2.2

Deferred Lighting on PS 3.0 with High Dynamic Range

Dean Calver

Introduction

Deferred lighting is a technique increasingly being used for real-time rendering. With the new capabilities of PS 3.0 and high-precision blending we can easily move to high-dynamic range. This article also covers high dynamic range tone-mapping for PS 3.0 in detail, using a GPU-efficient method. The technique is independent of the method of creating the buffer that needs to be tone-mapped, and therefore can also be used with conventional rendering techniques.

Overview

The use of geometry buffers (G-Buffers) to store the lighting parameters for post-processing lighting in real-time has been used in PC hardware since the introduction of PS 1.1. Although the fixed point precision and short pixel shaders meant that only the simplest lighting models could be used, the constant cost for lighting was already attractive enough to be used in several games (particularly on the Xbox where the lower-level access to the GPU allowed more functionality to be extracted from the relatively limited hardware). PS 2.0, floating-point render targets, and greater internal pixel shader precision considerably increased the usefulness of the technique but the lack of dynamic branching and the inability to use alpha blending meant there was still some way to go.

The introduction of graphic hardware with PS 3.0 support and blending to float surfaces has opened up new ways of working with deferred lighting, including HDR.

Implementation

This article assumes you have a working PS 2.0 deferred lighting system, as the article is built upon an existing article published on the Internet [Calver02]. Another good article that covers the general implementation of deferred shading and lighting in detail is "Deferred Shading with Multiple Render Targets" [Thibieroz02].

To recap, a deferred lighting system breaks the render pipeline into three major phases:

1. Geometry phase
2. Lighting phase
3. Post-processing phase

The geometry phase deals with filling the G-Buffer with parameters and except for the ability to support multiple light models, is largely the same as before. The lighting phase changes significantly in order to accommodate multiple lighting models and output results into a much higher precision buffer. Finally, the post-processing phase has the responsibility of tone mapping the high dynamic range into a low fixed range capable of being displayed on a monitor.

Multiple Light Models

With the length of PS 3.0 shaders and dynamic branching we can now allow multiple light models in the lights shaders. This dramatically increases the variety of surfaces through the high number of different code paths possible. However there are a number of caveats that make this simple process more involved due to the differences in how a GPU handles dynamic branching.

A GPU is designed to work in parallel and for pixel shaders the basic unit of operation is usually a quad of 2×2 pixels running in lock step with each other. The main reason for this structure is to allow the computation of gradients of any variable in a pixel shader by just taking the difference between the value of its neighbors at any point.

Since any pixel in a 2×2 quad can take different branches, dynamic branching breaks lock stepping and therefore breaks the method used to calculate gradients. As such, any operation that requires gradient calculation cannot work inside a dynamic branch. Not only does this affect obvious gradient instructions like DDX and DDY, but also it affects all texture loads since gradients are used internally for determining which MIP map level to use. Therefore to make texture loads useable inside a dynamic branch this value must be passed explicitly. A new HLSL instruction called tex2Dlod achieves this purpose.

As we generally won't be using MIP maps in light shaders, this limitation does not prove to be much of a difficulty as we can simply pass 0 as the LOD parameter to the function to retrieve data from the top level.

The basic approach consists of storing a light model ID in a channel of the geometry buffer that selects the light model to use for this pixel. Then each light retrieves this value and uses a switch/case statement to execute the desired code for this shader.

Unfortunately PS 3.0 does not support enough functionality to allow fast switch/case statements. So we have to use a series of "if" statements to select the correct light model, which unfortunately means we have to be frugal with the number of different light models supported. Ideally we would use a jump table but this isn't yet possible (if future architecture allows indexed jumps then this should be used). Care should be taken as all compares are floating point and therefore exact equality isn't guaranteed. One possibility is to store the light models IDs exactly in the floating point mantissa; these will then compare exactly, with the only restriction being the limited number of models supported. As `float16` has ten bits for the mantissa (giving

1024 IDs) this is unlikely to be a problem in this context. The approach used here is a simple epsilon on compares for simplicity.

```
const float FloatLMEpsilon    = 1e-5f;
const float SpecialLightModel = 1.0f;
const float ExtraLightModel   = 2.0f;

PixelShader( ... )
{
   float lmID = GetLightModelIDFromGBuffer();

   if( lmID < (SpecialLightModel - FloatLMEpsilon) )
   {
          // lmID is between 0 and (1-epsilon)
          // we have to use tex2Dlod in the if statement
          float4 param = tex2Dlod( texture, uv, 0 );
   } else if( lmID < (ExtraLightModel - FloatLMEpsilon) )
   {
          // lmID is between (1-epsilon) and (2-epsilon)
          // we have to use tex2Dlod in the if statement
          float4 param = tex2Dlod( texture2, uv, 0 );
   } else
   {
          // lmID is above  (2-epsilon)
          // we have to use tex2Dlod in the if statement
          float4 param = tex2Dlod( texture3, uv, 0 );
   }
}
```

HDR

High dynamic range rendering pioneered by Debevec [Debevec97] has quickly become the de facto standard for both offline and cutting edge real-time systems. The principle is simple: all rendering should be performed at a high range and only at the end of the graphics pipeline should this be reduced into a range that fits the output gamut. This last stage is known as tone-mapping and usually takes the form of an approximation of the eyes' response to light.

