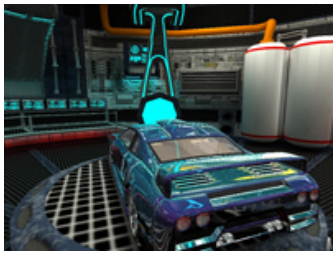
CubeMapGS Sample

Collapse All

The CubeMapGS sample demonstrates rendering a cubic texture render target with a single DrawIndexed() call using two new features in Direct3D 10: render target array and geometry shader. A render target array allows multiple render target and depth stencil textures to be active simultaneously. By using an array of six render targets, one for each face of the cube texture, all six faces of the cube can be rendered together. When the geometry shader emits a triangle, it can control which render target in the array the triangle gets rasterized on. For every triangle that gets passed to the geometry shader, six triangles are generated in the shader and output to the pixel shader, one triangle for each render target.



Path

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

Sample Overview

Environment mapping is a popular and well-supported technique in 3D graphics. Traditionally, dynamic cubic environment maps are created by obtaining a surface for each face of a cube texture and setting that surface as the render target to render the scene once for each face of the cube. The cube texture can then be used to render environment-mapped objects. This method, while it works, increases the number of rendering passes from one to seven, greatly reducing the application frame rate. In Direct3D 10, applications can use geometry shaders and render target arrays to alleviate this problem.

How the Sample Works

A geometry shader in Direct3D 10 is a piece of code that runs for each primitive to be rendered. In each invocation, the geometry shader can output zero or more primitives to be rasterized and processed in the pixel shader. For each output primitive, the geometry shader can also control to which element slice of the render target array the primitive gets emitted.

The render target array in Direct3D 10 is a feature that enables an application to render onto multiple render targets simultaneously at the primitive level. The application uses a geometry shader to output a primitive and select which render target in the array should receive the primitive. This sample uses a render target array of 6 elements for the 6 faces of the cube texture. The following code fragment creates the render target view used for rendering to the cube texture.

```
// Create the 6-face render target view
D3D10_RENDER_TARGET_VIEW_DESC DescRT;
DescRT.Format = dstex.Format;
DescRT.ViewDimension = D3D10_RTV_DIMENSION_TEXTURE2DARRAY;
DescRT.Texture2DArray.FirstArraySlice = 0;
DescRT.Texture2DArray.ArraySize = 6;
DescRT.Texture2DArray.MipSlice = 0;
pd3dDevice->CreateRenderTargetView( g_pEnvMap, &DescRT, &g_pEnvMapRTV );
```

By setting DescRT.TextureCube.FirstArraySlice to 0 and DescRT.TextureCube.ArraySize to 6, this render target view represents an array of 6 render targets, one for each face of the cube texture. When the sample renders onto the cube map, it sets this render target view as the active view by calling ID3D10Device::OMSetRenderTargets() so that all 6 faces of the texture can be rendered at the same time.

```
// Set the env map render target and depth stencil buffer
ID3D10RenderTargetView* aRTViews[ 1 ] = { g_apEnvMapOneRTV[view] };
pd3dDevice->OMSetRenderTargets( sizeof(aRTViews) / sizeof(aRTViews[0]),
    aRTViews, g_pEnvMapOneDSV );
```

Rendering the CubeMap

At the top level, rendering begins in Render(), which calls RenderSceneIntoCubeMap() and RenderScene(), in that order. RenderScene() takes a view matrix a projection matrix, then renders the scene onto the current render target. RenderSceneIntoCubeMap() handles rendering of the scene onto a cube texture. This texture is then used in RenderScene() to render the environment-mapped object.

In RenderSceneIntoCubeMap(), the first necessary task is to compute the 6 view matrices for rendering to the 6 faces of the cube texture. The matrices have the eye point at the camera position and the viewing directions along the -X, +X, -Y, +Y, -Z, and +Z directions. A boolean flag, m_bUseRenderTargetArray, indicates the technique to use for rendering the cube map. If false, a for loop iterates through the faces of the cube map and renders the scene by calling RenderScene() once for each cube map face. No geometry shader is used for rendering. This technique is essentially the legacy method used in Direct3D 9 and prior. If m_bUseRenderTargetArray is true, the cube map is rendered with the RenderCubeMap effect technique. This technique uses a geometry shader to output each primitive to all 6 render targets. Therefore, only one call to RenderScene() is required to draw all 6

faces of the cube map.

The vertex shader that is used for rendering onto the cube texture is VS_CubeMap, as shown below. This shader does minimal work of transforming vertex positions from object space to world space. The world space position will be needed in the geometry shader.

```
struct VS_OUTPUT_CUBEMAP
{
    float4 Pos : SV_POSITION;    // World position
    float2 Tex : TEXCOORD0;      // Texture coord
};
VS_OUTPUT_CUBEMAP VS_CubeMap( float4 Pos : POSITION, float3 Normal : NORMAL, float2 Tex : TEXCOORD )
{
    VS_OUTPUT_CUBEMAP o = (VS_OUTPUT_CUBEMAP)0.0f;

    // Compute world position
    o.Pos = mul( Pos, mWorld );

    // Propagate tex coord
    o.Tex = Tex;

    return o;
}
```

One of the two geometry shaders in this sample, GS_CubeMap, is shown below. This shader is run once per primitive that VS_CubeMap has processed. The vertex format that the geometry shader outputs is GS_OUTPUT_CUBEMAP. The RTIndex field of this struct has a special semantic: SV_RenderTargetArrayIndex. This semantic enables the field RTIndex to control the render target to which the primitive is emitted. Note that only the leading vertex of a primitive can specify a render target array index. For all other vertices of the same primitive, RTIndex is ignored and the value from the leading vertex is used. As an example, if the geometry shader constructs and emits 3 vertices with RTIndex equal to 2, then this primitive goes to element 2 in the render target array.

