Developer's Guide

DirectX* Developer's Guide for Intel® Processor Graphics

Maximizing Performance on the New Intel Microarchitecture Codenamed Ivy Bridge

Abstract

This guide was created to enable the developer to optimize Microsoft DirectX* applications for use on Intel processors based on the Intel® microarchitecture codenamed Ivy Bridge. Improvements to the hardware and game optimizations are discussed. Ways in which a developer can improve power efficiency of an application are presented. Sample code is included demonstrating some of the suggested techniques to assist the programmer/architect further.

Version 2.0- April 18, 2012

Table of Contents

1 Introduction

This guide was created to assist developers in maximizing application performance on new Intel® graphics technologies, primarily focused on the new Intel® microarchitecture codenamed Ivy Bridge. The document will introduce the new architecture and detail best practices with a focus on the DirectX 9, 10 and 11 APIs. Following the guidelines laid out in this document will help the developer's game reach optimal performance on Intel processor graphics while providing a great gaming experience to the largest possible market. As reported by Mercury Research in the second quarter of 2011, Intel's combined integrated graphics line currently enjoys the largest market share, standing at 50.4% for desktop parts, 59% for mobile parts.

With the introduction of the Intel microarchitecture codenamed Sandy Bridge, the graphics processor has moved onto the same die as the CPU, and is now referred to as "processor graphics". In addition, processor graphics has enjoyed numerous architectural improvements that yield significant performance improvements over previous generations of Intel® integrated graphics parts.

The new generation of microarchitecture, codenamed Ivy Bridge, provides another jump in functionality and performance over Sandy Bridge microarchitecture. Ivy Bridge microarchitecture is manufactured on the new 22nm process technology, incorporating Intel's new tri-gate (or 3D) transistor technology. This innovative new process yields greater performance at much lower power. For example, the new tri-gate transistors exhibit 36% faster switching speed than the equivalent transistors on the legacy process at the same voltage. Tri-gate transistors also leak about 10x less current in their off state, resulting in a power saving of approximately 50% when using a comparable performance profile.

The table below compares the main features for both Intel® HD Graphics 3000 found on the 2nd Generation Intel® Core™ processors and Intel® HD Graphics 4000 found on the 3rd Generation Intel® Core™ processors.

Intel® HD Graphics 3000	Intel ® HD Graphics 4000
DirectX 10.1 DirectX Shader Model 4.1 OpenGL 3.0	DirectX 11 DirectX Shader Model 5.0 OpenGL 3.X
12 Execution Units (EUs)	16 Execution Units (EUs)
5 threads/EU	8 threads/EU
1 texture sampler	2 texture samplers
32nm process (CPU, GPU)	22nm process (CPU, GPU)
Traditional "planar" transistor technology	New "tri-gate" transistor technology

