GPU Programming Guide
GeForce 8 and 9 Series

December 19, 2008

HISTORY OF MAJOR REVISIONS

Version	Date	Changes
1.0	12/19/2008	Initial

Table of Contents

Chapter 1. About This Document

1.1. Introduction

This guide will help you to get the highest graphics performance out of your application, graphics API, and graphics processing unit (GPU). Understanding the information in this guide will help you to write better graphical applications.

This document specifically focuses on the GeForce 8 and 9 Series GPUs, however many of the concepts and techniques can be applied to graphics programming in general. For specific advice and programming guide on earlier GPUs such as the GeForce 6 & 7 series please consult the earlier "GPU Programming Guide : GeForce 6 & & Series (and earlier)".

For information about anything and everything graphics and GPU related please check http://developer.nvidia.com/page/documentation.html

This document is organized in the following way:

- Chapter 1(this chapter) gives a brief overview of the document's contents.
- Chapter 2 explains how to optimize your application by finding and addressing common bottlenecks.
- Chapter 3 lists tips that help you address bottlenecks once you've identified them. The tips are categorized and prioritized so you can make the most important optimizations first.
- Chapter 4 presents several useful programming tips for GeForce 8 & GeForce 9 Series GPUs. These tips focus on features, but also address performance in some cases.
- Chapter 5 presents general advice for programming Microsoft's DirectX 10 API and considerations when porting from DirectX 9.
- Chapter 6 provides some general advice on programming graphics with NVIDIA GPUs.
- Chapter 7 provides an overview of NVIDIA's performance tools.

Chapter 2.How to Optimize Your Application

This section reviews the typical steps to find and remove performance bottlenecks in a graphics application.

Also refer to http://developer.nvidia.com/object/practical_perf_analysis.html for more information on performance analysis.

2.1. Making Accurate Measurements

Many convenient tools allow you to measure performance while providing tested and reliable performance indicators. For example, NVIDIA's PerfHUD tool allows you to measure total milliseconds (ms) per frame and displays the current frame rate. See the PerfHUD documentation for more information.

Here are some tips to help you ensure accurate and valid measurements:

- **Verify that the application runs cleanly.** For example, when the application runs with Microsoft's DirectX Debug runtime, it should not generate any errors or warnings.
- **Ensure that the test environment is valid.** That is, make sure you are running release versions of the application and its DLLs, as well as the release runtime of the latest version of DirectX.
- **Use release versions (not debug builds) for all software.**
- **Make sure all control panel display settings are set correctly.** Typically, this means that they are at their default values. Anisotropic filtering and antialiasing settings particularly influence performance.

- **Disable vertical sync.** This ensures that your frame rate is not limited by your monitor's refresh rate. This can be overridden in the NVIDIA control panel 3D settings. Look for "Vertical Sync" in "Manage 3D settings" section and switch to "Force OFF".
- **Run on the target hardware.** If you're trying to find out if a particular hardware configuration will perform sufficiently, make sure you're running on the correct CPU, GPU, and with the right amount of memory on the system. Bottlenecks can change significantly as you move from a low-end system to a high-end system.
- **Confirm system power/performance settings.** On some systems, especially laptops and notebooks the power settings will artificially lower the CPU performance to preserve battery power. Make sure CPU is at 100%.
- **Favor QueryPerformanceCounter() over RDTSC.** When manually timing sections of code it is important to have reliable data.
- **Benchmark/Time with averages.** Never time a single run, or a single function execution. Computers are very complex and multi-threaded, pipelined architectures. The timings of individual operations will vary wildly during execution. The best method is to execute a test a number of times and average the results.
- **Your ultimate goal is higher frames per second.** Measuring how a given change or optimization changes the fps of your application as averaged over a handful of frames (~1 second) is probably the best process to go about optimization.

2.2. Finding the Bottleneck

2.2.1. Understanding Bottlenecks

At this point, assume we have identified a situation that shows poor performance. Now we need to find the performance bottleneck. The bottleneck generally shifts depending on the content of the scene. To make things more complicated, it often shifts over the course of a single frame. So "finding the bottleneck" really means "Let's find the bottleneck that limits us the most for this scenario." *Eliminating this bottleneck achieves the largest performance boost.*

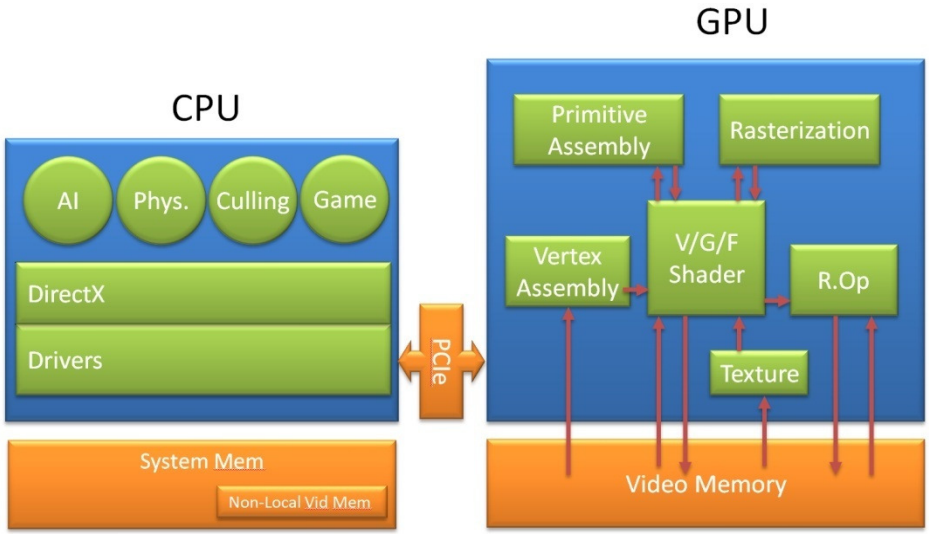


Figure 1. Potential Bottlenecks

In an ideal case, there won't be any one bottleneck—the CPU, PCIE bus, and GPU pipeline stages are all equally loaded (see Figure 1). Unfortunately, that case is impossible to achieve in real-world applications—in practice, something always holds back performance.

The bottleneck may reside on the CPU or the GPU. PerfHUD's dynamic performance dashboard (see Chapter 7 for more information about PerfHUD and other tools) shows how many milliseconds the GPU is idle during a frame. If the GPU is idle for even one millisecond per frame, it indicates that the application is at least partially CPU-limited. If the GPU is idle for a large percentage of frame time, or if it's idle for even one millisecond in all frames and the application does not synchronize CPU and GPU, then the CPU is the biggest bottleneck. Improving GPU performance simply increases GPU idle time.

Another easy way to find out if your application is CPU-limited is to ignore all draw calls with PerfHUD (removing most of the GPU load, and removing a small amount of CPU). In the Performance Dashboard, simply press Ctrl + N. Note that this also omits some CPU overhead from state changes and will not omit GPU work for stretchrect commands, etc. If you ignore draw calls and performance doesn't change, then you are very likely CPU-limited and you should use a tool like Intel's VTune or AMD's CodeAnalyst to optimize your CPU performance.

2.2.2. Basic Tests

You can perform several simple tests to identify your application's bottleneck. You don't need any special tools or drivers to try these, so they are often the easiest to start with.

- **Eliminate all file accesses.** Any hard disk access will kill your frame rate. This condition is easy enough to detect—just take a look at your computer's "hard disk in use" light or disk performance monitor signals using Windows' perfmon tool, AMD's CodeAnalyst, (http://www.amd.com/codeanalyst) or Intel's VTune (http://www.intel.com/software/products/vtune/). Keep in mind that hard disk accesses can also be caused by memory swapping, particularly if your application uses a lot of memory.
- **Run identical GPUs on CPUs with different speeds.** It's helpful to find a system BIOS that allows you to adjust (i.e., down-clock) the CPU speed, because that lets you test with just one system. If the frame rate varies proportionally depending on the CPU speed, your application is *CPU-limited*.
- **Reduce your GPU's core clock.** You can use publicly available utilities such as Coolbits (see Chapter 6) to do this. If a slower core clock proportionally reduces performance, then your application is limited by the vertex shader, rasterization, or the fragment shader (that is, *shader-limited*).
- **Reduce your GPU's memory clock.** You can use publicly available utilities such as Coolbits (see Chapter 6) to do this. If the slower memory clock affects performance, your application is limited by texture or frame buffer bandwidth (*GPU bandwidth-limited*).

Generally, changing CPU speed, GPU core clock, and GPU memory clock are easy ways to quickly determine CPU bottlenecks versus GPU bottlenecks. If underclocking the CPU by *n* percent reduces performance by *n* percent, then the application is CPU-limited. If underclocking the GPU's core and memory clocks by *n* percent reduces performance by *n* percent, then the application is GPU-limited.

2.2.3. Using PerfHUD

PerfHUD provides a vast array of debugging and profiling tools to help improve your application's performance. Here are some guidelines to get you started. The *PerfHUD User Guide* contains detailed methodology for identifying and removing bottlenecks, troubleshooting, and more. It is available at http://developer.nvidia.com/object/PerfHUD_home.html.

1. Navigate your application to the area you want to analyze.
2. If you notice any rendering issues, use the Debug Console and Frame Debugger to solve those problems first.
3. Check the Debug Console for performance warnings.
4. When you notice a performance issue, switch to Frame Profiler Mode (if you have a GeForce 6 Series or later GPU) and use the advanced profiling features to identify the bottleneck. Otherwise, use the pipeline experiments in Performance Dashboard Mode to identify the bottleneck.

2.3. Bottleneck: CPU

If an application is CPU-bound, use profiling to pinpoint what's consuming CPU time. The following modules typically use significant amounts of CPU time:

- Application (*the executable as well as related DLLs*)
- NVIDIA Driver
  - XP (*nv4disp.dll, nvoglnt.dll*)
  - Vista *(nv*.dll nvd3dum.dll etc)*
- DirectX D3D Runtime (*d3d*.dll, d3d9.dll, d3d10.dll etc*)
- DirectX Hardware Abstraction Layer (*hal32.dll*)

Because the goal at this stage is to reduce the CPU overhead so that the CPU is no longer the bottleneck, it is relatively important to know what consumes the most CPU time. The usual advice applies: choose algorithmic improvements over minor optimizations. And of course, find the biggest CPU consumers to yield the largest performance gains.

Next, we need to drill into the application code and see if it's possible to remove or reduce code modules. If the application spends large amounts of CPU in `hal32.dll`, `d3d*.dll`, or `nvoglnt.dll`, this may indicate API abuse. If the driver consumes large amounts of CPU, is it possible to reduce the number of calls made to the driver?

For example, under DirectX 9 improving batch sizes helps reduce driver calls. Detailed information about batching and DirectX 9 specific optimizations is available in http://developer.nvidia.com/object/optimizations_for_dx9.html

Additional information about DirectX 10 can be found in Chapter 5 and at http://developer.download.nvidia.com/presentations/2008/GDC/GDC08-D3DDay-Performance.pdf

PerfHUD also helps to identify driver overhead. It can display the amount of time spent in the driver per frame (plotted as a red line) and it graphs the number of batches drawn per frame.

