NVIDIA GPU Programming Guide

Version 2.4.0

Notice

Trademarks

Copyright

HISTORY OF MAJOR REVISIONS

Version	Date	Changes
2.4.0	07/08/2005	Updated cover Added GeForce 7 Series content
2.3.0	02/08/2005	Added *2D & Video Programming* chapter Added more SLI information
2.2.1	11/23/2004	Minor formatting improvements
2.2.0	11/16/2004	Added normal map format advice Added ps_3_0 performance advice Added *General Advice* chapter
2.1.0	07/20/2004	Added *Stereoscopic Development* chapter
2.0.4	07/15/2004	Updated MRT section
2.0.3	06/25/2004	Added *Multi-GPU Support* chapter

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, but keep in mind that it is never too early to send an e-mail to devsupport@nvidia.com asking for help or advice.

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 7 Series, GeForce 6 Series, and NV4X-based Quadro FX GPUs. These tips focus on features, but also address performance in some cases.
- Chapter 5 offers several useful programming tips for NVIDIA® GeForce™ FX and NV3X-based Quadro FX GPUs. These tips focus on features, but also address performance in some cases.

- Chapter 6 presents general advice for NVIDIA GPUs, covering a variety of different topics such as performance, GPU identification, and more.
- Chapter 7 explains NVIDIA's Scalable Link Interface (SLI) technology, which allows you to achieve dramatic performance increases with multiple GPUs.
- Chapter 8 describes how to take advantage of our stereoscopic gaming support. Well-written stereo games are vibrant and far more visually immersive than their non-stereo counterparts.
- Chapter 9 provides an overview of NVIDIA's performance tools.

1.2. Sending Feedback

If you have comments or suggestions for this document, please send them to devsupport@nvidia.com

Chapter 2. How to Optimize Your Application

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

2.1. Making Accurate Measurements

Many convenient tools allow you to measure performance while providing tested and reliable performance indicators. For example, NVPerfHUD's yellow line (see the NVPerfHUD documentation for more information) measures total milliseconds (ms) per frame and displays the current frame rate.

To enable valid performance comparisons:

- ❑ **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 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.

- ❑ **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.

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.*

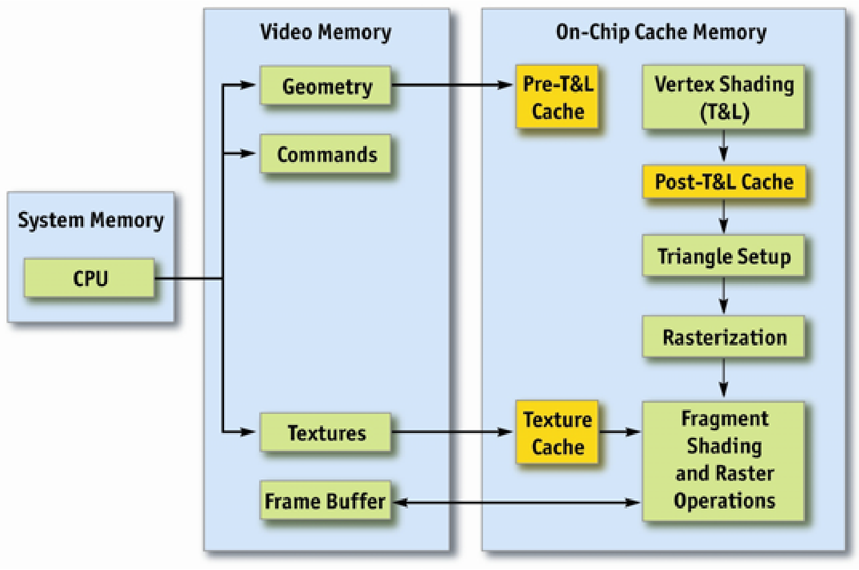


Figure 1. Potential Bottlenecks

In an ideal case, there won't be any one bottleneck—the CPU, AGP 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. NVPerfHUD's green line (see Section 10.1 for more information about NVPerfHUD) 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.

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/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html) 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 under-locking 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 NVPerfHUD

To verify CPU bottlenecks, run your application on a special driver, called a *Null Hardware* (Null HW) driver. This driver works like a normal driver (it runs through all the same code paths of a normal driver), except it never actually hands any work to the GPU. A Null HW driver therefore emulates an infinitely fast GPU. *If the performance of a Null HW driver is no better than a normal driver, the application is completely CPU-bound.*

NVPerfHUD 2.0 has a special mode that suppresses all draw calls of an application and thus emulates a Null Hardware Driver. However, omitting all draw calls also lightens the CPU load, since the driver no longer has to process and commit any of the state-changes made prior to each draw call. NVPerfHUD also has a variety of other useful features that can help you to identify performance problems.

If NVPerfHUD shows the GPU is never idle, then the application is GPU-limited. The blue line on NVPerfHUD shows how many milliseconds the driver is waiting for the GPU, so it's able to verify GPU-bound performance.

In addition, NVPerfHUD 3.0 has a "Frame Analysis" mode that allows you to freeze your application and single-step though the current frame, one draw call at a time. You can drill down to see what is happening inside the GPU at each stage in the graphics pipeline using Advanced State Inspectors for the Index Unit, Vertex Shader, Pixel Shader & Render Operations unit.

The *NVPerfHUD User Guide* contains detailed methodology for identifying and removing bottlenecks, troubleshooting, and more. It is available at http://developer.nvidia.com/object/nvperfhud_home.html.

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*)

❑ Driver (*nv4disp.dll, nvoglnt.dll*)

❑ DirectX Runtime (*d3d9.dll*)

- ❑ 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 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`, `d3d9.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? Improving batch sizes helps reduce driver calls. Detailed information about batching is available in the following presentations:

http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.ppt

http://download.nvidia.com/developer/presentations/GDC_2004/Dx9Optimization.pdf

NVPerfHUD 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 3.0 can be useful in a variety of situations to collapse multiple batches into one, and reduce both batch and draw overhead. See Section 4.1 for more on Shader Model 3.0.

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 http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_PipelinePerformance.ppt.

NVPerfHUD 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. NVPerfHUD similarly permits control over pixel shader execution times by forcing all or part of the shaders to run in a single cycle.

If you determine that the GPU is the bottleneck for your application, use the tips presented in Chapter 3 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 FX, GeForce 6 Series, and GeForce 7 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.

- ❑ **Poor Batching Causes CPU Bottleneck**
    - ❑ Use fewer batches
        - ❑ Use texture atlases to avoid texture 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.

- **Vertex Shaders Cause GPU Bottleneck**
    - Use indexed primitive calls
        - Use DirectX 9's mesh optimization calls [`ID3DXMesh:OptimizeInplace()` or `ID3DXMesh:Optimize()`]
        - Use our NVTriStrip utility if an indexed list won't work http://developer.nvidia.com/object/nvtristrip_library.html
- **Pixel Shaders Cause GPU Bottleneck**
    - Choose the minimum pixel shader version that works for what you're doing
        - When developing your shader, it's okay to use a higher version. Make it work first, then look for opportunities to optimize it by reducing the pixel shader version.
    - If you need `ps_2_*` functionality, use the `ps_2_a` profile
    - Choose the lowest data precision that works for what you're doing:
        - Prefer `half` to `float`
    - Use the `half` type for everything that you can:
        - Varying parameters
        - Uniform parameters
        - Variables
        - Constants
    - Balance the vertex and pixel shaders.
    - Push linearizable calculations to the vertex shader if you're bound by the pixel shader.
    - 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**

  - ❑ 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**
  - ❑ How to take advantage of antialiasing

3.2. Batching

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.

3.3. Vertex Shader

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` 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.

3.4. 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. 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, the high-end GeForce FX GPUs can deliver more than 20 operations per clock cycle! And the latest GeForce 6 and 7 Series GPUs can deliver many times more performance.

3.4.1. Choose the Lowest Pixel Shader Version That Works

Choose the lowest pixel shader version that will get the job done. For example, if you're doing a simple texture fetch and a blend operation on a texture that's just 8 bits per component, there's no need to use a `ps_2_0` or higher shader.

3.4.2. Compile Pixel Shaders Using the `ps_2_a` Profile

Microsoft's HLSL compiler (`fxc.exe`) adds chip-specific optimizations based on the profile that you're compiling for. If you're using a GeForce FX GPU and your shaders require `ps_2_0` or higher, you should use the `ps_2_a` profile, which is a superset of `ps_2_0` functionality that directly corresponds to the GeForce FX family. Compiling to the `ps_2_a` profile will probably give you better performance than compiling to the generic `ps_2_0` profile. Please note that the `ps_2_a` profile was only available starting with the July 2003 HLSL release.

In general, you should use the latest version of fxc (with DirectX 9.0c or newer), since Microsoft will add smarter compilation and fix bugs with each release. For GeForce 6 and 7 Series GPUs, simply compiling with the appropriate profile and latest compiler is sufficient.

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 FX, GeForce 6 Series, and GeForce 7 Series GPUs support 32-bit and 16-bit floating point formats (called `float` and `half`, respectively), and a 12-bit fixed-point format (called `fixed`). The `float` data type is very IEEE-like, with an `s23e8` format. The `half` is also IEEE-like, in an `s10e5` format. The 12-bit `fixed` type covers a range from [-2,2) and is not available in the `ps_2_0` and higher profiles. The `fixed` type is available with the `ps_1_0` to `ps_1_4` profiles in DirectX, and with either the `NV_fragment_program` extension or Cg in OpenGL.

The performance of these various types varies with precision:

- ❑ The `fixed` type is fastest and should be used for low-precision calculations, such as color computation.
- ❑ 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, with more than 99% of the rendered pixels 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 (available in the July 2003 HLSL update) 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` or `fixed` types, make sure you use them for varying parameters, uniform parameters, variables, and constants. If you're using DirectX's assembly language with the `ps_2_0` profile, use the `_pp` modifier to reduce the precision of your calculations.