Table 1:1: Intel® HD Graphics 3000 vs. Intel® HD Graphics 4000

2 Architectural Overview

2.1 Processor Graphics – an Overview of the GPU

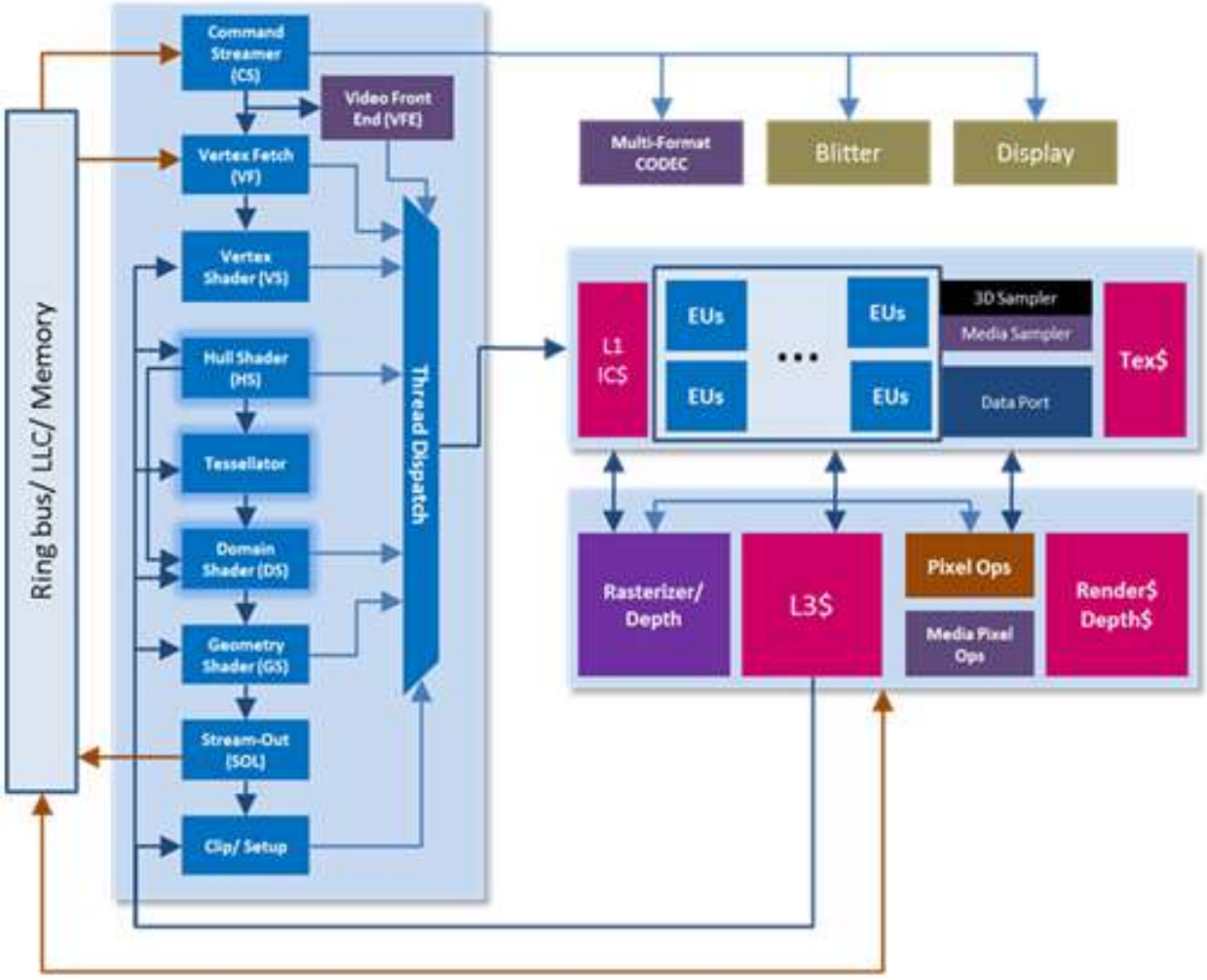


Table 2:1: Overview of the Processor Graphics

Intel 2nd Generation Core™ processor graphics come in two versions, called "Intel HD Graphics 2000" and "Intel HD Graphics 3000". Intel HD Graphics 2000 is the base level part, containing 6 Execution Units (EUs) and one texture sampler. Similarly, the Intel HD Graphics 2500 is the base level part in Intel 3rd Generation Core™ processors. Intel HD Graphics 3000 for Sandy Bridge microarchitecture and Intel HD Graphics 4000 for Ivy Bridge microarchitecture are the higher-end parts, containing 12 EUs, and 16 EUs respectively. Intel HD Graphics 4000 includes a second texture sampler as well.

2.2 Intel® Turbo Boost

2nd Generation Intel Core™ CPUs and processor graphics both benefit from Intel Turbo Boost Technology (sometimes called "Turbo mode") – a feature which allows either the CPU or graphics to increase frequency when the total system load allows for it. For example, if an application runs a CPU intensive section then the processor can increase the frequency above its rated upper power level for a limited amount of time using the

Intel Turbo Boost technology[1]. Conversely, if the CPU is not maxed out and the graphics are fully loaded, it is possible for the system to increase the frequency of the graphics.

3 Optimization Guidnce

This section contains details on optimizations that benefit Intel HD Graphics, specifically those found on Intel microarchitecture codenamed Sandy Bridge and Intel microarchitecture codenamed Ivy Bridge. This guide is not definitive, and outlines the most common issues that are likely to be encountered.

3.1 Tools

The first step to finding and fixing performance problems in an application is using the correct tools. Below is a list of some useful tools Intel engineers often use to identify performance issues.

3.1.1 Intel® Graphics Performance Analyzers

Intel Graphics Performance Analyzers (Intel GPA) is a tool provided by Intel. It provides extensive functionality allowing developers to perform an in-depth analysis on graphics API calls, and determine where the primary performance issues arise. As the name implies Intel GPA contains 4 different types of graphics analyzers: The Monitor, the Frame Analyzer, the Platform Analyzer, and the System Analyzer.

Intel GPA can be found here: www.intel.com/software/GPA/

Figure 3:1: Looking at the HUD in the Intel® GPA Monitor

Intel GPA Monitor has a heads-up display (HUD), which runs on top of an application and provides quick view information on current frame rate, as well as 4 real-time metrics graphs. These graphs can be configured to show metrics for the DirectX* pipeline, CPU and core utilization, and on systems with Intel processor graphics it is also possible to show extensive graphics hardware metrics. The HUD also provides access to simple experiments (called overrides) to help developers quickly detect performance issues. For example, the HUD has hot-keys to force rendering to only use simple shaders (i.e. a quick test to see if the application is Arithmetic Logic Unit (ALU) bound), or 2x2 textures (bandwidth bound), etc. The GPA documentation has more

[1] http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

details on the HUD and Monitor features and functionality, and there are many videos to assist developers in getting started quickly.

Figure 3:2: Part of the Intel GPA is the Frame Analyzer

Intel GPA Frame Analyzer provides functionality to perform draw call level analysis on a single frame. Frames may be captured from the Monitor, and then analyzed later in the Frame Analyzer. This in-depth analysis provides access to a number of metrics, including data on all Intel graphics performance counters, such as the amount of time the execution units are stalled for a particular call or group of calls. See the Intel GPA documentation for more details on the Frame Analyzer or view the available videos.

The Platform Analyzer found in Intel GPA is especially useful when a developer instruments an application to provide details on where performance bottlenecks may occur. The API to utilize is quite easy to insert and the benefits are especially helpful when viewing multi-threaded programs.

The last analyzer in Intel GPA is the System Analyzer. This tool allows the developer to test a system that may not have enough display real estate to show all of the required information. By connecting to the system through an internet connection with a testing system, commands can be easily relayed and results can be easily collected. The same experiments (or overrides) found with the Monitor HUD can be used with this analyzer.