Other areas to check when performance is CPU-bound:

- **Is the application locking resources, such as the frame buffer or textures?** Locking resources can serialize the CPU and GPU, in effect stalling the CPU until the GPU is ready to return the lock. So the CPU is actively waiting and not available to process the application code. Locking therefore causes CPU overhead.
- **Does the application use the CPU to protect the GPU?** Culling small sets of triangles creates work for the CPU and saves work on the GPU, but the GPU is already idle! Removing these CPU-side optimizations actually increase performance when CPU-bound.
- **Consider offloading CPU work to the GPU.** Can you reformulate your algorithms so that they fit into the GPU's vertex or pixel processors?
- **Use shaders to increase batch size and decrease driver overhead.** For example, you may be able to combine two materials into a single shader and draw the geometry as one batch, instead of drawing two batches each with its own shader. Shader Model 3 and 4 can be useful in a variety of situations to collapse multiple batches into one, and reduce both batch and draw overhead.

2.4. Bottleneck: GPU

GPUs are deeply pipelined architectures. If the GPU is the bottleneck, we need to find out which pipeline stage is the largest bottleneck. For an overview of the various stages of the graphics pipeline, see the GDC 2007 performance analysis presentation below.

http://developer.download.nvidia.com/presentations/2007/siggraph/perftools_sigg07.ppt

PerfHUD simplifies things by letting you force various GPU and driver features on or off. For example, it can force a mipmap LOD bias to make all textures 2 × 2. If performance improves a lot, then texture cache misses are the bottleneck. PerfHUD similarly permits control over pixel shader execution times by forcing all or part of the shaders to run in a single cycle.

PerfHUD also gives you detailed access to GPU performance counters and can automatically find your most expensive render states and draw calls, so we highly recommend that you use it if you are GPU-limited.

If you determine that the GPU is the bottleneck for your application, use the tips presented in Chapter 3, Chapter 4, and Chapter 5 to improve performance.

Chapter 3. General GPU Performance Tips

This chapter presents the top performance tips that will help you achieve optimal performance on GeForce Series GPUs. For your convenience, the tips are organized by pipeline stage. Within each subsection, the tips are roughly ordered by importance, so you know where to concentrate your efforts first.

A great place to get an overview of modern GPU pipeline performance is the *Graphics Pipeline Performance* chapter of the book *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. The chapter covers bottleneck identification as well as how to address potential performance problems in all parts of the graphics pipeline.

Graphics Pipeline Peformance is freely available at http://developer.nvidia.com/object/gpu_gems_samples.html.

3.1. List of Tips

When used correctly, recent GPUs can achieve extremely high levels of performance. This list presents an overview of available performance tips that the subsequent sections explain in more detail.

- **API Overhead Causes CPU Bottleneck**
  - Use fewer batches
    - Use texture atlases/texture arrays to avoid state changes. http://developer.nvidia.com/object/nv_texture_tools.html
    - In DirectX, use the Instancing API to avoid SetMatrix and similar instancing state changes.
    - Reduce state changes

- **Vertex Processing Cause GPU Bottleneck**
  - Use indexed primitive calls
    - Use DirectX `ID3DXMesh/ID3DX10Mesh` optimization calls [`OptimizeInplace()` or `Optimize()`]
    - Use our NVTriStrip utility if an indexed list won't work http://developer.nvidia.com/object/nvtristrip_library.html
  - Vertex Setup Attribute bottlenecks
    - Large vertices/many vertices often results in attribute bottlenecks.
    - Recalculate data in the vertex shader
    - Reduce vertex size
    - **Only use dynamic vertex buffers when really needed.**
- **Shaders Cause GPU Bottleneck**
  - Choose the highest shader model you can
  - Always use the latest version of fxc
  - Choose the lowest data precision that works for what you're doing:
    - Prefer `half` to `float` for everything that you can.
  - Pixel Shader bottleneck?
    - Push linearizable calculations to the vertex shader if you're bound by pixel shader.
    - Take advantage of Early Z culling optimizations
  - Interpolant attribute bottleneck?
    - Push vertex operations to pixel shader
    - Avoid passing constants values as interpolants. Make the same constants accessible to the shader stages that need them.
  - Geometry Shaders:
    - Make sure that you really need them and there is no better alternative, like instancing.
    - Use the smallest possible maxvertexcount value
    - Use the smallest vertices
  - Don't use uniform parameters for constants that will not change over the life of a pixel shader.
  - Look for opportunities to save computations by using algebra.
  - Replace complex functions with texture lookups

- Per-pixel specular lighting
- Use FX Composer to bake programmatically generated textures to files
- But `sincos`, `log`, `exp` are native instructions and do not need to be replaced by texture lookups

- **Texturing Causes GPU Bottleneck**
  - Always use mipmapping!
  - Use trilinear and anisotropic filtering prudently
    - Match the level of anisotropic filtering to texture complexity.
    - Use our Photoshop plug-in to vary the anisotropic filtering level and see what it looks like. http://developer.nvidia.com/object/nv_texture_tools.html
    - Follow this simple rule of thumb: If the texture is noisy, turn anisotropic filtering on.
- **Rasterization Causes GPU bottleneck**
  - Double-speed z-only and stencil rendering
  - Early-z (Z-cull) optimizations
- **Antialiasing**
  - Ensure it is only enabled when necessary

3.2. Graphics API Overhead (CPU)

3.2.1. Use Fewer Batches

"Batching" refers to grouping geometry together so many triangles can be drawn with one API call, instead of using (in the worse case) one API call per triangle. There is driver overhead whenever you make an API call, and the best way to amortize this overhead is to call the API as little as possible. In other words, reduce the total number of draw calls by drawing several thousand triangles at once. Using a smaller number of larger batches is a great way to improve performance. As GPUs become ever more powerful, effective batching becomes ever more important in order to achieve optimal rendering rates.

Investigate ways that you can reduce the number of batches and API calls in your engine.

See section 0 for some information about instancing in DirectX 10.

3.2.1.1. Texture Atlases and Texture Arrays

It is often possible to group many batches of objects using the same mesh with different textures using a texture atlas (DirectX 9 and DirectX 10) or texture array (DirectX 10 only). When combined with instancing, this can be used to easily render an entire field of trees, bushes, etc with a single batch.

3.2.2. Reduce state changes and constant changes

Group batches by render stage to avoid excessive state changes. In addition don't re-set render state if it is not necessary. If a state is set to the proper value, then it will cause overhead to set it to the same value.

Be aware that this grouping may not be desired as it might break other sorting that is required by the engine.

If you are not performing a depth pre-pass then in many cases it is better to allow some fragmentation of constants and state changes in favor of sorting front to back.

3.2.2.1. Set shader constants in groups

When setting constant data to the shaders it is more efficient to set this data in one call rather than setting many individual constants in separate calls.

3.3. Vertex Processing

3.3.1. Use Indexed Primitive Calls

Using indexed primitive calls allows the GPU to take advantage of its post-transform-and-lighting vertex cache. If it sees a vertex it's already transformed, it doesn't transform it a second time—it simply uses a cached result.

In DirectX, you can use the `ID3DXMesh/ID3DX10Mesh` class's `OptimizeInPlace()` or `Optimize()` functions to optimize meshes and make them more friendly towards the vertex cache.

You can also use our own NVTriStrip utility to create optimized cache-friendly meshes. NVTriStrip is a standalone program that is available at http://developer.nvidia.com/object/nvtristrip_library.html.

3.3.2. Attribute bottleneck (Vertex Setup)

You may want to reduce the size of your vertices to get better attribute performance. In these cases it sometimes makes sense to add more shader work to reconstruct data. See section 4.4 for more information about vertex setup attribute bottlenecks.

3.4. Shaders

High-level shading languages provide a powerful and flexible mechanism that makes writing shaders easy. Unfortunately, this means that writing *slow* shaders is easier than ever. If you're not careful, you can end up with a spontaneous explosion of slow shaders that brings your application to a halt.

GeForce 8 series and later NVIDIA GPUs make use of a unified shader architecture. This means that all shader types use the same hardware to execute instructions. The driver will balance which hardware is allocated to vertex, geometry or pixel dynamically allowing the GPU to adjust itself to more efficiently execute differing shader loads. Pre-G80 GPUs will still need to manually balance vertex and pixel workloads.

This doesn't mean you should not take care to execute the appropriate operations on the appropriate shader!

The following tips will help you avoid writing inefficient shaders for simple effects. In addition, you'll learn how to take full advantage of the GPU's computational power. Used correctly, you can achieve an instructions/ clock rate many times higher than a naïve implementation.

3.4.1. Choosing the latest shader model

With the advent of GeForce 8 series with its unified shader architecture, and more work towards better and more optimized compilers/drivers, it is no longer necessary to pick a low shader model. For all G8x and later cards you can safely pick shader model 3 for DirectX 9 and shader model 4 for DirectX

10 for all shaders and not worry about trying to fit your code into older limits. But keep in mind that it is still important to make sure your shader code is as optimal as it can be.

3.4.2. Compile with the latest compiler available.

In general, you should use the latest version of fxc (include with Microsoft's DirectX SDK), since Microsoft will add smarter compilation and fix bugs with each release. Check the latest DirectX SDK version at http://msdn.microsoft.com/directx

When compiling shaders to binary and using runtime linking ensuring that you use the latest compiler can result in significant optimizations and bug fixes.

3.4.3. Choose the Lowest Data Precision That Works

Another factor that affects both performance and quality is the precision used for operations and registers. The GeForce Series GPUs support 32-bit and 16-bit floating point formats (called `float` and `half`, respectively). The `float` data type is very IEEE-like, with an `s23e8` format. The `half` is also IEEE-like, in an `s10e5` format.

The performance of these various types varies with precision:

- If you need floating-point precision, the `half` type delivers higher performance than the `float` type. Prudent use of the `half` type can triple frame rates, and more than 99% of the rendered pixels will be within one least-significant bit (LSB) of a fully 32-bit result in most applications.
- If you need the highest possible accuracy, use the `float` type.

You can use the `/Gpp` flag to force everything in your shaders to `half` precision. After you get your shaders working and follow the tips in this section, enable this flag to see its effect on performance and quality. If no errors appear, leave this flag enabled. Otherwise, manually demote to `half` precision when it is beneficial (`/Gpp` provides an upper performance bound that you can work toward).

When you use the `half` types, make sure you use them for varying parameters, uniform parameters, variables, and constants.

Many color-based operations can be performed with the `half` data types without any loss of precision (for example, a `tex2D*diffuseColor` operation).

For instance, the result of any normalize can be half-precision, as can colors. Positions can be half-precision as well, but they may need to be scaled in the vertex shader to make the relevant values near zero.

For instance, moving values to local tangent space, and then scaling positions down can eliminate banding artifacts seen when very large positions are converted to half precision.

