HDRFormats10 Sample

Collapse All

High dynamic range lighting effects require the ability to work with color values beyond the 0 to 255 range, usually by storing high range color data in textures. Floating point texture formats are the natural choice for HDR applications, but they may not be available on all target systems. This sample shows how high dynamic range data can be encoded into integer formats for compatibility across a wide range of devices.

Path

Source	*SDK root*\Samples\C++\Direct3D10\HDRFormats10
Executable	*SDK root*\Samples\C++\Direct3D10\Bin*x86 or x64*\HDRFormats10.exe

Sample Overview

High dynamic range lighting techniques add realism to your application; furthermore, these techniques are fairly straightforward to implement, integrate well into an existing render pipeline, and require a modest amount of processing time; however, most HDR techniques rely in floating-point texture formats, which may not be available on the target system. One way to enable HDR techniques on devices without floating-point texture support is to encode high range floating-point data into an integer format, decoding on-the-fly as needed. This sample supports both Direct3D 9 and Direct3D 10 codepaths. Examples for Direct3D 10 codepath will be discussed.

Be aware that integer textures are not perfect replacements for floating-point textures; the disadvantages of integer encoding are a loss of precision and/or a loss a range from the floating-point source, plus a performance hit resulting from the encoding/decoding process. This encoding/decoding happens seemlessly within the pixel shaders, so it's not prohibitively expensive but it can cause a slight frame-rate drop. For static elements, such as a skyboxes, the encoding can be done once during initialization, saving some cycles during run-time.

Implementation

The application uses two high dynamic range techniques: tone mapping and blooming. If you are not familiar with how these techniques work or want more information about high dynamic range images in general, look at the HDR Lighting programming guide first, and view the HDRLighting and HDRCubeMap samples included with the DirectX Documentation.

For the techniques used in this sample, the ability to store HDR data in textures is critical. The scene must be rendered to an HDR render target, the environment map texture needs to be HDR, and the temporary textures used to measure scene luminance all need to store HDR data. A device which supports floating-point textures will offer the best performance and simplest implementation, but with a little extra work integer textures can be used to store HDR data. This sample implements three encoding schemes:

RGB16

Using a 16-bit per channel integer format, 65536 discrete values are available to store per-channel data. This encoding is a simple linear distribution of those 65536 values between 0.0f and an arbitrary maximum value (100.0f for this sample). The alpha channel is unused. Here is the formula for decoding the floating-point value from an encoded RGB16 color:

```
decoded.rgb = encoded.rgb dot max_value
```

RGB32

Using a 32-bit per channel integer format gives better range of color than using a16-bit per channel integer formats. 4294967296 discrete values are available to store per-channel data. This encoding is a simple linear distribution of those 4294967296 values between 0.0f and an arbitrary maximum value

(10000.0f for this sample). The alpha channel is unused. The formula for decoding the value from an encoded RGB32 color is exactly the same as it is for an RGB16 color:

```
decoded.rgb = encoded.rgb dot max_value
```

R9G9B9E5

This encoding is more sophisticated than RGB16 and allows for a far greater range of color data by using a logarithmic distribution. Each channel stores the mantissa of the color component, and the alpha channel stores a shared exponent. The added flexibility of this encoding comes at the cost of extra computation. Here is the formula for decoding the floating-point value from an encoded R9G9B9E5 color:

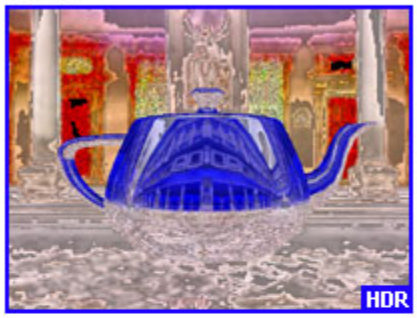
```
decoded.rgb = encoded.rgb * pow( 2, encoded.a )
```

The images below show the textures used to render the scene in top-down order according to when they're used. Images with a blue border contain high dynamic range data, shown here with RGBE8 encoding.

