzer, right-click on the shader associated with a selected draw call, selecting the batch of draw calls using the same shader (Figure 3). On the Texture Tab, right-click on the texture associated with a selected draw call, selecting the batch of draw calls using the same texture. After identifying expensive draw call(s) or batch(es) of draw calls, perform an in-depth analysis on them using the methods and experiments described in sections 3.3 (draw batch size), 3.4 (experiments with pixels, shaders, textures, etc.), 3.5 (draw order), and 3.6 (changes in rendering targets).

3.2 Examine the percentage of frame time for selected erg(s)

It is often useful to understand the percentage of total frame time your game spends on a particular visual effect, such as character rendering or HDR (High Dynamic Range) tone mapping. On Figure 3, you can see an example of the time for selected erg(s), which is the unit of work or energy, based on the centimeters, grams and seconds measurements standards, where 1 erg is equivalent to $10^{-7}$ joule. If the time looks too expensive, consider selecting a more optimized effect, and even disabling an expensive effect for certain classes of devices.

3.3 Analyze the primitive batch size of draw calls

The expense of DirectX draw calls can be minimized by batching the appropriate number of primitive calls into one draw call. For Intel HD Graphics, we recommend a batch size for primitive calls of 200 to 1000. Use Frame Analyzer’s
Prim Count (Figure 4) to analyze draw calls with small primitive batch sizes, and consider merging them into larger batches. Evaluate draw calls with large primitive batch sizes by observing the screen pixel coverage and LoD for their objects. Techniques for simplifying LoD include shader management to control pixel complexity (see the pixel-related experiments discussed in section 3), use of Continuous LoD (CLoD) to optimize the polygon mesh, and use of Hierarchical LoD (HLoD) for hierarchical aggregation of objects in a scene. Consider using textures instead of rendering individual primitives when appropriate.

![Figure 4: Primitive Count and Vertex/Pixel Process Duration](image)

3.4 Perform experiments

Intel GPA allows you to analyze your game in its normal environment using "what-if" experiments within its System Analyzer to diagnose at a high level where your game's performance bottlenecks are concentrated. Use the experiments for specific types of rendering to help you pinpoint bottlenecks and see what is required to optimize your game without modifying the code for every portion of the graphics pipeline. After you have determined what changes should be made, modify your code and verify that these changes have resulted in the performance gains and level of visual quality that you desire.

3.4.1 Experiment with vertex processing vs. pixel processing

After selecting a batch of draw calls for optimization, compare the values for Vertex Shader Duration vs. Pixel Shader
Duration in the Details Tab (Figure 4) to see if your game has a bottleneck in vertex processing or pixel processing. Frame Analyzer’s Erg Bar Chart, Details Tab and Shader Tab show geometry shader (GS) duration per draw call, within the GPU Breakdown chart option, and across all draw calls, respectively, that can help you with this analysis.

Vertex processing bottlenecks can be addressed by reducing the mesh LoD complexity for object meshes, using vertex shaders (VS) with simpler transform and lighting (T&L) algorithms, and applying an occlusion culling query to eliminate hidden draws. For pixel shader (PS) bottlenecks, try the numerous pixel-related experiments discussed in section 3.4.

**Figure 5:** Intel GPA Frame Analyzer Experiments Window with sample display

### 3.4.2 Experiment with pixel process texture analysis

From the Experiments tab in Frame Analyzer, select the 2x2 Textures experiment to evaluate the impact of reducing texture size. The 2x2 Textures experiment substitutes your game’s original texture access with an Intel GPA-default simple texture that resides entirely in texture cache, eliminating the bandwidth and latency for accessing texture from memory. If this experiment reduces GPU time significantly, the total number of textures and/or their overall texture complexity is a likely bottleneck. Continue investigating the list of textures in Texture tab to evaluate the size and frequency of each texture accessed for the scene, and the impact on frame rate and visual quality.
3.4.3 Experiment with Texture Clamp to MIP

The Clamp to MIP experiment evaluates the impact of reducing texture detail on frame rate and the associated visual quality (Figure 6). Select a frequently accessed, large texture from the list of textures used in a scene, increase the MIP level to reduce the texture size, and observe the resulting GPU time and the variation of Render Target Viewer in Normal mode. If GPU time decreases significantly without a noticeable difference in visual quality, consider reducing the texture resolution for the scene.