Some of the performance recommendations outlined in this document will be accompanied with information on how to identify related issues using Intel GPA.

3.1.2 Microsoft GPUView*

Microsoft GPUView* can be installed as part of the Microsoft Windows Performance Toolkit, which ships with the Windows SDK. GPUView is an extremely useful tool for determining if an application is CPU or GPU bound. More information on downloading the tools can be found here: http://msdn.microsoft.com/en-us/performance/cc825801.aspx

Figure 3:3: This photo shows Microsoft GPUView* results from a desktop system.

3.1.3 VTune™ Amplifier XE

As an Intel Parallel Studio XE tool, VTune™[2] helps the developer gain information on code performance. It can be used within Microsoft* Visual Studio* or on its own with the GUI Client. Intel VTune is especially useful for profiling game engine code and tuning its performance. A trial version download of VTune is available here: http://software.intel.com/en-us/articles/intel-software-evaluation-center/

3.2 General Recommendations

3.2.1 Targeting Intel Processor Graphics

To provide an optimal user experience, it is recommended that the developer identify the hardware platform the program is running on and set pre-defined suggested settings based on the known performance characteristics of that platform. This will provide the best out-of-box experience for customers. For Intel HD

[2] VTune is a trademark of Intel Corporation in the U.S. and/or other countries.

Graphics (based on the Sandy Bridge and Ivy Bridge microarchitectures) we recommend testing on the target hardware platform to verify playability as this varies widely depending on the game genre.

The GPUDetect sample has an implementation to accurately query available video memory for Intel processor graphics. With this information, the default resolution, shader complexity, texture resolution, etc. for the game can be determined to give the best trade-off between performance and visual quality.

2nd Generation Intel Core™ processors with Intel's HD Graphics 3000 are capable of playing the majority of AAA PC games at medium or better quality settings at a resolution of 1280x720 or better, and the features of Intel HD Graphics 4000 have significantly improved performance.

3.2.2 Driver Overhead

Limiting render state changes and shader switches between draws is a good general rule. Switching Pixel and Vertex Shaders are among the most expensive switches to make, so sorting by these before rendering is recommended. Sorting draws by material or by using instancing to batch similar geometry are examples of how sorting can be accomplished. Instancing can significantly reduce draw calls and thus reduce driver overhead. An example implementation is shown in the Batch Size sample. Use instancing, texture arrays and constant buffer updates to minimize state and shader changes.

3.2.3 Take Advantage of Early Z Rejection

Early Z rejection is where, in certain cases, hardware can make a determination that a group of pixels will not be visible due to being occluded by pixels closer to the camera. In these cases, the cost of ALU computation plus texture fetch is avoided for those pixels, as the hardware will reject them and move on to the next block of pixels.

There are various methods that can be utilized to take full advantage of early Z rejection, and minimize the cost of redundant pixel operations in a frame. A common optimization is to sort opaque geometry from front to back, to ensure that overdraw is minimized and thus the number of redundant pixel shader operations is minimized as well.

Some games may benefit from rendering a depth-only pass (meaning no color buffer writes or pixel shader execution) followed by a normal render pass. This uses the higher performance of Early-Z to reject occluded fragments which in turn reduces compute times and raster operations. For further performance gains, the developer may want to use a modified version of a color-pass vertex shader, stripping code that produces interpolated parameters for the pixel shader.

Refer to the Early Z Rejection sample bundled with this guide for an implementation of both z pre-pass and sorting.

Before attempting this optimization, consider the performance trade-offs between the additional vertex processing, CPU overhead of added draw submissions, and fill rate costs of a depth-only pass against the performance overhead of rendering without one. Using a tool (such as Intel GPA) to examine overdraw in a frame can help determine if an application has the potential for significant savings in this area.

Avoid writing depth values in a pixel shader. Writing the depth value will skip the Early-Z hardware optimization since it potentially changes the visibility of the fragments.

3.2.4 Consider Using Post-Process Anti-Aliasing Methods

The performance of Multi-Sample Anti-Aliasing scales with increasing scene complexity and resolution, whereas post-process anti-aliasing methods are generally not very sensitive to the complexity of the scene. For the scenes present in many modern games post-process anti-aliasing is usually a performance win, and can result in similar quality.

Intel has provided two samples which demonstrate possible techniques. The Morphological Anti-Aliasing Sample demonstrates how to filter edges based on their shape, and the Temporal Super Sampling Anti-Aliasing in the Dynamic Resolution Rendering Sample provides a technique for intelligently using previous frame data to increase effective resolution without ghosting artifacts.

3.2.5 Other Tips

1. Use visibility tests to reject objects that fall outside the view frustum to reduce the impact of objects that are not visible.
2. 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. Allow sufficient time between Occlusion Query tests and verifying the results to avoid serializing the rendering. See the Microsoft DirectX 10 "Draw Predicated" sample included in the Microsoft DirectX SDK for more information on this.
3. Draw the opaque overlays in the scene, such as a heads up display (HUD), first and write them to the Z buffer (at the closest possible distance). This takes advantage of early Z rejection, and effectively reduces the screen rendering area leading to considerable performance improvement.

3.3 Recommendations on Vertex Processing

3.3.1 Optimizing Geometry for the Caches

Use IDirect3DDevice9::DrawIndexedPrimitive (DirectX 9) or ID3D10Device::DrawIndexed (DirectX 10 and 11) to maximize reuse of the vertex cache.