The principal problem with supporting HDR in real-time graphics hardware has been the lack of render targets with enough precision. With the support of float render targets brought in by the PS 2.0 generation of cards (NVIDIA, GeforceFX, and ATI 9500+) it looked like this problem was solved, but quickly another problem emerged: cards of this generation are unable to blend to these high-precision targets, and without alpha blending the only legal option is ping-ponging between render-targets.

It is possible to set a texture as both an input texture and a render-target at the same time but many graphics cards do not have a way of synchronizing the texture and render-target caches and visual anomalies can result. As such this is defined as an illegal set-up under Direct3D and OpenGL.

Ping-ponging consists of setting two render textures; one as input and one as output, and manually performing the blend inside the shader by reading one texture and writing out the result of the blend to the other. After every alpha blend the input and output are swapped and the process is repeated. While this works, the amount of copying and number of render state changes can cause a considerable impact on performance (the NVIDIA NV40 chip is the first to support full alpha blending to a `float16` buffer, eliminating the expensive ping-pong process completely).

The following code is used to check whether the graphic hardware supports a `D3DFMT_A16B16G16R16F` texture as a render target *and* has support for post pixel shader blending with this format.

The DirectX 9 API call

```
D3D->CheckDeviceFormat( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                        DisplayFormat,
                        D3DUSAGE_QUERY_POSTPIXELSHADER_BLENDING |
                        D3DUSAGE_RENDERTARGET,
                        D3DRTYPE_TEXTURE,
                        D3DFMT_A16B16G16R16F );
```

will return `D3D_OK` if the card has the necessary support.

If both capabilities are supported we create a `D3DFMT_A16B16G16R16F` render target and let every light accumulate into it using standard additive blending. Using a `float16` buffer allows us to have a large range of lights, from incredible dark spaces (which the tone-mapping will brighten) to lights (like the sun) thousands of times brighter than normal, without worrying about over/under flow.

When the lighting phase has finished, the HDR destination lighting buffer has to be post-processed via tone-mapping.

Tone-Mapping

Tone-mapping is required to work around the low dynamic range nature of the display unit. Normally the human visual system adapts to the high dynamic range in the real world via a method known as visual adaptation [Durand02]. Because the monitor itself is unable to produce high enough dynamic range to trigger this visual adaptation we must do it ourselves before outputting the image to the display unit.

There are a number of separate processes we can simulate to implement tone-mapping, including dark adaptation, light adaptation, and chromatic adaptation. The model we choose to implement here is a simple model of light adaptation. Light adaptation is the fast change that happens when going from a lit environment to a dark one.

The method used here is basically the operator presented in "Photographic Tone Reproduction for Digital Images" [Reinhard02] but missing the dodging and burning feature. This method is easy to implement in real-time but further research in this field might have already been conducted by the time you read this. This technique only models luminance changes so the color of the scene is converted into luminance before use.

The key to this algorithm is to calculate the average luminance of the scene at full dynamic range and then use it and the white point to scale the pixel value. The white point is by default the maximum luminance in the scene.

Calculating $\bar{L}_w$

Equation (2.2.1) calculates the average luminance in world luminance (high dynamic range).

$$\bar{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\log(\delta + L_w(x,y))\right) \tag{2.2.1}$$

Where N is the number of pixels, x,y are the 2D coordinates of each pixel, δ is a small constant number and L_w is the luminance function.

This equation doesn't translate easily to GPU operations as it needs to read every pixel in the frame buffer, and even current Pixel Shader 3.0 hardware doesn't allow us to implement this in a single pass. It does however allow us to reduce it to only two passes in many cases. Potentially future Pixel Shader 3.0 hardware will support more instructions which could reduce this even further to a single pass. The single pass version is an easy modification from the version provided here and source is provided in the sample program running on the reference driver.

δ is a tiny epsilon to stop log(0) being introduced at black pixels.

In the first pass we create a small destination render target and draw a quad covering the entire render target. At each pixel on the destination surface the pixel shader sums a vast number of luminance values from the sources and stores the result. The second pass then sums this much smaller buffer and runs the final part of the calculation producing a texture containing $\bar{L}_w$.

