Whitepaper

Shader Model 3.0
Using Vertex Textures

Philipp Gerasimov
Randima (Randy) Fernando
Simon Green
NVIDIA Corporation

Shader Model 3.0: Using Vertex Textures

Since the introduction of programmability to the GPU, the capabilities of the vertex and pixel processors have been different. Now, with Shader Model 3.0, GeForce 6 Series GPUs have taken a huge step towards providing common functionality for both vertex and pixel shaders. This paper focuses specifically on one such Shader Model 3.0 feature: vertex texture fetch. Vertex texture fetch allows vertex shaders to read data from textures, just like pixel shaders can.

In modern graphics processors, vertex processing performance tends to be underutilized as most applications are either bound by pixel shaders, memory bandwidth, or the CPU. This fact means that you can safely introduce complexity in your vertex shaders, resulting in higher image quality without any performance degradation. This additional complexity is useful for a number of effects, including displacement mapping, fluid and water simulation, explosions, and more.

Figure 1. The Visual Effect of Displacement Mapping

This whitepaper begins by describing the specifications for vertex textures in both DirectX and OpenGL. We then explain how to add vertex textures to your applications and discuss filtering as well as performance. Finally, we include a case study of a game to show how vertex textures can be used in real-world projects. Two screenshots from the game, entitled *Pacific Fighters*, are shown in Figure 1.

Specification

Vertex textures are available in both DirectX and OpenGL. The following sections explain how to use them in each API.

DirectX 9

The Microsoft DirectX 9.0 SDK documentation contains the full specification for vertex textures (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/assemblylanguageshaders/vertexshaders/vertextextures.asp).

Here are the most important facts:

- Vertex Shader 3.0 (the vs_3_0 compilation target) supports texture fetches using the **`texldl`** texture instructions.
- Four texture samplers are supported:
  - `D3DVERTEXTEXTURESAMPLER1`
  - `D3DVERTEXTEXTURESAMPLER2`
  - `D3DVERTEXTEXTURESAMPLER3`
  - `D3DVERTEXTEXTURESAMPLER4`
- Vertex texture functionality is identical to ordinary pixel textures, except for the following restrictions:
  - Bilinear or trilinear filtering is not supported directly in hardware (though it can be implemented in the vertex shader)
  - Anisotropic filtering is not supported directly in hardware.
  - Rate of change information (that is, automatic mipmap level of detail calculation) is not available.
- A new register set **`(s0..s3)`** has been introduced to represent texture samplers in vertex shaders.
- The **`MaxVertexShader30InstructionSlots`** member value of the **`D3DCAPS9`** limits the number of texture instructions and the **`MaxVShaderInstructionsExecuted`** member value limits the number of total vertex shader instructions, including the number of texture fetches.
- DirectX 9 also supports vertex textures in software vertex processing mode, so an application could use vertex textures even on hardware without native vertex texture support.
- GeForce 6800 supports **`D3DFMT_R32F`** and **`D3DFMT_A32B32G32R32F`** texture formats (2D, cubemap, volume) for vertex texturing.

OpenGL

Vertex texture lookups are available in OpenGL via the **NV_vertex_program3** extension, the specification for which is available here:

http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_vertex_program3.txt

This extension is implemented as an option to the standard ARB vertex program language. This means that you can use the existing ARB API calls to load programs and set parameters. Vertex programs wishing to use the added functionality only have to add the following line to the beginning of the code:

```
OPTION NV_vertex_program3;
```

Adding Vertex Textures to an Application

To use vertex textures, an application needs to do the following:

- Check hardware capabilities for availability of vertex textures
- Create vertex texture resources
- Add necessary code to the vertex shader

The following sections explain how to perform each of these steps in both DirectX and OpenGL.

Checking Hardware Capabilities

DirectX 9

You first need to check the capability bits ("caps bits") of DirectX to see if your GPU supports Shader Model 3.0. Based on that, you may have to fall back to software vertex processing if there isn't support for the features you need.

An application can query the supported formats for vertex textures by calling **IDirect3D9::CheckDeviceFormat** with the **D3DUSAGE_QUERY_VERTEXTEXTURE** flag. Software vertex processing supports all texture formats.

OpenGL:

In OpenGL, all you need to do is check for the presence of the **NV_vertex_program3** extension. If you are using the GLUT library, the **glutExtensionSupported** function can perform that check. The number of supported vertex texture images is queried as follows:

```
glGetIntegerv(MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB, &vtex_units);
```

GeForce 6 Series GPUs support a maximum of four active vertex textures. However, these four textures can be referenced as many times as you like, up to the maximum instruction limit.