3.4.4. Save Computations by Using Algebra

Once you've got your shader working, look at your computations and figure out if you can collapse them by using mathematical properties. This is especially true for library functions shared across multiple shaders. For example:

- Generic sphere map projection is often expressed in terms of

  $p = \mathtt{sqrt}(Rx^2 + Ry^2 + (Rz + 1)^2)$

  This expands to :

  $p = \mathtt{sqrt}(Rx^2 + Ry^2 + Rz^2 + 2Rz + 1)$

  If you know the reflection vector is normalized (see Sections 3.4.7 and 3.4.5), the sum of the first three terms is guaranteed to be 1.0. This expression can then be refactored as:

  $p = \mathtt{sqrt}(2 * (Rz + 1)) = 1.414 * \mathtt{sqrt}(Rz + 1)$

- Fold the multiplication by 1.414 into another constant (see Section 3.4.7), saving a dot product.
- `dot(normalize(N), normalize(L))` can be computed far more efficiently.
  - It's usually computed as `(N/|N|) dot (L/|L|)`, which requires two expensive reciprocal square root (`rsq`) computations.
  - Doing a little algebra gives us:
  - `(N/|N|) dot (L/|L|)`
  - `= (N dot L) / (|N| * |L|)`
  - `= (N dot L) / (sqrt( (N dot N) * (L dot L) )`
  - `= (N dot L) * rsq( (N dot N) * (L dot L) )`
  - which requires only one expensive `rsq` operation.

3.4.5. Don't Write Overly Generic Library Functions

Functions that are shared across multiple shaders are frequently written very generically. For example, reflection is often computed as:

```
float3 reflect(float3 I, float3 N) {
       return (2.0*dot(I,N)/dot(N,N))*N – I;
}
```

Written this way, the reflection vector can be computed independent of the length of the normal or incident vectors. However, shader authors frequently want at least the normal vector normalized in order to perform lighting calculations. If this is the case, then a dot product, a reciprocal, and a scalar multiply can be removed from `reflect()`. Optimizations like these can dramatically improve performance.

3.4.6. Don't Compute the Length of Normalized Vectors

A common (and expensive) example of an overly-generic library function is one that computes the lengths of the input vectors locally. However, the vectors have often been normalized prior to calling the function. Compilers don't detect this, which means substantial per-pixel arithmetic is performed to compute 1.0.

If your library functions must work correctly independent of the vector's lengths, consider making length a scalar parameter to the functions. That way, the shaders that normalize vectors before calling the function can pass down a constant value of 1.0 (providing all the benefits of not computing the length), and those that don't normalize vectors can compute the length.

3.4.7. Fold Uniform Constant Expressions

Many developers compute expressions involving dynamic constants in their pixel shaders. If more than one uniform constant (or a uniform and an in-lined constant) is used in an expression, there is often a way to fold the constants together and improve performance. For example:

```
half4 main(float2 diffuse : TEXCOORD0,
           uniform sampler2D diffuseTex,
           uniform half4 g_OverbrightColor) {
  return tex2D(diffuseTex, diffuse) * g_OverbrightColor * 3.0;
}
```

`g_OverbrightColor` can be premultiplied by 3.0 on the CPU, saving a per-pixel multiplication on potentially millions of pixels each frame.

You may need to distribute or factor expressions in order to fold as many constant expressions as possible. In addition, you can use HLSL preshaders to perform precomputation on the CPU before a shader runs.

Another common example is computing `materialColor * lightColor` at each vertex. Because this expression has the same value for all vertices in a given batch, it should be calculated on the CPU.

You should also compute matrix inverses and transposes on the CPU instead of on the GPU, because they only need to be calculated once instead of per-vertex or per-fragment. The `/Zpr` (pack row-major) and `/Zpc` (pack column-major) compiler options can help store matrices the way you want.

3.4.8. Uniform Parameters Caveat

Don't Use Uniform Parameters for Constants That Won't Change Over the Life of a Pixel Shader

Developers sometimes use uniform parameters to pass in commonly used constants like 0, 1, and 255. This practice should be avoided. It makes it harder for compilers to distinguish between constants and shader parameters, reducing performance. When using constant values in a shader make sure to use the keyword `const` to let the compiler know the data value will never change.

3.4.9. Pixel Shader Bottlenecks?

Achieving high performance is all about removing bottlenecks—which really means that you have to balance every piece of the pipeline: the CPU, the PCIE bus, and the stages of the graphics pipeline. The GeForce 8 Series and later GPUs make use of a unifed shader system. This means that the same hardware is being used to execute vertex, geometry and pixel operations. The decision of where to execute shader operations should be dictated by where that data is needed and depends on a few factors. Attempt to execute all operations to the shader where they will require the least number of executions.

If you expect your application to be run at higher resolutions or if you're doing complex shading, the pixel shader is more likely to become the bottleneck. So, look for opportunities to move calculations to the vertex shader. You can use our ShaderPerf tool to find out how many cycles your shaders are using. Also,

note that newer hardware such as GeForce 8 and 9 Series GPUs will allow more complex pixel shaders before becoming shader-bound.

Please read the next section 3.4.10 before pushing operations to the VS as this may actually REDUCE performance.

One more thing to check when you find Pixel Shader bottlenecks is how much of the pixel shading work was wasted effort because some of the shaded pixels were later occluded by other objects. PerfHUD can help you determine this to some degree. If you conclude this is a real problem you should consider some of the following:

1. Perform a depth prepass of the scene, taking advantage of double speed Z (Section 3.6.1).
2. Ensure you are getting the most out of coarse and fine-grained Z cull optimization. See section 4.8 for more details.

   NOTE: although alpha-tested geometry (or geometry rendered with shaders that use texkil, discard or clip) will not get Double-speed Z in some cases, such us when using costly Pixel Shaders, it may be still worth to render this geometry in a depth-only prepass of the scene, since that will guarantee that no shading is wasted in the final shading pass thanks to the EarlyZ optimization.
3. If not performing a depth prepass, render opaque objects front to back.
4. Use a deferred shading implementation.

3.4.10. Interpolants and Post-transform cache pressure.

In contrast to the advice that says to push linearizable calculations form the PS to the VS, a graphics programmer needs to be aware that in G8x and later processors the power of the shader core often means that the application is NOT limited on shader operations. In this case moving operations from the PS to the VS will increase the number of attributes output from the VS and thus DECREASE the performance. This is due, primarily to two reasons:

- **Increased vertex size results in lower vertex cache performance.** A larger vertex means that less vertices will fit in the Post-transform cache and thus there is more pressure on that cache.
- **More interpolants reduces vertices/clock.** Interpolating vertex data is fixed function work that a modern GPU does. This work is not free and in some cases calculating too many interpolants will result in an attribute bottleneck. See section 4.4.

3.4.10.1. When interpolating between VS and PS don't pack

When passing data from a vertex shader to a pixel shader, the number of scalar attributes is the only thing that matters.

Packing too much information into a calculation can make it harder for the compiler to optimize your code efficiently. For example, if you are passing down a tangent matrix, do not include the view vector in the 3 q components.

This mistake is illustrated below:

```
// Bad practice
tangent = float4(tangentVec, viewVec.x)
binormal = float4(binormalVec, viewVec.y)
normal = float4(normalVec, viewVec.z)
```

Instead, place the view vector in a fourth interpolant.

Note: this is not true for vertex declarations (i.e. INPUT to a vertex shader). When creating a vertex buffer, the number of attributes AND the number of vector values are both relevant and packing is a valid optimization. See section 4.4 for more information about vertex setup and attribute boundedness.

In certain cases there is no other choice but to pack multiple attributes into a single vector attribute due to the limited number registers that can be taken as input by the pixel shader. In any other case, however, the advice applies.

3.4.11. Geometry Shaders?

Remember that geometry shaders work on primitives. This means that if you are transforming the 3 vertices for a triangle in the geometry shader then you will likely being performing a redundant transformation on the same vertex for every primitive that shares it. Only use the GS on when you really need it. (See section 3.4.14) for more information about GS performance.

3.4.12. Use the `mul()` Standard Library Function

Instead of performing matrix multiplication manually, use the `mul()` Standard Library function. This will avoid some row-major/column-major issues that may appear when applications pass down matrices in interpolants.

3.4.13. Use `D3DTADDRESS_CLAMP` (or `GL_CLAMP_TO_EDGE`)

Instead of `saturate()` for Dependent Texture Coordinates, try using `D3DTADDRESS_CLAMP` (or `GL_CLAMP_TO_EDGE`) .

Using `saturate()` can cost extra on some GPUs. If the clamped result is used as a texture coordinate, it is preferable to use the texture hardware's ability to clamp texture coordinate to the [0..1] range, rather than doing this in the shader.

3.4.14. Too many generated primitives in Geometry Shader

Geometry Shaders have the ability to output new primitives generated procedurally in the shader. Be careful to use this feature judiciously as on all current generation hardware the performance of the geometry shader is directly proportional to the number of output attributes. In general outputting the same number as input or a few more is acceptable, but 10x the number of input primitives will start to slow the shader down to the point where it will become the bottleneck. See section 4.6 for more information.

3.5. Texturing

3.5.1. Use Mipmapping

To prevent minified textures from causing "sparkling" artifacts, always use mipmapping in your applications. You'll achieve better image quality, improved texture cache behavior, and higher performance. You get all this for just 33% more memory usage, which is a great trade-off. 3D textures, in particular, can benefit greatly from mipmapping—we've seen performance increases of 30% to 40% when mipmapping was enabled.

When creating mipmaps, don't simply use a box filter to generate smaller and smaller mipmaps. Also, never use DirectX's built in mipmap generation.

Instead, use a Gaussian or Mitchell filter to take more samples—this will produce a higher quality result. But spending a little more time in the preprocess to create mipmaps, you can make your application look better continuously at runtime. Our Photoshop plug-in (part of the NVIDIA Texture Tools suite) can quickly create high-quality mipmaps for you. The suite is available at http://developer.nvidia.com/object/nv_texture_tools.html.

3.5.2. Use Trilinear and Anisotropic Filtering Prudently

Trilinear and anisotropic filtering both help to improve image quality, but they each bring a performance penalty. Try to use trilinear and anisotropic filtering only where they're needed. In general, you'll want to use them on textures that have a lot of high-contrast detail. For anisotropic filtering, you may also want to consider the orientation of the texture. If you know a texture will be oblique to the viewer (for example, a floor texture), increase the level of anisotropic filtering for that texture. For multi-textured surfaces, you should have an appropriate level of filtering for each of the different layers.

Our Adobe Photoshop plug-in is helpful for determining the level of anisotropic filtering to use. This tool allows you to try different filtering levels and see the visual effects. It is available at http://developer.nvidia.com/object/nv_texture_tools.html. Your artists may want to use this tool to help them decide which textures require anisotropic or trilinear filtering.

3.5.3. Replace Complex Functions with Texture Lookups

Textures are a great way to encode complex functions—think of them as multidimensional arrays that you can index on-the-fly. The GeForce family can access textures efficiently—often at the same cost as an arithmetic operation. You can use our FX Composer tool to prototype this kind of optimization. FX Composer is available at http://developer.nvidia.com/FXComposer.

