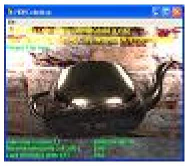
HDRCubeMap Sample

Collapse All

This sample demonstrates cubic environment-mapping with floating-point textures and high dynamic range lighting. In Direct3D 9, floating-point formats have become available for textures. These formats can store color values higher than 1.0, which can make lighting effects more realistic on the environment-mapped mesh when the material absorbs part of the light.

Note that not all cards support all features for the environment-mapping and high dynamic range lighting techniques.



Path

Source	*SDK root*\Samples\C++\Direct3D\HDRCubeMap
Executable	*SDK root*\Samples\C++\Direct3D\Bin\x86 or x64\HDRCubeMap.exe

Sample Overview

The sample shows off two rendering techniques: cubic environment-mapping and high dynamic range lighting. Cubic environment-mapping is a technique in which the environment surrounding a 3D object are rendered into a cube texture map, so that the object can use the cube map to achieve complex lighting effects without memory intensive lighting calculations. High dynamic range (HDR) lighting is a technique to render highly realistic lighting effects by using floating-point textures and high-intensity lights. The floating-point texture formats are introduced in Direct3D 9. Unlike traditional textures in integer format, floating-point textures are capable of storing a wide range of color values. Because color values in floating-point textures don't get clamped to [0, 1], much like lights in real-world, these textures can be used to achieve greater realism.

The scene that HDRCubeMap renders is an environment-mapped mesh, along with several other objects making up the environment around the mesh. The mesh has a reflectivity setting adjustable from 0 to 1, where 0 means the mesh absorbs light completely, and 1 means it absorbs no light and reflects all. The lights used in the scene consist of four point lights, whose intensity is adjustable by the user.

Implementation

The scene that the sample renders consists of these objects:

- A room mesh. This includes the walls, floor, and ceiling.
- Four small sphere meshes representing light sources.
- A mesh in the center of the room with the environment map applied to it.
- Two bi-plane meshes. These orbit around the environment-mapped mesh and are reflected on the environment map. They give the visual cue that the environment map is dynamic, and changes as the surrounding environment changes.

When the sample loads, it creates two cubic textures, one in A8R8G8B8 format and the other in A16B16G16R16F format. When rendering, the sample uses one of these to construct an environment map based on the user's choice. This is done to show the visual difference between integer and floating-point textures when used with high dynamic range lighting. A stencil surface with a size equal to the size of a cube texture face is also created at this time. This stencil surface will be used as the stencil buffer when the sample renders the scene onto the cube texture. The sample also creates an effect object that contains all of the shaders and techniques needed for rendering the scene. The camera used in the sample is a CModelViewerCamera. This camera always looks at the origin, where the environment-mapped mesh is. The user can move the camera position around the mesh and rotate the mesh with mouse control. The camera object computes the view and projection matrices for rendering, as well as the world matrix for the environment-mapped mesh.

The Rendering Code

The rendering of the sample happens in three functions: RenderSceneIntoCubeMap, Render, and RenderScene, using vertex and pixel shaders.

RenderScene is the function that does the actual work of rendering the scene onto the current render target. This is called both to render the scene onto the cube texture and onto the device backbuffer. It takes three parameters: a view matrix, a projection matrix, and a flag indicating whether it should render the environment-mapped mesh. The flag is needed because the environment-mapped mesh is not drawn when constructing the environment map. The rest of the function is straightforward. It first updates the effect object with the latest transformation matrices, as well as the light position in view space. Then, it renders every mesh object in the scene by using an appropriate technique for each one.

RenderSceneIntoCubeMap's job is rendering the entire scene (minus the environment-mapped mesh) onto the cube texture. First, it saves the current render target and stencil buffer, and sets the stencil surface for the cube texture as the device stencil buffer. Next, the function iterates through the six faces of the cube texture. For each face, it sets the appropriate face surface as the render target. Then, it computes the view matrix to use for that particular face, with the camera at the origin looking out in the direction of the cube face. It then calls RenderScene with bRenderEnvMappedMesh set to false, passing along the computed view matrix and a special projection matrix. This projection matrix has a 90 degree field of view and an aspect ratio of 1, since the render target is a square. After this process is complete for all six faces, the function restores the old render target and stencil buffer. The environment map is now fully constructed for the frame.