If you're using the OpenGL `ARB_fragment_program` language, use the `ARB_precision_hint_fastest` option if you want to minimize execution time, with possibly reduced precision, or use the `NV_fragment_program` option if you want to control precision on a per-instruction basis (see http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_fragment_program_option.txt).

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

On GeForce FX hardware in OpenGL, you can speed up shaders consisting of mostly floating-point operations by doing operations (like dot products of normalized vectors) in fixed-point precision.

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 3.4.8 and 3.4.6), the sum of the first three terms is guaranteed to be 1.0. This expression can then be refactored as:

  `p = sqrt(2 * (Rz + 1)) = 1.414*sqrt(Rz + 1)`

- Fold the multiplication by 1.414 into another constant (see Section 3.4.8), 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 Pack Vector Values into Scalar Components of Multiple Interpolants

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.

3.4.6. 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.

3.4.7. 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.

3.4.8. 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.

3.4.9. 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.

3.4.10. Balance the Vertex and Pixel Shaders

Achieving high performance is all about removing bottlenecks—which really means that you have to balance every piece of the pipeline: the CPU, the AGP bus, and the stages of the graphics pipeline. The decision to use a vertex shader or a pixel shader depends on a few factors:

- ❑ **How tessellated are your objects?** You may want to lighten the load on the vertex shader if you have millions of vertices in each frame. This is especially true if you're using a multipass algorithm.
- ❑ **What resolution are you targeting?** If you expect your application to be run at higher resolutions, the pixel shader is more likely to become the bottleneck. So, you may want to push more computations to the vertex shader.
- ❑ **How long are your pixel shaders?** If you're doing complex shading, the pixel shader will probably be your bottleneck. If your pixel shaders compile to more than 20 cycles (on average) and occupy more than half the screen, your application will likely be pixel shader-bound on GeForce FX hardware. So, look for opportunities to move calculations to the vertex shader. (See Section 3.4.11 for examples.) You can use our NVShaderPerf tool to find out how many cycles your shaders are using. Also, note that newer hardware such as GeForce 6 and 7 Series GPUs will allow more complex pixel shaders before becoming pixel shader-bound.

3.4.11. Push Linearizable Calculations to the Vertex Shader If You're Bound by the Pixel Shader

The rasterizer takes per-vertex values and interpolates them per-fragment while accounting for perspective correction. Take advantage of the hardware that already takes care of this for you by moving linear calculations to the vertex shader. You may be able to perform the calculation for fewer vertices and still receive an interpolated result in the pixel shader.

For example, you can move from world space to light space for attenuation. Or, if you're doing bump mapping, you can make the move into tangent space per-vertex, unless you're doing per-pixel reflection into a cube map.

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

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. Use Lower-Numbered Interpolants First

You will see higher performance if you use lower-numbered texture coordinate sets (`TEXCOORD` sets) first. Start by using `TEXCOORD0`, and move upwards from there to `TEXCOORD1`, `TEXCOORD2`, and so on.

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. 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 multitextured 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 FX 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:

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

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.

3.5.3.2. Normalizing Vectors

If you're writing a `ps_1_*` shader, use normalization cube maps to normalize vectors quickly. For higher quality, you can use two 16-bit signed cube maps: one for `x` and `y`, and the other for `z`.

Another optimization is based on the fact that, in practice, the vectors V that are normalized are often of norm close to 1 because they're interpolated or filtered normals. This means that you can approximate 1 /||V|| by the first terms of the Taylor expansion of 1 / sqrt(x) at x = 1:

$$1 / \mathrm{sqrt}(x) \sim 1 + \tfrac{1}{2} (1 - x)$$

such that:

$$V / ||V|| = V / \mathrm{sqrt}(||V||\^{}2) = V + \tfrac{1}{2} V (1 - ||V||\^{}2)$$

This formula can be written in two assembly instructions:

```
dp3_sat r1, r0, r0
mad_d2 r1, r0, 1-r1, r0_x2
```

with `r0` containing V and the final value of `r1` containing V / ||V||.

The code above is only valid for `ps_1_4` because of the `_x2` register modifier. For lower pixel shader versions, the following formula can be used instead:

```
dp3_sat r1, r0_bx2, r0_bx2
mad r1, r0_bias, 1-r1, r0_bx2
```

This is assuming that `r0` contains ½ (V + 1), which is rarely a constraint as V often needs to be passed on range-compressed from [-1, 1] to [0, 1] to the pixel shader.

GeForce 6 and 7 Series GPUs have a special half-precision normalize unit that can normalize an fp16 vector for free during a shader cycle. Take advantage of this feature, simply perform a normalization on an fp16 quantity and the compiler will generate a `nrmh` instruction.

For more on normalization, please see our Normalization Heuristics and Bump Map Compression whitepapers, available at the following URLs:

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

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

3.5.3.3. The `sincos()` Function

Despite the preceding advice, the GeForce FX family and later 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. Performance

3.6.1. Double-Speed Z-Only and Stencil Rendering

The GeForce FX, GeForce 6 Series, and GeForce 7 Series GPUs 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
- ❑ The active depth-stencil surface is not multisampled
- ❑ Texkill has not been applied to any fragments
- ❑ 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
- ❑ No user clip planes are enabled

3.6.2. Early-Z Optimization

Early-z optimization (sometimes called “z-cull”) 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. To take advantage of z-cull, follow these guidelines:

- ❑ Don’t create triangles with holes in them (that is, avoid alpha test or texkill)
- ❑ Don’t modify depth (that is, allow the GPU to use the interpolated depth value)

Violating these rules can invalidate the data the GPU uses for early optimization, and can disable z-cull until the depth buffer is cleared again.

3.6.3. Lay Down Depth First

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 NVSDK 7.1 and later.

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:

1. 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.
2. Create vertex and pixel shaders
3. Load remaining textures

3.7. Antialiasing

GeForce FX, GeForce 6 Series, and GeForce 7 Series GPUs all 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.

One issue that is now solved with DirectX 9.0b or later is using antialiasing with post-processing effects. The `StretchRect()` call can copy the back buffer to an off-screen texture in concert with multisampling.

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 `StretchRect()` 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.

Chapter 4.
GeForce 6 & 7 Series Programming Tips

This chapter presents several useful tips that help you fully use the capabilities of GeForce 6 & 7 Series as well as NV4X-based Quadro FX GPUs. These are mostly feature oriented, though some may affect performance as well.

4.1. Shader Model 3.0 Support

Microsoft DirectX 9.0 introduced several new standards for advanced vertex and pixel shader technology, version 2.0 and version 3.0. Shader Model 2.0 hardware has been available since late 2002, and the vast majority of GPUs shipped today support Shader Model 2.0 or better. Shader Model 2.0 includes technologies useful for advanced lighting and animation techniques, but has limited shader program length, and complexity, which limits the fidelity of the effects that can be achieved.

As developers push against the limits inherent in Pixel Shader 2.0 and Vertex Shader 2.0, they have started to adopt the newer, more advanced Shader Model 3.0. This shader model has advances in several areas, in both pixel and vertex shader processing.

4.1.1. Pixel Shader 3.0

The following is a feature summary outlining the key differentiators between Pixel Shader 2.0 and 3.0.

Pixel Shader Feature	**Shader 2.0**	**Shader 3.0**	**Description**
Shader length	96	65535+	Allows more complex shading, lighting, and procedural materials
Dynamic branching	No	Yes	Saves performance by skipping complex shading on irrelevant pixels
Shader antialiasing	Not supported	Built-in derivative instructions	Developers can calculate the screen space derivatives of any function, allowing them to adjust shading frequencies or over-sampling to eliminate artifacts
Back-face register	No	Yes	Allows two-sided lighting in a single pass
Interpolated color format	8-bit integer minimum	32-bit floating point minimum	Higher range and precision color allows high-dynamic range lighting at the vertex level
Multiple render targets	Optional	4 required	Allows advanced lighting algorithms to save filtering and vertex work – thus more lights for minimal cost
Fog and specular	8-bit fixed function minimum	Custom fp16-fp32 shader program	Shader Model 3.0 gives developers full and precise control over specular and fog computations, previously fixed-function
Texture coordinate count	8	10	More per-pixel inputs allows more realistic rendering, especially for skin

4.1.2. Vertex Shader 3.0

Here is a similar listing of key features developers enjoy when moving from Vertex Shader Model 2.0 to 3.0.

Vertex shader feature	Shader 2.0	Shader 3.0	Description
Shader length	256 Instructions	65535 instructions	More instructions allow more detailed character lighting and animation
Dynamic branching	No	Yes	Saves performance by skipping animation and calculations on irrelevant vertices
Vertex texture	No	Any number of lookups from up to 4 textures	Allows displacement mapping, particle effects
Instancing support	No	Required	Allows many varied objects to be drawn with only a single command

4.1.3. Dynamic Branching

One major feature of both Shader 3.0 models (vertex and pixel) is Dynamic Branching. Put simply, this allows a shader author to create true loops and conditionals in their shader programs. For instance, one could write a shader that looped through a certain number of vertex lights, determine which ones might influence a particular vertex, and then pass down the index of each relevant light to the pixel shader. The pixel shader could then use this 'light index' to determine which light parameters to apply. The pixel shader would then loop over the active lights, then use dynamic branching to exit the shader early once all lights are processed.

Most light types only apply to the front side of an object—the side facing the light. Therefore, you can use both vertex and pixel branching to skip processing for lights that the shader detects as facing away from the light (using the new "back-face" register). This can save significant processing time, and speed up the shader. Similar speedups can be used to skip processing of character bone animation as well as many similar algorithms.