Pixel Shader 3.0 guarantees 512 actual instructions, but the total number of instructions executed is variable; current hardware allows loops up to 65,535 instructions to be executed in a single shader. The number of instructions we have to execute per pixel read wholly decides how many passes we have to perform.

We drop back to pixel shader assembler for this shader, as this is such an expensive operation that we want total control and an exact count of instructions used.

To get a single log luminance sample we need to execute $logL_w = \log(\delta + L_w(x,y))$, using Equation 2.2.2 to convert RGB to luminance:

$$L = 0.27R + 0.67G + 0.06B \tag{2.2.2}$$

The pixel shader 3.0 assembler code for the summation with x = i1, y = i2 is then:

```
// v0.xy = start uv co-ordinate for this block      Instruction count
// c0.z = 0
// c1.xyzw = xy = delta uv for each pixel, z = 0, w = 1
// s0 = frame buffer sampler
// c3 = 0.27, 0.67, 0.06, sqrt(delta)
mov r0.xy, v0.xy // set uv to start of this block                  1
mov r0.zw, c0.z  // clear accumulator                             1
mov r1.w, c3.w   // set w = sqrt(delta) so r1.w * r1.w = delta    1
mov r2.w, c1.x   // optimise the inner loop                       1

rep i2           // loop across y                                 3
    rep i1       // loop across x                                 3
        texld r1.xyz, r0, s0  // get the pixel from the framebuffer 1
        dp4 r2.x, r1.xyzw, c3.xyzw  // delta + L(x,y)            1
        log r2.x, r2.x        // log(x)                          1
        add r0.xw, r0.xwww, r2.wxxx // Inc u and sum log-luminance 1
    endrep       // end x loop                                    2
    mad r0.xy, r0.xy, c1.zw, c1.zy  // increment v and set u to 0 1
endrep           // end y loop                                    2
mov oC0.xyzw, r0.wwww        // output accumulator                1
```

giving us nine cycles per x pixel with another six cycles to increment a y line.

If we calculate luminance in blocks of 128 × 56 pixels by setting i1 = 128 and i2 = 56 this code consumes 64,853 cycles. For a 1280 × 1024 frame buffer, we would need to sample into 10 × 19 destination textures which then would be read by the next pass of the algorithm.

We can optimize this further by noticing that the loops themselves are the most expensive part. By unrolling the x loop as much as memory allows and removing the x loop we can greatly speed this code up. In theory the shader compilers inside the driver should do this automatically where appropriate, but here we do it manually just in case it's missed.

By removing the x loop the code for each log luminance sample goes from nine cycles to four cycles. We can repeat it 125 times and still fit in instruction limits. Unfortunately the pixel shader assembler doesn't have a repeat block macro, so you will have to do it manually using cut and paste or using your own code creator.

```
// v0.xy = start uv co-ordinate for this block      Instruction count
// c0.z = 0
// c1.xyzw = xy = delta uv for each pixel, z = 0, w = 1
// s0 = frame buffer sampler
// c3 = 0.27, 0.67, 0.06, sqrt(delta)
mov r0.xy, v0.xy  // set uv to start of this block                 1
mov r0.zw, c0.z   // clear accumulator                            1

mov r1.w, c3.w    // set w = sqrt(delta) so r1.w * r1.w = delta    1
mov r2.w, c1.x    // optimise the inner loop                       1
```

```
rep i2          // loop across y                                    3
    BEGIN_REPEAT_BLOCK(125) // imaginary macro to physically repeat
                            // the code 125 times
    texld r1.xyz, r0, s0      // get the pixel from the frame buffer1
    dp4 r2.x, r1.xyzw, c3.xyzw// delta + L(x,y)                     1
    log r2.x, r2.x            // log(x)                             1
    add r0.xw, r0.xwww, r2.wxxx    // Inc u and sum log-luminance   1
    END_REPEAT_BLOCK               // total cost of block 500
    mad r0.xy, r0.xy, c1.zw, c1.zy // increment v and set u to 0    1
endrep          // end y loop                                       2
mov oC0.xyzw, r0.wwww       // output accumulator                  1
```

Each y loop takes 506 instructions so we can work on 125 × 128 pixel block taking 64,773 cycles in total, writing into an 11 × 8 render target. Loop unrolling will cause problems at the right edge of the texture; one possible solution is to have a non-unrolled version for the last column, but another is to use `BORDER` mode texture addressing to return a luminance of 0. This will introduce a tiny error (0 actually equals delta) but it shouldn't be visible in practice.

This optimization is largely theoretical at this stage. While it is more efficient in terms of cycles, the actual gains will be graphic chipset-specific as this stresses the texture fetch units and pixel threading system which can vary greatly across different implementations (it may be more efficient for a specific chip to use different numbers and methods).

