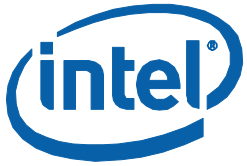


Intel Graphics Media Accelerator Developer's Guide

How to maximize graphics performance on Intel Integrated Graphics

Document Number: 321671-003US

Revision: 2.6.7

Contributors: Jeff Freeman, Chris McVay, Chuck DeSylva, Luis Gimenez, Katen Shah

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
321671-001US	1.0	Re-drafted for Intel® 4-Series Chipsets.	Sept 2008
321671-001US	1.1	Re-drafted for Intel® 4-Series Chipsets.	Sept 2008
321671-002US	2.6.6	Intel® Graphics Media Accelerator Developer's Guide.	March 2009
321671-003US	2.6.7	Intel® Graphics Media Accelerator Developer's Guide.	April 2009

Contents

List of Figures

List of Tables

§

1 About this Document

This document provides development hints and tips to ensure that your customers will have a great experience playing your games and running other interactive 3D graphics applications. This document also describes the Intel® Graphics Media Accelerator used in the Intel® 4 Series Chipsets (the Intel® 4500, X4500, and X4500HD GMAs) with a focus on performance analysis on Microsoft DirectX* and includes a section detailing performance analysis with the Intel® Graphics Performance Analyzer (Intel® GPA). These chipsets are used in desktop G41, G43, and G45 and mobile GM45 systems.

In general, the core of the graphics media accelerators is broken into generations; these generations are known as Intel Graphics Media Accelerator (GMA) Series 3 and 4. Each year, more capabilities and better performance are provided by new integrated graphics cores. Intel Integrated Graphics is currently the number one graphics solution chosen by new PC purchasers. Therefore, it makes sense to write your 3D applications to this broad market and optimize the experience for the greatest number of people. By following the tips and tricks in this document, you have the opportunity for your application to shine with the graphics volume market leader.

1.1 Intended Audience

This document is targeted at experienced graphics developers who are familiar with OpenGL*/Microsoft DirectX*, C/C++, multithread and shader programming, Microsoft Windows* operating systems, and 3D graphics.

1.2 Conventions, Symbols, and Terms

The following conventions are used in this document.

Table 1 Conventions and Symbols Used in this Document

Source code:

```
for(int i=0;i<10; ++i ){
        cout << i << endl;
```

The following terms are used in this document.

Table 2 Terms Used in this Document

1. Intel Integrated Graphics Hardware (IIG).

a. GPU – Graphics Processing Unit
b. GMCH – Graphics and Memory Controller Hub –parent component architecture and chipset housing the Intel integrated graphics hardware (GPU)
    1. GMA – Graphics and Media Accelerator–component name describing the GPU chipset component in the GMCHIntel® 3 Series Chipsets includes the desktop products Intel® GMA X3000 Express Chipset (Intel® G965 Express Chipset) Intel® X3500 Express Chipset (Intel® G35 Express Chipset) and mobile products: Intel® GMA X3100 Express Chipset (Intel® GM965 and Intel® GL960 Express Chipset)
    2. Intel® 4 Series Express Chipsets includes desktop chipsets: Intel® GMA X4500 Express Chipset and Intel® GMA X4500HD Express Chipset (Intel® G41 Express Chipset, Intel® G43 Express Chipset, and Express Chipset G45 Express Chipset), and mobile chipset Intel® 4500MHD (GM45) Express Chipsets.
c. UMA – Unified Memory Architecture - an architecture where the graphics subsystem does not have exclusive dedicated memory and uses the host system's memory (SDRAM)
d. DVMT – Dynamic Video Memory Technology – a memory allocation scheme in UMA systems which allocates an exclusive, dynamically resizable chunk of main memory to the graphics (driver)
e. VF – Vertex Fetch
f. VS – Vertex Shader
g. PS – Pixel Shader
h. GS – Geometry Shader
i. EU – Execution Unit, a vector machine component
j. CS – Command Stream manager component controlling 3D and media
k. I$ - Instruction cache
l. SO – Stream Output

2. SWGP – Software geometry processing, a superset of CPU-based processing that includes CPU vertex processing. SWGP is not equivalent to the Microsoft DirectX* reference device.
3. SWVP – Software vertex processing
4. HWVP – Hardware vertex processing

1.3 Related Information

Intel® 3 Series Express Chipsets including the Intel® 3000 GMA and Intel® X3000 GMA Developer's Guide: http://software.intel.com/en-us/articles/intel-gma-3000-and-x3000-developers-guide/.

2 About Intel Integrated Graphics

2.1 Graphics and Media Accelerator Roadmap

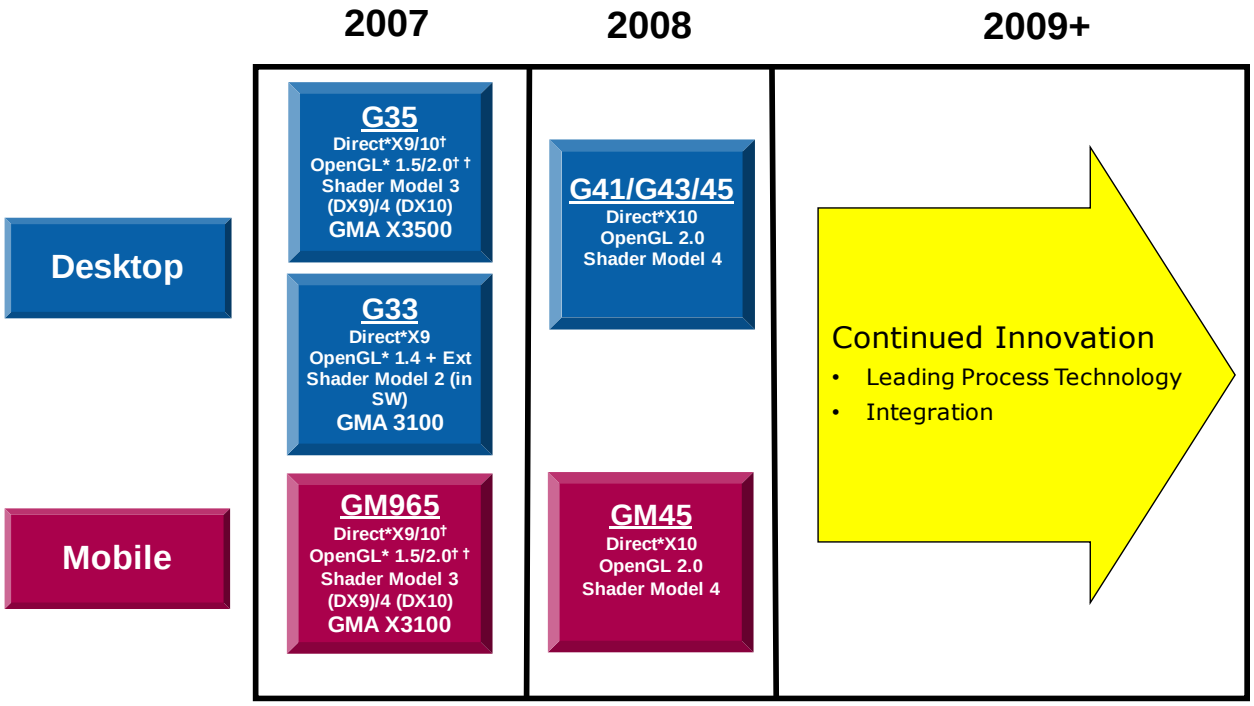


Figure 1 Integrated Graphics Roadmap 2008-2009

2.2 Intel® 4 Series Express Chipsets Architecture

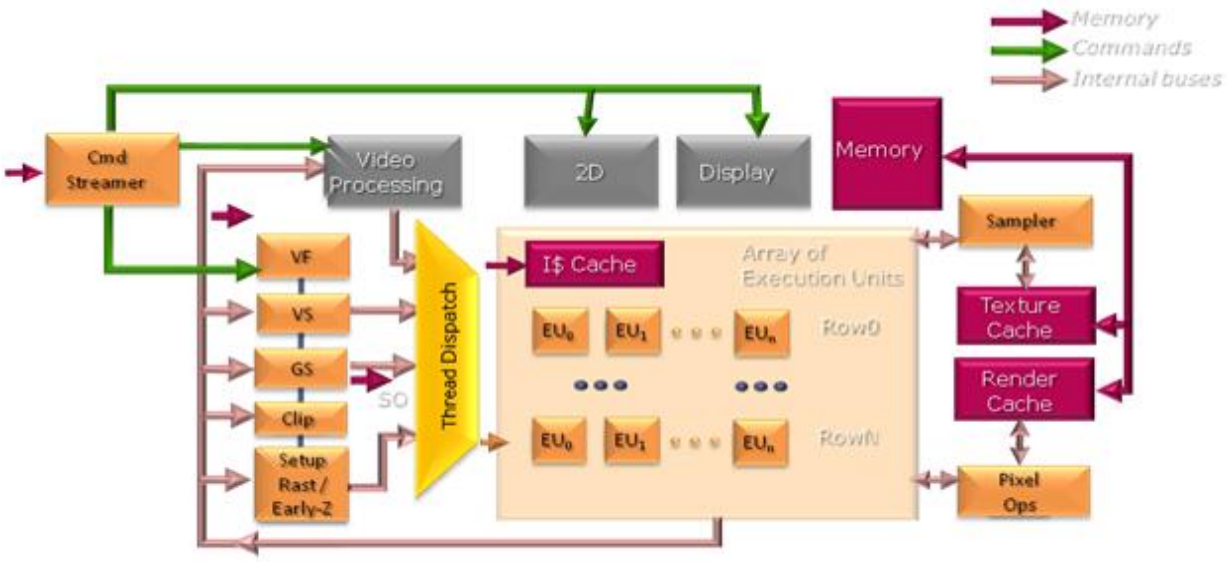


Figure 2 Intel® 4 Series Express Chipsets Architecture Diagram

The latest Intel integrated graphics products have been architected to support Microsoft DirectX* 10, including a consistent feature set with no need for checking individual capability bits. Intel® 3 Series Express Chipsets and Intel® 4 Series Express Chipsets also take advantage of a generalized unified shader model including support for Shader Model 4.0. The graphics core executes vertex, geometry, and pixel shaders on the programmable array of Execution Units (EUs). The EUs have programmable SIMD (Single Instruction, Multiple Data) widths and are capable of executing multiple threads to optimize throughput.