Pre- and post-transform vertex cache sizes can vary significantly across different hardware, and even across different generations of Intel graphics platforms. To maximize performance of vertex processing, optimize the ordering of vertices and triangle indices in vertex and index buffers.

The D3DX library provides an API for applying a generalized vertex cache optimization that works well for all cache sizes. In DirecX 9, this functionality is exposed via the D3DXOptimizedFaces and D3DXOptimzeVertices functions. In DirecX 10/11, it is exposed via the ID3DX10Mesh::Optimize and ID3DX10Mesh::GenerateAdjancencyAndPointReps interface methods.

Since the D3DX vertex cache optimization works well across all hardware, it can be applied at authoring time (e.g. when exporting mesh data from a content creation pipeline, or baking final geometry data into the game's data files), cutting down level load times. The code snippet in Appendix A demonstrates how to optimize vertex and index data using DirectX 9.

Also note that most professional 3D modeling packages provide export options for optimizing meshes by ordering the vertices and triangle indices in a generalized, vertex cache-friendly manner. Please refer to the 3D modeling application documentation to learn more about these options.

3.3.2 Tessellation

Tessellation is introduced in the DX11 runtime and is supported by Ivy Bridge microarchitecture processor graphics. Tessellation is enabled via 3 new stages: Hull Shader (HS), Fixed Function Tessellator (TE) and Domain Shader (DS).

Tessellation's primary function is to increase visual fidelity of characters and terrain by increasing the amount of polygon complexity. By incorporating tessellation in-line the GPU generates higher resolution models based on input of low-resolution models while minimizing memory bandwidth and CPU to GPU traffic.

It also supports displacement mapping and scalable LOD (Level of Detail) techniques. DX11 Tessellation is described in detail in Microsoft's Windows SDK.

There are some optimizations to keep in mind while implementing tessellation:

- Given that tessellation requires more computes to produce triangles (VS+HS+DS), it is especially important to avoid submitting objects (patches) that are completely out of view and do not contribute to the final image. Developers might also consider adding a visibility cull test within Hull Shaders and thereby culling patches that are outside the view frustum, etc.

- Developers should take advantage of tessellation factor clamping through use of the MaxTessFactor declaration in the Hull Shader and adaptive tessellation in order to avoid over-tessellation of patches. Common adaptive techniques include: view dependent adaptive tessellation to only tessellate along silhouette edges where the added detail of tessellation plus displacement mapping can better be seen, and screen-space adaptive tessellation where there is a minimum triangle size in screen space and the tessellation factors are set so that triangles will not be any smaller than the threshold.

- Tessellation is powerful since it helps avoid input geometry bandwidth limitations. However, in cases where input bandwidth is not a concern, the developer needs to trade-off the advantages of tessellation with the increased compute requirements of executing Hull and Domain Shaders. This is especially important where the impact to visual quality is minimal.

3.4 Recommendations on Pixel Processing

3.4.1 Render Targets

It is recommended that no more than 3 render targets are used for best performance. The more render targets used, the more bandwidth is required. To determine if this is a problem, use a tool such as Intel GPA to determine if the program is "fill" bound. Refer to the article Practical Game Performance Analysis Using Intel Graphics Performance Analyzers to learn how to detect fill rate issues with Intel GPA.

Further, it is also recommended that smaller render target formats are used. For example, for floating point render targets, choose 32 bit/pixel formats such as DXGI_FORMAT_R11G11B10_FLOAT rather than 64 or 128 bit formats. Larger formats require more bandwidth. Again, use a tool such as Intel® GPA to determine if fill rate is an issue.

3.4.2 Textures

3.4.2.1 Optimal Texture Formats and Sizes

Use compressed (D3DFMT_DXT<n>, ATI1N and ATI2N on DirectX9, or DXGI_FORMAT_BC<n> on DirectX 10 and above) textures where possible. As a general rule, the smaller the texture format, the better the performance seen. Compressed textures are the ideal case. 32-bit floating point textures are the worst case.

For DirectX9, the table below shows the format extensions supported by both Sandy Bridge and Ivy Bridge microarchitecture HD Graphics, however we recommend developers refer to the code in GPUDetect for runtime detection of support on Intel hardware, as this may depend on the driver version on a customers PC.

DirectX 9 Format Extensions Supported	Description
NULLRT	Null render target for use when rendering to depth or stencil buffer with no colour writes. Consumes no memory.
ATI1N	One component compressed texture format similar to DXGI_FORMAT_BC4.
ATI2N	Two component compressed texture format similar to DXGI_FORMAT_BC5.
INTZ	Allows native depth buffer to be used as a texture.
DF24	Allows native depth buffer to be used as a texture.
DF16	Allows native depth buffer to be used as a texture.
ATOC	Alpha to coverage for transparency multisampling.
ATIATOC	Alpha to coverage for transparency multisampling.

Table 3:1: Supported DirectX 9 Format Extensions

Using MIP levels on all textures (where applicable) will help performance as well as quality. It is best to use the smallest size textures possible (making appropriate trade-offs between size and quality). Even though the hardware supports sampling up to 8k by 8k textures, prefer smaller textures.