Any time you can encode a complex sequence of arithmetic operations in a texture, you can improve performance. Keep in mind that some complex functions, such as `log` and `exp`, are micro-instructions in `ps_2_0` and higher profiles, and therefore don't need to be encoded in textures for optimal performance.

3.5.3.1. Per-Pixel Lighting

Using a 2D Texture

One common situation where a texture can be useful is in per-pixel lighting. You can use a 2D texture that you index with (`N dot L`) on one axis and (`N dot H`) on the other axis. At each (u, v) location, the texture would encode:

```
max(N dot L,0) + Ks*pow((N dot L>0) ? max(N dot H,0) : 0), n)
```

This is the standard Blinn lighting model, including clamping for the diffuse and specular terms.

Using a 1D ARGB Texture

A useful trick is to use a 1D ARGB texture, indexed by `(N dot H)`. The texture encodes `(N dot H)` to various exponents in each channel. For example, it may encode:

$((N \cdot H)^4, (N \cdot H)^8, (N \cdot H)^{12}, (N \cdot H)^{16})$

Then, each material is assigned a four-component weighting constant that blends these values, giving a monochrome specular value for shading. The beauty of this approach is that it works on GeForce 4-class hardware and is flexible enough to enable a variety of appearances.

Using a 3D Texture

You can also add the specular exponentiation to the mix by using a 3D texture. The first two axes use the 2D texture technique described in the previous section, and the third axis encodes the specular exponent (shininess).

Remember, however, that cache performance may suffer if the texture is too large. You may want to encode only the most frequently used exponents and keep the size very small.

3.5.3.2. The `sincos()` Function

Despite the preceding advice, the GeForce family GPUs support some complex mathematical functions natively in hardware. One such function that is convenient is the `sincos` function, which allows you to simultaneously calculate the sine and cosine of a value.

3.6. Rasterization

This is a list of some ideas to enhance performance that will require some work in adjusting the architecture of your graphics engine. While these require more up-front work they can often provide a much larger performance benefit than simply optimizing a shader.

3.6.1. Double-Speed Z-Only and Stencil Rendering

All GeForce Series GPUs (FX and later) render at double speed when rendering only depth or stencil values. To enable this special rendering mode, you must follow the following rules:

- Color writes are disabled
- Texkill has not been applied to any fragments (clip, discard)
- Depth replace (`oDepth`, `texm3x2depth`, `texdepth`) has not been applied to any fragments
- Alpha test is disabled
- No color key is used in any of the active textures

See section 6.4.1 for information on NULL render targets with double speed Z.

3.6.2. Z-cull Optimization

Z-cull optimization improves performance by avoiding the rendering of occluded surfaces. If the occluded surfaces have expensive shaders applied to them, z-cull can save a large amount of computation time. See section 4.8 for a discussion on Z-cull and how to best use it.

3.6.3. Lay Down Depth First ("Z-only rendering")

The best way to take advantage of the two aforementioned performance features is to "lay down depth first." By this, we mean that you should use double-speed depth rendering to draw your scene (without shading) as a first pass. This then establishes the closest surfaces to the viewer. Now you can render the scene again, but with full shading. Z-cull will automatically cull out fragments that aren't visible, meaning that you save on shading computations.

Laying down depth first requires its own render pass, but can be a performance win if many occluded surfaces have expensive shading applied to them. Double-speed rendering is less efficient as triangles get small. And, small triangles can reduce z-cull efficiency.

Another related technique is Deferred Shading, which you can find in the NVSDK

3.6.4. Allocating Memory

In order to minimize the chance of your application thrashing video memory, the best way to allocate shaders and render targets is:

5. Allocate render targets first
    - Sort the order of allocation by pitch (width * bpp).

- Sort the different pitch groups based on frequency of use. The surfaces that are rendered to most frequently should be allocated first.

6. Create vertex and pixel shaders
7. Load remaining textures

3.7. Antialiasing

All GeForce Series GPUs have powerful antialiasing engines. They perform best with antialiasing enabled, so we recommend that you enable your applications for antialiasing.

If you need to use techniques that don't work with antialiasing, contact us—we're happy to discuss the problem with you and to help you find solutions.

With DirectX 9, the `StretchRect()` call can copy the back buffer to an off-screen texture in concert with multisampling. This allows applications to make use of multi-sampling with post-processing.

With DirectX 10, you can use `ResolveSubresource()` to copy a MSAA resource to a non-MSAA resource.

For instance, if 4x multisampling is enabled, on a 100 × 100 back buffer, the driver actually internally creates a 200 × 200 back buffer and depth buffer in order to perform the antialiasing. If the application creates a 100 × 100 off-screen texture, it can `ResolveSubresource()` the entire back buffer to the off-screen surface, and the GPU will filter down the antialiased buffer into the off-screen buffer.

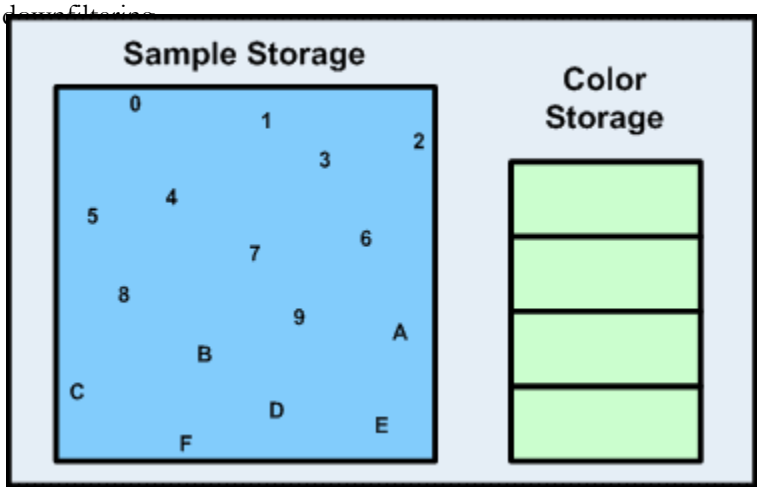
Then glows and other post-processing effects can be performed on the 100 × 100 texture, and then applied back to the main back buffer.

This resolution mismatch between the real back buffer size (200 × 200) and the application's view of it (100 × 100) is the reason why you can't attach a multisampled z buffer to a non-multisampled render target.

3.7.1. Coverage Sampled Anti-Aliasing (CSAA)

Coverage Sampling Antialiasing (CSAA) is one of the key new features of the GeForce 8 series GPUs (and later). CSAA produces antialiased images that rival the quality of 8x or 16x MSAA, while introducing only a minimal performance hit over standard (typically 4x) MSAA. It works by introducing the concept of a new sample type: a sample that represents coverage.

This differs from previous antialiasing (AA) techniques where coverage was always inherently tied to another sample type. In supersampling for example, each sample represents shaded color, stored color/z/stencil, and coverage, which essentially amounts to rendering to an oversized buffer and



MSAA reduces the shader overhead of this operation by decoupling shaded samples from stored color and coverage; this allows applications using antialiasing to operate with fewer shaded samples while maintaining the same quality color/z/stencil and coverage sampling. CSAA further optimizes this process by decoupling coverage from color/z/stencil, thus reducing bandwidth and storage costs.

See the NVSDK sample "Coverage Sampling Antialiasing" for more information.

http://developer.nvidia.com/object/coverage-sampled-aa.html

Chapter 4. GeForce 8 and 9 Series Programming Tips

This chapter presents several useful tips that help you fully use the capabilities of GeForce 8 & 9 Series and GTX 2xx Series as well as G8x based(and later) Quadro GPUs. These are mostly feature oriented, though some may affect performance as well. This chapter focuses on Microsoft's DirectX API. However, many concepts can be directly translated to OpenGL as well.

4.1. Introduction to GeForce 8/9 series architecture

The GeForce 8 Series GPU revolutionize graphics processing by being the first DirectX 10 supported GPU. It features a unified shader architecture which allows for dynamic load balancing of shader workloads. The GeForce 9 series follows the same architecture as the 8 series with some minor enhancements. The 9800GTX features 128 unified shader cores @ 1688Mhz for unrivaled single GPU performance.

4.2. Shader Model 4.0

Shader Model 4 is only available in DirectX 10 or OpenGL extensions. With shader model 4 there is no longer an intermediary shader assembly format. HLSL is compiled directly to shader binary and a graphics programmer need not worry about attempting to optimize or write shader asm code. In addition shader model 4 has been designed with a "common shader core", meaning that all general operations are available in all shader types. This closely models the unified architectures of modern GPUs.

With respect to features, SM4 is a super set of shader model 3, including everything available in previous shader models (except some specific shader model 1 one-off features). The following table illustrates some new features in SM4 above SM3.

See Chapter 5 for more information about DirectX 10 specific issues.

Pixel Shader Feature	Shader 3.0	Shader 4.0	Description
Shader length	65535+	Unlimited	Allows more complex shading, lighting, and procedural materials
Sample from MSAA surface?	No	Access to individual samples.	Developers can control how multiple MSAA samples are combined and used for custom effects, and general purpose calculations.
Constants	Register file	Constant buffers and Texture buffers	Allows more general use of shaders without the input/output register restrictions of previous models.
System value semantics	No	Yes	Provide convenient access to vertex, primitive and pixel IDs. Useful for buffer indexing.

4.3. Shader Model 4 System Values (SV_)

There are a handful of new system values that are available to all shader code via the SV_* semantic. Simply define a shader constant variable and assign it the desired semantic and you will have access to that data. Below is a table of some of the available constants. For the complete list please refer to Microsoft's DirectX 10 documentation.

Semantic	*Type*	*Value*	*Notes*
SV_InstanceID	uint	Unique Id for each instance in a draw call.	Monotonically increasing. Reset to 0 each time.
SV_VertexID	uint	Unique Id for each vertex in a draw call.	Monotonically increasing. Reset to 0 each time. For indexed draw calls, SV_VertexID is equal to the index from the IB.
SV_PrimitiveID	uint	Unique Id for each pixel in a draw call.	Monotonically increasing. Reset to 0 each time.

4.3.1. System Values Performance Tips

System values can be used to reduce bottlenecks. Below are a few performance notes and caveats.

4.3.1.1. Reduce vertex size with SV_InstanceID during instancing

A good use of SV_InstanceID is to encode per instance data into a buffer and use the Load() call in HLSL to read data using SV_InstanceID to offset. Thus you are able to significantly shrink the size of your vertices. This can help to avoid the primitive assembly overhead that generally occurs when using instancing.

Be aware that using the SV_ semantics do add a fixed attribute overhead or 8 scalar attributes to the shader using it. So if you are attribute bound and trying to use a system value semantic to reduce your attribute count make sure you can reduce it more than 8 attributes or it will not make much of a difference.

4.3.1.2. Take care when using SV_PrimitiveID to sample textures

When using SV_PrimitiveID to index into a texture for sampling you may encounter poor performance as the order may induce texture cache thrashing.