![Figure 6: Experiment with Changing Texture Clamp to MIP](image)

**Figure 6: Experiment with Changing Texture Clamp to MIP**

3.4.4 Experiment with the alpha test

In most cases, when the Clamp to MIP and 2x2 Texture experiments dramatically improve performance, the bottleneck is in the texture size. There are exceptions to this rule. In one title we evaluated, the 2x2 Textures experiment reduced GPU time significantly, pointing to a bottleneck in the texture size. However, raising the MIP level as well did not decrease GPU time. The two experiments not only reduced the texture size, but also condensed the range of texel color (including the alpha value), which changed the alpha test load for the game.

We then selected the State tab’s experiment that disables the alpha test (Figure 7), resulting in a significant improvement to the frame rate. To verify the bottleneck’s root cause is the alpha test rather than texture size, try disabling the alpha test or adjusting the alpha reference value. The alpha test is important when rendering concave objects with transparent or translucent effects, such as leaf textures.
3.4.5 Experiment with the filtering algorithm

To evaluate the filtering algorithm of texture sampler as a bottleneck, try changing the filtering algorithm to a simpler one. For example, Anisotropic Filtering (AF) requires more memory bandwidth and is computationally intensive, especially at a high anisotropy level. Remember that AF is a method for improving the image quality of textures on surfaces that are at oblique viewing angles with respect to the camera where the projection of the texture appears to be non-orthogonal. While it eliminates aliasing effects, it also reduces blur at extreme viewing angles (unlike bilinear and trilinear filtering). Review the textures you are using to identify ones that do not benefit from AF, such as low-frequency lightmap textures.

3.4.6 Analyzing Render Target usage

If you use off-screen render targets, it’s important to understand how they affect performance. High resolution render targets require more memory, increasing pixel workloads and fill rate. Fetching textures from high resolution render-targets often is a bottleneck due extensive texture cache misses. Floating point render targets, usually used in post-processing pipelines such as HDR effects, are slower than other formats. While analyzing performance, examine the list of active render targets, specifically looking at their size and format (Figure 7). It is always a good practice to
use the minimum required size and format.

3.4.7 Analyzing the API Log

Frame Analyzer gives you the ability to examine the list of Direct 3D (D3D) API calls associated with every draw call in your frame, including vertex/index streams setup, state/sampler state changes, and setups for constants and pixel, vertex, and geometry shaders (Figure 8). This log allows you to analyze API usage to reduce driver/CPU bottlenecks. For example, group primitives that have similar rendering states and shaders with sequential draw calls, rather than continually changing parameters and incurring the overhead of changing the graphics state. The API log also helps you catch duplicate calls in your code. Finally, rather than examining all calls for all erg(s), use this feature to examine those ergs that require the most processing time.

![Figure 8: Analyzing the API Log](image)

3.4.8 Analyzing Shader code

If you identified that a particular draw call is pixel or vertex shader limited, use the Shaders tab (Figure 9) to review the shader code. You can look at DX Shader Assembler code and HLSL (High Level Shader Language) listing if you compile the shaders in runtime. In particular, look at the number of instructions and the number of shader constants.
3.4.9 Experiment with Simple Pixel Shader

The Simple Pixel Shader experiment within Frame Analyzer (Figure 5) substitutes the application’s original pixel shader with a very simple pixel shader, rendering the pixel with a default color which eliminates texture access and pixel shader calculation costs. Whether you use a programmable or fixed rendering pipeline in your game, this experiment will automatically use the simple shader, thereby allowing you to determine what portion of your rendering time is spent within the shaders for the selected erg(s).