Render is the top level rendering function called once per frame by DXUT. It first calls FrameMove for the camera object, so that the matrices managed by the camera can be updated. Next, it calls RenderSceneIntoCubeMap to construct the cube texture to reflect the environment for that frame. After that, it renders the scene by calling RenderScene with the view and projection matrices from the camera, and bRenderEnvMappedMesh set to true.

The Shaders

All of the rendering in this sample are done by programmable shaders, divided into three effect techniques: RenderLight, RenderHDREnvMap, and RenderScene.

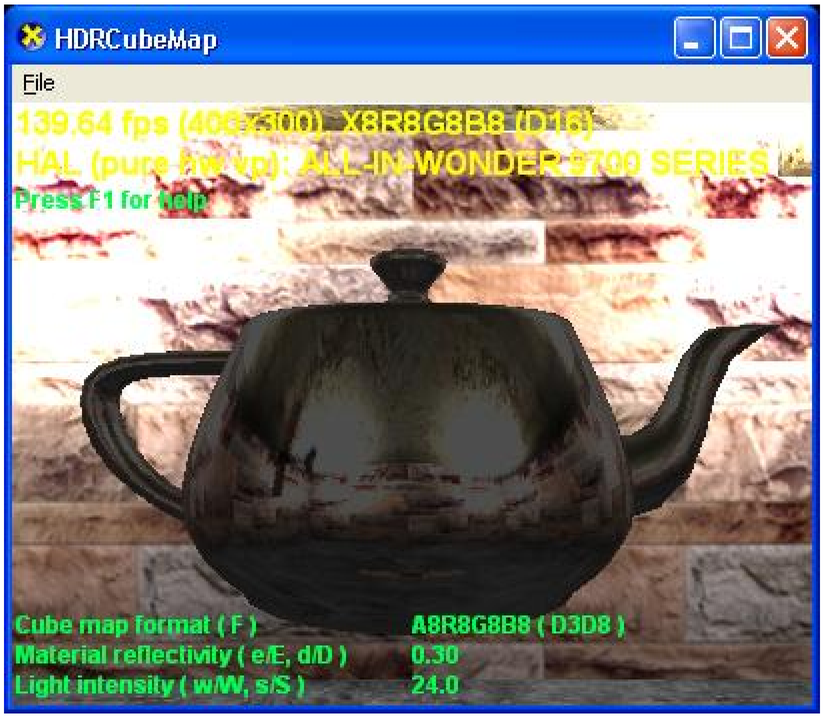
The RenderLight technique is used for rendering the spheres representing the light sources. The vertex shader does a usual world-view-projection transformation, then assigns the light intensity value to the output diffuse. The pixel shader propagates the diffuse to the pipeline.

The RenderHDREnvMap technique is used for rendering the environment-mapped mesh in the scene. Besides transforming the position from object space to screen space, the vertex shader computes the eye reflection vector (the reflection of the eye-to-vertex vector) in view space. This vector is passed to the pixel shader as a cubic texture coordinate. The pixel shader samples the cube texture, applies the reflectivity to the sampled value, and returns the result to the pipeline.

The RenderScene technique is used to render everything else. Objects rendered by this technique employ per-pixel diffuse lighting. The vertex shader transforms the position from object to screen space. Then, it computes the vertex position and normal in view space and passes them to the pixel shader as texture coordinates. The pixel shader uses this information to perform per-pixel lighting. A for-loop is used to compute the amount of light received from the four light sources. The diffuse term of the pixel is computed by taking the dot product of the normal and the unit vector from the pixel to the light. The attenuation term is computed as the reciprocal of the square of the distance between the pixel and the light. These two terms are then modulated to represent the amount of light the pixel receives from one of the light sources. Once this process is done for all lights in the scene, the values are summed and modulated with the light intensity and the texel to form the output.