4.4. Vertex Setup/Attribute bottleneck issues

Given the large shader operation processing power of the GeForce 8, 9 and GTX2xx series cards, the bottlenecks have shifted from previous generations of cards. In many cases we encounter the case where the application is unable to feed the shader processing cores with enough operations to keep them busy all the time. This situation results in lost performance and a lower frame rate than might be otherwise possible. One reason for this could be becoming attribute bound.

4.4.1. Vertex assembly on a GPU

When rendering a mesh, the application binds one or more vertex buffers with a custom vertex declaration. Before the vertex shader can operate on a vertex, that vertex needs to be assembled into a single data chunk. This is also called "setup". During setup, each float of vertex data is fetched from the appropriate location in video memory based on the vertex stream and offset information. Each float of vertex data is called an "attribute".

4.4.2. Attribute bottleneck

As the size of a vertex used by the vertex shader increases, the number of attributes that need to be fetched and assembled increases. On the GeForce 8, 9 and GTX2xx series cards there is a fixed number of attributes that can be fetched per clock cycle. Thus, you can imagine that if a vertex becomes extremely large, the vertex setup stage of the graphics pipeline will quickly

become the bottleneck and slow down rendering by starving the rest of the GPU pipeline.

4.4.3. Detecting Attribute bottlenecks

It is fairly easy to determine if you are attribute bound. There are generally two methods.

4.4.3.1. Increase vertex size and test for slowdown

This is the easiest method, as it does not require that you significantly rewrite your shader code. This would be to simply add in some dummy data to your vertex declaration and see if the performance suffers. If it goes down then you are either attribute bound, or very close to being attribute bound and should consider ways to reduce your vertex size.

4.4.3.2. Decrease vertex size and test for speedup

This is the more intuitive method as it will mirror your attempts to gain performance by reducing vertex size. However, you should be careful and try to keep shader logic, because the changes can result in code simplification and make the experiment invalid if you are shader bound.

You can use the following simple steps for checking:

1) Replace attribute reference from the shader body by a new variable:
   floatN TestAttribute = IN.TestAttribute;
2) Replace attribute by a variable with predictable behaviour
   TestAttribute = (floatN(IN.UsedAttribute) + TestAttribute) * eps + Const
   eps – small value, used to minimize the influence of attributes used
3) If the performance hasn't change much, we can remove TestAttribute from the code:
   TestAttribute = floatN(IN.UsedAttribute) * eps + Const;

If you can see a speedup by moving from step 2 to step 3, this is a good indicator that you are attribute bound, if not, you may try to perform the above steps for remaining attributes.

4.4.4. Fixing Attribute bottlenecks

All you have to do is reduce the vertex size and make sure your attributes are packed well.

4.4.4.1. Not only total number of attributes matters

The important metric is not only the total number of scalar attributes, but a number of vector attributes used as well. For example, the following have the same number of scalar attributes, but may not result in the same performance on GeForce 8 series or later cards.

```
float4 myData;
```

and

```
float3 myDataOne;
float1 myDataTwo;
```

Full attributes are better for the vertex declaration.

4.4.4.2. Suggestions

1. First check for and remove unused attributes

This is the easiest and least likely to be found. Sometimes a programmer will allocate a float4 while developing and never use the data in the w component. This is an extra attribute and will possibly increase the setup time. You can easily check the assembly for these (unused attributes are removed from the input signature) and use that space for some other valuable attributes.

2. Try to perform logical grouping of your attributes to reduce the total number of vector attributes used

Logical grouping means combining a number of separate attributes into a single attribute, up to float4, all components of which will be used in the same passes.

For example, if you are using a pair of texture coordinates, it is better to pack them into a single float4, than using two separate float2 attributes. Almost always vertex position requires just a single float3 value, if you can logically combine it with a separate float value used, do it.

3. Recalculate attributes

You do not need to pass down all the data you will be using. In many cases there are dependencies in the data such that you can pass down a subset and recalculate the rest. For example, you do not need to pass a normal, binormal and tangent to the vertex shader. Passing down the normal and tangent alone and recalculating the binormal using a cross product will result in more shader operations but a smaller vertex size. Depending on your bottleneck this could be a significant speedup.

4. Use vertex shader load()

Given that VTF is blazingly fast on GeForce 8 and later cards, using the system value SV_VertexID to create texture coordinates and fetch vertex data from a texture in video memory is now a viable option for those who are encountering an attribute bottleneck. Note, this is not always faster, so care should be taken to test the effectiveness of this optimization. In particular, adding SV_VertexID or SV_InstanceID will result in additional overhead, so make sure you are able to remove at least a pair of vector attributes used, the more the better.

4.5. Vertex Texture Fetch

From GeForce series 8 and later, NVIDIA graphics chips are now using a unified shader architecture. This means that shader code for vertex, geometry and pixel shaders is executed on the same hardware. This also means that vertex and geometry shader texture fetch uses the same hardware as pixel, and thus benefits from the same caching and speed.

Thus it is now possible to make use of a large texture or buffer of constant data and load values from that resource in the shader. This is often required for processing that access into a constant data set that cannot fit into traditional constant memory. For example, NVIDIA SDK sample "Skinned Instancing" uses VTF to fetch animation bone matrices from a texture that contains all frames of all animations for a character. This allows thousands of independently animated character meshes to be rendered in just a hand full of draw calls.

4.6. Geometry Shader

Shader Model 4 and DirectX 10 level GPUs have added a brand new shader stage. This stage is called the Geometry Shader. It is executed just after the vertex shader and allows primitive level operations. It inputs all vertices for a primitive and potentially neighboring primitives. It outputs 0 or more primitives to be rasterized (or written to memory with stream out, see section 4.7).

The geometry shader also allows creation (or destruction) of small amounts of primitives. Due to the requirement that the output of a Geometry Shader

thread must preserve the order in which it is generated, the GPU must support buffering the output data in this correct order for a number of threads running in parallel.

GeForce 8, 9 and GTX2xx series have limited output buffer sizes, which is at least sufficient to support the maximum output size allowed in Direct3D 10 (1024 32-bit scalars). The output buffer size may vary by GPU.

The performance of a GS is inversely proportional to the output size (in scalars) declared in the Geometry Shader, which is the product of the vertex size and the number of vertices (maxvertexcount). This performance degradation however occurs at particular output sizes, and is not smooth.

For example, on a GeForce 8800 GTX a GS that outputs at most 1 to 20 scalars in total would run at peak performance, whereas a GS would run at 50% of peak performance if the total maximum output was declared to be between 27 and 40 scalars.

More concretely, if your vertex declaration is:

```
float3 pos: POSITION;

float2 tex: TEXCOORD;
```

Each vertex is 5 scalar attributes in size.

If the GS defines a maxvertexcount of 4 vertices, then it will run at full speed (20 scalar attributes).

But if you increase the vertex size by adding a float3 normal, the number of scalars will increase by 4 * 3, putting your total at 32 scalar attributes. This will result in the GS running at 50% peak performance.

Thus, it is important to understand that the main use of the geometry shader is **NOT for doing heavy output algorithms such as tessellation.**

In addition, because a GS runs on primitives, per-vertex operations will be duplicated for all primitives that share a vertex. This is potentially a waste of processing power.

A geometry shader is most useful when doing operations on small vertices or primitive data that requires outputting only small amounts of new data. But in general, the potential for wasted work and performance penalties for using a GS makes it an often unused feature of Shader model 4.

4.6.1. GS Performance Bottleneck ("maxvertexcount")

Keep maxvertexcount as low as possible!

In addition, one thing to be aware of when outputting many attributes from a Geometry Shader program is that if you are outputting a dynamically variable number of attributes for each primitive then ALL primitives executing the geometry shader will run at the worst case of the geometry shader output.

That is, when declaring the geometry shader you must define the maximum number of vertices that the shader will output. It is THIS declaration that determines the speed with which the GS runs.

4.6.2. A decent use of Geometry Shaders: Point Sprites

In contrast to all the performance hazards of using the GS, one case generally will run very well, and be simple to implement. This case is point sprites. Given that the point sprite fixed function capability has been removed in DirectX 10, you can now simply generate a primitive from a single input vertex. This has the benefit of generally reducing vertex setup attribute boundedness for that batch, as well the generated vertices will generally be small in size and thus the performance of the GS will stay fast. But be mindful of the number of output scalar attributes. You can still easily hit the performance cliff if you make your vertices large. See the previous example.

4.7. Stream out

Stream out is a new feature available in DirectX 10 and OpenGL for G80 and later GPUs. With stream out the programmer is able to bypass the rasterization and later stages of the graphics pipeline and write the output of the geometry/vertex shader directly into video memory. This is a versatile and useful tool to enable custom processing without the overhead of the pixel pipeline. Some examples below.

4.7.1. Skinned Characters Optimization

With stream out a game engine can pose/skin a matrix palette skinned character and save the posed mesh in model space out as a collection of vertices in video memory. Then in subsequent passes, rather than skinning the character each time it is rendered the pre-skinned mesh is used. This is an optimization applied to animated characters. In most game engines a character will be

rendered at least twice. Once for the shadow map and once for the main In some deferred rendering engines, or if there are more than one shadow casting light the number of times rendered may be even more. By pre-computing (each frame) the animated character the engine can save the cost of re-evaluating the animation palette and at least a matrix multiply per vertex.

A sample render flow is below.

1. Render animated character mesh using simple vertex shader
2. Using stream out bypass rasterization and save posed mesh to memory
3. Render shadow map generation sourcing posed mesh
   - Vertex shader does not do skinning operations
4. Render main render sourcing posed mesh
   - Vertex shader does not do skinning operations

4.7.2. Blending Morph Targets

By using stream out an engine can support effectively infinite morph targets. Simply define to buffers one as input and one as output. Starting with the initial mesh:

1. Evaluate the input mesh using the morph target
2. Write the morphed mesh using stream out
3. Swap input and output buffers and change morph operation.
4. Repeat for 1-3 for as many morph targets as you desire.

4.8. ZCULL and EarlyZ: Coarse and Fine-grained Z and Stencil Culling

NVIDIA GeForce 6 series and later GPUs can perform a coarse level Z and Stencil culling. Thanks to this optimization large blocks of pixels will not be scheduled for pixel shading if they are determined to be definitely occluded.

In addition, GeForce 8 series and later GPUs can also perform fine-grained Z and Stencil culling, which allow the GPU to skip the shading of occluded pixels.

These hardware optimizations are automatically enabled when possible, so they are mostly transparent to developers. However, it is good to know when they cannot be enabled or when they can underperform to ensure that you are taking advantage of them.