3.4.2.2 Filtering Recommendations

Minimize the use of anisotropic filtering – only use it where absolutely necessary, and make appropriate trade-offs on the level of anisotropic filtering required when it is used. For floating point texture formats, minimizing use of anisotropic filtering and trilinear filtering is recommended, since there is a significant penalty for sampling in these cases

3.4.2.3 Using Stencil as Texture

A new feature in Direct3D 11 allows reading from the stencil buffer in a depth-stencil surface via texture sampling. Due to the Sandy Bridge and Ivy Bridge microarchitetures, using this method is not recommended as it can cause some performance degradation (the performance hit increasing exponentially with the resolution of the texture in question).

Currently, sampling from a depth/stencil with 24 bit depth, 8 bit stencil is supported but may have a performance cost. Sampling from MSAA 24-bit depth, 8 bit stencil, and from 32-bit depth, 8-bit stencil is not supported. A fix may ship in a future driver release to allow these scenarios to work, although it should be noted that any driver workaround would still come at a performance cost.

This issue will be addressed in a future release of Intel processor graphics. However, for current generations it is recommended that using stencil surfaces as textures is avoided.

3.4.2.4 Volumetric Textures

Using volume textures will incur a performance penalty. Judging on a case by case basis will determine if volume textures are the right approach for the desired effect given the performance hit. The texture sampler counters in Intel GPA provide a good method to measure the performance penalty.

3.4.3 Atomic Operations Performance Impact

Global atomic operations to the same address should be avoided. To identify this issue, look for atomic operations to the same address either in pixel shaders or compute shaders by looking for excessive EU stalls and atomic usage in Intel GPA.

3.4.4 Optimizing Pixel Shaders

3.4.4.1 Consider better performing alternative methods when using Texkill/Clip with Depth Write Enabled

When the texkill/Clip instruction is paired with depth write enabled, be aware of better performing alternatives. The best approach is to disable the depth write enabled state if it is not necessary. If depth write is disabled, the geometry being rendered may have to be sorted to appear correct. The developer should conduct performance analysis on a relevant frame to see if sorting yields better performance than enabling the depth write state. The developer should pre-process the draw order if it does not need to change over the course of the application.

An alternative is to choose alpha test instead of texkill. In general, on Sandy Bridge and Ivy Bridge microarchitectures hardware, alpha test is faster than using texkill, even in cases where texkill skips a large chunk of pixel shader code. Again, the developer should evaluate the choice between alpha test and texkill by conducting performance analysis on a relevant frame and use the better performing of the two options.

3.4.4.2 Spatially Sorting Geometry to Improve Texkill/Clip/Alpha Test Performance with Depth Write Enabled

If the usage of Texkill/Clip or alpha test with depth write enabled is necessary, spatially sorting geometry to avoid drawing consecutive overlapping polygons will help performance. The less polygons simultaneously competing to render to the same pixel in the render target, the faster the operation will run. For example, if the application is rendering dense grass with alpha blending and the geometry is sorted in a front to back order, it is possibile that there are a high number of consecutive overlapping polygons. Most likely, the same set of data can be sorted to maintain the correct appearance while modifying the draw order to avoid drawing consecutive overlapping polygons.

Consider the view frustum in the image below, where each lettered rectangle could be considered a blade of grass. The depth sorted order would be A, B, C, D. In the depth sorted case there are consecutive polygons with B covering A and D covering C. The same set of polygons, spatially sorted, could be drawn in the order A, C, B, D, to avoid consecutive drawing of overlapping polygons.

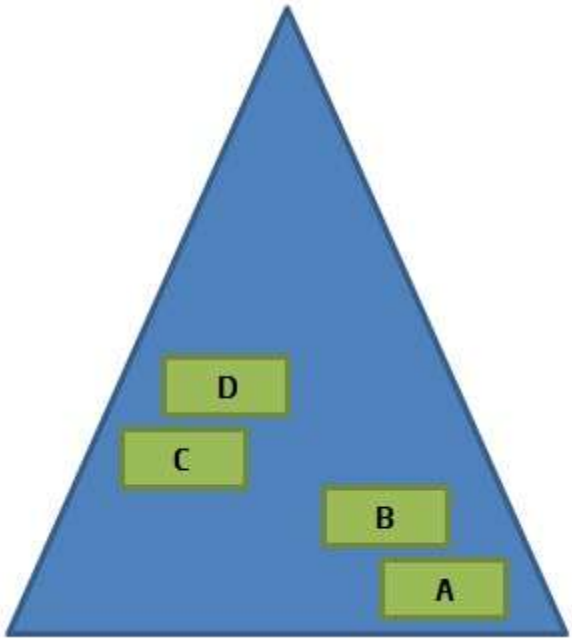


Figure 3:4: A View Frustum Example

3.5 Compute Shaders

With the advent of Direct3D 11, Microsoft introduced a new shader called the compute shader. Although the compute shader shares the use of High Level Shader Language (HLSL) with other shaders, it is significantly different. This general purpose shader uses the capabilities of available GPU parallel cores to execute computation workloads.[3]

3.5.1 Variable SIMD Size

One of the unique features of Ivy Bridge microarchitecture is that the SIMD size is variable - the hardware is capable of dispatching threads to the execution units as SIMD8, SIMD16, or SIMD32. The driver chooses among these options based on a combination of factors (register pressure, thread group dimensions, etc) and maps the HLSL "logical" threads onto physical hardware threads accordingly. There have been issues noted in some publicly available applications and libraries where the author of the HLSL algorithm assumed a SIMD dispatch width of 16 or 32 – that is, the algorithm assumes that 16 or 32 SIMD lanes are executed in lock step and thus certain patterns of data accesses are "safe" from concurrency. Developers testing these algorithms on hardware that matches the assumptions will not see any issue, but when these workloads are dispatched on Ivy Bridge microarchitecture and the driver chooses a SIMD mode narrower than the developer's algorithm assumed, the execution can cause concurrent accesses resulting in data corruption or possibly an infinite loop (hang/TDR).