Creating Resources

DirectX 9

An application can create vertex textures using **IDirect3D9::CreateTexture**, **IDirect3D9::CreateCubeTexture**, **IDirect3D9::CreateVolumeTexture**, or any texture creation function from the D3DX library.

A vertex texture must be created in the scratch pool (**D3DPOOL_SCRATCH**) for use with software vertex processing.

OpenGL

Vertex textures are bound using the standard texture calls, using the **GL_TEXTURE_2D** texture targets. Currently only the **GL_LUMINANCE_FLOAT32_ATI** and **GL_RGBA_FLOAT32_ATI** formats are supported for vertex textures. These formats contain a single or four channels of 32-bit floating point data, respectively. Be aware that using other texture formats or unsupported filtering modes may cause the driver to drop back to software vertex processing, with a commensurate drop in interactive performance.

Here is some sample code for loading and binding a vertex texture:

```
GLuint vertex_texture;
glGenTextures(1, &vertex_texture);
glBindTexture(GL_TEXTURE_2D, vertex_texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST_MIPMAP_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE_FLOAT32_ATI, width, height, 0,
GL_LUMINANCE, GL_FLOAT, data);
```

Accessing Vertex Textures in the Vertex Shader

DirectX 9:

An application sets vertex textures via the **IDirect3DDevice9::SetTexture** call, with one of **D3DVERTEXTEXTURESAMPLER1** through **D3DVERTEXTEXTURESAMPLER3** as a sampler index. The vertex textures created in the default pool (**D3DPOOL_DEFAULT**) can also be set as a pixel texture.

In a vertex shader the samplers must be declared using the **dcl_samplerType** instruction (just as in ps_2_0 and ps_3_0 pixel shaders).

```
// ASM example

dcl_texcoord0 v0
dcl_2D        s0

texldl r0, o0, s0
```

```
// HLSL / Cg example

sampler2D tex;

vDisplacement = tex2Dlod ( tex, t );  // t.w selects the mipmap level
```

OpenGL:

Texture lookups are performed in an OpenGL vertex program using the TEX, TXB, TXL or TXP instructions, just like in a fragment program. (Or in a high-level language such as Cg, as shown in the code sample above.) A significant difference from fragment programs texturing is that vertex texture lookups do not automatically calculate level of detail.

Level of detail is a measure of how magnified or minified the texture image is on the screen. It is normally calculated based on the rate of change of the texture coordinates from pixel to pixel. Since vertex textures are only accessed at the vertices there is no easy way for the hardware to calculate this value. If you want to use mipmapped textures you will have to calculate the LOD yourself in the vertex program and use the TXL instruction.

Here is an example ARB vertex program that illustrates how to perform a simple texture lookup:

```
!!ARBvp1.0
OPTION NV_vertex_program3;
PARAM mvp[4] = { state.matrix.mvp };
PARAM texmat[4] = { state.matrix.texture };
PARAM scale = program.local[0];
TEMP pos, displace;
// vertex texture lookup
TEX displace, texcoord, vertex.texcoord, 2D;
// try and do as much work here as possible that isn't
// dependent on the texture lookup, to cover latency
MOV result.texcoord[0], texcoord;
MOV pos.w, 1.0;
// scale vertex along normal
MUL displace.x, displace.x, scale;
MAD pos.xyz, vertex.normal, displace.x, vertex.position;
 // transform position to clip space
DP4 result.position.x, mvp[0], pos;
DP4 result.position.y, mvp[1], pos;
DP4 result.position.z, mvp[2], pos;
DP4 result.position.w, mvp[3], pos;
END;
```

Mipmaps

Similar to conventional pixel shader textures, mipmapping can provide significant performance and image quality benefits for vertex textures. Since vertices don't have screen space partial derivatives (because pixel-level information is not yet available this early in the graphics pipeline), the default level of detail (LOD) for all vertex

texture instructions is 0. To use mipmaps we need to calculate the LOD inside the shader. For example:

```
#define maxMipLevels 10.0f

Out.HPOS = mul( ModelViewProj, vPos );
float mipLevel = ( Out.HPOS.z / Out.HPOS.w ) * maxMipLevels;
float vDisplacement = tex2Dbias( tex, float4( t, mipLevel, mipLevel );
```

This code bases the level of detail on the vertex depth, which is an inexpensive, yet useful approximation. You can also modify this technique further by taking the fractional part of `mipLevel` and using that to interpolate between mipmap levels:

```
#define maxMipLevels 10.0f

Out.HPOS = mul( ModelViewProj, vPos );

float mipLevel = ( Out.HPOS.z / Out.HPOS.w ) * maxMipLevels;

float mipLevelFloor = floor(mipLevel);
float mipLevelCeiling = mipLevelFloor + 1;
float mipLevelFrac = frac(mipLevel);

float vDisplacementFloor = tex2D( tex, float4( t, mipLevelFloor,
mipLevelFloor );

float vDisplacementCeiling = tex2Dbias(tex,
float4(t,mipLevelCeiling,mipLevelCeiling );

float vDisplacement = vDisplacementFloor + vDisplacementCeiling
```

Filtering

Texture filtering is allowed with vertex texturing but the available filter types depend on hardware (or reference rasterizer) support. GeForce 6 Series hardware only supports the nearest-neighbor filtering mode, but you can implement your own filtering functionality in the vertex shader.