Coarse Z/Stencil culling (also known as ZCULL) will not be able to cull any pixels in the following cases:

1. If you don't use Clears (instead of fullscreen quads that write depth) to clear the depth-stencil buffer.
2. If the pixel shader writes depth.
3. If you change the direction of the depth test while writing depth. ZCULL will not cull any pixels until the next depth buffer Clear.
4. If stencil writes are enabled while doing stencil testing (no stencil culling)
5. On GeForce 8 series, if the DepthStencilView has Texture2D[MS]Array dimension

Also note that ZCULL will perform less efficiently in the following circumstances

1. If the depth buffer was written using a different depth test direction than that used for testing
2. If the depth of the scene contains a lot of high frequency information (i.e.: the depth varies a lot within a few pixels)
3. If you allocate too many large depth buffers.
4. If using DXGI_FORMAT_D32_FLOAT format

Similarly, fine-grained Z/Stencil culling (also known as EarlyZ) is disabled in the following cases:

1. If the pixel shader outputs depth
2. If the pixel shader uses the .z component of an input attribute with the SV_Position semantic (only on GeForce 8 series in D3D10)
3. If Depth or Stencil writes are enabled, or Occlusion Queries are enabled, and one of the following is true:
   - Alpha-test is enabled
   - Pixel Shader kills pixels (clip(), texkil, discard)
   - Alpha To Coverage is enabled
   - SampleMask is not 0xFFFFFFFF (SampleMask is set in D3D10 using OMSetBlendState and in D3D9 setting the D3DRS_MULTISAMPLEMASK renderstate)

Chapter 5. DirectX 10 Considerations

With the introduction of DirectX 10, we have seen a significant change in the performance characteristics of 3D applications. It is not necessarily the case that simply converting your DirectX 9 application to DirectX 10 will result in a speedup. In fact, in many cases, a slowdown is perceived. This is often due to a naïve porting effort. It is essential to understand the important differences between DirectX 9 and 10 in order to properly effect a codebase transition.

In DirectX10 Microsoft has moved validation of api call parameters from runtime to creation time. This means that, generally, runtime calls to DirectX10 have less CPU overhead than DirectX9. This translates to an increase in the number of draw calls you can issue per frame and still maintain a high frame rate.

However, naively porting your DirectX9 code to DirectX10 will induce slowdowns. In particular, the most common source of a performance slowdown when porting is due to improper use of shader constants and constant buffers. In addition are to keep focus on when porting would be resource updates. Refer to the specific subsections in this chapter for information about these concepts.

In addition, you can gain additional performance benefit by using built in the new features in DirectX10 such as instancing, texture arrays, stream out, etc.

Refer to Microsoft's "Direct3D 9 to Direct3D 10 considerations" for more information.

5.1. DirectX 10 States and Constants

A major cause of poor performance in naïve DirectX 10 ports!

5.1.1. Immutable State blocks

A large change in DirectX10 is the use of render state blocks. Previously, each render state was separately set, and those commands were injected into the command buffer. In DirectX10 there are 5 groups of render state that are set together and bound into an immutable state block object. You create these state blocks which are unique based on a hash of all the values in them.

The important thing to remember about state blocks is that creating state blocks takes time. A game should not dynamically create state blocks.

Always create your state blocks at load time.

One recommendation that has not changed is to sort draw calls by render states.

5.1.2. Constant blocks

Constants are declared in buffers in DirectX 10. These buffers are atomically updated. This means, that if any data in a constant buffer changes, the entire buffer needs to be re-uploaded. This can be a significant bandwidth hit if your application has large constant buffers.

5.1.2.1. Group and sort constants by update frequency

The recommendation is to separate shader constants into buffers based on update frequency. And example would be per-frame, per-mesh, per-draw and global constant buffers. A more simple illustration of this is below.

```
cbuffer PerFrameConstants
{
    float4x4 mView;
    float    fTime;
    float3   fWindForce;
    // etc.
};
```

All constants in *PerFrameCosntants* changes only per frame, they are constant between meshes and draw calls, and thus the buffer only needs to be uploaded once per frame.

```
cbuffer SkinningMatricesConstants
{
    float4x4 mSkin[64];
};
```

In contrast to *SkinningMatricesConstants* which is constant per mesh, and will only need to be re-uploaded when the mesh changes. Since many meshes have more than one draw call and a matrix pallete for animation is very large, separating this into its own buffer will often save a lot of bandwidth.

In general the graphics engineer will need to strike a good balance between:

- Amount of constant data to upload
- Number calls required to do it (== # of cbuffers)
- Don't go overboard (3-5 constant buffers is enough)

Another key concept to remember is that it is entirely possible to share constant buffers between pixel and vertex shaders in the same call.

Example:

- PerFrameGlobal (time, per-light properties)
- PerView (main camera xforms, shadowmap xforms)
- PerObjectStatic (world matrix, static light indices)
- PerObjectDynamic (skinning matrices, dynamic lightIDs)

5.1.2.2. Constant grouping within a block

Another decision to make is the ordering of individual constants withing a state block. Given that constant blocks can be thought of a buffers that are loaded from, you can imagine that constant data needs to be cached as it is read from video memory. Thus two constant reads that are spatially local will be faster due to higher percentage of hits in the cache.

Thus the recommendation is to sort constants in a block by access pattern.

For example, given the block of shader code below, there are a number of ways to order the constants in the constant block.

```
float4 PS_main(PSInput in)
```

```
{
  float4 diffuse = tex2D0.Sample(mipmapSampler,
in.Tex0);
  float ndotl = dot(in.Normal, vLightVector.xyz);
  return ndotl * vLightColor * diffuse;
}
```

```
// BAD ordering
cbuffer PerFrameConstants
{
   float4   vLightVector;
   float4   vOtherStuff[32];
   float4   vLightColor;
};
```

```
// GOOD ordering
cbuffer PerFrameConstants
{
   float4   vLightVector;
   float4   vLightColor;
   float4   vOtherStuff[32];
};
```

The above GOOD and BAD orderings illustrate what you should do when ordering constants in a block. Due to the fact that vLightVector and vLightColor are accessed in order, placing a large block of unused data in between then inside a constant block will result in a cache miss when loading vLightColor. Placing them together in the block will result in a higher likelihood that the cach line loaded when accessing vLightVector will include vLightColor.

5.1.3. Don't use global constants!

If you place all of your constant variables outside of an explicit block they will automatically be compiled into a cbuffer called $Globals.

The $Globals buffer typically has poor performance due to:

- Wasted CPU cycles (and bandwidth) updating unused constants
- Cbuffer contention
- Poor cache reuse due to improper ordering of constants.

You can check to see if a constant is being used with D3D10_SHADER_VARIABLE_DESC.uFlags which is part of the DirectX 10 reflection API.

5.1.3.1. D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY

When compiling SM3 shaders to SM4+ and using the D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY flag, you can group constants into constant block with a preprocessor conditional to declare cbuffers. This allows you to share shaders between DirectX 9 and DirectX 10 without sacrificing constant buffer performance.

For example:

```
#ifdef DX10
cbuffer MyBuffer {
#endif
```

5.1.4. When to use a tbuffer?

A tbuffer uses texture memory to store constant data. As a result is follows the more traditional texture sample path when loading that data. It is appropriate to use in situations where the data in the buffer is very large, and you are randomly (non-sequentially) accessing it. It can also be used if you are unable to fit your data in the constant buffer size limit.

Type	Max Size
cbuffer	4096*float4(16 bytes)
tbuffer	128 MiB

tbuffers have larger latency for data loading, but if you can mask that latency with shader operations then they will be as fast as cbuffers.

5.2. Resource Management

5.2.1. Resource creation and destruction

Resource creation and destruction calls have significant overhead. They allocate memory, validate the call, and include various driver checks. This results in time to create resources.

If possible keep the number of resource creations and destructions to a minimum. Attempt to create all resource that you will use either up front (i.e during application or level load), or save them for non-performance critical times (i.e cut-scenes). At runtime when a resource changes, rather than delete and re-create, attempt to reuse an existing resource and simply update the data.

5.2.2. Updating resources

There are a few recommendations for updating resource data with Direct 10.

5.2.2.1. Textures (UpdateSubresource is bad!)

When updating textures avoid the UpdateSubresource() call. This call is not performant for larger data resources and is very bad for really large textures.

We recommend using a ring buffer of intermediate staging textures and then copying them to video memory resources using the following method.

1. D3D10_USAGE_STAGING textures.
2. Map(D3D10_MAP_WRITE,…) with flag
   - D3D10_MAP_FLAG_DO_NOT_WAIT to avoid stalls
3. Copy textures to video memory (D3D10_USAGE_DEFAULT)
   - CopyResource() or CopySubresourceRegion()

5.2.2.2. Buffers

When updating buffers in general you have two options.

1. Map(D3D10_MAP_WRITE_DISCARD,…);
2. UpdateSubResource();

In general, for buffers either one works well. With Map the CPU can avoid updating data in the buffer that the shader doesn't care about. However, the entire buffer will need to be copied to the GPU after being updated.

For very large dynamic Vertex/Index buffers

This is a special case for very large VB/IBs. The recommendation in this case is to use a large shared ring buffer type construct. This would be essentially placing the separate buffers back to back in a large buffer and adjusting an offset. Then,

- Map(D3D10_MAP_DISCARD,…) when buffer is full or the first time initializing it.

- Map(D3D10_MAP_No_OVERWRITE,…) when updating the next buffer in the ring.
- Avoid Updatesubresource() as it is very bad in this case.

5.3. Alpha Test in DirectX 10

In DirectX 10, Microsoft has removed the fixed function alpha test renderstates. To perform alpha test operations the only way is to use the discard() or clip() functions in your pixel shader.

One performance mistake that is common when implementing alpha test in DirectX 10 is to create a shader that handles both alpha testing and non-alpha testing objects.

The following is an example of this **PERFORMANCE MISTAKE:**

```
if ( bAlphaTestOn  && alpha < threshold) {
  discard;
}
```

When a shader potentially uses the discard() function the hardware is unable to rely on the information in the Z buffer and thus is forced to disable early Z functionality. So by combining alpha test and non alpha test operations together into a single shader, an application will disable early for all rendered objects including opaque objects.

The recommendation is to create two shaders. One shader is for opaque objects that do not require alpha test, and one is for transparent objects that will be using alpha test. By doing this you can ensure optimal z culling and performance. Please refer to section 4.8 for more information about Z culling.

5.4. Batching and Instancing

As mentioned above, in DirectX 10 runtime API call validation has been moved to creation time, resulting in a reduction of call overhead. In DirectX 9 this was seen most prominently in draw call overhead. Many applications would become CPU bound due to this validation when issuing many draw calls per frame. DirectX 10 has improved this situation. However, runtime api overhead still exists, and can be reduced by using geometry instancing.

In DirectX10 instancing support has been build into the API directly. In addition to being able to specify the instance count as a Draw*() parameter, there are explicit specifications in the vertex declaration to support per instance data.

Because of this the concept of drawing everything in a scene in a single draw call has changed to:

"Be as efficient as possible in reducing API calls, including state changes, and constant changes. Not just draw calls."

Instancing still plays a large role in enabling this.

For example, it is still valid to group all blades of grass in a field into a single call. Instancing can allow those grass blades to be independently perturbed by winds or other interactions.