2.3 Intel® 3 Series Chipsets and Intel® 4 Series Express Chipsets Features

Intel Graphics Core	Intel® GMA X3000 Express Chipset	Intel® GMA X3100 Express Chipset	Intel® GMA X3500 Express Chipset	Intel® GMA X4500/ X4500HD Express Chipset	Intel® GMA 4500MHD Express Chipset
Intel Chipset	**G965**	**GM965, GL960**	**G35**	**G41, G43 and G45**	**GM45**
Segment	**Desktop**	**Mobile**	**Desktop**		**Mobile**
Architecture	**Intel® Series 3 Chipsets**			**Intel® 4 Series Express Chipsets**	
Video Memory	Up to 384 MB	Up to 384 MB	Up to 384 MB	> 512MB	> 512MB
DirectX* Support	9.0Ex	10	10	10	10
OpenGL Support	2.0	2.0	2.0	2.0	2.0
Shader Model Support	3.0 (HWVP)	4.0	4.0	4.0	4.0

Table 3 Intel® GMA Series 3 and 4 Feature Specifications

3 Quick Tips: Graphics Performance Tuning

3.1 Primitive Processing

Support for both Hardware Vertex Processing (HWVP) and Software Geometry Processing (SWGP) is included. SWGP is a superset of software-based processing that includes software vertex processing (SWVP). HWVP peak vertex throughput has been significantly improved in Intel GMA Series 4 – twice as fast as the previous generation, and by default, HWVP is enabled. However, CPU vertex processing may offer even greater performance enhancements on the latest Intel multi-core processors. The driver will always export full HWVP support. Specifically on Microsoft DirectX*9, we recommend using D3DCREATE_PUREDEVICE during device creation. This allows SWGP to be enabled based on performance that is determined by the specific configuration, workload, and Intel integrated graphics capability. SWGP has optimizations beyond the Microsoft DirectX* runtime Processor Specific Graphics Pipeline (PSGP) which takes advantage of the evolving set of CPU instructions and capabilities. For example, the VS/Clip stages behave as a pass-through and reallocate compute resources back for pixel processing and can cause an overall performance gain especially on pixel shading intensive applications. In some cases we have witnessed gains up to 30% improvement by using SWGP, although this is dependent on the particular configuration and workload.

Developers can test gains from SWGP by using the Intel® Graphics Performance Analyzers suite to force workloads that would otherwise be performed on Intel integrated graphics to the CPU. See the section "Enhancing Graphics Performance on Intel GMA Series 4 with Intel® Graphics Performance Analyzer" for more information on this tool.

3.1.1 Tips On Vertex/Primitive Processing

1. Use IDirect3DDevice9::DrawIndexedPrimitive (DX 9) or ID3D10Device::DrawIndexed (DX 10) to maximize reuse of the vertex cache.
   a. The vertex cache size will increase over time and can be discovered using D3DQUERYTYPE_VCACHE.
2. Ensure adequate batching of primitives to amortize runtime and driver overhead.
   a. Maximize batch sizes: 200 to 1000 is recommended and within this range, bigger is better.
   b. Minimize render state changes between batches to reduce the number of pipeline flushes.

   c. Use instancing to enable better vertex throughput especially for small batch sizes. This also minimizes state changes and Draw calls.
3. Use static vertex buffers as much as possible.
4. Use visibility tests to reject objects that fall outside the view frustum to reduce the impact of objects that are not visible.
   a. Set D3DRS_CLIPPING to FALSE for objects that do not need clipping.
5. Take advantage of Early-Z rejection.
   a. Render with a Z-only pass (meaning no color buffer writes or pixel shader execution) followed by a normal render pass in roughly front to back order. This uses the higher performance of Early-Z to reject occluded fragments which reduces compute times and raster operations.
   b. Balance a Z-only pass against the inherent cost of an additional pass – do not do this at the cost of including more render state changes or worse batching due to sorting.
   c. Avoid using modified Z values (depth) in the pixel shader. Modifying the depth value will skip the Early-Z hardware optimization algorithm since it changes the visibility of the fragment.
6. Use the Occlusion Query feature of Microsoft DirectX* to reduce overdraws for complex scenes. Render the bounding box or a simplified model – if it returns zero, then the object does not need to be rendered at all.
   a. Consider drawing over-lays such as heads up displays (HUD) first if they are opaque and then writing them to the Z buffer which can effectively reduce the screen rendering area leading to a considerable performance improvement.

3.2 Shader Capabilities

While both generations of integrated graphics support the Microsoft DirectX*10 Unified Shader Model 4.0, Intel® GMA Series 4 significantly improves compute capability over Intel® 3 Series Chipsets. There is a 25% increase in raw computations as well as significant improvements in performance of transcendental instructions. Intel® 4 Series Express Chipsets also have increased support of latency coverage at higher frequencies and shaders with longer instruction lengths. GMA Series 3 and 4 support the new Geometry Shader and Stream-out functionality associated with D3D10.

	2007		2008
Product	**G35**	**GM965**	**G41/43/45, GM45/47**
Gfx Arch	**Intel GMA Series 3**		**Intel GMA Series 4**
Shader Model Profile	vs_3_0, ps_3_0; vs_4_0, ps_4_0, gs_4_0		vs_4_0, ps_4_0, gs_4_0
Max # of Instructions	SM 3.0 = 512; SM 4.0 = Unlimited		
Max # of Constants	SM 3.0 = 256; SM 4.0 = 4Kx16 (Constant Buffers)		
Max # of Temp Registers	Temporary storage per shader execution instance is 4096 elements that can be used in any combination of registers/arrays, i.e., the total number of r# and x# declared must <= 4096		
Precision	32-bit floating point		
Vertex Texture/Instancing	SM 3.0 / 4.0		
Flow Control	Static and Dynamic		

Table 4 Intel® GMA Series 3 and 4 Shader Specifications

3.2.1 Tips on Shader Capabilities

1. Utilize the highest possible shader model, e.g. SM 4.0 over 3.0 and lower.
2. Use programmable shaders over fixed functions as much as possible.
   a. For example, use shader based fog instead of fixed function fog. Fixed Function fog has been deprecated on SM 3.0 and 4.0.
3. Use flow control wisely.
   a. Dynamic flow control can provide significant benefits by skipping a large number of computations. Ensure that this is used where a large portion of the shader can be skipped.
   b. Use predication over dynamic flow control for shorter branching instruction sequences.
   c. The pixel shader operates on up to 16 pixels in parallel. This means the benefits will depend on the likelihood of the number of pixels taking the same branch.
4. Balance texture samples and shader complexity.

   a. Recommend greater than 4:1 ratio of ALU:Sample for better latency coverage. A larger ratio may be better for floating point textures, higher order filtering, and 3D textures.
   b. Although large shaders can be supported via cache structure, it is important to be aware of limited number registers that are available per EU and running out of these can drop the efficiency of the execution units.
5. Space your texture sampling calls away from where it is used in pixel shaders when possible and practical to maximize EU utilization.
6. Optimize your shader performance by adequate use of your integrated graphics:
   a. Reduce the use of macro/transcendental functions where possible. Instructions like LOG, LIT, ARL, POW, EXP are more expensive.
   b. Use full precision for non-transcendental instructions.
   c. Mask alpha if you are not using it.
7. Minimize the usage of geometry shaders.
8. In general, minimize use of StreamOut and DrawAuto() for optimal performance.

3.3 Texture Sample and Pixel Operations

	2007		2008
Product	**G35**	**GM965**	**G41/43/45, GM45/47**
Gfx Arch	**Intel GMA Series 3**		**Intel GMA Series 4**
Format Support	16/32-bit fixed point 16/32-bit floating point operations		
Max # of Samples	Up to 16		
Vertex Textures	Yes		
Max 2D/3D/Cube Textures	8Kx8K/2Kx2K/8K		
Filtering Type Support	BLF, TLF and Dynamic AF w. up to 16 sub-samples		
Texture Compression	DX9: DXT1/3/5; DX10: BCx		
Non Power of 2 Textures	Yes		
Render to Texture	Yes, Incl. Off-screen Surface Support		
Multi-Sample Render	Single Sample Only		
Multi-Target Render	Max = 8		
Alpha-Blend FP formats	Both FP16/FP32 formats are supported		

Table 5 Intel® GMA Series 3 and 4 Texture Sampling and Pixel Specifications

	2007		2008
Product	**G35**	**GM965**	**G41/43/45, GM45/47**
Gfx Arch	**Intel GMA Series 3**		**Intel GMA Series 4**
32-bit Texels			
Point/Bilinear	1X		1X
Trilinear	0.5X		1X
Anisotropic	0.5X/n		0.5X/n
64-bit Texels			
Point/Bilinear	0.5X		1X
Trilinear	0.25X		0.5X
Anisotropic	0.25X/n		0.25X/n
128-bit Texels			
Point	0.25X		0.25X

Table 6 Intel® GMA Series 3 and 4 Sampler Filtering Specifications

All sampler filtering types are supported, including dynamic anisotropic filtering. Intel GMA Series 4 doubles 32 bits-per-pixel fixed trilinear filtering and 16 bits-per-pixel float bilinear performance as shown in Table 6.

3.3.1 Tips on Texture Sampling / Pixel Operations

1. Use compressed textures and mip-maps in the same format when possible and minimize the use of large textures even though the architecture supports up to 8K×8K. For optimal performance use texture sizes that are 256x256 or less.
2. Minimize the use of Trilinear and Anistropic Filtering especially for floating point textures where the performance of bilinear and trilinear is not equivalent.
   a. Utilize a type of filtering based on the usage in a scene rather than using it everywhere.
3. Avoid using 32-bit floating point textures.

   a. FP32 filtering is optional on Microsoft DirectX* 9 and Microsoft DirectX*10. Be sure to check the caps (DirectX* 9) or call ID3D10Device::CheckFormatSupport (DirectX* 10) for the latest supported feature set based on the installed driver.

4. Keep multiple render targets to <4. Keep the size under 128x128 for optimal performance.
5. Minimize the number of Clear calls.
   a. Clear surfaces, Color and Z/Stencil buffer at the same time when required.
6. Minimize lock/blit of Z and/or stencil buffer to minimize bandwidth impact.
7. Utilize shadow maps instead of stencil shadows as they are fill intensive.
8. Multi-Texture Rendering is better than multi-pass rendering since MTR reduces state changes, driver overhead, and CPU load. In addition, Intel integrated graphics utilizes main system memory for graphics. The intermediate pixels computed in a multi-pass rendering need to be transported back to main memory and then back to the graphics subsystem when needed again, causing a full round-trip over the bus per render target for each pass.