The DirectX specification makes no stipulation about the hardware implementation in terms of the level of hardware SIMD dispatch width (also known as wavefront size). Algorithms that access data across logical HLSL threads should use the appropriate synchronization mechanisms provided by the specification (barriers, atomics) rather than relying on a device specific implementation and possibly non-portable assumptions about SIMD width in their algorithm design. Developers should verify that their algorithms work properly on Ivy Bridge microarchitecture hardware.

[3] http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx

3.5.1 Use of Structured Buffers

Certain screen-space effects, such as Gaussian blur can be produced by using structured UAV buffers. Because structured buffers are 1-demensional, they are mapped as linear memory. Screen-space operations are actually 2D, consisting of a horizontal pass followed by a vertical pass. This means that the horizontal pass is very fast but the vertical pass becomes very slow. Developers should utilize the RWTexture resources rather than relying on RWStructuredBuffers in these instances. RWTexture resources are faster because they use a tiled memory layout internally that results in a better locality of reference or cache hit rate.

3.6 Queries

3.6.1 The Problem with Queries

DirectX supports use of query objects in order to allow applications to determine GPU related status. The query interface is designed to be polled by the application through the use of the GetData() method. Polling using GetData() in a loop while waiting on the results of a query should be avoided. The reason for this is that polling causes the CPU utilization to spike to 100%. The OS and hardware detect this utilization increase, and as a result they increase the CPU frequency to maximum turbo. The effect of this is an exponential increase in power consumption – for no corresponding increase in performance. This problem is exacerbated on low voltage and ultra-low voltage platforms. On these platforms, the chip level thermal design point (TDP) limits how much power can be consumed at any given time. When an application issues query.GetData() in a tight loop and the CPU frequency is set to maximum, it actually limits the amount of power available to the GPU. As a result, the GPU frequency is lowered, and this will almost always result in a significant performance drop in 3D applications.

3.6.2 Alternate Approaches

Several approaches can be used to mitigate the impact of query.GetData() polling, but in current versions of Windows (up through Windows 7), there is no foolproof way of eliminating the problem. One common application of queries is to use them to detect end of frame conditions. In most cases, this is not required, and it should be avoided. For queries that cannot be avoided entirely, it is recommended to schedule additional work after the query is issued and before the GetData() check is issued. That way the CPU is doing useful work instead of busy waiting. To facilitate this approach, move the query.Issue() operation to a position as early in the frame rendering pipeline as possible. That way, the queries are more likely to be completed by the time GetData() is issued.

3.7 Memory and Memory Bandwidth Considerations

3.7.1 Determining the Amount of Available Memory

Graphics applications often check for the amount of available free video memory early in execution. Developers typically want to know the amount of memory that the graphics device can access at full performance. Intel processor graphics enjoys an increased flexibility in memory usage. Since the graphics share the memory controller and last-level cache with the CPU, it has full access to system memory. This design choice will result in an inaccurate picture of what memory is available if a simple queries for "dedicated" video memory is performed.

The GPUDetect sample packaged with this guide has an implementation to accurately query available video memory for Intel processor graphics. In addition, the Microsoft DirectX SDK (June 2010) includes the

VideoMemory sample that demonstrates several commonly used methods used to detect the total amount of video memory. Out of all these tests, GPUDetect and GetVideoMemoryViaWMI are recommended as the best alternatives. All other methods either return the local/dedicated graphics memory and consequently will report inaccurate results on Intel processor graphics, or will report the sum of the dedicated memory and the shared memory, something that is not usefully comparable to the memory used by discrete graphics devices. For more information on GetVideoMemoryViaWMI, see the sample code here: http://msdn.microsoft.com/en-us/library/ee419018(v=VS.85).aspx

4 Power Efficiency

4.1 Overview

Mobile and ultra-mobile computing are becoming ubiquitous. As a result, battery life becomes a significant issue for users. As manufacturing processes continue to shrink and improve, we see improved performance-per-watt characteristics of CPUs and processor graphics. However, there are many techniques that can be utilized to help improve power utilization on mobile devices in software.

4.2 Efficient Power Practices

4.2.1 Limit Frame Rate

Frame rates of 1000fps for cut-scene playback can sometimes be observed in shipping titles. This causes unnecessary power consumption in the graphics device (and on the CPU), since there will be no visible difference between 1000+ fps and locking the present interval to the 60Hz display refresh. Excessively high frame rates can also be observed in menus, loading screens and other low-GPU-intensive places in games. By locking to 60Hz in all these places, noticeable improvements in power consumption can be observed.

For best user experience (particularly on mobile platforms), games should also consider limiting frame rate during game play to keep the device operating at lower frequencies – this helps on battery life and thermals. Game developers may want to allow the user to disable this for benchmarking purposes (with appropriate caveats on impact to battery life/thermals); the developer may also want to give the end user control to set the frame rate cap based on their preferred experience.