If the simple pixel shader reduces the GPU time significantly, investigate the complexity of the shader using Frame Analyzer’s Shaders Tab to display the source codes or assembler codes of the effect file (.fx) used by selected draw calls. Identify the expensive shaders, specifically those shaders with algorithms that have large instruction counts and large register counts. Compare the DX state values (Figure 7) defined in the shader functions with the current DX states in Frame Analyzer, associating the draw calls with the shader functions in used in the frame. There are many techniques for simplifying shader complexity, such as reducing rendering depth, utilizing Early-Z Rejection, using lower precision or moving per-fragment work to the vertex shader.

3.4.10 Experiment with pixel overdraws

Enable Overdraw mode (Figure 5) in the Render Target Viewer to observe the filling history of any screen pixel to see whether excess draw calls have rendered to that specific pixel. To address overdraws, Disable Erg Experiment (Figure 5) to evaluate the benefits of reducing unnecessary draw calls, or enabling Early-Z rejection. You may combine multiple override modes and state modifications for a deeper analysis of your game. You can also examine pixel history for the particular erg(s) rendered to a pixel, and whether the rendering was optimized (for example, if Z-rejection occurred) (Figure 10).
3.4.11 Understanding the overdraws in your frame

If you see a large number of overdraws in the frame, or notice many ergs rendering to that pixel without trivial rejection, you might want to examine in details the area on the screen which is covered by geometry rendered by these calls. You can see this information in the Selected Ergs area (Figure 5). You can also hide other calls to find situations where most of the pixel area for the erg is hidden and overridden by subsequent draw calls. If these calls occurred later than the selected erg(s), your game is not taking advantage of trivial Z-rejection, so consider changing the rendering order in your algorithm to minimize the expense of fully rendering each pixel multiple times with different draw calls.

3.5 Understanding Intel® HD Graphics hardware metrics

Beginning with version 3.0, Intel GPA supports additional hardware metrics for the latest Intel Integrated Graphics chipsets, beginning with chipsets that support Intel HD Graphics. The new metrics are displayed in Intel GPA System Analyzer’s Metrics Tree under the GPU category (Figure 11). You can view a real-time graph from any of these metrics by dragging the metric name to the right side of System Analyzer window. For each metric, an aggregate value for the whole frame is displayed.
The new metrics are also available in Intel GPA Frame Analyzer in the Details tab for every Erg (Figure 12). When using Intel GPA with older Intel graphics chipsets or with non-Intel GPUs, only three metrics are available on the Details tab — GPU Duration, Vertex Shader Duration, and Pixel Shader Duration. When running on Intel HD Graphics chipsets, 24 new hardware metrics are displayed. A detailed description of the new metrics and the Intel Integrated Graphics block diagram can be found in Appendix A of the Intel GPA documentation.
The new metrics are also available in Intel GPA Frame Analyzer in the Details tab for every Erg (Figure 12). When you make changes to the frame by modifying the rendering or the states, editing shader and texture settings, applying experiments from the Experiments tab, etc., System Analyzer updates the metrics to reveal the impact of your changes, displaying the old and new values associated with your changes. The metrics for the changes are colorized to make it easier to see the effect of changes (Figure 13).
The most valuable metrics in tuning your game's performance are related to the GPU Backend, the array of the Execution Units which process different type of threads: pixel, vertex, geometry shader threads, clipping or media threads. **GPU Backend Active**, **GPU Backend Busy**, **GPU Backend Stalled** metrics show the percentage of the execution time the Execution Units spend on processing threads.

If the **GPU Backend Stalled** percentage is high, you will want to look at the metrics for **GPU Backend Stalled on Samples**, **GPU Backend Stalled on Mathbox**, **GPU Backend Stalled on Data Port** to pinpoint the location of the stalls.