3.4 Microsoft DirectX*10 Optional Features

D3D10 does specify some optional features even though the CAP bit concept from the previous API has been removed. The following features are not required or optional:

1. MSAA support is not supported on Intel® GMA Series 3 and 4.
2. 32-bit FP filtering is not supported on Intel® GMA Series 3 and 4.
3. RGB32 render targets are not supported on Intel® GMA Series 3 and 4.
4. 16bit UNORM blending is supported in Intel® GMA X4XXX and later.
5. D3D10 specifies a large number of resource types and data formats including many of them that are optional. Utilize ID3D10Device::CheckFormatSupport to determine what is supported.

3.5 Managing Constants on Microsoft DirectX*

Constants are external variables passed as parameters to the shaders; their values remain “constant” during each invocation of the shader program. Despite their name, constants are one of the most frequently changing values in a Microsoft DirectX* application. A shader program can initialize a constant variable statically to a value in the shader file or at runtime through the application.

Most of the recommendations described here are not completely new and may have been described elsewhere. However, it is still very much applicable to Intel integrated graphics and the recommendations attempt to detail them in a cohesive manner. In addition to these points it is worth noting that care should be taken when porting from Microsoft DirectX*9 to Microsoft Direct*X10 to maintain performance. For more information on this topic, see the Intel publication “DirectX Constants Optimizations For Intel® Integrated Graphics” 0 available soon on the Intel Software Network.

3.5.1 Tips on Managing Constants on Microsoft DirectX*9

1. The driver optimizes access to the most frequent used constants. Use under 32 constants to achieve the highest performance gain from this feature. Limit the use of dynamic indexed constants (C[ax], C[r]) as these cannot be optimized by the driver, causing high latency in shaders. These constants are normally found in vertex shaders.
2. Higher performance is obtained with local constants over global constants.
3. Immediate constants provide better performance than dynamic indexed constants. In dynamic indexed constants the driver cannot determine a prior the index value and needs to create a full size constant buffer space in memory, instead of using the hardware constant buffer.
4. To take advantage of the optimization, limit the use of global constants and the use of dynamically indexed constants C[ax] as these skip the IIG optimization algorithm within the Intel Driver.

3.5.2 Tips on Managing Constants on Microsoft DirectX*10

1. The driver optimizes access to the most frequent used constants. Use under 32 constants per shader to achieve the highest performance gain from this feature. Limit the use of dynamic indexed constants (C[ax], C[r]) normally found in vertex shaders as these cannot be optimized by the driver, causing high latency in shaders.
2. Avoid creating an uber constant buffer that houses all of the constants, especially if porting from Microsoft DirectX* 9, which can result in a large global buffer. If any constant value is changed, it results in reloading the whole buffer to the GPU, causing a significant performance impact. It is generally preferred to have a larger number of small size constant buffers than a single uber buffer. When possible, share constant buffers between different shaders.
3. For optimal constant buffer management, smaller packed constant buffers grouped by frequency of update and access pattern are ideal for higher performance. As an example: organize Per Frame/ Per Pass/ Per Instance constant buffers first which tend to be smaller in size and have a low update rate followed by Per Draw/Per Material constant buffers which may also be small but have a higher update rate. Finally, define large constant buffers like skinning constants.
4. If there are constants that are unused by most of the shaders, then moving those to the bottom will allow for binding of a smaller buffer to those shaders.
5. Another optimization that could be made is to breakup constant buffers based on features that are optional in games (e.g. shadows, post-processing effects, etc.). Essentially due to performance constraints for integrated platforms, some of these features are either going to be disabled or run with a lower setting – given this it would be beneficial to break up constants into separate buffers and then disable the updates to these constant buffers based on the settings selected by the user.
6. When using indexed constant buffers, it is recommended to keep the buffer size tailored to actual needs. For example, if the shader iterates over five elements

only, declare a 5-element constant buffer for this shader rather than a general purpose 50-element constant buffer shared among shaders. This allows the driver to optimize placement so that it incurs a low latency path.

3.6 Graphics Memory Allocation

Integrated graphics will continue to use the Unified Memory Architecture (UMA) and Dynamic Video Memory Technology (DVMT) as noted in the chart below. As with past integrated graphics solutions, UMA specifies that memory resources can be used for video memory when needed. DVMT is an enhancement of the UMA concept, wherein the optimum amount of memory is allocated for balanced graphics and system performance. DVMT ensures the most efficient use of available memory - regardless of frame buffer or main memory size - for balanced 2D/3D graphics performance and system performance. DVMT dynamically responds to system requirements and application's demands, by allocating the proper amount of display, texturing, and buffer memory after the operation system has booted. For example, a 3D application when launched may require more vertex buffer memory to enhance the complexity of objects or more texture memory to enhance the richness of the 3D environment. The operating system views the Intel graphics driver as an application, which uses a high speed mechanism for the graphics controller to communicate directly with system memory called Direct AGP to request allocation of additional memory for 3D applications, and returns the memory to the operating system when no longer required.

	2007		2008	
Product	**G35**	**GM965**	**G41,G43,G45**	**GM45**
Segment	**Desktop**	**Mobile**	**Desktop**	**Mobile**
Gfx Arch	**Intel GMA Series 3**		**Intel GMA Series 4**	
Memory BW (GBps)	10.7 – 12.8	8.5 – 10.7	12.8 – 23.1	10.7 – 17.1
UMA Capability	2x DDR2-667/800	2x DDR2-533/667	2x DDR3-800/1066/1333	2x DDR3-667/800/1067
Max DVMT (XP) 1 or 2GB System Memory	384MB		> 512MB	
Max DVMT (Vista) 1GB / 2GB System Memory	256MB/384MB		256MB/>512MB	

Table 7 Intel® GMA Series 3 and 4 Memory Specifications

3.6.1 Tips On Resource Management

1. Allocate surfaces in priority order. The render surfaces that will be used most frequently should be allocated first. On Microsoft DirectX* 10, memory is taken care of for you by the OS. On Microsoft DirectX* 9:

a. Use D3DPOOL_DEFAULT for lockable memory (dynamic vertex/index buffers).
b. Use D3DPOOL_MANAGED for non-lockable memory (textures, back buffers, etc).

2. On D3D10 use of the Copy…() methods are preferred over calling the Update…() operations. Partial or sub-resource copies should be used sparingly, i.e., when updating all or most of the LODs of a resource use CopyResource() or multiple CopySubResource().

3.7 Microsoft DirectX* Considerations Prior to Microsoft Directx*10

3.7.1 Creating a Microsoft DirectX*9 Device and Identifying Intel® GMA

The following code shows how to correctly initialize and detect Microsoft DirectX* 9 Software Vertex Processing (SWVP). This sample also shows how to switch to software vertex processing for legacy integrated graphics hardware for the devices that support it, and conversely, hardware vertex processing for the devices that support that.

```
HRESULT hr;
DWORD BehaviorFlags = 0;
IDirect3DDevice9* pDevice = NULL;

UINT nMinRequiredVertexShaderLevel = yourMinimumVSLevel; // i.e.D3DVS_VERSION(3,0)
UINT nMinRequiredPixelShaderLevel = yourMinimumPSLevel;  // i.e.D3DPS_VERSION(2,0)

// Clear any vertex processing flags
BehaviorFlags &= ~(D3DCREATE_HARDWARE_VERTEXPROCESSING |
                   D3DCREATE_MIXED_VERTEXPROCESSING |
                   D3DCREATE_SOFTWARE_VERTEXPROCESSING);

// We'll try to get 'PURE' hardware first
BehaviorFlags |= D3DCREATE_PUREDEVICE;

hr = pD3D->CreateDevice(Adapter,
                    DeviceType,
                    hFocusWindow,
                    BehaviorFlags | D3DCREATE_HARDWARE_VERTEXPROCESSING,
                    pPresentationParameters,
                    &pDevice);
if(D3D_OK == hr)
{
         // NOTE: We're using pDevice->GetDeviceCaps and not pD3D->GetDeviceCaps
         hr = pDevice->GetDeviceCaps(&Caps9);
}

if((D3D_OK != hr) ||
   (Caps9.VertexShaderVersion < nMinRequiredVertexShaderLevel) ||
   (Caps9.PixelShaderVersion < nMinRequiredPixelShaderLevel))
{
         // We didn't get a 'PURE' hardware, so clear the flag.
         BehaviorFlags &= ~D3DCREATE_PUREDEVICE;
```

```
        hr = pD3D->CreateDevice(Adapter,
                                DeviceType,
                                hFocusWindow,
        BehaviorFlags |
                                D3DCREATE MIXED VERTEXPROCESSING,
                                pPresentationParameters,
                                &pDevice);
        if(D3D_OK == hr)
        {
                hr = pDevice->GetDeviceCaps(&Caps9);
        }

        if ((D3D_OK != hr) ||
        (Caps9.VertexShaderVersion < nMinRequiredVertexShaderLevel) ||
        (Caps9.PixelShaderVersion < nMinRequiredPixelShaderLevel))
        {
                hr = pD3D->CreateDevice(Adapter,
                        DeviceType,
                        hFocusWindow,
                        BehaviorFlags |
                        D3DCREATE SOFTWARE VERTEXPROCESSING,
                        pPresentationParameters,
                        &pDevice);
                if(D3D_OK == hr)
                {
                        pDevice->GetDeviceCaps(&Caps9);

                        if(Caps9.PixelShaderVersion <
                                nMinRequiredPixelShaderLevel)
                        {
                                // Minimum specs for this application are
                                // higher than this system can handle
                                // Exit this application gracefully…
                                pDevice->Release;
                                pDevice = NULL;
                                hr = E FAIL;
                        }
        }
}
```

3.7.2 Checking for Available Memory

The operating system will manage memory for an application on Microsoft DirectX* 10. On Microsoft DirectX* 9, a check that is often performed before actually executing the application is the amount of available free graphics or video memory. As a result of the dynamic allocation of graphics memory performed by the Intel Integrated Graphics devices (based on application requests), you need to take care in ensuring that you understand all of the memory that is truly available to the graphics device. Memory checks that only supply the amount of 'local' graphics memory available do not supply an appropriate value for the Intel Integrated Graphics devices. To accurately detect the amount of memory available to the Intel Integrated Graphics devices, check the total video memory availability. All video memory on Intel® GMA Series 3 and 4, even the dynamically allocated DVMT (Dynamic Video Memory Technology) memory, is considered to be "Local Memory". "Non-Local Video Memory" will show as ZERO (0). This should not be used to determine "AGP" or "PCI Express" compatibility.