The second pass takes this smaller texture and sums each pixel, then completes the calculation. N is set to the number of samples from the original destination light buffer. This pass is so small compared to the last pass, that for most frame buffer sizes this can be completely unrolled and still fit within 512 instructions.

This second pass doesn't perform the dot product or the logarithm operations as they have already been done on the previous pass.

```
// c0.x = 1/N
// c0.z = 0
// c1.xyzw = xy = delta uv for each pixel, z = 0, w = 1
// s0 = frame buffer sampler
mov r0.xy, v0.xy      // set uv to start of this block
mov r0.zw, c0.z       // clear accumulator
mov r2.w, c1.x        // u delta
BEGIN_REPEAT_BLOCK(ysize)          // ysize = 8 in the example above
    BEGIN_REPEAT_BLOCK(xsize)      // xsize = 11 in the example above
    texld r2.x, r0, s0             // get the pixel from the last pass
    add r0.xw, r0.xwww, r2.wxxx    // Sum log-luminance and inc u
    END_REPEAT_BLOCK               // end x loop
    mad r0.xy, r0.xy, c1.zw, c1.zy // increment v and set u to 0
END_REPEAT_BLOCK      // end y loop
mul r0.x, r0.w, c0.x // multiple by 1/N
exp r0.x, r0.x        // and convert back into a luminance value
mov oC0.xyzw, r0.xxxx
```

Using $\bar{L}_w$

The equation suggested in [Reinhard02] is used:

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{\text{white}}^2}\right)}{1 + L(x,y)} \tag{2.2.3}$$

Where $L_d(x, y)$ is the tone mapped pixel and L_{white} is the white point, which is the HDR value that causes complete white out. L_{white} can be set to infinity (or a very large value) to bring all luminance values produced into a displayed value. $L(x,y)$ is produced via Equation 2.2.3:

$$L(x,y) = \frac{a}{\bar{L}_w} L_w(x,y) \tag{2.2.4}$$

with a being the Key Value (basically a tuneable magic number between 0 and 1, the default suggested in the paper is 0.18) and L_w and $\bar{L}_w$ being the same as the previous operations.

Equation 2.2.2 calculates the new monochromic luminance value for a pixel. To color it, we implement a simple scale of the original pixel color into the new scale.

The following HLSL shader is used:

```
Sampler TexToToneMap;
sampler BarLwTexture;
float4 main( in float2 uv : TEXCOORD0,
       uniform in float RecipA,
       uniform in float RecipLwhiteSqrd )
{
   // get average Lw from the texture we calculated previously
   float BarLw = tex2D( BarLw, float2(0,0) );
   float aOverBarLw = KeyA * (1.f/BarLw);

   // calc L(x,y) by getting the Luminance of this pixel and
   // scaling by a/BarLw
   float4 hdrPixel = tex2D( TexToToneMap, uv );
   float lumi = dot( hdrPixel, float3( 0.27, 0.67, 0.06 ) );
   float Lxy = aOverBarLw * lumi;

   // now calculate Ld
   float numer = (1 + (Lxy * RecipLwhiteSqrd)) * Lxy;
   float denum = 1 + Lxy;
   float Ld = numer / denum;

   // we now have to convert the original  color into this range.
   float3 ldrPixel = (hdrPixel / lumi) * Ld

   // currently don't process hdr alpha just saturate on the output
to LDR
   return float4(ldrPixel, hdrPixel.a);
}
```

The render target this shader outputs into is a low dynamic range surface capable of being output to the display unit.

Conclusion

Pixel Shader version 3.0 enables us to reduce the number of render target switches to implement tone-mapping and float16 blending provides us the ability to use HDR without restriction or limitation. Adding these capabilities to a deferred lighting renderer allows us to get even closer to off-line renderer quality.

References

[Calver02] D. Calver, "Photorealistic Deferred Lighting," available online at *http://www.beyond3d.com/articles/deflight/*

[Thibieroz02] "Deferred Shading with Multiple Render Targets," *ShaderX² Tips and Tricks,* Wordware Publishing 2002.

[Debevec97] P. Debevec, "*Recovering High Dynamic Range Radiance Maps From Photographs,*" Proceedings of SIGGRAPH 97, available online at *http://www.debevec.org/Research/HDR/debevec-siggraph97.pdf.*

[Durand02] F. Durand et al., "*Interactive Tone Mapping,*" Proceeding of the Eurographics Workshop on Rendering 2002, available online at *http://graphics.csail.mit.edu/~fredo/PUBLI/EGWR2000/durandInteractiveTM.pdf.*

[Reinhard02] E. Reinhard et al., "*Photographic Tone Reproduction for Digital Images,*" ACM, available online at *http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf.*