High Dynamic Range Realism

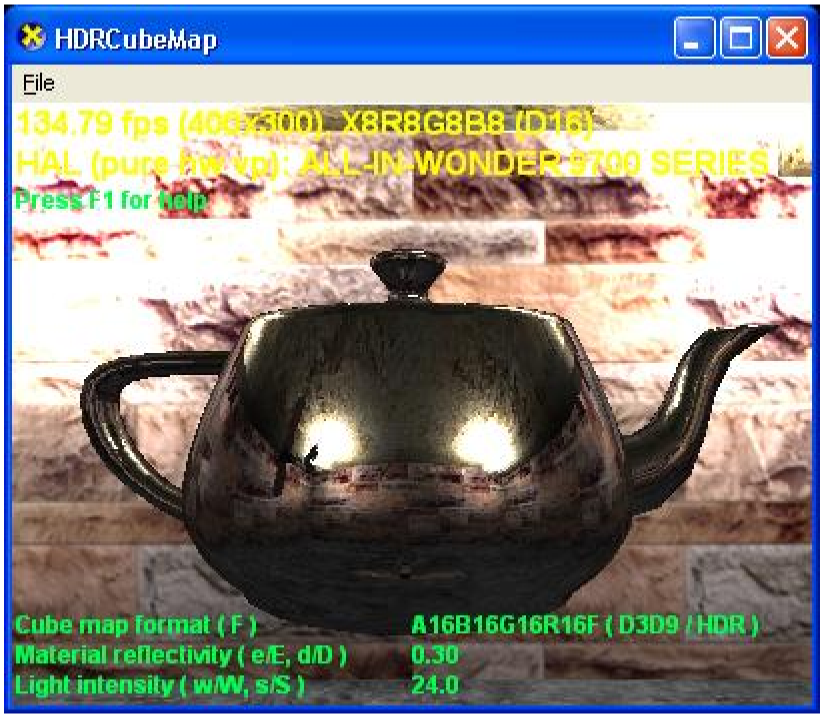
Prior to Direct3D 9, only integer texture formats are supported by Direct3D. When a texture of this type is used, if a color value is higher than 1.0, it is clamped to 1.0 before being written so that the value can fit in an 8-bit integer. This means that when the cube texture is written, its texel color values will have a maximum of 1.0. When this texture is later used for environment-mapping on a material that is not 100 percent reflective (say, only 30 percent reflective), even the brightest texels in the texture will end up showing at 0.3 on the mesh, regardless of how high the original light intensity is. Consequently, realism is compromised. This is demonstrated in the figure below.



139.64 fps (400x300), X8R8G8B8 (D16)
HAL (pure hw vp): ALL-IN-WONDER 9700 SERIES
Press F1 for help

```
Cube map format ( F )                 A8R8G8B8 ( D3D8 )
Material reflectivity ( e/E, d/D )    0.30
Light intensity ( w/W, s/S )          24.0
```

When Direct3D 9's new floating-point cube texture is used with lights whose intensity is higher than 1, the texture is capable of storing color values higher than 1.0, thus preserving true luminance even after the reflectivity multiplication. The figure below shows this.



134.79 fps (400x300), X8R8G8B8 (D16)
HAL (pure hw vp): ALL-IN-WONDER 9700 SERIES
Press F1 for help

```
Cube map format ( F )                 A16B16G16R16F ( D3D9 / HDR )
Material reflectivity ( e/E, d/D )    0.30
Light intensity ( w/W, s/S )          24.0
```

Alternatives

It is possible to achieve a high level of realism in a high dynamic range lighting environment without using floating-point cube textures. An application can accomplish this by applying the material reflectivity multiplier when constructing the cubic environment map. Then, during rendering, the pixel shader will sample the cube texture and output the value straight back to the pipeline. This method works around color clamping by scaling before the clamping instead of after. The drawback of this method is that if the scene has multiple environment-mapped objects with different reflectivity, a different environment map will generally need to be rendered for each of these objects.