The code snippet below outlines the function call necessary to most accurately check the memory available for use by the Intel Integrated Graphics controller within Microsoft DirectX*9. Recall that Integrated Graphics utilizes main system memory

with UMA, causing this call to return the total amount of system memory available for use by the graphics device:

```
int AvailableTextureMem = pd3dDevice->GetAvailableTextureMem();
```

3.8 Surviving a GPU Switch

Intel in combination with third party graphics vendors jointly developed a switchable graphics solution that allows end users to switch on-the-fly between two heterogeneous GPUs without a reboot. This functionality incorporates the energy efficiency of Intel integrated graphics with the 3D performance of discrete graphics in a single notebook solution. This technology is applicable to the ~30 million discrete notebooks purchased annually. Currently most applications running on PC platforms with heterogeneous GPUs do not survive when GPUs are switched at run-time, since they do not re-query underlying graphics capability when the active adapter changes.

Keys to handling GPU changes:

- New applications should be aware of multi-GPU configurations and handle the messages D3DERR_DEVICELOST and WM_DISPLAYCHANGE.
- Legacy applications, if possible, should develop and distribute patches for old games to handle the messages D3DERR_DEVICELOST and WM_DISPLAYCHANGE.

3.8.1 Microsoft DirectX*9 Algorithm

Microsoft DirectX* 9 applications should follow the below procedure to query GFX adapter's capabilities (re-create DX object/device) on D3DERR_DEVICELOST:

1. Manually save the current context including state and draw information in the application.
2. Query if the GPU adapter has changed using the Windows API's EnumDisplaySettings or EnumDisplayDevices.
3. If the adapter has changed, then:
   a. Recreate a Microsoft DirectX* device.
   b. Restore the context.
   c. Continue rendering from last scene rendered before the D3DERR_DEVICELOST event occurred.

3.8.2 DirectX 10 Algorithm

There is no concept of D3DERR_DEVICELOST as a return status in Microsoft DirectX* 10. The changes in Microsoft DirectX* 10 applications are:

1. Check for WM_DISPLAYCHANGE windows message in the message handler.

2. Query if the GPU adapter has changed using the Windows API's EnumDisplaySettings or EnumDisplayDevices.
3. If yes, then save off the current context including state and draw information in the application and then:
   a. Recreate the Microsoft DirectX* device.
   b. Restore the context.
   c. Continue rendering from the last scene rendered before the WM_DISPLAYCHANGE message occurred.

4 Performance Analysis on Intel Integrated Graphics

Though the principle behind performance analysis on Intel integrated graphics is similar to other GPU devices, there are significant differences due to the UMA model used in IIG. Diagnosing a performance bottleneck often involves several steps with the potential of revealing other performance issues along the way. This section will break down the graphics stack to reveal key areas to focus on when troubleshooting, diagnosing, and resolving performance bottlenecks with Intel integrated graphics.

4.1 Diagnosing Performance Bottlenecks

At a very high level, the graphics stack includes a rendering system that takes polygons, textures, and commands as input to display the resulting picture on an output device.

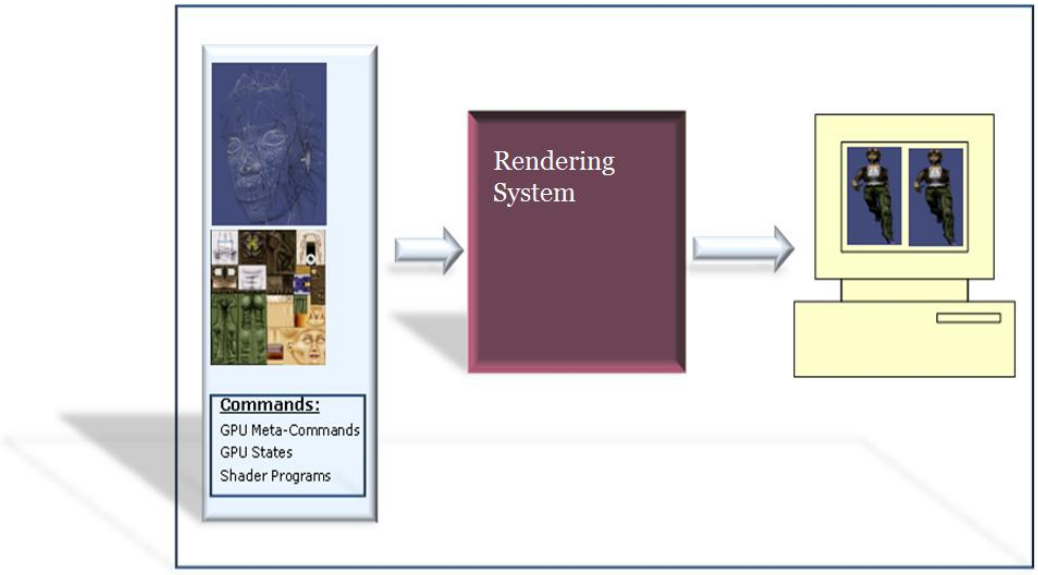


Figure 3 A simplified Graphics Stack

The graphics stack consists of the CPU, main memory, and the bus which delivers the visual payload of data to the Intel integrated graphics chipset. Several scenarios involving these components can affect overall performance. Considering that each of these computational systems resides along a highway where data is flowing, the following could occur:

- If any of these channels are *under-utilized* the system may be under-performing in terms of overall capacity to do more work.
- If any of these channels are *over-utilized*, the system may be under-performing in terms of capacity to keep the data moving fast enough.

For optimal performance, the application should maximize the performance of the graphics subsystem and operate the other channels optimally to keep the graphics subsystem continuously productive with minimal starving or blocking situations.

As noted in the Intel® GMA Architecture Diagram, Intel integrated graphics employs main system memory via DVMT as well as the CPU via the driver to create a closely knit computation facility. Analyzing a performance issue and breaking it down into parts is therefore crucial to isolating the issue to a part and understanding a way to resolve the bottleneck. These concepts help define a performance analysis methodology that can be used to diagnose issues.

4.2 Performance Analysis Methodology

In terms of performance, we will consider a single high-performance graphics application such as a PC game and use this scenario as the foundation of our performance analysis methodology. In order to make the visual experience seem real and engaging, it is important to be balanced yet aggressive in utilizing the resources of the CPU, GPU, bus, and main memory.

Here are some aspects of each of these resources that affect performance. Some of this is simple in concept but more difficult in practice in a performance application such as a game:

The Graphics Stack Domains

- The **CPU** is a domain consisting of the processor itself and the software running on it. Performance sensitive areas include the application and API's it uses, the driver, and how the software prepares data to be passed on to another facility. Notable hardware components in this domain include the CPU speed, cache size, cache coherency, and utilization of hardware threads.
- The **GPU** domain includes the graphics subsystem, shader programs, the part of the texture processing that occurs on that subsystem, and state changes.
- The **Main Memory** domain constitutes the physical RAM and memory allocated to the game as well as secondary storage used by virtual memory.
- In a close relationship with the memory domain, the **Bus** constitutes the connection between main memory and the graphics subsystem and in terms of this breakdown, is focused on delivery of the graphics payload to the GPU.

The goal of this breakdown is to localize the current performance bottleneck to one of these domains. Performance issues in a PC game will likely fall across several computational facilities, but isolating performance to a single one and focusing efforts there will help to choose a strategy and toolset to employ. The next section covers Intel® GPA, a utility set including the Intel® GPA's System Analyzer, a tool useful in isolating issues to one of these domains, and the Intel® GPA Frame Analyzer, an in-

depth frame analysis utility useful in exploring issues specific to the integrated graphics part for Intel® GMA Series 4.

4.2.1 Game Performance Analysis – "Playability"

Analyzing game performance issues is not an exact science. A holistic approach would consider the general performance of a game, such as if the game consistently runs at a frame rate deemed outside of an acceptable range with a specific graphics feature or feature set enabled in the game. More recent analysis efforts have focused on "slow frames" or specific areas in a game that render below acceptable frame rate ranges. These slow frames are good markers as a starting point for identifying the bottleneck – the sequence of events leading up to the rendering of that scene. When a graphics performance issue is suspected, Intel® GPA, a new tool released by Intel, can help determine which computational domains are affected and where to focus a more thorough breakdown of a single or set of performance issues in a game.

Introducing Intel® GPA

Intel® GPA is a tool set for isolating graphics performance issues within applications running on Intel integrated graphics. It provides an all-in-one tool, to explore CPU and GPU loads. This section will briefly cover some aspects of Intel® GPA in terms of this performance analysis methodology. While this isn't a comprehensive list of possible scenarios it does provide a typical set of checks performed when optimizing games for Intel integrated graphics. Many of these scenarios can be exposed with Intel® GPA as a companion to a C/C++ debugger.

Intel® GPA is a suite of graphics performance optimization tools composed of the Intel® GPA System Analyzer and the Intel® GPA Frame Analyzer. These tools allow optimization across all currently supported Intel GPUs running Microsoft DirectX* workloads.

The Intel® GPA System Analyzer is a high-level tool intended to enable an engineer to understand game performance across the CPU and GPU. This is an interactive real time tool that displays various metrics and allows DirectX* level overrides. Key features include:

1. Drag and drop metric display.
2. DirectX* and graphics driver overrides including a simple pixel shader, null hardware, and null driver.
3. Frame capture and transition to the Intel GPA Frame Analyzer.

The Intel® GPA Frame Analyzer is an interactive deep single frame GPU analysis tool that enables an engineer to analyze performance at the frame, region, and draw call level. Frame captures are file based and can be shared between engineers and different GPUs for analysis. The major features of the frame analyzer are:

1. Draw Call Bar Chart Visualization: a visualization of any selected metric for each and every draw call in the frame. The default metric is GPU duration.
2. Scene Overview: a sort-able tree view of performance metrics at the frame level, region level (default region = render target change), and draw call level. All metrics are available in this view.

3. Render Target Viewer: a thumbnail and full-sized view of all render targets associated with the current draw call selection set, including highlighting options for selected draw calls.
4. Experiments Tab: a set of selectable experiments including a simple pixel shader, 2x2 textures, and 1x1 scissor rect that modifies the current draw call selection set. Performance impact of these changes can be viewed in the bar chart and scene overview.
5. Texture Tab: a thumbnail and full-sized view of all textures associated with the current draw call selection.
6. Shader Tab: a shader viewer and on-the-fly editor. Includes the ability to modify a shader via an in-line-edit, cut and paste, and file change. Modifies all shaders within the current draw call selection set. Performance impact of these changes can be viewed in the bar chart and scene overview.
7. State Tab: a view that displays and allows modification of all DirectX* state for the current draw call selection set.
8. API Log: a chronologic view of all DirectX* APIs organized by draw call.
9. System Info: a listing of important system information from the system that rendered the captured frame. (driver version, OS version, DirectX* version, GPU version info, etc).