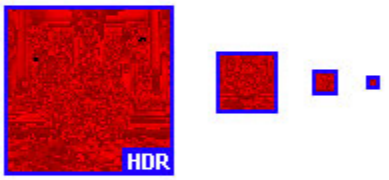
Skybox

Env Map

Scene

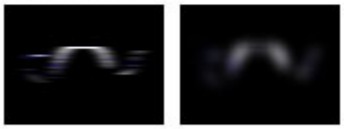


Luminance Calculation



Bright Pass

Bloom Filter



Final

Rendering with HDR

The process of rendering the scene is nearly identical between the native floating-point texture mode and the encoded integer texture modes. Pixel shaders which read from high dynamic range textures simply require an extra decoding step when reading from encoded integer textures and require a similar encoding step when writing HDR data to integer textures. By using uniform arguments to the shader, the same shader can be compiled differently and used by every encoding scheme. For example, here is the pixel shader that lights the model:

```
//--------------------------------------------------------------------------
// Name: ScenePS
```

```
// Type: Pixel Shader
// Desc: Environment mapping and simplified hemispheric lighting
//--------------------------------------------------------------------------------
float4 ScenePS( SceneVS_Output Input,
                uniform bool RGB9E5,
                uniform bool RGB32,
                uniform bool RGB16 ) : SV_TARGET
{
    // Sample the normal map
    float3 bump = g_tNormal.Sample( LinearSampler, Input.Tex );
    bump *= 2.0;
    bump -= float3(1,1,1);

    //move bump into world space
    float3 binorm = normalize( cross( Input.Normal_World, Input.Tangent_World ) );
    float3x3 wtanMatrix = float3x3( binorm, Input.Tangent_World, Input.Normal_World );
    bump = mul( bump, wtanMatrix ); //world space bump

    // Sample the environment map
    float3 vReflect = reflect( Input.ViewVec_World, bump );
    float4 vEnvironment = g_tCube.Sample( LinearSampler, vReflect );

    if( RGB9E5 )
        vEnvironment.rgb = DecodeRGB9E5( vEnvironment );
    else if( RGB32 )
        vEnvironment.rgb = DecodeRGB32( vEnvironment );
    else if( RGB16 )
        vEnvironment.rgb = DecodeRGB16( vEnvironment );

    // Simple overhead lighting
    float3 vColor = saturate( MODEL_COLOR * bump.y );

    // Add in reflection
    vColor = lerp( vColor, vEnvironment.rgb, MODEL_REFLECTIVITY );

    if( RGB9E5 )
        return EncodeRGB9E5( vColor );
    else if( RGB32 )
        return EncodeRGB32( vColor );
    else if( RGB16 )
        return EncodeRGB16( vColor );
    else
        return float4( vColor, 1.0f );
}
```

Normal Map Formats

In the Direct3D 10 codepath, the sample renders the object with a normal map. There are various normal map formats to choose from.

R8G8B8A8

The one in use for most Direct3D 9 generation shaders is the R8G8B8A8 format. In R8G8B8A8 the x, y, and z components of the normal are stored in 8bit R, G, and B channels in the texture. The values are biased, meaning that for each channel, 128 represents the value 0. 0 represents -1, and 255 represents +1.

The Alpha channel remains unused for this sample, but can be used to store height information for effects such as displacement or parallax mapping. Please refer to the DisplacementMapping10 sample for more information.

Floating Point Formats

Floating point normal formats provide more precision than integer formats such as R8G8B8A8. In addition, since floating point textures can be signed, there is no need to bias the texture to represent negative values. However, in this sample, the source of the normal map is already a biased R8G8B8A8 texture, so we still unbias in the shader. The sample simply gains precision from the floating point normal map

formats.

Compressed Formats

New additions to Direct3D 10 are the compressed formats for storing normal maps. These are DXGI_FORMAT_BC5_SNORM and DXGI_FORMAT_BC5_UNORM respectively. They store data in a 0 to 1 range, but offer compression of the normal map. Unlike the HDR formats discussed above, there is no need to decode these formats in the shader. It is already handled by the hardware.

BC5_UNORM textures are unsigned. They store data in the range from 0 to 1. As with the R8G8B8A8 format, the data must be unbiased in the shader to get negative values.

BC5_SNORM textures are signed, meaning that they can store values from -1 to 1. If the normal maps are authored with this in mind, no unbiased needs to be done inside the shader to get negative values. Since the source normal map for the sample is already biased, the sample does not use the BC5_SNORM format to it's fullest.