4.1.4. Easier Code Maintenance

As game engines become more and more complex, they often create many different versions of each shader in order to fit them all in to the Pixel Shader 2.0 program length limitations. This adds to code maintenance, shader compile time, and level load time, as well as taking up valuable system memory at runtime. Shader Model 3.0 eliminates this issue, through its comprehensive looping and branching, allowing the engine to write a single vertex and single pixel shader containing appropriate static and dynamic branching in order to select the correct execution path at runtime, thus greatly simplifying the shader combinatorial explosion issue.

4.1.5. Instancing

Another key feature of Shader Model 3.0 is the support for the Microsoft DirectX® Instancing API. Currently, games face limits on the number of unique objects they can display in the scene, not because of graphics horsepower, but often because of the CPU-side overhead of either storing or submitting many slightly different variations of the same object. For instance, a forest is made up of trees that are often similar to each other, but each would be in a different position, have differing height, leaf color, and so on. In order to add the desired variation, developers have to choose between storing many separate copies of the tree, each slightly different, or making expensive render state changes in order to rotate, scale, color and place each tree.

Instancing allows the programmer to store a single tree, and then several other vertex data streams to specify the per-instance color, height, branch size and so on. For instance, a single 1,000-vertex tree model would contain the vertex positions and normals, and a 200-element vertex streams would contain positions, colors, and heights. Instancing allows the programmer to submit a single draw call, which renders each of the 200 trees, using the same data for the basic tree shape, but then vary it through the per-instance streams.

Our instancing code sample is available at
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

4.1.6. Summary

In summary, DirectX 9.0 Shader Model 3.0 is a significant step forward in terms of ease of use, performance, and shader complexity. Dynamic branching brings speed-ups to many algorithms which contain early-out opportunities, while also simplifying shader code paths in graphics engines and tools. Lastly, instancing allows extreme complexity for very low CPU and memory overhead.

4.2. GeForce 7 Series Features

With 8 vertex pipelines and 24 pixel pipelines, the GeForce 7800 GTX features a superscalar architecture and the NVIDIA CineFX 4.0 engine. The CineFX 4.0 engine includes a redesigned vertex shader unit that shortens the time required to set up complex geometry processing. This results in higher vertex throughput.

In addition, a new pixel shader unit delivers up to 2x the floating point shading power of previous generation products due to additional computational resources in the shader core.

An improved texture unit incorporates new hardware to speed up anisotropic and floating-point filtering operations. Also included are new features such as transparency antialiasing, improved multisampling, advanced video processing, and more.

Learn more about the GeForce 7 Series at http://www.nvidia.com/page/geforce_7800.html.

4.3. Transparency Antialiasing

Transparency antialiasing is a new feature available for GeForce 7 Series GPUs. When alpha testing is used to create transparent regions in textures, transparency antialiasing will antialias the resulting edges, resulting in higher image quality.

With GeForce 7800 GTX, NVIDIA has exposed a render state that allows individual primitives to be multisampled. Multisampling is accomplished by using the computed alpha as a measure of how many samples the pixel will take. A texel with an alpha value of one will contribute to all samples in the destination pixel, while texels with successively lower alpha values will contribute to fewer samples. Transparency multisampling is especially effective for textures where soft edges are desirable (as is the case with most vegetation).

When using transparency multisampling, your art assets must have a gradual falloff towards their edges in the alpha channel. The alpha channel should be created or scaled in the shader such the computed alpha goes from the alpha reference value to 0.0f at the boundaries. This ensures that the coverage values will fall off nicely, producing smooth antialiased edges.

Learn more about transparency antialiasing at http://developer.nvidia.com/object/transparency_aa.html.

4.4. sRGB Encoding

sRGB encoding is a format that uses gamma conversion to provide more precision near zero, mimicking the human visual system. sRGB also works with DXT compression formats, allowing applications to benefit from both increased color fidelity and reduced storage size.

On GeForce 6 & 7 Series GPUs, you may want to use a floating-point format in some cases, as this offers several advantages:

- □ Higher precision across the entire range.
- □ Much larger, linear, dynamic range
- □ Linear blending for frame buffers

For textures, sRGB is vastly preferable to floating-point in most cases, because of its smaller memory footprint and bandwidth requirements.

4.5. Separate Alpha Blending

The GeForce 6 & 7 Series GPUs allow you to specify a separate blending function for color and alpha. This gives you more flexibility with respect to what values are stored in the alpha channel of your textures, for example. One use for this would be to modulate the color channels while preserving alpha.

4.6. Supported Texture Formats

The following table shows the texture formats supported by GeForce 6 & 7 Series GPUs.

Integer Formats	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
R8G8B8	N	N	N	N	N	N	N	N	N
A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	Y	Y	Y	Y	Y	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N
G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

Float Formats	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
R16F	N	N	N	N	N	N/A	N	N	N
G16R16F	Y	Y	Y	Y	Y	N/A	Y	N	N
A16B16G16R16F	Y	Y	Y	Y	Y	N/A	Y	Y	N
R32F	Y	Y	Y	Y	N	N/A	Y	N	Y
G32R32F	N	N	N	N	N	N/A	N	N	N
A32B32G32R32F	Y	Y	Y	Y	N	N/A	Y	N	Y

Shadow Map	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
D24X8	Y	N	N	Y	Y	N/A	Y	N/A	N
D24S8	Y	N	N	Y	Y	N/A	Y	N/A	N
D16	Y	N	N	Y	Y	N/A	Y	N/A	N

4.7. Floating-Point Textures

GeForce 6 & 7 Series GPUs offer improved support for floating-point textures. The following table shows the various features that are supported for both 16-bit per component (fp16) and 32-bit per component (fp32) floating-point textures.

Texture Component Type	Nearest Filtering	Bilinear and Trilinear Filtering	Anisotropic Filtering	Mipmap Support	3D Textures	Cube Maps	Non-power-of-2 Textures
16-bit	Yes	Yes	Yes	Yes	Yes	Yes	Yes
32-bit	Yes	No	No	Yes	Yes	Yes	Yes

4.7.1. Limitations

Please note that we do not support the R16F format – use G16R16F instead. In addition, you can only blend to an A16B16G16R16F surface, not a G16R16F or R32F surface. However, filtering is supported for G16R16F textures.

4.8. Multiple Render Targets (MRTs)

GeForce 6 & 7 Series GPUs support MRTs, which allow a pixel shader to write out data to up to four different targets. MRTs are useful whenever the pixel shader computes more than four float values and needs to store these intermediate results in a texture.

Examples uses of MRTs include particle physics simultaneously computing positions and velocities, and similar GPGPU algorithms. Deferred shading is another technique that computes and stores multiple float4 values simultaneously: it computes all material properties, such as for example, surface normal, diffuse and specular material properties and stores these in separate textures. These properties become used when lighting the scene with multiple lights in subsequent passes.

The DirectX caps bit `NumSimultaneousRTs` indicates how many render targets the graphics device can render to simultaneously. For GeForce 6 & 7 Series GPUs that caps bit is four. To enable MRTs use the

SetRenderTarget(index, pRenderTarget) API call. This call binds the passed in render-target texture to the given render index. A pixel shader then outputs to those bound render-target textures using the oC0, oC1, oC2, and oC3 output registers. Remember to reset the render targets for indices one through three to NULL to turn MRT rendering off.

MRTs restrict other GPU features. Most important, hardware-accelerated antialiasing is inapplicable to MRT render targets. Furthermore, all render targets must have the same width, height, and bit depth. For the GeForce 6 & 7 Series, that means one can freely mix-and-match within the 32-bit formats (i.e., A8R8G8B8, X8R8G8B8, G16R16F, and R32F), or use up to four render-targets with 64bits each (i.e., use the A16R16G16B16F format), or up to four render-targets with 128 bits each (i.e., use the A32R32G32B32F format).

Furthermore, the post pixel shader blend operations alpha-blending, alpha-testing, fogging, and dithering are only available for MRT if the D3DMISCCAPS_MRTPOSTPIXELSHADERBLENDING cap bit is set and querying a render-format for USAGE_QUERY_POSTPIXELSHADERBLENDING returns yes.

GeForce 6 & 7 Series chips support this capability for all MRT formats, except R32F, G16R16F, and A32R32G32B32F. Thus, note in particular that GeForce 6 & 7 Series GPUs (aside from the GeForce 6200) do support post-blend operations on A16R16G16B16F floating point render-targets. The DirectX specification further modifies the behavior of these MRT post-blend operations: MRT rendering ignores the dithering state, respects the fog state only for render-target zero (render-targets one through three act as if fog is disabled), and alpha-testing uses only the value of oC0.a to determine whether or not to discard all four render-target pixels.

Finally, using MRTs has performance implications. MRTs have a large associated frame-buffer bandwidth cost, especially when using the wider bit-depths. For example, rendering to four A32R32G32B32F surfaces consumes 16 times the frame-buffer bandwidth of rendering to a single A8R8G8B8! In addition, the GeForce 6 & 7 Series has a performance sweet spot when using three or fewer render-targets simultaneously.

The following general performance advice thus applies: Use MRTs only as needed, i.e., when MRT saves multiple passes. Minimize the number of render-targets and their bit-depths, for example by tightly packing your data and not wasting the alpha channels. Make sure to allocate MRT render-targets early (see Section 3.6.4. Allocating Memory).

Also, split MRT outputs into groups of 3 or fewer, when doing so does not increase the total number of passes. For example, if your application renders an ambient pass, followed by a pass which outputs to 4 MRTs, consider outputting one of the targets during the ambient pass, and then outputting to just 3 MRTs. Doing so is particularly beneficial if one of the targets can be stored at a lower precision than the others, and is easily computed independently of the other targets (e.g., a material diffuse texture map). You can learn more about deferred shading in version 7.1 of our SDK or by downloading our Deferred Shading demo clip at ftp://download.nvidia.com/developer/Movies/NV40-LowRes-Clips/Deferred_Shading.avi.