Figure 4 Intel® GPA System Analyzer in Action

4.2.2 Localizing Bottlenecks to a Graphics Stack Domain

CPU Load

It is typically easiest to begin with a focus on the CPU domain. It is usual to see a CPU load across all cores in the range of 20-40% in a typical game. This load can easily go up to 90%+ and not be a bottleneck. Intel® VTune™ Performance Analyzer can provide a detailed hot-spot analysis to determine if the driver, API, or game itself is the problem area. Intel® Thread Profiler can further reduce the search area for threading and concurrency issues.

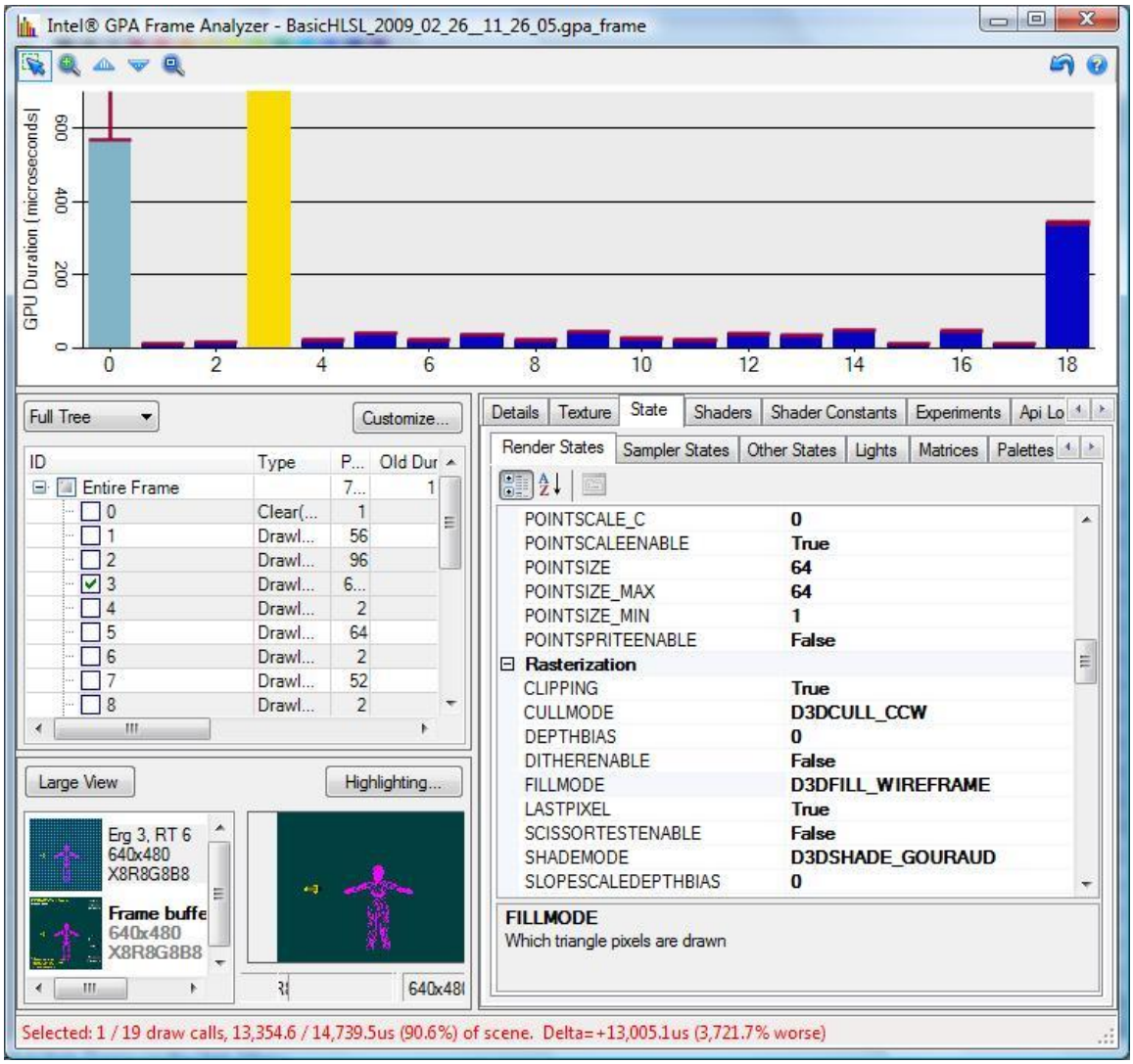
GPU Load

If Intel® GPA's System Analyzer shows a relatively low overall CPU load and the frame rate is below a desired range, it is reasonable to assume a GPU bounded condition.

Capturing a frame at this point and running Intel® GPA's Frame Analyzer on that frame will help explore the issue in greater detail.

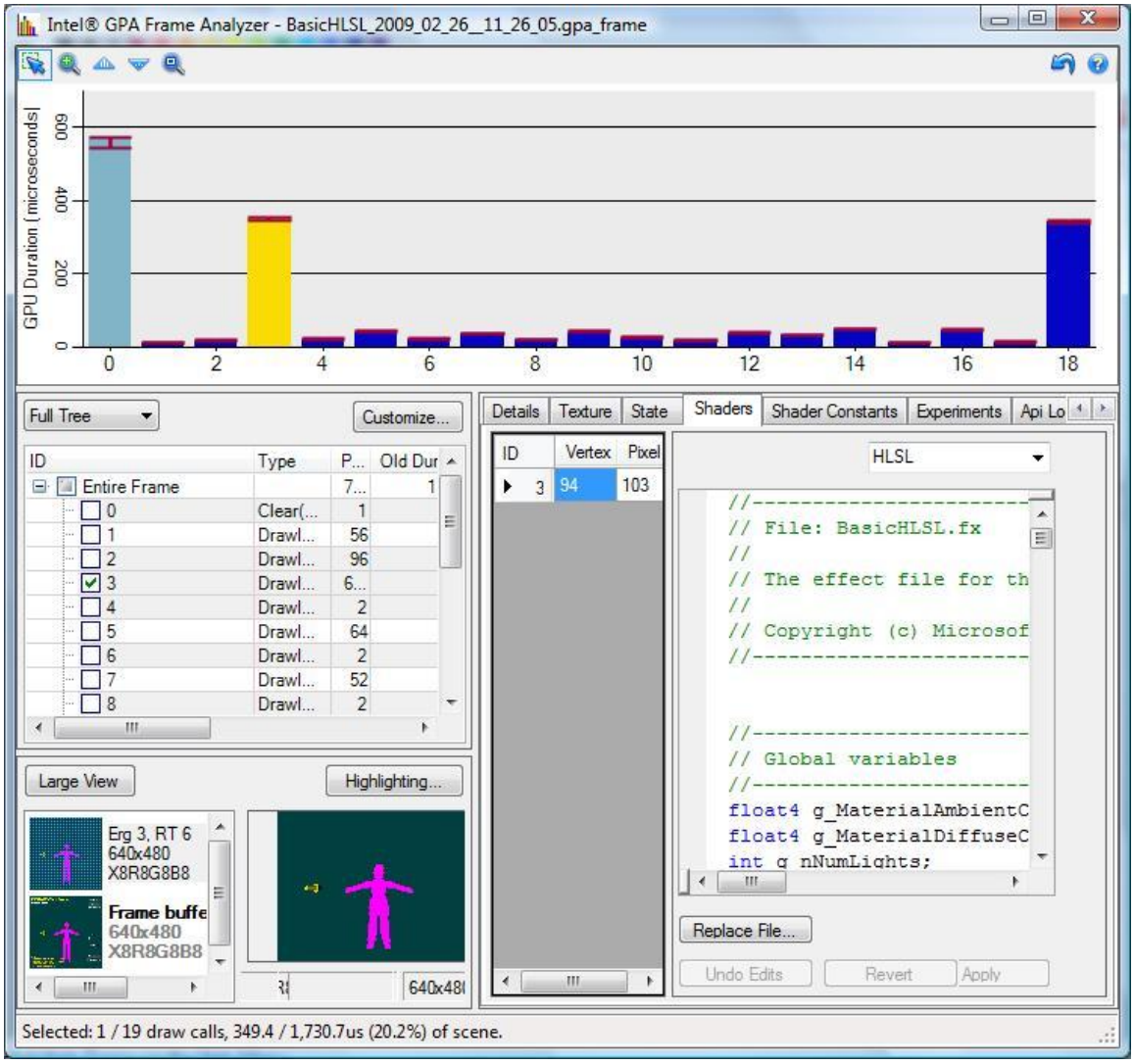
If a GPU bounded situation is suspected, here are some tips to confirm it:

- Render in wireframe mode as this may increase the load on the vertex shader due to the absence of clipping. If you are vertex bound, the frame rate should drop. However, this will reduce the number of pixels emitted and hence the pixel shader load will decrease. Intel GPA provides a feature to render in wire frame mode by changing the D3DFILL_SOLID rasterization render state to D3DFILL_WIREFRAME without changing the code. In the figure below, the Microsoft DirectX* SDK sample's Tiny mesh is changed to render in wireframe mode.