Bilinear Filtering

```
#define textureSize 512.0f
#define texelSize 1.0f / 512.0f

float4 tex2D_bilinear( uniform sampler2D tex, float2 t )
{
  float2 f = frac( t.xy * textureSize );

  float4 t00 = tex2D( tex, t );
  float4 t10 = tex2D( tex, t + float2( texelSize, 0.0f );

  float4 tA  = lerp( t00, t10, f.x );
```

```
  float4 t01 = tex2D( tex, t + float2( 0.0f, texelSize ) );
  float4 t11 = tex2D( tex, t + float2( texelSize, texelSize ) );

  float4 tB  = lerp( t01, t11, f.x );

  return lerp( tA, tB, f.y );
}
```

Bilinear Filtering With Mipmapping

```
float4 tex2D_bilinear( uniform sampler2D tex, float4 t )
{
  float2 f = frac( t.xy * miplevelSize );

  float4 t00 = tex2Dbias( tex, t );
  float4 t10 = tex2Dbias( tex, t + float4( texelSize, 0.0f, 0.0f, 0,0f );

  float4 tA  = lerp( t00, t10, f.x );

  float4 t01 = tex2Dbias( tex, t + float4( 0.0f, texelSize, 0.0f, 0.0f ) );
  float4 t11 = tex2Dbias( tex, t + float4(texelSize, texelSize, 0.0f, 0.0f));

  float4 tB  = lerp( t01, t11, f.x );

  return lerp( tA, tB, f.y );
}
```

Since neighboring texels are in most cases present in the cache, bilinear filtering is reasonably efficient from a performance standpoint. If you need only one component from the texture, you can try using a 4-component texture with the four neighboring samples packed into the four components for bilinear filtering.

Bicubic, trilinear, and other types of the filtering can be performed with the vertex shader. However, trilinear filtering takes a higher performance hit because the shader needs to access texels from different mipmap levels.

Performance Tips

The vertex processor on the GeForce 6800 is capable of processing on more than 600 million vertices per second! Of course, this number is a result of simply doing the minimum amount of work on the vertex processor. More interestingly, what is the maximum vertex throughput if vertex texture fetch is used? For a basic displacement mapping shader with nearest filtering, we measured 33 million displaced vertices per second.

33 million displaced vertices per second translates to more than a million displaced vertices per frame at 30 frames per second. That's more vertices per frame than just about any currently shipping game. Furthermore, typically not all vertices in a frame need to be displaced.

However, you can do even better. The aforementioned scenario assumes that you are querying the vertex texture for each vertex. In practice, you can take advantage of the efficient branching implementation in the GeForce 6 Series GPUs to test if a vertex texture fetch is necessary. For example, you may be able to do a V dot N test to see if a particular vertex is close to a silhouette or not. That way, you could avoid displacing vertices that are not along silhouettes. You could then use the performance you saved to do filtering on the other vertices.

Another advantage of the dynamic branching in the vertex shaders is the possibility of early-out. Since graphics hardware culls vertices only after they have been processed by the vertex-shader, the vertex shader could be doing a lot of work for vertices that ultimately get thrown away. Thus, an expensive vertex shader should test early on if a vertex is going to be clipped, and if so early-out via a dynamic branch. Here is an example of clipping on the GPU:

```
// OpenGL example
float4 vClipPos = mul( ModelViewProj, vPos );
float3 bClip = abs( vClipPos.xyz ) < ( vClipPos.www + vClipOffset );

if( all(bClip) )
{
  DoLightingAndDisplacement();
}
```

Vertex Textures as Constant Memory

Since vertex texture reads are much slower than constant reads (see previous section), we strongly advise against using vertex textures as constant memory. The pixel processing architecture in the GPU is highly optimized to hide texture fetch latency, but the vertex shader is not nearly as efficient. Therefore, you should limit your vertex texture fetches to a small number of coherent accesses per vertex.