Check out NVIDIA SDK 10.5 sample "Instancing Tests" for a test application where you can evaluate how instancing can help your content. It allows for loading a custom mesh and includes source code so you can easily evaluate different rendering techniques.

http://developer.download.nvidia.com/SDK/10.5/Samples/InstancingTests.zip

Chapter 6.General Advice

This chapter covers general advice about programming GPUs that can be leveraged across multiple GPU families.

6.1. Identifying GPUs

In the past, developers often queried a GPU's device ID (through Windows) to find out what GPU they were running on. The device IDs have historically been monotonically increasing. However, with the GeForce 6 & 7 Series (and later) GPUs, this is no longer the case. Therefore, we recommend that you rely on caps bits (in DirectX) or the extensions string (in OpenGL) to establish the features of the GPU you're running on. If you're using OpenGL's renderer string, don't forget that NV40-based chips do not all have an "FX" moniker in their name (they are named "GeForce 6xxx" or "Quadro FX x400"). Similarly, G70-based chips are named "GeForce 7xxx".

Device IDs are often used by developers to try to reduce support calls. If you mishandle Device IDs, you will instead **create support calls**. Often, when we create a new GPU, many applications will not recognize it and fail to run.

One key idea that cannot be stressed enough is that **not recognizing a Device ID does not give you any information.** Do not take any drastic action just because you do not recognize a Device ID.

The only reasonable use of Device ID is to take action when you **recognize** the ID, and you know there is a special capability or issue you wish to address.

Some games are failing to run on GeForce 8 & 9 SeriesGPUs because they mis-identify the GPU as a TNT-class GPU, or don't recognize the Device ID. This behavior creates a support nightmare, as the G80 generation of chips is the most capable ever, and yet some games won't run due to poor coding practices.

Device IDs are also **not a substitute** for caps (DirectX 9) and extension strings(OpenGL). Caps have and do change over time, due to various reasons. Mostly, caps will be turned on over time, but caps also get turned off, due to specs being tightened up and clarified, or simply the difficulty or cost of maintaining certain driver or hardware capabilities.

One benefit of using DirectX 10 is that Microsoft has removed the concept of caps bits and a GPU that supports DirectX 10 must support all features of DirectX 10.

Render target and texture formats also have been turned off from time to time, so be sure to check for support.

For an up to date list of Device IDs refer to
http://developer.nvidia.com/object/device_ids.html

If you are having problems with Device IDs, please contact our Developer Relations group at devrelfeedback@nvidia.com.

The current list of Device IDs for all NVIDIA GPUs is here:
http://developer.nvidia.com/object/device_ids.html.

6.2. Hardware Shadow Maps

NVIDIA hardware from the GeForce 3 GPU and up supports hardware shadow mapping in OpenGL and DirectX. "Hardware shadow mapping" means that we have dedicated special transistors specifically for performing the shadow map depth comparison and percentage-closer filtering operations. We recommend that you take advantage of this feature, as it produces higher quality filtered shadow map edges very efficiently. Because dedicated transistors exist for hardware shadow mapping, you will lose performance and quality if you try to emulate our shadow mapping algorithm with ps_2_0 of higher.

For a lot more information about shadowing and the many different techniques available check out the following link.

http://developer.nvidia.com/object/doc_shadows.html

There are many techniques available. However, the general recommendation is that unless you know what you are doing you should just implement simple multi-tap cascaded shadow maps. In general, 3 levels are sufficient to provide good shadow detail for any scene.

6.3. Depth Bounds Test (DBT)

Depth Bounds Test and is available on all GPUs from GeForce 6 Series and later.

Depth Bounds Test (DBT) allows the programmer to enable an additional criterion to allow discarding of a pixel before blending to the rendertarget.

The extension adds a new per-fragment test that is, logically, after the scissor test and before the alpha test. The depth bounds test compares the depth value stored at the location given by the incoming fragment's (xw,yw) coordinates to a user-defined minimum and maximum depth value. If the stored depth value is outside the user-defined range (exclusive), the incoming fragment is discarded. Unlike the depth test, the depth bounds test has NO dependency on the fragment's window-space depth value.

6.3.1. Important Notes

- It is important to note that DBT **does not depend on the depth of the pixel out from the pixel shader** only the previously stored render target depth position.
- The DBT comes logically after the scissor test and before alpha testing.
- The min/max values are clamped to [0..1]. NANs are converted to 0.
- The depth bound test is enabled if all the following is true:
  - min < max (after clamping)
  - min > 0 or max < 1 (after clamping)
  - A depth buffer is in use

6.3.2. API Usage

To check for DBT support:

```
if(dev->CheckDeviceFormat(Adapter, D3DDEVTYPE_HAL, AdapterFormat, 0,
D3DRTYPE_SURFACE, (D3DFORMAT MAKEFOURCC('N','V','D','B') ) == S_OK)

{

MessageBox(NULL, _T("Device/driver does not support depth bounds
test!"), _T("ERROR"), MB_OK|MB_SETFOREGROUND|MB_TOPMOST);

   return E_FAIL;

}
```

Enabling DBT under DirectX 9 is as follows:

```
float zMin,zMax;
zMin = .25f;
zMax = .75f;
dev->SetRenderState(D3DRS_ADAPTIVETESS_X,MAKEFOURCC('N','V','D','B'));
dev->SetRenderState(D3DRS_ADAPTIVETESS_Z,*(DWORD*)&zMin);
dev->SetRenderState(D3DRS_ADAPTIVETESS_W,*(DWORD*)&zMax);
```

To disable it, simply change the fourCC code for the adaptiveness.

```
dev->SetRenderState(D3DRS_ADAPTIVETESS_X,0);
```

6.3.3. What is DBT good for?

Depth Bounds Test can be used in any case where you wish to manually restrict pixels being written to your render target based on a stored depth value. Because the depth of the actual pixel as output from the shader does not get evaluated, it is sometimes tricky to understand the usefulness.

Below is an example taken from the OpenGL extension document on depth bounds test:

"This functionality is useful in the context of attenuated stenciled shadow volume rendering. To motivate the functionality's utility in this context, we first describe how conventional scissor testing can be used to optimize shadow volume rendering.

If an attenuated light source's illumination can be bounded to a rectangle in XY window-space, the conventional scissor test can be used to discard shadow volume fragments that are guaranteed to be outside the light source's window-space XY rectangle. The stencil increments and decrements that would otherwise be generated by these scissored fragments are inconsequential because the light source's illumination can pre-determined to be fully attenuated outside the scissored region. In other words, the scissor test can be used to discard shadow volume fragments rendered outside the scissor, thereby improving performance, without affecting the ultimate illumination of these pixels with respect to the attenuated light source. This scissoring optimization can be used both when rendering the stenciled shadow volumes to update stencil (incrementing and decrementing the stencil buffer) AND when adding

the illumination contribution of attenuated light source's. In a similar fashion, we can compute the attenuated light source's window-space Z bounds (zmin,zmax) of consequential illumination. Unless a depth value (in the depth buffer) at a pixel is within the range [zmin,zmax], the light source's illumination can be pre-determined to be inconsequential for the pixel. Said another way, the pixel being illuminated is either far enough in front of or behind the attenuated light source so that the light source's illumination for the pixel is fully attenuated. The depth bounds test can perform this test."

Another example is deferred renderers. Given that lighting is rendered by first converting the light to screen space, can make use of DBT to cull pixels in areas of the screen where the light depth (previously written) will not be visible.

6.4. FOURCC Codes

FourCC codes are special codes that give developers access to certain custom formats that are specific to an IHV. NVIDIA provides a number of extremely useful FourCC codes to enable maximum performance and efficiency on the latest GPUs.

6.4.1. NULL Rendertarget ("NULL")

Supported GPUs: GeForce 6 Series and Later

When rendering a shadow map occlusion pass using hardware shadow maps it is necessary to bind a color buffer. This is a DirectX API restriction. Another restriction is that the color and depth buffers be the same size. Because rendering a shadow map when using hardware shadow maps does not actually use the color buffer this results often in the need to create an unused color buffer to bind to the pipeline. In most cases the shadow map resolution doesn't match a previously allocated color buffer. This is a waste of precious video memory.

This is the reason to use a NULL rendertarget. A NULL rendertarget advertises any resolution that you want but does not actually allocate any memory and thus takes up no space in video memory.

6.4.1.1. Usage

Below is a code example showing how to check for hardware shadow map support and create an appropriate NULL render target.

```
m_zFormat     = D3DFMT_D24X8;

m_colorFormat = D3DFMT_A8R8G8B8;

m_nullFormat  = (D3DFORMAT)MAKEFOURCC('N','U','L','L');

// Check for hardware shadowmap support

if(FAILED(CheckResourceFormatSupport(m_pd3dDevice, m_zFormat,
D3DRTYPE_TEXTURE, D3DUSAGE_DEPTHSTENCIL)))

{

  MessageBox(NULL, _T("Device/driver does not support hardware shadow
maps!"), _T("ERROR"), MB_OK|MB_SETFOREGROUND|MB_TOPMOST);

  return E_FAIL;

}

 // Check for NULL render target support

if(FAILED(CheckResourceFormatSupport(m_pd3dDevice, m_nullFormat,
D3DRTYPE_SURFACE, D3DUSAGE_RENDERTARGET)))

{

  MessageBox(NULL, _T("Device/driver does not support hardware shadow
maps with NULL colorbuffers!"), _T("ERROR"),
MB_OK|MB_SETFOREGROUND|MB_TOPMOST);

  return E_FAIL;

}

if(FAILED(m_pd3dDevice->CreateRenderTarget ( TEXDEPTH_WIDTH,
TEXDEPTH_HEIGHT, m_nullFormat, D3DMULTISAMPLE_NONE, (DWORD)0, FALSE,
&m_pSMColorSurface, NULL)))

   return E_FAIL;

m_pSMColorTexture = NULL;
```

6.4.2. Direct DepthBuffer Access ("INTZ" and "RAWZ")

Supported GPUs: GeForce 6 Series and later (INTZ requires 8 Series)

There are two fourcc codes that allow direct depth buffer access. The first "RAWZ" gives a 4 component value that represents the raw data stored in the depth buffer. The second "INTZ" returns a simple depth value representing the depth that is stored in the depth buffer. INTZ is the recommended code to use, but it is only supported on GeForce 8 Series GPUs and later.

You can use direct access to the depth buffer to enable effects (such as custom shadow mapping) without having to render depth to a color buffer. By performing a Z only render you can make use of the double speed Z rendering and still get direct access to the rendered depth.

6.4.2.1. Usage

"No z-compare" z-buffers are exposed as a separate FOURCC format ('RAWZ' on NV4x and 'INTZ' on G8x).

To create a RAWZ z-buffer, just do:

1. On GeForce 6/7 series use: `(D3DFORMAT)MAKEFOURCC('R','A','W','Z')`
2. On GeForce 8 series use: `(D3DFORMAT)MAKEFOURCC('I','N','T','Z')`

```
m_pd3dDevice->CreateTexture(TEXDEPTH_WIDTH, TEXDEPTH_HEIGHT, 1,
D3DUSAGE_DEPTHSTENCIL, (D3DFORMAT)MAKEFOURCC('I','N','T','Z'),
D3DPOOL_DEFAULT, &m_pSMZTexture, NULL)
```

Then use it just as if it was a normal z-buffer (render to it, etc).