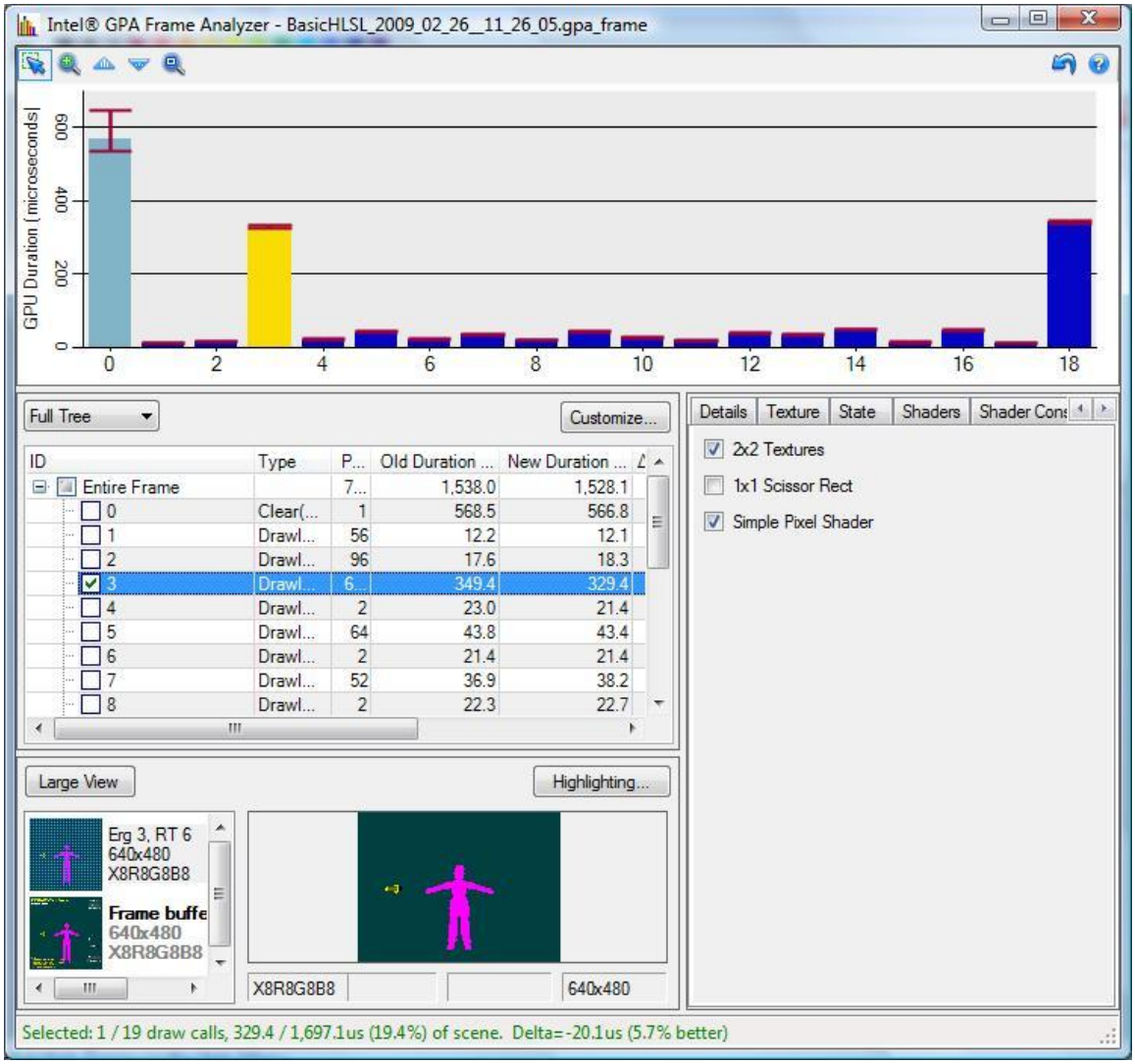


- Look for complex shaders and short circuit them to return immediately. This will help determine if the content of the shader is a bottleneck for the GPU. Intel® GPA provides a feature to have a shader output a single color (pink, configurable in the screen shot below) that can identify shader utilization on Intel integrated

graphics noting the shader(s) to the right. The figure below shows the Microsoft DirectX* SDK sample's Tiny mesh with the associated shader short-circuited.



- Be aware that a large numbers of small shaders could cause thrashing in memory, generating performance overhead.
- If texture size, format, or number of textures is suspected of causing excessive bandwidth overhead, utilize Intel® GPA's Frame Analyzer to render the captured frame with a trivial shader/texture combination. In the figure below, the draw call for the Tiny mesh was rendered with a trivial 2x2 texture and a simple pixel shader along with the original and new duration of time taken to render this frame without altering a single line of code.



Bus Load

Evaluation within the bus domain is a less precise science. With current technologies, sustained bandwidth is roughly 65-75% of the computed peak bandwidth, meaning that there is a significant portion of bandwidth that is likely consumed by the CPU and not available as a resource to a bandwidth hungry application such as a game even if no other high-bandwidth applications are running on the CPU.

A good general rule is that if the time averaged bandwidth value is in the vicinity of 70%, the application may be bandwidth limited. If bus bounded situation is suspected, the following checks may help confirm it:

- Monitor the memory bandwidth usage with Intel® GPA. From this, you should be able to get a reasonable estimate of the bandwidth the application is using. Note that the Intel® GPA tool shows the overall bus usage and not just that used by the title application, so this will be a rough estimate in terms of analyzing a single

game application, but it provides some data to start with. If the bandwidth while the game is running is utilizing 90%+, the game is likely bandwidth limited.

- Recall from the Intel® GMA Series 4 architecture that the Intel integrated graphics GPU uses system memory. Add in more main memory to the system or if the system is maxed out, take some out and see if the frame rate varies significantly. A significant performance change indicates a memory limitation.
- Turn off or reduce the resolution of the textures using the Intel® GPA Frame Analyzer 2x2 texture experiment. This should reduce the load on the bus significantly and help narrow down where the memory usage is coming from. The use of large textures tends to be a significant cause of bus bounded scenarios.
- Temporarily short-circuit multi-pass rendering loops and rerun the test to check for performance improvements. Memory bandwidth over the bus can be a major constraint for integrated graphics with multi-pass rendering.
- Using the Intel® GPA override, change the polygon vertex winding order from the default counter-clockwise or clockwise or vice-versa to the other. If the game is bandwidth limited, *generally*, this should not change the frame rate.

5 Enhancing Graphics Performance on Intel® GMA Series 4 with Intel® GPA

5.1 Case Study: Gas Powered Games – "Demigod"*

Intel actively engages with members of the Intel® Software Partner Program. Participation in the program provides independent software vendors, who develop commercial software applications on Intel technology, with a portfolio of benefits to support them across the entire product planning cycle - from planning, to developing, to marketing and selling of their application.

This program has provided a long list of commercial applications with engineering support from Intel with a wide range of work focusing on performance optimizations most recently focusing on multi-core and graphics. Games have long been a focus as a high-performance version of software running on in the consumer space. The following section deep dives into one such engagement with Redmond, WA based game developer Gas Powered Games* http://www.gaspowered.com/. Their action/role-playing/real-time strategy title "Demigod"* was analyzed using Intel® GPA on Intel integrated graphics with the Intel® G45 Express Chipset and Intel® GM45 Express Chipset.

Whether a game is playable or not on a platform is somewhat subjective and requires a fair bit of judgment and engineering intuition with respect to the genre of the title such as first-person shooter, real-time strategy, or massive-multiplayer-online-game, etc. Performance expectations and game play tend to affect the features, detail, and responsiveness expected from the game and the hardware. Often times, specific scenes that under-perform on the graphics hardware can yield a great deal of information about potential performance optimizations.

5.1.1 Stage 1: Graphics Domain

In the Demigod title, several different approaches were taken to localize GPU workload as a performance sensitive area in the game when running on Intel integrated graphics. The goal of this performance analysis was to yield the greatest performance increase with the least amount of fidelity loss to bring the frame rate within a playable range. In keeping with this goal, low fidelity settings were selected as a base case. A test level was selected for the game and performance sampling was started with the Intel GPA System Analyzer. This sampling yielded some interesting metrics noting a low frame rate and fairly significant graphics utilization.

Given the relatively low overall CPU utilization and memory bandwidth load, we can presume that this is not indicative of a single slow frame but rather an overall GPU bounded performance problem with the scene itself.

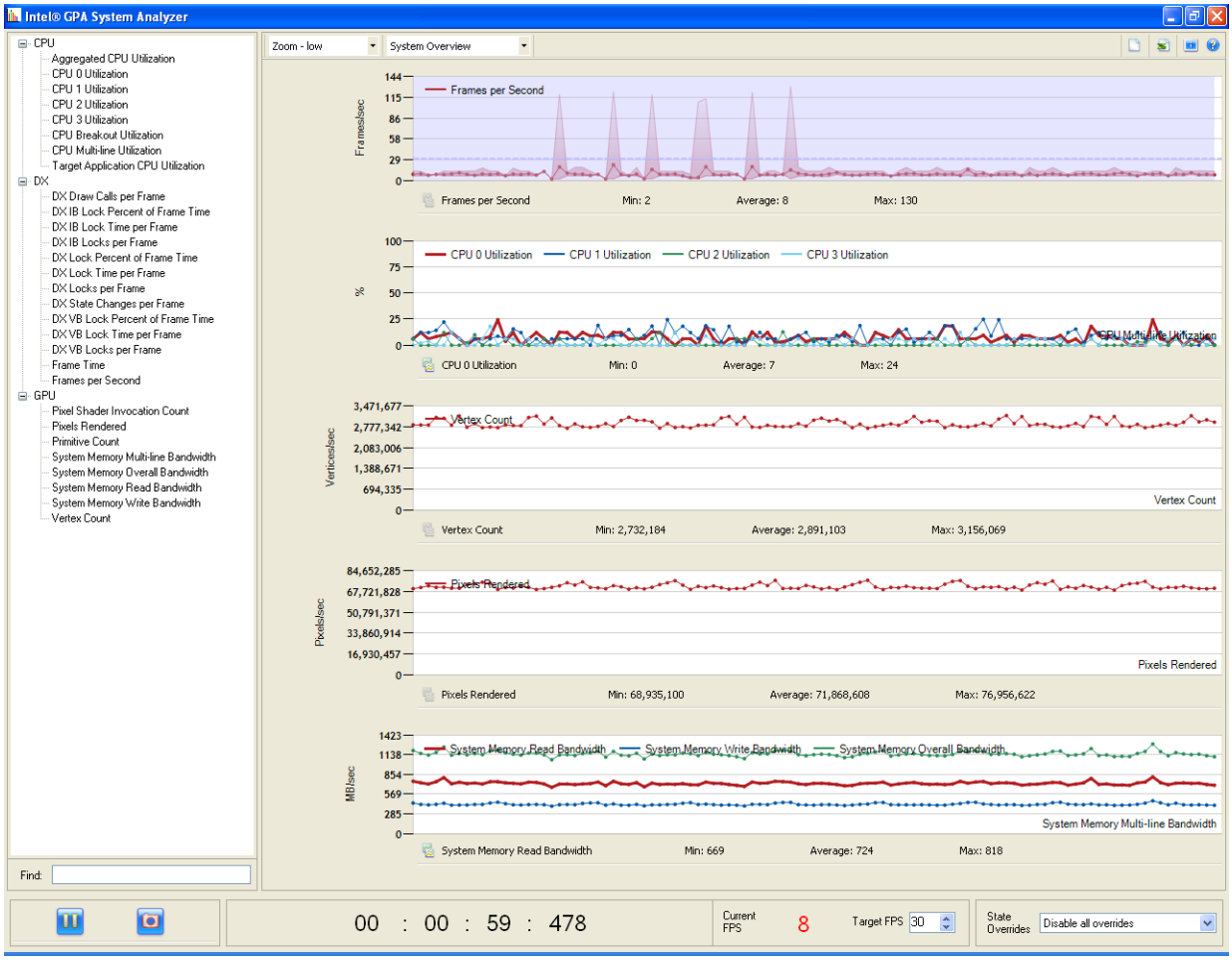


Figure 5 Intel® GPA System Analyzer: Sampling of a Scene in Demigod Indicating a High GPU Load