4.9. Vertex Texturing

The GeForce 6 & 7 Series GPUs support vertex texturing, but vertex textures should not be treated as constant RAM. Vertex textures generate latency for fetching data, unlike true constant reads. Therefore, the best way to use vertex textures is to do a texture fetch and follow it with many arithmetic operations to hide the latency before using the result of the texture fetch.

Vertex Textures **are not a substitute for large arrays of constants.** They instead are designed for sparse per vertex data, such that there are only a small number of vertex texture fetches per vertex.

Learn more about vertex textures by reading our Using Vertex Textures whitepaper, which is available at: http://developer.nvidia.com/object/using_vertex_textures.html.

4.10. General Performance Advice

The GeForce 6 & 7 Series architecture contains numerous improvements that make it more efficient and versatile. Here are some tips to help you take advantage of its capabilities:

- **Use write masks and swizzles.** The GeForce 6 & 7 Series shader architecture is able to schedule portions of 4-component vectors on different units (through co-issue and dual-issue), which improves shader utilization. By using write masks and swizzles, you can help the compiler to identify these types of schedule opportunities.
- **Use partial precision whenever possible.** There are two reasons to use partial precision with GeForce 6 & 7 Series GPUs. First, the GeForce 6 & 7 Series have a special free fp16 normalize unit in the shader, which allows 16-bit floating-point normalization to happen very efficiently in parallel

with other computations. To take advantage of this, simply use partial precision whenever appropriate in your programs. The second reason is that partial precision helps to reduce register pressure, potentially resulting in higher performance.

- ❑ **Use dynamic branching when the branches will be fairly coherent.** As mentioned in Section 4.1.3, dynamic branching can make code faster and easier to implement. But in order for it to work optimally, branches should be fairly coherent (for example, over regions of roughly 30 x 30 pixels).

4.11. Normal Maps

If normal map storage is an issue for your application, normal map compression for unit-length tangent-space normals can be achieved on GeForce 6 & 7 GPUs by utilizing a technique known as hemisphere remapping to eliminate the need to store one of the components:

```
N.z = sqrt( 1 – N.x*N.x – N.y*N.y );
```

This compiles to 3-5 pixel shader instructions, so this technique may or may yield a performance improvement, depending on whether these instructions can be co-issued with other (previously existing) shader instructions, and whether or not fetching textures is a bottleneck.

If you decide to pursue hemisphere remapping, the preferred texture formats on GeForce 6 & 7 Series GPUs are `D3DFMT_V8U8` in DirectX, and `GL_LUMINANCE8_ALPHA8` in OpenGL. These formats are 16 bits/pixel, which provide 2:1 lossless compression.

Note that hemisphere remapping does reduce some flexibility, since only positive, unit-length normals are generated. Techniques which rely on negative values (such as object-space normal mapping) or non-unit normals (such as the antialiasing technique described in http://developer.nvidia.com/object/mipmapping_normal_maps.html) will not be possible.

To learn more about normal maps, please see our *Bump Map Compression* whitepaper at http://developer.nvidia.com/object/bump_map_compression.html.

To create high quality normal maps that make a low-poly model look like a high-poly model, use NVIDIA Melody. Simply load your low poly working model, then load your high-poly reference model, click the "Generate Normal Map" button and watch Melody go to town. Melody is available at http://developer.nvidia.com/object/melody_home.html.

Chapter 5.
GeForce FX Programming Tips

This chapter presents several useful tips that help you fully use the capabilities of the GeForce FX family. These are mostly feature oriented, though some may affect performance as well.

5.1. Vertex Shaders

The powerful GeForce FX vertex engine can achieve over 200 million triangles per second on the GeForce FX 5900. It supports full dynamic branching, which allows a shader author to branch, based on the number and type of lights, bones, and so forth.

Each active branch thread will slow the overall execution of the program, so branch only to save significant vertex calculations or to increase primitive batch sizes through the API.

5.2. Pixel Shader Length

The GeForce FX can natively handle 512 pixel instructions per pass in Direct3D and 1,024 in OpenGL. In DirectX, compile to the `ps_2_a` or `ps_2_x` profiles. In OpenGL, you can use GLSL, Cg (by compiling to the `arbfp1` or `fp30` profiles), or use the `ARB_fragment_program` extension directly.

Quadro FX cards can handle 2,048 pixel instructions per pass.

5.3. DirectX-Specific Pixel Shaders

The latest DirectX 9 `ps_2_0` and higher shading models require that math and temporaries be computed at 24-bit or higher precision by default (the GeForce FX family uses the 32-bit `float` type for this case). Applications can specify a `_pp` modifier on the assembly to achieve 16-bit floating-point precision.

When you use HLSL or Cg, this task is very easy—declare your variables as `float` for 32-bit precision, and `half` for 16-bit precision.

We recommend writing your shaders the most convenient way first, and then optimizing for register use and half-precision as needed.

Fixed-point blending is mainly used in the fixed-function texture blending pipeline and in the texture arithmetic sections of shader models `ps_1_0` – `ps_1_4`.

In DirectX, you cannot request fixed-point precision through `ps_2_0`. If you can fit a program into `ps_1_1` – `ps_1_4`, it will often run faster because of the increased usage of the fixed-point shading hardware.

5.4. OpenGL-Specific Pixel Shaders

The `ARB_fragment_program` extension requires 24-bit floating-point precision at a minimum, by default. You can place various flags at the top of the `ARB_fragment_program` source code to control precision:

- ❑ `NV_fragment_program`. Allows the `half` (16-bit floating-point) and `fixed` (12-bit fixed-point) formats to be used explicitly for maximum control and performance.
- ❑ `ARB_precision_hint_fastest`. The Unified Compiler will determine the appropriate precision for each shader operation at run-time, increasing overall performance with minimal visual artifacts.
- ❑ `ARB_precision_hint_nicest`. Forces the entire program to run in `float` precision.

5.5. Using 16-Bit Floating-Point

Many developers haven't worked with `half` before, and view `float` as a format that "just works," with very few concerns about range and precision. `half` has 10 mantissa bits and five bits of exponent. `float` has 23 bits of mantissa and 8 bits of exponent. Each mantissa bit can be viewed as a tick mark on a ruler. This dictates the maximum precision of the format. In `half`, the precision between each mantissa bit is .1%. The exponent value can be viewed as the length of the ruler. As the ruler gets longer with higher exponents, each tick on the ruler represents a greater unit of length. This is how floating-point formats automatically trade precision for range.

With `half`, you can only exactly represent the integers from –2048 to 2048, with no fractional bits left over. If you perform view- or world-space calculations in the pixel shader, you may run out of precision. For instance, if your character is at position 4096 and the light is at position 4097, both characters are represented by the same 16-bit floating point number. When you subtract these values, you get zero. Then, squaring and normalizing the result yields `INF`. The workaround is simple, elegant, and even faster than the original shader: Move matrix and vector subtraction operations into the vertex shader.

Problems with lighting calculations when you first use a GeForce FX GPU can often be traced to misuse of the `half` format. This is not surprising because the `half` type is not a common format for games, although it is used extensively in films for colors.

Vertex shaders are required at a minimum to support `float`, so they can easily handle large world and view spaces. Also, it's easy to see why linear calculations belong in the vertex shader to begin with—why recalculate something at each pixel when it can be calculated only per vertex and iterated for free by the GPU?

Therefore, our recommendation is to move vertices into light space or tangent space in the vertex shader, and pass the resulting position down to the pixel shader. One nice approach is to subtract light positions from vertex positions in the vertex shader. Then uniformly scale the vector by a constant to make the value closer to zero, and normalize the vector in the fragment shader. (Note that the uniform scale doesn't affect the normalized result.)

Often the goal is to perform lighting in tangent space anyway. This is especially useful because it lets you center the coordinate system on the vertex. Working near zero gives you the sign bit to work with as well, so you have an extra mantissa bit to work with.

In general, perform constant calculations on the CPU, linear calculations in the vertex shader, and nonlinear calculations in the pixel shader.

5.6. Supported Texture Formats

Integer Formats	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
R8G8B8	N	N	N	N	N	N	N	N	N
A8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
X8R8G8B8	Y	Y	Y	Y	Y	Y	Y	Y	N
R5G6B5	Y	Y	Y	Y	Y	Y	Y	Y	N
X1R5G5B5	Y	Y	Y	Y	Y	Y	Y	Y	N
A1R5G5B5	Y	Y	Y	Y	Y	Y	N	N	N
A4R4G4B4	Y	Y	Y	Y	Y	Y	N	N	N
R3G3B2	N	N	N	N	N	N	N	N	N
A8	N	N	N	N	N	N/A	N	N	N
A8R3G3B2	N	N	N	N	N	N	N	N	N
X4R4G4B4	N	N	N	N	N	N	N	N	N
A2B10G10R10	N	N	N	N	N	N	N	N	N
A8B8G8R8	N	N	N	N	N	N	N	N	N
X8B8G8R8	N	N	N	N	N	N	N	N	N
G16R16	Y	Y	Y	Y	Y	N	N	N	N
A2R10G10B10	N	N	N	N	N	N	N	N	N
A16B16G16R16	N	N	N	N	N	N	N	N	N
A8P8	N	N	N	N	N	N	N	N	N
P8	N	N	N	N	N	N	N	N	N
L8	Y	Y	Y	Y	Y	N	N	N	N
L16	Y	Y	Y	Y	Y	N	N	N	N
A8L8	Y	Y	Y	Y	Y	N	N	N	N
A4L4	N	N	N	N	N	N	N	N	N

Float Formats	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
R16F	N	N	N	N	N	N/A	N	N	N
G16R16F	Y	Y	Y	Y	N	N/A	Y	N	N
A16B16G16R16F*	Y	N	N	N	N	N/A	Y	N	N
R32F	Y	Y	Y	Y	N	N/A	Y	N	N
G32R32F	N	N	N	N	N	N/A	N	N	N
A32B32G32R32F*	Y	N	N	N	N	N/A	Y	N	N

Shadow Map	2D	Cube	3D	MIP	Filter	sRGB	Render	Blend	Vertex
D24X8	Y	N	N	Y	Y	N/A	Y	N/A	N
D24S8	Y	N	N	Y	Y	N/A	Y	N/A	N
D16	Y	N	N	Y	Y	N/A	Y	N/A	N

* Exposed in DX9.0c without wrapping or mipmapping capability

5.7. Using `ps_2_x` and `ps_2_a` in DirectX

GeForce FX supports several features beyond `ps_2_0` functionality, including derivative calculations (through DDX and DDY), longer shaders, and predication support. You can access this functionality from a high-level shading language in two ways. One is to compile using HLSL or Cg with the `ps_2_x` profile, which uses the above functionality but does not check against a particular set of capability bits. The better approach is to use the `ps_2_a` profile, which is a profile that exactly matches the GeForce FX family's shading capabilities and produces more optimized code.