6.4.2.2. Reconstructing Z for RAWZ (GeForce 6/7 Series GPUs)

REMINDER: Since the GeForce 8 Series and later GPUs can read depth buffers directly, there's no need for the 'RAWZ' format. Thus, this mode is exposed as FOURCC of 'INTZ' on G8x. When reading it in the shader, no reconstruction is necessary - z value is replicated to all four components of a texture fetch.

If you are using RAWZ due to running on a GeForce 6/7 Series GPU then in the shader you reconstruct the z-value as follows:

```
float z = dot(tex2D(RawZSampler, tcoord).arg,
float3(0.99609380937181767057285729 4849,
0.00389099144285866277567522380800 39,
1.51991853236666514674813430000 15e-5));
```

Unfortunately, the result provided by this simple approximation is not always accurate enough. In fact, due to rounding errors, it can be as bad as being accurate to only 8 bits. Better accuracy (at the cost of more instructions) can be obtained by dropping any extra rounding errors as follows:

```
float3 rawval = floor( 255.0 * tex2D(RawZSampler, tcoord).arg +
0.5);
```

```
float z = dot( rawval, float3(0.99609380937181767057285729 4849,
0.00389099144285866277567522380800 39,
1.51991853236666514674813430000 15e-5) / 255.0);
```

Chapter 7. Performance Tools Overview

This section describes several of our tools that will help you identify and remedy performance bottlenecks as well as assist you in content creation. Our tools are constantly being updated and new tools being developed. For up to date information as well as the latest releases and usage information please visit

http://developer.nvidia.com/page/tools.html

This chapter will overview some of the most used tools.

7.1. PerfKit

http://developer.nvidia.com/PerfKit

NVIDIA PerfKit is a comprehensive suite of performance tools to help debug and profile OpenGL and Direct3D applications. It gives you access to low-level performance counters inside the driver and hardware counters inside the GPU itself. The counters can be used to determine exactly how your application is using the GPU, identify performance issues, and confirm that performance problems have been resolved.

NVIDIA PerfKit includes support for 32-bit and 64-bit Windows XP and Vista platforms.

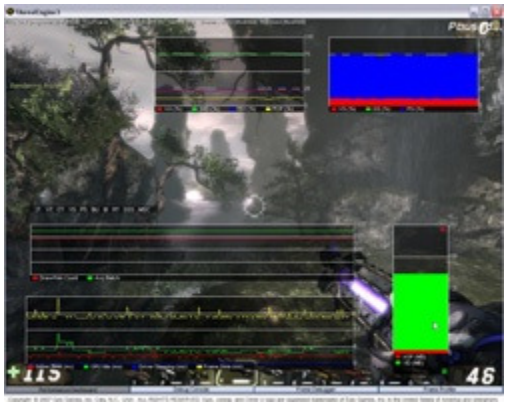
The performance counters are available directly in your OpenGL and DirectX applications and in tools such as Intel® VTune™ for Windows and Graphic Remedy's gDEBugger via the Windows Management Instrumentation (WMI) Performance Data Helper (PDH) interface. A plug-in supporting Microsoft PIX

for Windows is also provided, giving you low-latency access to PerfKit performance counters directly from the driver.

The following tools are included (plus more) in the NVIDIA PerfKit.

7.1.1. PerfHUD

http://developer.nvidia.com/PerfHUD



NVIDIA PerfHUD is a powerful real-time performance analysis tool for Direct3D applications. PerfHUD is widely used by the world's best game developers and was a 2007 Game Developer Magazine Frontline Award Finalist.

Check out screenshots of PerfHUD running in some today's most successful games, testimonials by game developers all over the world, or reviews of PerfHUD 5 from iXBT and Beyond3D.

- GeForce GTX 200 Series GPU Support
- A robust input layer for intercepting the mouse and keyboard
- Works with standard drivers on Windows Vista
- SLI Support
- Texture Visualization and Overrides
- API Call List
- Dependency View
- CPU/GPU Timing graph

7.1.2. PerfSDK

http://developer.nvidia.com/object/nvperfsdk_home.html

NVPerfSDK is a component of NVPerfKit that provides a programmatic API for accessing performance counters in the graphics driver and GPU. It allows you to query performance counters from your own applications, enabling you to

build customized profiling functionality. NVPerfSDK is available in 32-bit and 64-bit Windows as well as 32-bit and 64-bit Linux.

NVPerfSDK has the following features:

- NVIDIA NVPerfAPI for easy integration into applications
- Sample applications provided to learn from
- Simplified Experiments
  - Targeted, multipass experiments to determine GPU bottleneck
  - Automated analysis of results to show bottlenecked unit
- Use cases
  - Real time performance monitoring using GPU and driver counters, round robin sampling
  - Simplified Experiments for single frame analysis

To learn more about NVPerfSDK, we recommend that you view our "GPU Performance Tuning with NVIDIA Performance Tools" talk from GDC 2006.

7.1.3. GLExpert

http://developer.nvidia.com/object/glexpert_home.html

GLExpert is a component of NVPerfKit that helps OpenGL developers by identifying errors and performance issues.

GLExpert is controlled using the GLExpert tab of the NVIDIA Developer Control panel. You can choose what level of debugging information to report, as well as where the output is sent.

GLExpert provides the following features:

- Outputs to console/stdout or debugger
- Displays different groups and levels of information detail
- **OpenGL errors:** print as they arise
- **Software Fallbacks:** indicate when the driver is using software emulation
- **GPU Programs:** print errors during compilation or linking
- **VBOs:** show where buffers reside along with mapping details
- **FBOs:** print reasons why a configuration is unsupported

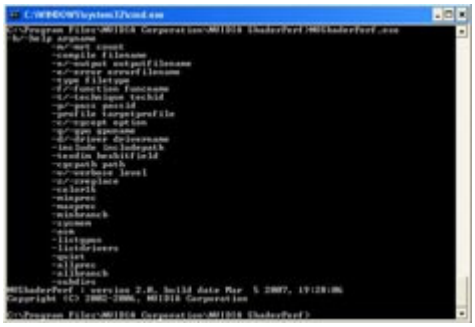
You can expect even more functionality in the future as GLExpert's feature list grows with driver advances.

To learn more about GLExpert, we recommend that you take a look at our "GPU Performance Tuning with NVIDIA Performance Tools" talk from GDC 2006.

7.1.4. ShaderPerf

http://developer.nvidia.com/ShaderPerf

NVIDIA ShaderPerf is a command-line shader profiling utility and C API that reports detailed shader performance metrics for a wide range of GPUs.



A graphical user interface (GUI) for shader performance analysis is available in FX Composer 2.5, and was built using the ShaderPerf API.

ShaderPerf 2.0 includes several new features:

- GeForce 8 series support
- Pixel Shader Differencing
- Vertex Shader Analysis

ShaderPerf outputs the following for any shader that you analyze:

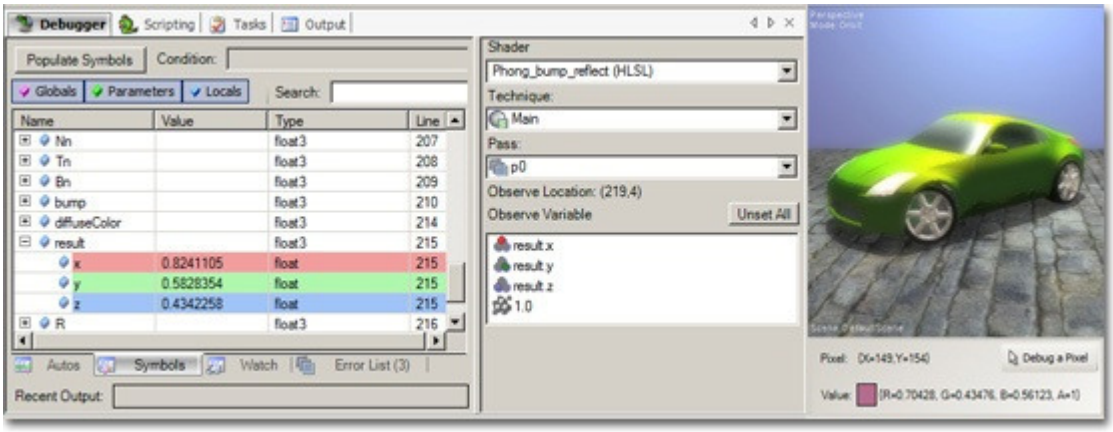
- Cycle count
- Register usage
- Driver-optimized shader instruction list
- Vertex and pixel throughput estimates

7.2. Shader Debugger

http://developer.nvidia.com/object/nv_shader_debugger_home.html

Modern shaders are growing in complexity, making them harder to understand and debug. To help developers address this problem, NVIDIA offers a full-

featured pixel shader debugger that allows shaders to be debugged just like CPU code.



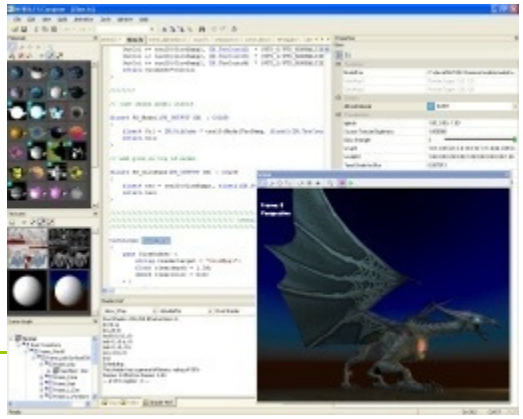
The NVIDIA Shader Debugger is a plug-in for FX Composer 2.5 that supports debugging of pixel shaders in the following shading languages:

- Microsoft DirectX 10 HLSL
- Microsoft DirectX 9 HLSL
- CgFX
- COLLADA FX Cg

7.3. FX Composer

http://developer.nvidia.com/FXComposer

FX Composer empowers developers to create high-performance shaders for DirectX and OpenGL in an integrated development environment with unique real-time preview and optimization features. FX Composer was designed with the goal of making shader



development and optimization easier for programmers while providing an intuitive GUI for artists customizing shaders for a particular scene.

FX Composer allows you to tune your shader performance with advanced analysis and optimization:

- DirectX 10 support, including geometry shaders and stream out.
- Visual Styles - the ability to create, define, and export multiple looks for a model.
- Particle systems
- Support for the NVIDIA Shader Debugger
    - Enables performance tuning workflow for vertex and pixel shaders
    - Optimization hints notify you of performance bottlenecks
- Capture of pre-calculated functions to texture look-up table
- Remote control over TCP/IP
- Unified Importing of Models

Developer Tools

Questions and Feedback

We would like to receive your feedback on our tools. Please visit our Developer Forums at http://developer.nvidia.com/forums.