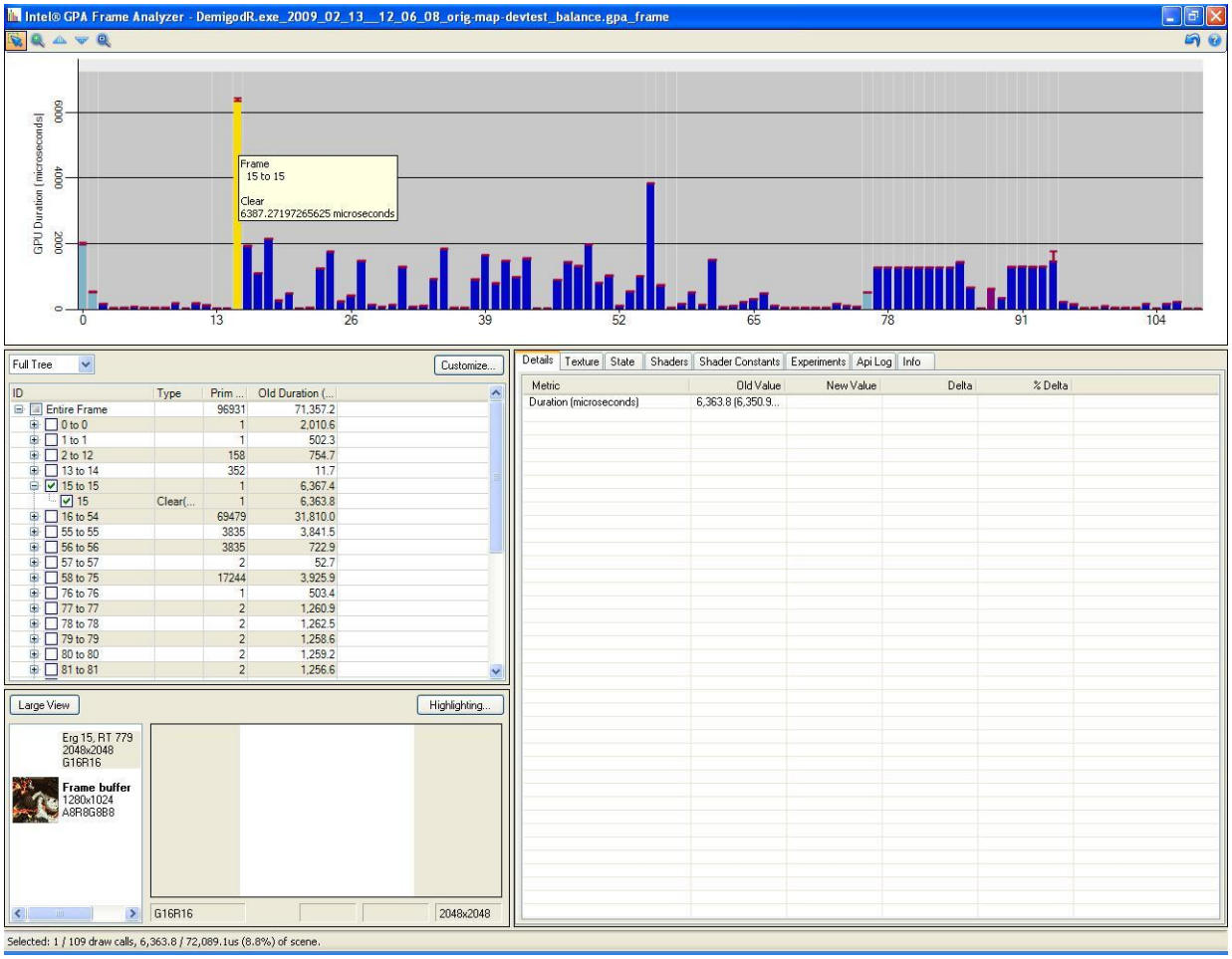
5.1.2 Stage 2: Scene Selection

Further analysis required selecting a specific scene that was yielding low frame rate numbers given that GPU workload remained high as indicated by overall frame rate and low CPU utilization in other scenes as well. The scene below was selected because it is representative of a typical environment rendered with lower graphic settings in which the game operates in terms of visuals, level of detail, characters, props, and graphical workload. The red square in the upper left-hand corner indicates the presence of Intel GPA and the frame rate is indicated in yellow noting 14 frames-per-second for this scene.

Figure 6 A typical Scene in Demigod: Graphics Detail is on the Lowest Game Setting

5.1.3 Stage 3: Isolating the Cause

In some cases enabling efforts are supported by the presence of source code. GPG/Demigod was one such case allowing for a detailed exploration of the code and how it matched up to what was going on in the rendered scene. Much like Intel® VTune™ Performance Analyzer can identify hot spots in code, the Intel® GPA Frame Analyzer is able to match up code hot spots to the unit of time captured within a sample set of frames and also work within a single rendered frame. The Intel GPA Frame Analyzer allows us to further explore the GPU performance of the game in greater detail. When we first started analysis we did not see high bus utilization which would be expected in cases where the GPU is handling too large of a vertex buffer so it was likely that the issue was elsewhere on the GPU side. The Intel® GPA Frame Analyzer provides a window into what is going on in the scene.



The figure below shows a significant spike in GPU processing (aggregate duration of time) over the sampling set. This indicated a large number of calls to Clear a buffer in the histogram.

Figure 7 Intel® GPA Frame Analyzer Sampling Indicating a Hot Spot in the Clear Call

Recall in section "Tips on Texture Sampling / Pixel Operations" that unnecessary calls to Clear have a performance impact on Intel integrated graphics. Noting that we have selected a low fidelity mode, the code was double checked and determined that while Shadows were disabled in low fidelity mode, the sizeable texture buffer was still getting allocated and Cleared. A simple condition to avoid this call when Shadows were disabled yielded a slight performance boost to 15 frames-per-second as noted below without changing the rendered scene.

Figure 8 After Disabling the Clear Call when Shadows are Disabled

Now that one problem was diagnosed, returning to the Intel® GPA System Analyzer yielded numbers similar to the previous result. This is somewhat expected because the first fix was not the root cause, but still worth evaluating as a possible change.

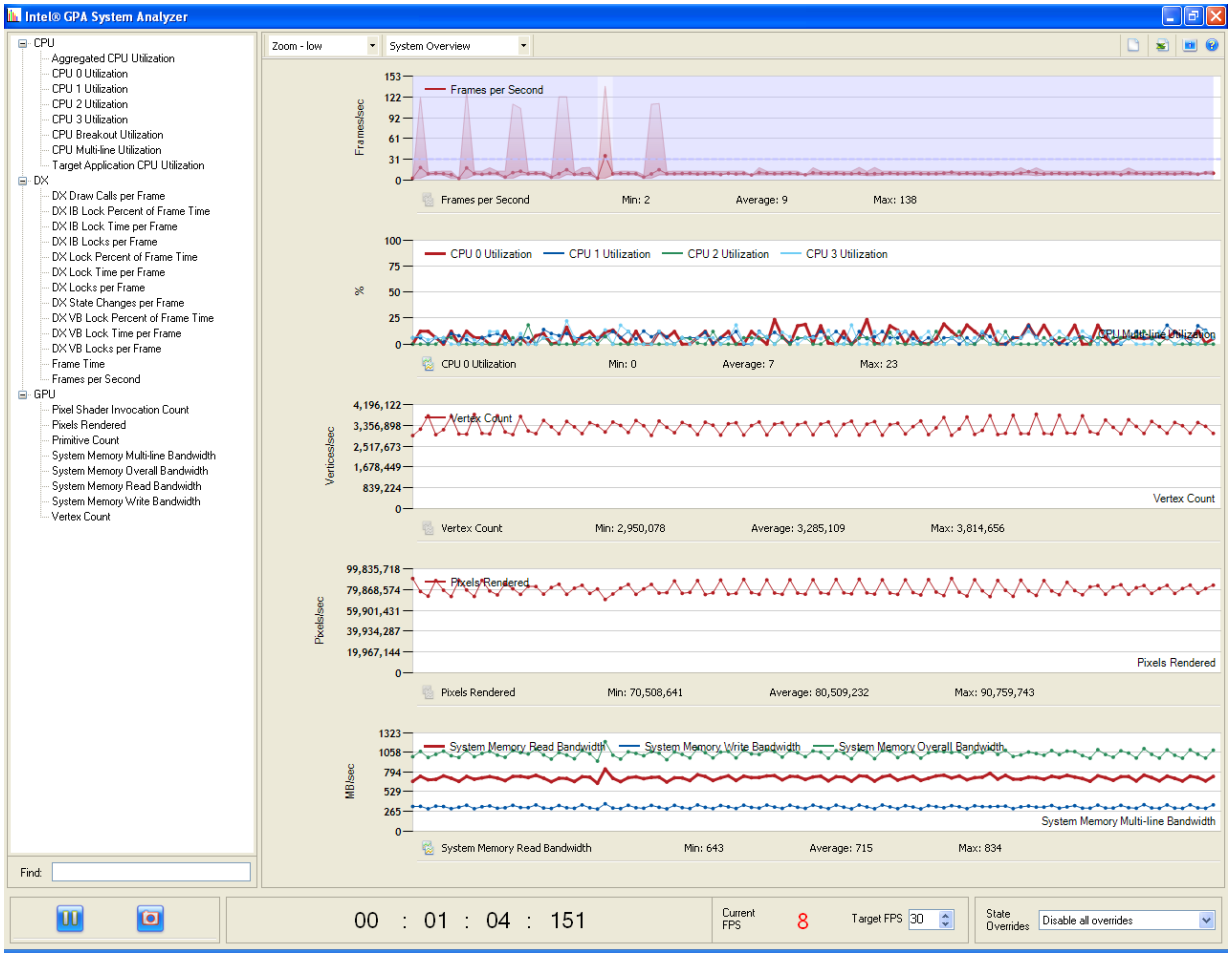


Figure 9 Intel® GPA System Analyzer after Skipping the Clear Call in Low Fidelity Mode

This supports the assumption that the GPU is still busy doing other work. The Intel® GPA Frame Analyzer also details shader activity within a sample set and a single frame. Returning to the Intel® GPA Frame Analyzer and looking at the shader activity during that frame as well as an analysis of the shader itself and the number of instructions executed by each, we found that a specific shader was consuming a lot of time on the GPU. The Intel® GPA Frame Analyzer offers the interesting "comment this out" functionality by overriding a shader to short circuit it and only does the work of outputting a single color – yellow. This will give us a visual indicator of what that shader does. Best of all, this can all be done without editing a file. Just change a setting in the Intel® GPA Frame Analyzer and the frame will be recomputed on-the-fly with the output applied by the test shader to render the same scene we saw before. The frame rate increase is a result of applying the simplified the Intel® GPA Frame Analyzer yellow shader.