5.8. Using Floating-Point Render Targets

The GeForce FX family supports four-component floating-point render targets in 64-bit and 128-bit, and one- and two-component 32-bit floating point render targets and mipmapped textures.

Under OpenGL, floating-point textures are exposed through the `NV_float_buffer` extension, and multiple low-precision components can be packed into a single larger-precision component using the pack/unpack instructions in `NV_fragment_program`. Applications can emulate `NEAREST_MIPMAP_NEAREST` floating-point cubemaps and volume textures using the pack/unpack instructions on ordinary RGBA8 textures (support is limited to a single fp32 or 2 fp16 components).

On GeForce FX hardware, floating-point render targets don't support blending and floating-point textures larger than 32-bits per texel don't support mipmapping or filtering.

5.9. Normal Maps

GeForce FX, GeForce4, and GeForce3 GPUs have dedicated hardware to normal map hemisphere remapping from a 2-component normal map. For these GPUs, you should use the CxV8U8 format, since on these parts it may be faster than to use this format rather than deriving Z in the shader.

For more information about normal maps, please see our Bump Map compression whitepaper at http://developer.nvidia.com/object/bump_map_compression.html.

To create high quality normal maps that make a low-poly model look like a high-poly model, use NVIDIA Melody. Simply load your low poly working model, then load your high-poly reference model, click the "Generate Normal Map" button and watch Melody go to town. Melody is available at http://developer.nvidia.com/object/melody_home.html.

5.10. Newer Chips and Architectures

The GeForce FX architecture is now available in a wide range of models and prices. Each product in the family has the same vertex and pixel shading features. These features let developers easily scale their games targeting GeForce FX and beyond—for example, by handling scalability through only geometry LOD or screen resolution. The only differences are with respect to performance: that is, the number of vertex and pixel processors.

GeForce 6 & 7 Series GPUs have much faster `float` performance, but continue to support `half` because of its higher performance. They are also more orthogonal with respect to the `float` and `half` types, floating-point render targets, and floating-point textures compared to their fixed-point counterparts.

5.11. Summary

The GeForce FX, GeForce 6 Series, and GeForce 7 Series architectures have the most flexible shader capabilities in the industry—from long shader programs to true derivative calculations. However, on GeForce FX hardware, pure floating-point shaders do not run as fast as a combination of fixed- and floating-point shaders.

For many shaders, the best way to achieve maximum performance on the GeForce FX architecture may be to use a mixture of `ps_1_*` and `ps_2_*` shaders. For instance, for per-pixel lighting it may be faster to do the diffuse lighting term in a `ps_1_1` shader, and the specular term in another pass using a `ps_1_4` or `ps_2_0` shader.

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 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 6 & 7 Series GPUs because they misidentify the GPU as a TNT-class GPU, or don't recognize the Device ID. This behavior creates a support nightmare, as the NV4X and G70 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 and extension strings. 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.

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

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.

If you're using shadow maps in a game engine, you may want to take a look at:

- http://developer.nvidia.com/object/hwshadowmap_paper.html
- http://developer.nvidia.com/object/cedec_shadowmap.html
- http://developer.nvidia.com/object/d3dshadowmap.html
- http://developer.nvidia.com/object/Shadow_Map.html
- *Perspective Shadow Maps* from SIGGRAPH 2002

- ❑ *Perspective Shadow Maps: Care and Feeding* in *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (http://developer.nvidia.com/GPUGems)

Simon Kozlov's "Perspective Shadow Maps: Care and Feeding" chapter in *GPU Gems* explains some improvements that make perspective shadow maps usable in practice. We have taken the concepts in Kozlov's chapter and implemented them in an engine of our own as proof of concept, and we've found that they work well in real-world situations. This example is available in version 7.0 and higher of our SDK. It's available at http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html.

In DirectX, you can create a hardware shadow map in the following way:

1) Create a texture with usage `D3DUSAGE_DEPTHSTENCIL`

2) The format should be `D3DFMT_D16, D3DFMT_D24X8` (or `D3DFMT_D24S8`, but you can't access the stencil bits in your shader)

3) Use `GetSurfaceLevel(0)` to get the `IDirectDrawSurface9` Interface

4) Set the Surface pointer as the Z buffer in `SetDepthStencilSurface()`

5) DirectX requires that you set a color render target as well, but you can disable color writes by setting `D3DRS_COLORWRITEENABLE` to zero.

6) Render your shadow-casting geometry into the shadow map z buffer

7) Save off the view-projection matrix used in this step.

8) Switch render targets and z buffer back to your main scene buffers

9) Bind the shadow map texture to a sampler, and set the texture coordinates to be:

```
V' = Bias(0.5/TexWidth, 0.5/TexHeight, 0) *
Bias(0.5, 0.5, 0) *
Scale(0.5,0.5,1) *
ViewProjsaved * World * Object * V
```

The matrices can be concatenated on the CPU, and the concatenated transformation can be applied in either a vertex shader or using the fixed-function pipeline's texture matrices.

10) If using the fixed-function pipeline or ps_1.0-1.3, set the projection flags to be `D3DTTFF_COUNT4 | D3DTTFF_PROJECTED`.

11) If using pixel shaders 1.4 or higher, perform a projected texture fetch from the shadow map sampler.

12) The hardware will use the shadow map texture coordinate's projected x and y coordinates to look up into the texture.
13) It will compare the shadow map's depth value to the texture coordinate's projected z value. If the texture coordinate depth is greater than the shadow map depth, the result returned for the fetch will be 0 (in shadow); otherwise, the result will be 1.
14) If you turn on `D3DFILTER_LINEAR` for the shadow map sampler, the hardware will perform 4 depth comparisons, and bilinearly filter the results for **the same cost** as one sample—this just makes things look better.
15) Use this value to modulate with your lighting

Early NVIDIA drivers (version 45.23 and earlier) implicitly assumed that shadow maps were to be projected. This behavior changed with NVIDIA drivers 52.16 and later—programmers now need to explicitly set the appropriate texture stage flags. In particular, to use shadow-maps in the ps.2.0 shader model one has to explicitly issue a projective texture look-up (for example, `tex2Dproj(ShadowMapSampler, IN.TexCoord0).rgb`). Emulating the same command by doing the w-divide by hand, followed by a non-projective texture look-up does not work! For example, `tex2D(ShadowMapSampler, IN.TexCoord0/IN.TexCoord0.w)` does not work.

Similarly, when using the ps1.1-1.3 shader models, drivers version 52.16 and later now require that the projective flag is explicitly set for the texture-stage sampling the shadow-map (for example, `SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_PROJECTED`).

NOTE: In ForceWare 61.71 and later, any texture instruction (`tex2D`, `tex2Dlod`, etc.) will work correctly with shadow maps.

Our SDK contains simple examples of setting up hardware shadow mapping in both DirectX and OpenGL. They are available at: http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html.

Chapter 7.
2D and Video Programming

In addition to delivering phenomenal 3D performance, GPUs are of course able to shoulder 2D and video programming tasks. This chapter focuses on performance advice for that area of computer graphics.

7.1. OpenGL Performance Tips for Video

When used properly, Quadro FX, GeForce 6 Series, and GeForce 7 Series GPUs can achieve extremely high levels of performance for video applications. This section explains some of the guidelines video developers should follow to get the most out of the GPU.

Video developers writing a playback application are concerned with uploading textures as quickly as possible to the video card. Video typically has a video frame size with a resolution that is not a power of 2, some typical video resolutions are:

Standard Definition Video (SD)	720 × 480i	720 × 576i
High Definition Video (HD)	1280 × 720p	1920 × 1080i

There are three methods that can be used for texturing video images:

1. `GL_TEXTURE_2D`
   POT (power of two) texture coordinates range from [0..u_{scale}] x [0..v_{scale}].
2. `GL_TEXTURE_2D`
   NP2 (non-power of two) texture coordinates range from [0..1] x [0..1].
3. `GL_TEXTURE_RECTANGLE_NV`
   NP2 (non-power of two) texture coordinates range from
   [0..width] x [0..height].

7.1.1. POT with and without Mipmaps

A POT texture with the source as a NP2 texture size requires the driver to pad extra data for each texture upload. With mipmapping enabled, the driver needs to create all the subsequent mipmap levels. Both of these impose a significant performance penalty. Texture coordinates must range from [0..u_{scale}] x [0..v_{scale}]. This [u_{scale}, v_{scale}] factor shown below handles all various image widths and heights.

$$\left[u_{scale} \quad , \quad v_{scale} \right] = \left[\frac{NP2_{width}}{POT_{width}} \quad , \quad \frac{NP2_{height}}{POT_{height}} \right]$$

This method to texture video supports all mipmap and non-mipmap texture filtering as well as all the texture wrap and border modes.