4.2.2 Consider Lowering Rendering Quality Settings When on Battery

The Win32 API GetSystemPowerStatus can be used to determine if a system is running on battery or not. The ACLineStatus member of the SYSTEM_POWER_STATUS struct will be set to 0 for no AC power (laptop on battery), 1 for AC (laptop plugged in), and 255 for unknown (i.e. a desktop system with no battery). Render quality can also be adjusted by experimentation with other fields in the SYSTEM_POWER_STATUS struct, e.g. BatteryLifePercent.

4.2.3 Avoid 'Busy Wait' Loops

A 'busy wait' is defined as a section in code where the application keeps polling for a value until it changes. Below is a typical example of a busy wait loop:

```
HRESULT res;
IDirect3DQuery9 *pQuery;

// create a query
```

```
res = pDevice->CreateQuery(.., &pQuery);
...

// busy-wait for query data

while ( (res = pQuery->GetData(…, 0)) == S_FALSE);
```

Code Example 4.1: A Simple Example of a Busy Wait Loop

The code above may cause unnecessary power consumption since the operating system and the underlying power management hardware has no way to determine that the application does not perform any useful computation, but rather just waits for data from the direct3D runtime subsystem. Such waiting loops should be avoided whenever possible, or if there is no other option, the developer should consider inserting a Sleep call between one poll operation and the next.

4.2.4 Use Multi-threading Where Possible

Splitting the application execution into multiple threads enables the operating system to take advantage of multiple CPU cores available in modern processors. Performing a given piece of computation across multiple CPU's benefits an application's power efficiency in two ways:

1. The computation is completed in a shorter period of time and therefore enables the operating system to switch the processor into 'power saving' mode.
2. As the CPU cores are active for a shorter period of time, the operating system and the power management hardware will tend to operate these cores in lower, more power efficient frequencies.

Appendix

Example Code

The following samples can be found at the Intel Samples Website. The source code, Visual Studio 2008/2010 solutions, and documentation are included in the downloads..

Early Z Rejection

http://software.intel.com/en-us/articles/vcsource-samples-early-z-rejection/

Appendix A Figure 1: Early Z Rejection

This sample demonstrates two ways to take advantage of early Z rejection. When rendering, if the hardware detects that after performing vertex shading a fragment will fail the depth test, it can avoid the cost of executing the pixel shader and reject the fragment. To best take advantage of this feature, it is necessary to maximize the number of fragments that can be rejected because of depth testing. This sample demonstrates two ways of doing this: front to back rendering and z pre-pass.

Front to back rendering requires sorting opaque geometry based on distance from the camera and rendering the geometry closest to the camera first. This rendering order will maximize depth test failure without additional D3D API calls.

For Z pre-pass, all opaque geometry is rendered in two passes. The first pass populates the Z buffer with depth values from all opaque geometry. A null pixel shader is used and the color buffer is not updated. For this first pass, only simple vertex shading is performed so unnecessary constant buffer updates and vertex-layout

data should be avoided. For the second pass, the geometry is resubmitted with Z writes disabled but Z testing on and full vertex and pixel shading is performed. The graphics hardware takes advantage of Early-Z to avoid performing pixel shading on geometry that is not visible.

Performing a Z pre-pass can be a significant gain when per pixel costs are the dominant bottleneck and overdraw is significant. The increased draw call, vertex and Z related per pixel costs of the pre-pass may be such that this is not a suitable performance optimization in some scenarios.

GPUDetect

http://software.intel.com/en-us/articles/vcsource-samples-gpu-detect/

This sample demonstrates how to get the vendor and ID of the GPU. For Intel processor graphics, the sample also demonstrates a default graphics quality preset (low, medium, or high), support of DX9 extensions, the recommended method for querying the amount of video memory, and if supported by the hardware and driver, the recommended method for querying the minimum and maximum frequency.

The minimum frequency can be used as an indicator of wether the processor is a low power configuration, and when used alongside the device ID it can provide a performance indicator.

Optimizing Geometry for the Caches

The following sample code demonstrates how to use the Direct3D D3DXOptimizeVertices and D3DXOptimizeFaces calls to optimize geometry for the vertex pre- and post-transform caches.

```
void OptimizeMesh( WORD * Indices, // [In/Out] - Index buffer data
                   DWORD NumFaces, // Number of faces in the mesh
                   Vertex * Vertices, // [In/Out] - Vertex buffer data
                   DWORD NumVertices ) // Number of vertices in the mesh
{
    HRESULT hr;

    // Create a "re-map" buffer for the new face ordering, and
    // calculate the new order.
    DWORD *Remap = new DWORD[ NumFaces ];
    hr = D3DXOptimizeFaces( IndicesIn, NumFaces, NumVertices,
                            FALSE, Remap );

    // Make a copy of the old indices, which we'll pull from for the new
    // re-ordered list of indices.
    WORD *OldIndices = new WORD[ NumFaces * 3 ];
    memcpy( OldIndices, IndicesIn, sizeof(WORD) * NumFaces * 3 );
    WORD * NewFaces = Indices;

    // Apply the new mapping.
    for( DWORD i = 0; i < NumFaces; ++i )
    {
        WORD *OldFaceBase = OldIndices + ( Remap[ i ] * 3 );
        NewFaces[0] = OldFaceBase[0];
        NewFaces[1] = OldFaceBase[1];
        NewFaces[2] = OldFaceBase[2];

        NewFaces += 3;
    }
```