Figure 10 Same Scene with a High GPU Load Shader Outputting Yellow

It looks like this particular shader is not significantly affecting the scene in Low Fidelity mode. Here is a pixel-by-pixel comparison of the key differences between this altered scene and the original. Looking at the code, it turns out that the shader applies a metallic feature to the structures in the scene, ignoring some differences in the lava's post-processing effects that will be explained later.

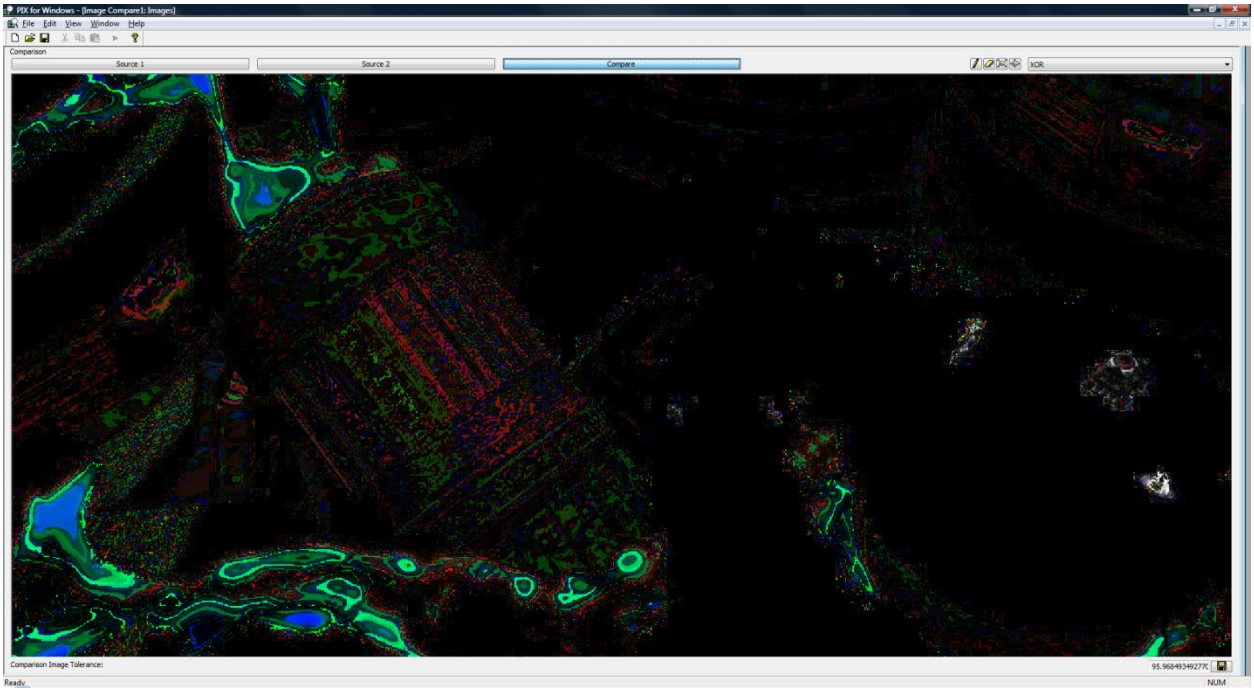


Figure 11 Pixel-by-pixel Image Comparison of the Intel® GPA Frame Analyzer's Yellow Shader and the Original

Removing this shader from processing bumped the frame rate up again to roughly 18 frames-per-second, while only removing a relatively low visual fidelity attribute in the scene. Returning to the Intel® GPA System Analyzer with the change to skip the Clear call and not utilizing the metallic shader yielded the following results.

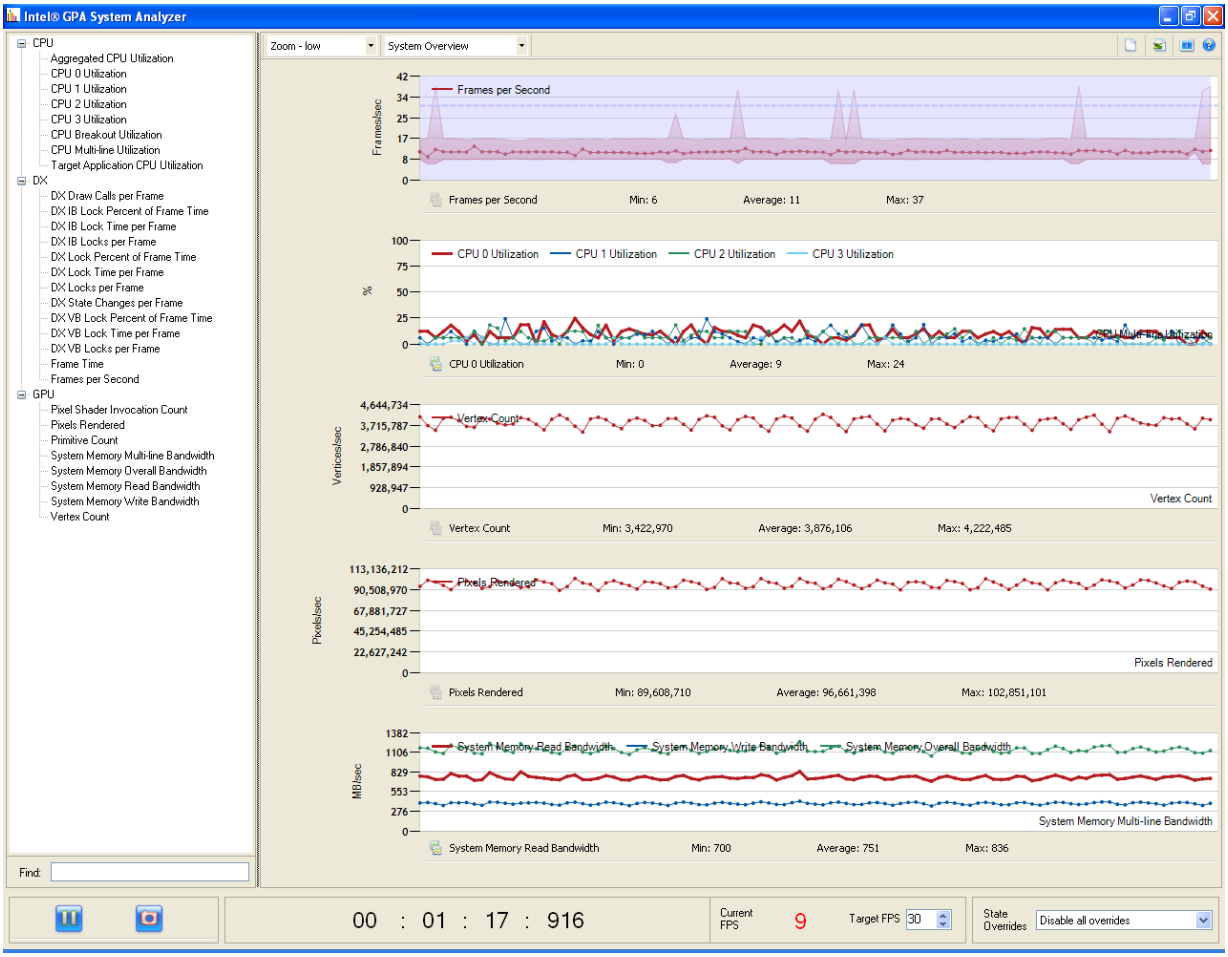


Figure 12 Intel® GPA System Analyzer after the Clear and Shader Change Applied

Based on this new sampling from the Intel® GPA System Analyzer, the CPU load is relatively the same to previous sample sets and as evident in the game, frame rate was still low indicating that there is still a GPU bounded problem. Returning again to the Intel® GPA Frame Analyzer, it appears that two post-processing effects on the lava in the scene were consuming a good deal of resources for integrated graphics. By disabling Bloom and Blur in the code that Demigod provided, the frame rate jumps up to 26 frames-per-second but a great deal of visual fidelity is lost which is not desirable.

Figure 13 Light Shaft Blur and Bloom Disabled - Clearly not a Desirable Change

After noticing the fidelity loss by disabling both settings, Bloom was left on but the blur post processing effect from the light shafts was disabled, yielding nearly the same performance gain (24 frames-per-second versus 26 found when both effects were disabled)

Figure 14 Final Result: Clear, Metallic Shader Removed, Light Shaft Blur Disabled with Bloom on

5.1.4 Key Takeaways from this Analysis

The final tally is a net increase of 14 to 24 frames-per-second running on Intel integrated graphics in low fidelity mode simply by removing a few high-end effects while preserving as much of the scene as possible. Reflecting back to the pixel-by-pixel comparison earlier in this section that includes the blur effect's removal, you'll see the net total difference was relatively small to the rendered scene bringing the game within a more playable range.

6 Support

- Intel's integrated graphics chipset development community forum:

  http://software.intel.com/en-us/forums/developing-software-for-visual-computing/

- Game programming resources:

  http://software.intel.com/en-us/visual-computing/

- Intel® Software Network:

  http://software.intel.com/en-us/

- Intel Software Partner Program:

  http://www.intel.com/software/partner/visualcomputing/

- Intel Visual Adrenaline graphics and gaming campaign:

  http://www.intel.com/software/visualadrenaline/

- Intel® VTune™ Performance Analyzer:

  http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

7 References

[1] "Copying and Accessing Resource Data (Direct3D 10)". Direct3D Programming Guide. Microsoft DirectX SDK (November 2008).

[2] "DirectX Constants Optimizations for Intel integrated graphics". Intel Software Network, Intel: http://software.intel.com/en-us/articles/directx-constants-optimizations-for-intel-integrated-graphics/