To specify a POT texture size, use this parameter for *target*.

```
glTexSubImage2D (GL_TEXTURE_2D, … )
```

7.1.2. NP2 with Mipmaps

A NP2 texture with mipmapping enabled is useful for compositing Standard Definition video with computer generated graphics. This technique has the same drawbacks as POT textures, as the driver will need to generate all mipmap levels and perform texture padding when textures are being uploaded.

Before using this option, the application first needs to query for the OpenGL extension string `ARB_texture_non_power_of_two`. If this capability is supported by the driver, this method may be used. Texture coordinates must range from [0..1] x [0..1].

This method to texture video supports all mipmap texture filtering and texture wrap and border modes.

To specify a NP2 texture size, use this parameter for *target*:

```
glTexSubImage2D (GL_TEXTURE_2D, … )
```

7.1.3. NP2 without Mipmaps (Recommended)

For video playback at HD resolutions, the goal is to upload each texture to the video card as quickly and efficiently as possible. This method eliminates the need for texture padding and the mipmap creation. NP2 textures have the benefit of using less video memory than POT textures and reduce bandwidth requirements significantly. Texture coordinates are non-normalized and range from [0..width] x [0..height]. There are some restrictions: NP2 textures may not use mipmap filtering and do not support the optional 1-texel border. For texture wrap and border modes, only the following are supported:

- ❑ `GL_CLAMP`
- ❑ `GL_CLAMP_TO_EDGE`
- ❑ `GL_CLAMP_TO_BORDER_ARB`.

All non-mipmap texture filtering is supported with one exception. For the GeForce FX family and the GeForce 6200 GPU, only `GL_NEAREST` filtering is supported for 16-bit floating point textures. For GeForce 6 & 7 Series as well as NV4XGL-based Quadro FX GPUs, filtering is supported with 16-bit floating point textures.

To specify a NP2 texture size, use the following parameter for *target*:

```
glTexSubImage2D (GL_TEXTURE_RECTANGLE_NV, … )
```

7.1.4. Texture Performance with Pixel Buffer Objects (PBOs)

PBOs enable the fast path for texture uploads/downloads in the driver. There is an excellent example with source code included in the NVIDIA SDK 8.5 (and newer versions) that uses ARB approved extensions for PBO's. This shows example demonstrates how to optimally use OpenGL Pixel Buffer Objects (PBOs) for transferring textures to and from the GPU.

This example can be found at:
http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html.

Chapter 8. NVIDIA SLI and Multi-GPU Performance Tips

This chapter presents several useful tips that allow your application to get the maximum possible performance boost when running on a multi-GPU configuration, such as NVIDIA's SLI technology. For more information, please see also ftp://download.nvidia.com/developer/presentations/2004/GPU_Jackpot/SLI_and_Stereo.pdf.

8.1. What is SLI?

SLI, which stands for "Scalable Link Interface", allows multiple GPUs to work together to deliver higher performance. SLI-certified motherboards are PCI Express motherboards with two x16 physical lanes; each one of these slots accepts a PCI Express graphics board. When the two graphics boards are then linked via an external SLI bridge connector, the driver recognizes the configuration and allows you to enter SLI Multi-GPU mode. In SLI Multi-GPU mode the driver configures both boards as a single device: the two GPUs look like a single logical device to all graphics applications.

This single logical device runs up to 1.9 times faster than a single GPU, since the driver splits the rendering load across the two physical GPUs. Note that running in SLI mode does not double available video memory. For example, plugging in two 256 MB graphics boards still only results in a device with at most 256 MB of video memory. The reason is that the driver generally replicates all video memory across both GPUs. That is, at any given time the video memory of GPU 0 contains the same data as the video memory of GPU 1.

When running an application on a SLI system, it may run in one of several modes: compatibility mode, alternate frame rendering (AFR), or split frame rendering (SFR) mode.

Compatibility mode simply uses only a single GPU to render everything (that is, the second GPU is idle at all times). This mode cannot show any performance enhancement, but also ensures an application is compatible with SLI.

For AFR the driver renders all of frame n on GPU 0 and all of frame n+1 on GPU 1.   Frame n+2 renders on GPU 0 and so on. As long as each frame is self-contained (that is, frames share little to no data) AFR is maximally efficient, since all rendering work, such as per-vertex, rasterization, and per-pixel work splits evenly across GPUs. If some data is shared between frames (for example, reusing previously rendered-to textures), the data needs to transfer between the GPUs. This data transfer constitutes communications overhead preventing a full 2x speed-up.

For SFR the driver assigns the top portion of a frame to GPU 0 and the bottom portion to GPU 1. The size of the top versus the bottom portion of the frame is load balanced: if GPU 0 is underutilized in a frame, because the top portion is less work to render than the bottom portion, the driver makes the top portion larger in an attempt to keep both GPUs equally busy. Clipping the scene to the top and bottom portions for, respectively, GPU 0 and 1, attempts to avoid processing all vertices in a frame on both GPUs.

SFR mode still requires data sharing, for example, for render-to-texture operations. Because AFR generally has less communications overhead and better vertex-load balancing than SFR, AFR is the preferred mode. Sometimes, however, AFR fails to apply, for example, if an application limits the maximum number of frames buffered to less than two.

8.2. Choosing SLI Modes

Applications in development default to compatibility mode in current drivers (66.93 and later). Application developers can force AFR and SFR modes by creating an application-specific profile in the driver, i.e., in the display driver control panel, click on the GeForce tab, select "Performance & Quality Settings," click on "Add Profile," and enter the name of your application's executable. Then enable "Show advanced settings" and choose "AFR" or "SFR".

Another option is for the application to programmatically choose an SLI mode. All NVIDIA drivers support a control panel API that lets applications query and set various control panel options. As part of that API the `NvCplGetDataInt()` entry point allows querying for how many total GPUs are installed and how many of those are currently in SLI mode. The `NvCplSetDataInt()` entry point lets application set whether the calling application runs in compatibility, AFR, or SFR mode.

Complete documentation of these interfaces and source code showing how to use these interfaces are available as part of the NVSDK; the NVCpl sample in particular exercises the SLI detection and mode-setting functionality (http://developer.nvidia.com/object/sdk_home.html).

The following is a list of dos and don'ts to get a maximum performance boost on an SLI system.

8.3. Avoid CPU Bottlenecks

If your application is CPU-bound running it on a more powerful graphics solution has little to no performance impact. To take advantage of a multi-GPU configuration, you must avoid becoming bottlenecked by the CPU. Chapter 2 of this Programming Guide provides guidance how to detect and avoid CPU bottlenecks.

Similarly, if your application artificially throttles its frame rate to an arbitrary and fixed value, then the frame rate cannot exceed that value. Please avoid that situation.

8.4. Disable VSync by Default

Enabling VSync forces the frame rate to (at best) match the monitor refresh rate. Multi-GPU configurations are likely to achieve frame rates that are much higher than monitor refresh rates, due to their higher graphics performance. These frame rates are thus only attainable if VSync is turned off.

For some applications, i.e., visual simulations, VSync is an absolute requirement to avoid tearing. In these cases SLI still demonstrates significant benefit when the graphics workload and fill-rate requirement are high. For example, using SLI it may become possible to run those applications in resolutions of 1920×1200 or 2048×1536 along with 4× or higher full-scene antialiasing at real-time rates.

Triple buffering is not a solution for attaining higher frame rates: Triple-buffering simply allocates an additional back buffer that a graphics adapter can render to.

With triple buffering, a graphics adapter can render into up to three buffers round-robin style. But if the graphics adapter consistently finishes rendering each buffer at a rate higher than the monitor refresh rate, then the number of back buffers is irrelevant; the monitor refresh continues to gate the overall frame rate.

Worse, triple buffering has two significant disadvantages:

1. It consumes more video memory (in the case that screen resolution is high and antialiasing is enabled, the amount of additional video memory consumed is significant).
2. It increases lag as more frames are in flight between issuing a rendering command and seeing it onscreen.

To turn off VSync in DirectX, the `PresentationInterval` member of the `D3DPRESENT_PARAMETERS` structure has to be set to `D3DPRESENT_INTERVAL_IMMEDIATE` when calling `IDirect3D9::CreateDevice()`.

8.5. DirectX SLI Performance Tips

8.5.1. Limit Lag to At Least 2 Frames

DirectX allows drivers to buffer up to three frames. Buffering frames is desirable to ensure that CPU and GPU can work independent of one another and thus achieve maximum performance. On the other hand, the more frames are buffered the longer it takes between issuing a command and seeing its result on-screen. This lag is generally undesirable, since humans can detect and object to lag-times of as little as 30 ms (depending on the test scenario).

Some games thus artificially limit how many frames are buffered. For example, locking the back-buffer forces a hard synchronization between the CPU and the GPU. Locking the back-buffer first stalls the CPU, drains all buffers, and then stalls the GPU. At the end of the lock all systems are idle, and the number of buffered frames is zero.

Stalling a system in this manner has a severe performance penalty and is to be avoided, especially on multi-GPU configurations.

A less objectionable solution to limiting the number of buffered frames is to insert tokens into the command stream (e.g., DirectX event queries) at the end of each frame. Checking that these events have been consumed before issuing additional rendering commands limits the number of buffered frames and thus lag to anywhere between one to three frames.

Multi-GPU systems are particularly sensitive to limiting the number of buffered frames. In general, a system with n GPUs requires at the very least n frames to be buffered to be maximally efficient.

Surprisingly, doing so does not increase lag, since a dual GPU system generally runs twice as fast as a single GPU system. For example, buffering two frames (that take 15 ms each to render) on a dual GPU system has the same 30 ms lag as buffering one frame (that takes 30ms to render) on a single GPU system.