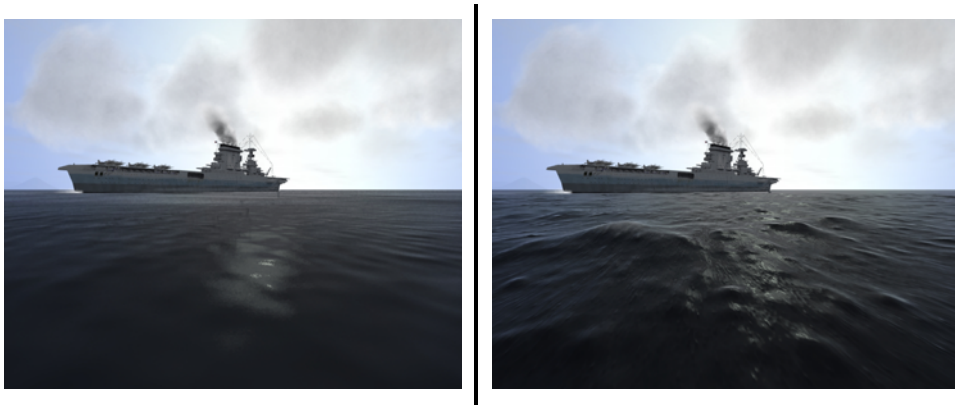
Case Study

Game developers are already adopting vertex textures for projects currently in development. In this section, we take a look at one such game, called *Pacific Fighters*, developed by 1C:Maddox Games and published by Ubi Soft Entertainment.

When considering displacement mapping in modern games, flight simulators often make excellent candidates. This is because they contain massive outdoor environments with terrain, rivers, and oceans – all of which can benefit from displacement mapping. In this section, we look at a case study of a development team who integrated displacement mapping into their flight simulator.

IL-2 Sturmovik from Ubisoft and 1C:Maddox Games is one of the most spectacular and modern fly simulators of recent years. The game's developer, 1C:Maddox Games, has always striven to incorporate the latest technologies to be on razor's edge of the game industry. Their latest product, called *Pacific Fighters*, takes full advantage of the GeForce 6 Series family of GPUs. "The ability to use textures in the vertex shader is one of the most awaited features of 3D accelerators," said Yuri Kryachko, lead 3D programmer of *Pacific Fighters*.

Because the game's environment has expansive ocean surfaces, 1C:Maddox Games used vertex textures to create one of the most realistic water surfaces in the game industry. Before using vertex textures, the developer used dynamic bump mapping to visualize waves but this technology did not provide anywhere near the level of realism that vertex textures and geometry displacement added to the water surface. The screenshots on the right of Figure 2 show the vast increase in realism achieved by adding displacement mapping to the engine.



Without Displacement Mapping | With Displacement Mapping

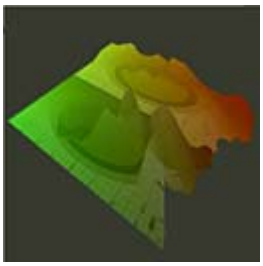
Figure 2. The Visual Effect of Displacement Mapping

The water shaders in *Pacific Fighters* are very complex (more than 140 instructions long) and used to calculate physically corrected water reflection, refraction and waves animation. They combine multiple dynamic normal maps to calculate the appropriate geometric displacement at each vertex. In addition, the shader uses multiple texture fetches to perform filtering, which results in higher visual quality.

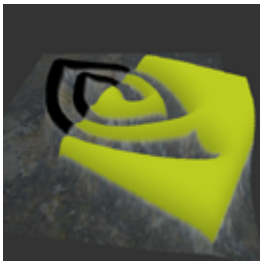
Yuri Kryacko explains, "Performance dramatically improved by using dynamic branching both in pixel and vertex shaders. We plan to optimize and improve visual quality of other parts of project, using vertex textures, new shaders, instancing, and CPU-free occluders to push our graphics engine to the next generation of realism."

Downloads

Learn more about vertex texture fetch by downloading our examples:



The "paint_sculpt" FX Composer effect uses the new features of PS_3_0 and VS_3_0 to allow you to interactively sculpt geometry in a paint-like application. paint_sculpt paints on any 3D surface with valid texture coordinates, and then the result of your painting actions are read back into that object via vertex texture fetch. The values then are applied to the surface geometry to permit you to freely sculpt any painted displacement you like in real time.



The “simple vertex texture” OpenGL example demonstrates the use of the NV_vertex_program3 extension to perform texture look-ups in a vertex program. It uses this feature to perform simple displacement mapping. The example also shows how to implement bilinear filtering of vertex texture fetches.

These examples, and hundreds of others, are available at:

http://download.nvidia.com/developer/SDK/Individual_Samples/samples.html

http://download.nvidia.com/developer/SDK/Individual_Samples/effects.html

Notice

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com