At the top level, the shader consists of a for loop that loops 6 times, once for each cube face. Inside the loop, another loop runs 3 times per cube map to construct and emit three vertices for the triangle primitive. The RTIndex field is set to f, the outer loop control variable. This ensures that in each iteration of the outer loop, the primitive is emitted to a distinct render target in the array. Another task that must be done before emitting a vertex is to compute the Pos field of the output vertex struct. The semantic of Pos is SV_POSITION, which represents the projected coordinates of the vertex that the rasterizer needs to properly rasterize the triangle. Because the vertex shader outputs position in world space, the geometry shader needs to transform that by the view and projection matrices. In the loop, the view matrix used to transform the vertices is g_mViewCM[f]. This array of matrix is filled by the sample and contains the view matrices for rendering the 6 cube map faces from the environment-mapped object's perspective. Thus, each iteration uses a different view transformation matrix and emits vertices to a different render target. This renders one triangle onto 6 render target textures in a single pass, without calling DrawIndexed() multiple times.

```
struct GS_OUTPUT_CUBEMAP
{
    float4 Pos : SV_POSITION;     // Projection coord
    float2 Tex : TEXCOORD0;       // Texture coord
    uint RTIndex : SV_RenderTargetArrayIndex;
};

[maxvertexcount(18)]
void GS_CubeMap( triangle VS_OUTPUT_CUBEMAP In[3], inout TriangleStream<GS_OUTPUT_CUBEMAP> CubeMapStream )
{
    for( int f = 0; f < 6; ++f )
    {
        // Compute screen coordinates
        GS_OUTPUT_CUBEMAP Out;
        Out.RTIndex = f;
        for( int v = 0; v < 3; v++ )
        {
            Out.Pos = mul( In[v].Pos, g_mViewCM[f] );
            Out.Pos = mul( Out.Pos, mProj );
            Out.Tex = In[v].Tex;
            CubeMapStream.Append( Out );
        }
        CubeMapStream.RestartStrip();
    }
}
```

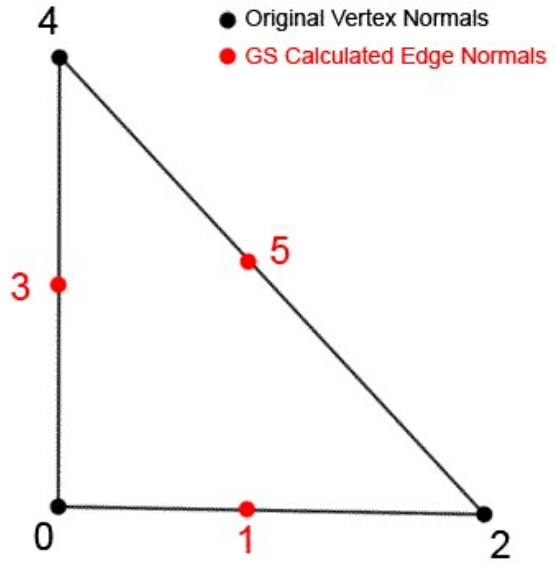
The pixel shader, PS_CubeMap, is rather straight-forward. It fetches the diffuse texture and applies it to the mesh. Because the lighting is baked into this texture, no lighting is performed in the pixel shader.

Rendering the Reflective Object

Three techniques are used to render the reflective object in the center of the scene. All three fetch texels from the cubemap just as they would from a cubemap that wasn't rendered in a single pass. In short, this technique is orthogonal to the way in which the resulting cubemap is used. The three techniques differ mainly in how they use the cubemap texels. RenderEnvMappedScene uses an approximated fresnel reflection function to blend the colors of the car paint with reflection from the cubemap. RenderEnvMappedScene_NoTexture does the same, but without the paint material. RenderEnvMappedGlass adds transparency to RenderEnvMappedScene_NoTexture.

Higher Order Normal Interpolation

Traditional normal interpolation has been linear. This means that the normals calculated in the vertex shader are linearly interpolated across the face of the triangle. This causes the reflections in the car to appear to *slide* when the direction of the normal changes rapidly across the face of a polygon. To mitigate this, this sample uses a quadratic normal interpolation. In this case, 3 extra normals are calculated in the geometry shader. These normals are placed in the center of each triangle edge and are the average of the two normals at the vertices that make up the edge. In addition, the geometry shader calculates a barycentric coordinate for each vertex that is passed down to the pixel shader and used to interpolate between the six normals.



In the pixel shader, these 6 normals weighted by six basis functions and added together to create the normal for that particular pixel. The basis functions are as follows.

```
2x^2 + 2y^2 + 4xy -3x -3y + 1
-4x^2 -4xy + 4x
2x^2 - x
-4y^2 -4xy + 4y
2y^2 - y
4xy
```