We thus recommend that applications check how many GPUs are available and, if they must, limit the number of buffered frames to at least that number of GPUs. The control panel API (see Section 8.2) allows querying whether a system is in SLI mode and how many GPUs are currently in use. In particular, the function `NvCplGetDataInt(NVCPL_API_NUMBER_OF_SLI_GPUS, &number)` returns the number of SLI-enabled GPUs in the system. The NVControlPanel_API.pdf document and the NVSDK sample NVCpl provides details.

8.5.2. Update All Render-Target Textures in All Frames that Use Them

The efficiency of a multi-GPU system is inversely proportional to how much data the GPUs share. In the best case, the GPUs share no data, thus have no synchronization overhead, and thus are maximally efficient.

To minimize the amount of shared data, each rendered frame should be independent of all previous frames. In particular, when using render-to-texture techniques, it is desirable that all render-target textures used in a frame are also generated during that same frame. Conversely, avoid updating a render target only every other frame, yet using the same render-target as a texture in every frame.

If an application is explicitly skipping render-target updates to increase rendering speed on single GPU systems, then it might be of advantage to modify that algorithm for multi-GPU configurations. For example, detect if the application is running with multiple GPUs and if so, update render-targets every frame (i.e., to increase visual fidelity) or update render-targets two frames in a row and then skip updates two frames in a row.

Alternatively, rendering to render-targets early on and only using the result late in the frame is also beneficial for SLI systems. It avoids stalling one GPU waiting for the results of the other GPU's render to texture operation.

8.5.3. Clear Color and Z for Render Targets and Frame Buffers

Clearing the color and z information of a render target prior to its use indicates to the driver and the GPU that any existing data in the render target is irrelevant. Conversely, not clearing that data indicates that the data may be relevant and thus needs to be maintained, and in the case of SLI configurations shared between GPUs.

Clearing z is generally advisable even knowing that all z-values in the render target are going to be overwritten later (see Section 3.6.2): clearing z is close to free and enables early z-cull optimizations. On SLI configurations, an added benefit is avoiding synchronizing this z-information across the GPUs.

Similarly, clearing color is advisable on SLI configurations as it avoids synchronization overhead between the GPUs. Thus, you should always clear color, even when knowing that every pixel in the render target is going to be overwritten later anyway.

8.6. OpenGL SLI Performance Tips

8.6.1. Limit OpenGL Rendering to a Single Window

Optimal scaling with multiple GPUs happens with a single OpenGL window. Any child windows your application creates should not contain an OpenGL context.

With the AFR mode you are limited to exactly one double buffered window with page flipping. Any additional OpenGL windows will prohibit page flipping and force the driver to fall back to a single GPU until window positioning makes page flipping possible again.

8.6.2. Request PDF_SWAP_EXCHANGE Pixel Formats

When creating an OpenGL context, make sure to ask for a pixel format with the `PFD_SWAP_EXCHANGE` flag set instead of `PFD_SWAP_COPY`. The swap exchange flag implies that the application does not rely on the back buffer content after a `SwapBuffers()` is performed.

This is essential for the AFR mode because the driver cycles through GPUs per frame so the back buffer may be stored on another GPU.

For the SFR mode `PFD_SWAP_EXCHANGE` is preferred over `PFD_SWAP_COPY` because the swap copy mode requires additional synchronization between the GPUs.

8.6.3. Avoid Front Buffer Rendering

Rendering to the front buffer requires heavy synchronization when rendering to multiple GPUs and should therefore be avoided at all costs. OpenGL overlays are recommended instead of front buffer rendering. They are supported on Quadro and work on both GPUs independently in an SLI configuration. Make sure to use a R5G5B5 high-color overlay for optimal performance on NVIDIA GPUs.

8.6.4. Limit pbuffer Usage

Rendering to a pbuffer on a multi GPU system requires the driver to broadcast the rendering to both GPUs because the rendered result may be used by either GPU later on. This limits the amount of scaling available when running your application on multiple GPUs.

In some cases it is possible for the driver to detect that your application only uses the output from a pbuffer in the same frame and can limit the pbuffer rendering to a single GPU.

8.6.5. Render Directly into Textures Instead of Using glCopyTexSubImage

If your application is doing any sort of render-to-texture operation the preferred method is to use pbuffers (`WGL_ARB_pbuffer`) with the render-to-texture extension (`WGL_ARB_render_texture`) or the upcoming framebuffer object extension (`GL_EXT_framebuffer_object`) to render directly into a texture. Using `glCopyTexSubImage()` to copy data from a pbuffer or the back buffer is not recommended as the texture needs to get updated on both GPUs.

For more information, please see the following resources on our Developer web site:

http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_framebuffer_object.txt

http://www.nvidia.com/dev_content/nvopenglspecs/WGL_ARB_pbuffer.txt

http://www.nvidia.com/dev_content/nvopenglspecs/WGL_ARB_render_texture.txt

8.6.6. Use Vertex Buffer Objects or Display Lists

To limit the CPU overhead when specifying geometry, your application should use vertex arrays with the vertex buffer object extension (`GL_ARB_vertex_buffer_object`) or display lists instead of immediate mode calls.

Here are some general VBO performance tips:

- ❑ Load your VBO working set first or textures may block faster memory until the working set stabilizes.
- ❑ Avoid huge batches when drawing geometry, there is a performance drop-off on NVIDIA GPUs when using more than 64k vertices per draw call.
- ❑ Use `unsigned short` for indices.
- ❑ Use `glDrawRangeElements()` instead of `glDrawElements()`.
- ❑ Use the correct VBO usage hints for the type of data in the VBO

For additional VBO performance tips, please see the following resources on our Developer web site:

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

http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_vertex_buffer_object.txt

8.6.7. Limit Texture Working Set

Texture data is shared between both GPUs, so your application should limit the amount of textures in use at a given time and not stream in new textures too often.

For game applications it is a good idea to render some pre-warming frames to get the textures into the right memory space to avoid low initial frame rates.

For procedural textures it's a good idea to generate them using a shader instead of downloading different versions of the texture every frame. This saves bandwidth and texture memory on both GPUs.

8.6.8. Render the Entire Frame

Avoid rendering to only a section of the frame using methods such as `glViewport()` or `glScissor()`. This disables load balancing with the SFR mode and hurts performance with the AFR method.

8.6.9. Limit Data Readback

Avoid reading back the color or depth buffers using `glReadPixels()` and never use `glCopyPixels()`. These commands cause pipeline stalls and inhibit parallelism. However, reading the contents of the back buffer for reuse in the same frame is ok.

8.6.10. Never Call glFinish()

The `glFinish()` command does not return until all pending OpenGL commands are complete. This kills asynchronous work and reduces any performance benefits from using multiple GPUs.

Chapter 9. Stereoscopic Game Development

This section explains how NVIDIA's stereo rendering implementation works, and how to take full advantage of it in your applications.

9.1. Why Care About Stereo?

Game developers often overlook one factor in their never ending quest for greater realism in games: People see with two eyes in the real world. While artificial stereoscopic viewing (on a screen versus in real life) is not a huge market, many gamers enjoy the extra sense of presence obtained by playing games with inexpensive shutter glasses along with NVIDIA's stereo override driver.

In addition, there are benefits to viewing your game in stereo during development. You immediately pick up on things that look fake. Keep in mind that motion parallax gives similar visual cues to stereo but the stereo viewer perceives instantaneously what users see if they move around and obtain depth information via motion parallax.

Using stereoscopic viewing while developing a game is a competitive advantage; you see and correct visual defects before they even come out in a game. This of course will also enhance the experience of those who play your game in stereo.

9.2. How Stereo Works

The NVIDIA 3D Stereo Driver allows full-screen stereoscopic viewing of DirectX and OpenGL based games. The 3D Stereo Driver supports red and blue anaglyph rendering as well as page-flipped viewing suitable for shutter glasses. With compatible hardware you'll see the image with the perception of depth. Please note that the Stereo Driver version must match the Display Driver version in order to function.

When playing 3D games with the stereoscopic driver enabled, the scene will be rendered from two viewpoints; each a little to the side of the real viewpoint, as though they were coming from the left and right eye. This works with both fixed-function rendering and vertex shaders.

For an accurate stereo effect, developers need consider a number of issues that we list below.

9.3. Things That Hurt Stereo

Here we list common things that negatively impact stereo, and provide some ideas to work around them.

9.3.1. Rendering at an Incorrect Depth

This problem is the number one thing you should take care of. The 3D Stereo Driver uses the depth to create the stereo effect, so anything that is not at the correct depth stands out like a sore thumb when viewed in Stereo.

❑ Place background images, sky boxes, and sky domes at the farthest possible depth. Otherwise, the world will look like it's in a little box in stereo.

❑ Place HUD items at their proper 3D depth. If you have name labels that hover over objects, putting them at the 3D depth of the object gives a better stereo effect than putting them on the near plane of the view frustum.

❑ It's also helpful to render the HUD as far into the scene as possible. This trick gives you a greater perceived depth in the rest of your scene while not causing eyestrain when looking at the HUD.

❑ Laser sights, crosshairs, and cursors do not look correct unless placed in the 3D world at the depth of the objects they are pointing to. When they aren't, it is almost impossible to use them since the user's eyes are converging on one depth, but the cursor is at another depth; users see two cursors, neither of which point at the correct place.

❑ Highlighting objects should happen at the depth of the object itself, not in screen space.

9.3.2. Billboard Effects

Billboard effects look flat and bad in regular 3D; they look even worse in stereo. In regular 3D you see the billboards as you move around, but in 3D Stereo the problem immediately pops out at you, even in a static scene. Most billboard effects look extremely flat in stereo, so use real geometry everywhere you can instead, even low resolution geometry looks better.

For particle effect billboards (sparks, smoke, dust, etc), these may or may not look ok, the best thing you can do is test your app in stereo to see what it looks like and judge if the quality is good enough, and make sure that the billboards have a meaningful depth in 3D.