The highest percentage of **Sampler** stalls reveals that shaders are overloaded with texture fetch instructions or there is a significant number of texture cache misses. To reduce the stalls, try minimizing the number of texture fetch instructions in the shaders, reducing texture size and optimize texture fetch patterns to improve texture cache efficiency. Take a look at the **Sampler Throughput** metric to see the number of bytes read from memory for texture requests, which indicates the level of texture cache misses. The **Texel Sampled** metric shows the number of texels sampled from Texture Units. By evaluating these two metrics, you can precisely calculate the percentage of texture cache misses. If you detect texture cache misses, try changing your texture fetch kernels to use cache more efficiently.

**Mathbox** is the Processing Unit dedicated to complex mathematical functions, like sin/cos, exponent and square root. If a significant percentage of stalls is related to Mathbox, your game is likely using shaders that contain a high number of complex math instructions. Try simplifying the math used by your game’s shaders wherever possible, and mix texture and math instructions to reduce latency. Using the texture look-up-table (LUT) is a less productive method for minimizing Mathbox stalls since latency for texture fetching is usually higher than latency for Mathbox operations.
**Data Port** is the Functional Unit that provides read-write access to memory. Data Port stalls are typically related to using shaders that depend on a high number of shader constants or memory reads, for example, in the DX10 Compute shader.

**Vertex Count, Primitive Count, and Vertex Shader Invocation Count** metrics indicate how well your geometry is optimized for the post-transform cache. Vertex Count is the total number of vertices that entered the pipeline, Primitive Count is the number of rendered primitives, and Vertex Shader Invocation Count is the number of vertex shader kernel executions. For example, if you render quad from 4 vertices, this will result in 2 triangles (Primitive Count = 2), with 6 total vertices to process (Vertex Count 6), and the vertex shader will be called 4 times (Vertex Shader Invocation = 4).

**Clipper Invocation Count, Post-Clip Primitive Count, Non-Culled Polygons** metrics are related to geometry culling and clipping. Clipper Invocation Count is equal to the number of primitives required to be clipped. For example, if you have 100 triangles to render with Clipping enabled, the Clipper Invocation Count will be 100. If clipping is disabled, the metric will be 0. Post-Clip Primitive Count shows the number of primitives which were not clipped. Non-Culled Polygons equals the number the polygons which were not back-face (front-face in case of front-face culling) culled.

**Post-GS Primitive Count** metric shows the number of primitives created in the geometry shader stage, which is useful if the geometry shader your game uses creates a variable number of primitives.

**Pixel Shader Invocation Count, Pixel Shader Threads and Pixel Rendered** metrics show the effectiveness of Early-Z culling for the selected Erg. The pixel shader thread is run on the group of 8 or 16 pixels, and a lower number of pixel threads results in a better frame-rate. If the number of rendered pixels is significantly lower than the number of invocations, many pixels were culled by Z-test, so Early-Z culling was not effective for this call. If you notice that situation, try changing the rendering order to improve Early-Z pixel rejection.

**Number of Render Target SubSpans Writes** is the number of 4-pixel quads written to the Render target. A higher number indicates a higher fill-rate, which reduces rendering performance.

If your game is running on Intel HD Graphics hardware, use Intel GPA to inspect the average number of pixels per frame. If the number of pixels looks high, use Frame Analyzer to determine which draw calls are causing the high number of pixels to be rendered, then optimize where possible.

Use the overdraw visualization mode in the Render Target View to find hot spots, then use pixel history in combination with overdraw analysis to inspect all draw calls responsible for hot spots. Then select key draw calls and inspect the metrics for pixels rendered to identify the draw calls to optimize.