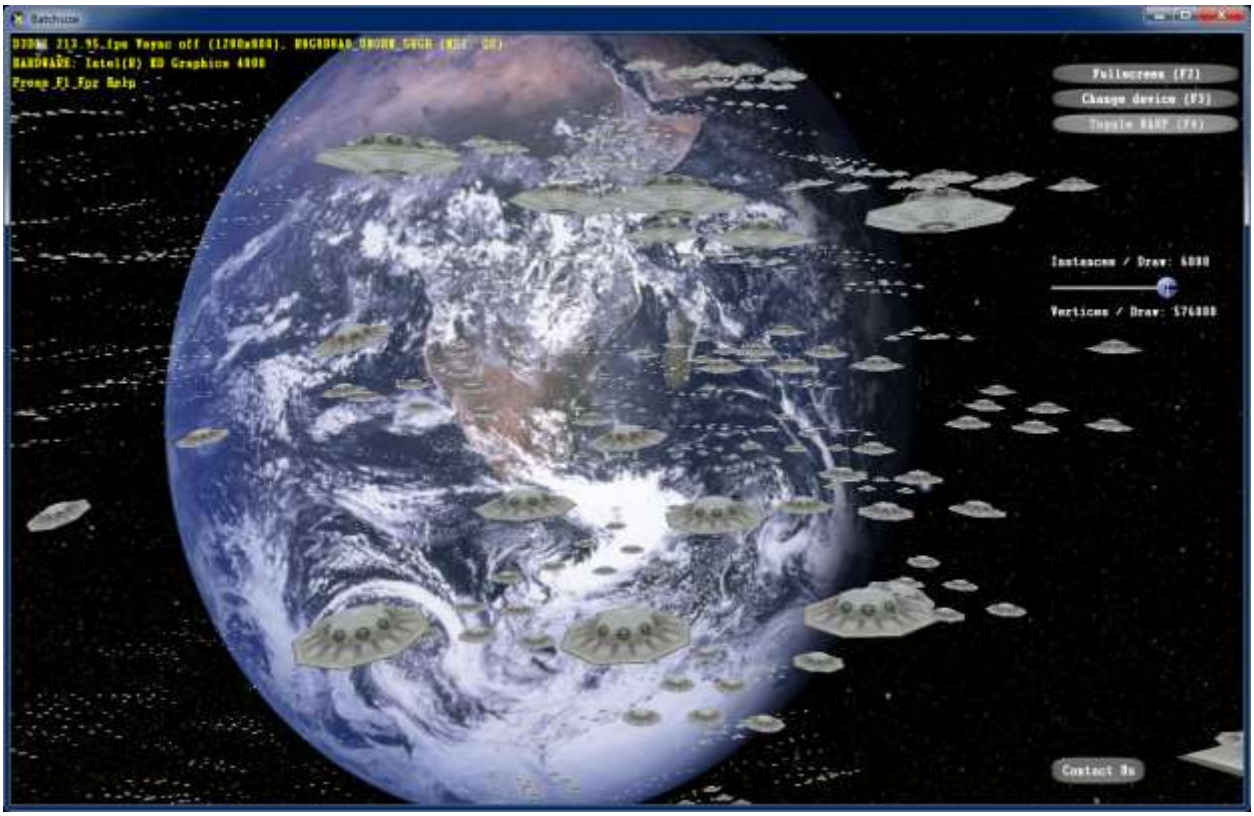
```
    delete[] Remap;
    delete[] OldIndices;

    // Create a "re-map" buffer for the new vertex ordering, and
    // calculate the new order.
    DWORD ActualVertexCount = 0;
    Remap = new DWORD[ NumVertices ];
    hr = D3DXOptimizeVertices( Indices, NumFaces, NumVertices,
                 FALSE, Remap );

    // Count how many vertices we actually have. Remap indices of
    // 0xffffffff indicate a vertex that was not referenced by any faces.

    DWORD dwVertexCount = 0;
    for( DWORD i = 0; i < NumVertices; ++i )
    {
        if( aRemap[ i ] == 0xffffffff )
            break;

        ++ActualVertexCount;
    }

        // Copy the vertex data into a temp buffer.
    Vertex *OldVertices = new Vertex[ NumVertices ];
    memcpy( OldVertices, Vertices, sizeof(Vertex) * NumVertices );

    // Perform the remapping
    const DWORD *CurrentRemap = Remap;
    const SVertex *OldVertex = OldVertices;
    for( DWORD i = 0; i < ActualVertexCount; ++i )
    {
        aVertices[ *( CurrentRemap++ ) ] = *( OldVertex++ );
    }

    // We'll also need to re-index our vertices to point to the new
    // vertex locations.
    for( DWORD i = 0; i < NumFaces * 3; i++ )
    {
        Indices[ i ] = ( WORD )aRemap[ Indices[ i ]];
    }

    delete[] Remap;
    delete[] OldVertices;
}
```

Code Example 4.2: Optimizing Geometry for the Caches

Batch Size

http://software.intel.com/en-us/articles/vcsource-samples-batch-size/

This sample demonstrates the use of instancing to limit the number of draw calls to minimize driver overhead and maximize hardware efficiency. The sample renders many small meshes. The meshes can be drawn individually, or grouped together as instances with per-mesh data (world transforms for example) stored in a separate, per-instance buffer. The number of meshes per draw call can be varied to study the driver overhead and GPU costs associated with the workload.



Appendix A Figure 2: Batch Size Sample

Notices

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101