9.3.3. Post-Processing and Screen-Space Effects

2D screen-space effects can greatly hurt the stereo effect. Things like blurry glow, bloom filters, image-based motion blur fall into this category. These effects are usually created by rendering the 3D geometry to a texture, then rendering a 2D screen aligned quad to the screen. The geometry in the texture is no longer at the correct depth in the world as the effect should be, thus working poorly in 3D Stereo.

You should provide the option to disable these effects for people playing in stereo and render the geometry to the back buffer.

9.3.4. Using 2D Rendering in Your 3D Scene

Any object rendered as 2D doesn't really have a real 3D depth, so it is placed at the monitor depth. This will look very flat in 3D Stereo. If you are mixing 2D and 3D in your HUD you may have inconsistent depths leading to eye strain. Again, render everything at the proper depth, in 3D, and test with stereo on.

9.3.5. Sub-View Rendering

When rendering a sub view to the screen, such as a picture in picture display, car mirror, or small map in the upper corner, you must set the viewport to cover the area before rendering. This trick prevents strange stereo effects bleeding outside of the intended section of the screen.

9.3.6. Updating the Screen with Dirty Rectangles

If you are determining only the parts of the screen that have changed and not updating the rest of the screen this can cause odd looking rendering in 3D stereo. Just render all visible objects each frame.

9.3.7. Resolving Collisions with Too Much Separation

If you are resolving collisions by pushing objects away from each other, make sure you don't push them away too far. It looks bad in normal 3D when moving around but stands out right away in stereo, making things appear to hover above the ground.

9.3.8. Changing Depth Range for Difference Objects in the Scene

Splitting the scene into multiple depth ranges can cause distortions in the stereo effect making some objects look shorter or elongated. All objects should be rendered in a consistent depth range for the best stereo effect.

9.3.9. Not Providing Depth Data with Vertices

When sending vertices for rendering to D3D for software transform and lighting, include the RHW depth information for stereo to function properly.

9.3.10. Rendering in Windowed Mode

NVIDIA's 3D Stereo only works when your application is in full-screen exclusive mode. If you don't support fullscreen mode, then game players can't take advantage of 3D Stereo.

9.3.11. Shadows

Rendering stencil shadows using a fullscreen shadow color quad will not work properly in stereo. However, re-rendering shadowed objects in the scene at their proper depth in shadow color will function correctly in stereo. Shadow maps function fine, and projection shadows function as long as you are projecting to the proper depth for the shadow.

9.3.12. Software Rendering

NVIDIA 3D Stereo drivers support DirectX and OpenGL automatically. If you use any other APIs to render 3D geometry it will not be rendered in stereo.

9.3.13. Manually Writing to Render Targets

Don’t lock render targets and do direct writes; doing so bypasses the stereo driver.

9.3.14. Very Dark or High-Contrast Scenes

Very dark scenes can become even darker when using 3D Stereo shutter glasses. Providing a brightness or gamma adjustment will help this problem. Using very bright objects on very bright and very dark objects causes ghosting, which hurts stereo. Testing your game in stereo quickly shows whether or not this is a problem.

9.3.15. Objects with Small Gaps between Vertices

Small gaps in meshes can become much more obvious when rendered in stereo. Make sure your meshes are tight and test in stereo to be sure this isn’t happening.

9.4. Improving the Stereo Effect

Here are some ideas you can use to create a more immersive experience using stereo.

9.4.1. Test Your Game in Stereo

The best thing you can do to get a great 3D Stereo experience is to test your game while running in stereo. Most problems are obvious right away and simple to fix. You can get low cost stereo kits from IODisplay (http://www.i-glasses.com). The NVIDIA 3D Stereo Driver also supports Red and Blue Anaglyph Stereo Mode, so if you have a pair of paper 3D glasses lying around it’s even easier to test.

9.4.2. Get “Out of the Monitor” Effects

You can design things in your game that are very close to the near plane but do not intersect the edges of the view frustum. This design lets you get the great “popping out of the monitor” stereo effect. Hovering orbs, space ships, flying characters, and so on are a few possibilities. These all really look fantastic in stereo.

9.4.3. Use High-Detail Geometry

Use more polygons for objects to get better realism in 3D. This strategy is of course always true, but even more so for stereo. Use polygons everywhere–for buildings, plants, trees, characters… anywhere possible!

9.4.4. Provide Alternate Views

Give the users a choice of viewpoint in your game, or at least control over the camera. First person, third person, top down, etc, some may look better than others in stereo.

9.4.5. Look Up Current Issues with Your Games

Once the 3D Stereo Driver is installed, you can view its control panel and look under “Stereo Game Configuration”. On this page you can see if your game is listed and what issues we have already found with it. Your NVIDIA Developer Relations contact can also help out here.

9.5. Stereo APIs

We are currently developing two separate APIs to better support stereo:

- ❑ **StereoBLT API – Display Pre-Rendered Stereo Images in 3D**
  Allows display of pre-created left/right images while the stereoscopic display is active.
- ❑ **IStereoAPI – Real-time Control Over Stereoscopic Rendering**
  Lets you query and control the convergence, stereo separation and other details of the stereo driver. This API consists of a header and library that you can use to control these settings in real time in your game. Modifications take effect immediately and can be changed every frame.

- **OpenGL Quad-Buffered Stereo.** This is available on the NVIDIA Quadro GPU family, requires no special stereo driver, and works in windowed mode too.

9.6. More Information

To learn more, contact your NVIDIA Developer Relations representative, or e-mail 3DStereoDev@nvidia.com to request more info or the prerelease Stereo APIs.

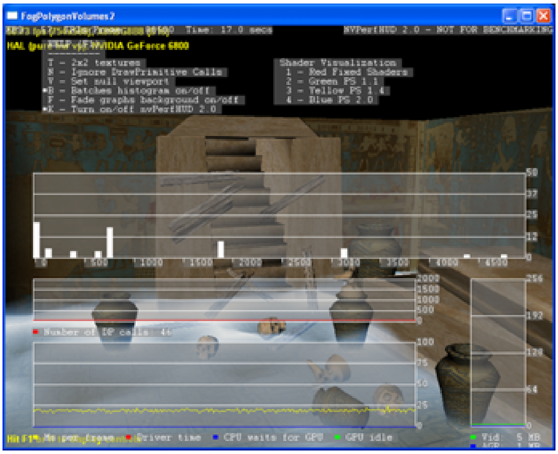
You can also find updated Stereo Information online at http://developer.nvidia.com/object/3D_Stereoscopic_Dev.html

Chapter 10.
Performance Tools Overview

This section describes several of our tools that will help you identify and remedy performance bottlenecks.

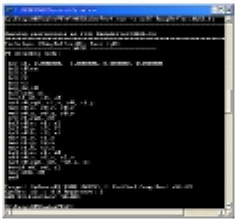
10.1. NVPerfHUD

NVPerfHUD displays four informative graphs overlaid on top of any DirectX application. These graphs show important statistics about your application, which helps you to identify potential bottlenecks. The graphs display sample data in a heart-rate monitor format. By scrolling from right to left, you can see the values of the last 256 frames.



You can get NVPerfHUD on the NVIDIA Developer Web Site at: http://developer.nvidia.com/object/nvperfhud_home.html.

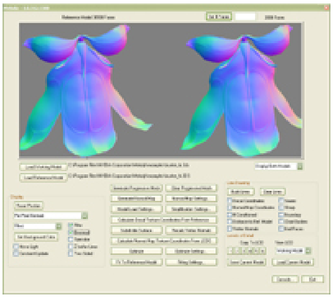
10.2. NVShaderPerf



The NVShaderPerf command line utility uses the same technology as the Shader Perf panel in FX Composer to report shader performance metrics. It supports DirectX and OpenGL shaders written in HLSL, GLSL, Cg, `!!FP1.0`, `!!ARBfp1.0`, `ps_1_x`, and `ps_2_x`. You can get performance reports for your shaders on the entire family of GeForce 6 & 7 Series and GeForce FX GPUs, including cycle count, register usage and a GPU utilization rating.

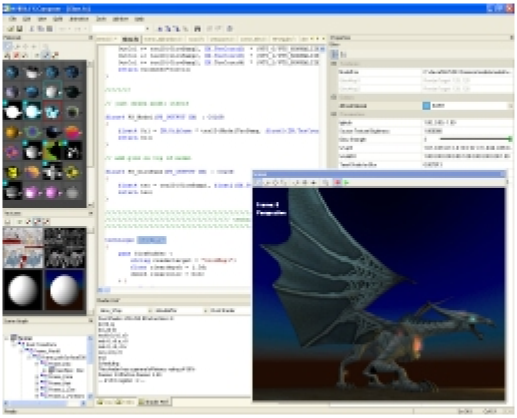
You can get NVShaderPerf at: http://developer.nvidia.com/object/nvshaderperf_home.html.

10.3. NVIDIA Melody



To create high quality normal maps that make a low-poly model look like a high-poly model, use NVIDIA Melody. Simply load your low poly working model, then load your high-poly reference model, click the "Generate Normal Map" button and watch Melody go to town. Melody is available at **http://developer.nvidia.com/object/melody_home.html**.

10.4. FX Composer



FX Composer empowers developers to create high performance shaders 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:

- Enables performance tuning workflow for vertex and pixel shaders
- Simulates performance for the entire family of GeForce 6 & 7 Series and GeForce FX GPUs
- Capture of pre-calculated functions to texture look-up table
- Provides empirical performance metrics such as GPU cycle count, register usage, utilization rating, and FPS.
- Optimization hints notify you of performance bottlenecks

You can download the latest version of FX Composer from: **http://developer.nvidia.com/fxcomposer**.

10.5. Developer Tools Questions and Feedback

We would like to receive your feedback on our tools. Please send your comments or concerns to sdkfeedback@nvidia.com.