### 3.6 Analyze the draw order of scene objects

The draw order of scene objects frequently yields opportunities for performance improvement. Grouping draw calls by similar object status and/or other graphics resources (such as textures) can reduce the overhead in the graphics rendering pipeline; rendering from front to back can also significantly reduce pixel overdraw and improve rendering times. If no source code is available for the game, use Frame Analyzer to analyze the draw order of scene objects by selecting the draw calls successively, then monitoring the change in the rendering target. When analyzing overall draw order, consider the following issues:

- Does changing your game’s draw order benefit performance? Grouping draw calls that use the same rendering states, shaders, or expensive textures to execute in sequence will reduce or avoid the relatively expensive overhead of changing the graphics state.
- Does your game utilize large draw call batches where it’s difficult to see the GPU time for each draw call in the batch? If so, opportunities for optimization cannot be identified by the methods discussed in section 3.1; the potential for optimization can only be found by analyzing draw call order.
• Does your game match the appropriate optimization techniques for optimal benefit to the performance of your game? For example, for the Early-Z test the graphics card maintains a record of the depth of every pixel written to the screen. If a new object is further away than one already rendered for that pixel, the rendering time required for that new pixel is eliminated, which is especially important for complex pixel shaders. The Early-Z rejection technique is suitable to objects with many pixels on the screen, as well as for those objects using complicated pixel shaders; however, this technique requires an additional rendering pass for building the scene depth. The draw order found with Frame Analyzer may indicate whether this technique is used. If the game scene is not appropriate for Early-Z rejection, the overhead of this technique may actually harm game performance.
• Are there redundant draw calls? Eliminate drawing hidden objects and re-rendering objects within a scene, and review the scene-culling algorithm for possible tuning.
• Remember that if you re-order object draws within the scene, you will also need to re-evaluate the scene for pixel overdraws, then examine any new opportunities for eliminating unnecessary draw calls or enabling Early-Z rejection.

3.7 Monitor the changes in rendering targets

After each set of optimizations or experiments, monitor the frame rate, frame time and visual quality of the rendering target. Comparing changes in the rendering target with the cost of related draw calls can provide helpful information on rendering quality and efficiency. Frame Analyzer provides the following functions for a comprehensive analysis of game scene rendering:
• Select the Normal option to observe the visible changes in the display for the selected draw calls.
• Select the Highlight option to see the scope of pixel coverage for the selected draw calls; remove any redundant or hidden draw calls.
• Select the Wireframe or Highlighted wireframe option to observe the complexity of object meshes and vertices for the selected draw calls (which is relevant to the vertex processing load).
• Select the Popout option to highlight invisible objects, and verify they are occluded by other objects before removing the draws for them.
• Enable Scrub mode to view individual screen changes for every selected draw call.

4. Conclusion

Optimizing game performance for integrated graphics can improve the popularity and sales of your game or graphics application. This article described practical performance analysis methods using Intel® Graphics Performance Analyzers to optimize games running on Intel® HD Graphics.

References

Biographies

Sheng Guo is an application engineer in Intel Developer Relations Division, focusing on enabling online game ISVs with Intel advanced technologies. He has a Master’s degree in Computer Science from Nanjing University and has solid skills and experience in performance optimization for real-time 3D graphics applications.

Philipp Gerasimov is a technical consulting engineer in Intel’s Advanced Visual Computing group within VCSD/SSG. Philipp works with different development groups at Intel and ISVs on Intel GPA support and future graphics architecture development. He also presents at numerous game development and computer graphics conferences. You can see his work on game performance optimizations at many modern computer games, like Crysis®, Painkiller 2®, Call of Juarez®, and Pacific Fighters®.

Bonnie Aona is a software engineer in the Intel Compilers and Languages Group within Software and Services Group (SSG) focusing on optimization of complex applications to achieve high performance and optimal parallelism for the target environment. Her career leverages complex technical analysis with software design for high performance applications for computer graphics, real-time systems, scientific research, automated manufacturing, e-Commerce, aerospace and healthcare. She holds a Masters degree in Electrical and Computer Engineering from University of California at Davis.