

Crystal/Candy Shader

John Isidoro
ATI Research

David Gosselin
ATI Research

Introduction

The goal of this chapter is to create a shader which gives the effect of a very shiny semi-transparent surface such as crystal or a piece of hard candy. This is accomplished using a combination of vertex and pixel shaders. The shader combines a cubic environment map to provide the reflections, a bump map to provide some detail, a fresnel term to control opacity, and a base map to provide the surface color.

Setup

This shader uses a model with one set of texture coordinates for both a base map (color) and bump map (normals encoded as colors). Additionally the shader uses a cubic environment map. In this example, the environment map was pre-rendered using the ATI *Sushi* graphics engine and is shown in the Figure 1.

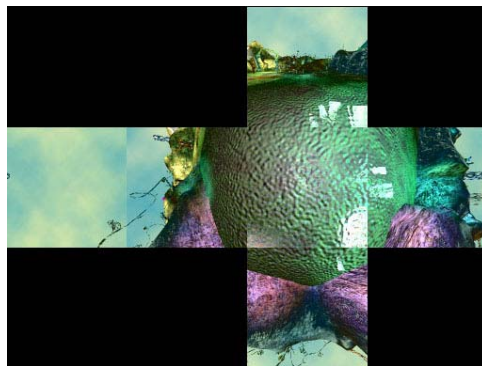


Figure 1 - Cubic Environment Map

Since the object needs to be semi-transparent it will be drawn in two passes both blended into the frame buffer. The first pass will draw the back faces of the object. This can be accomplished by setting the cull mode to either clockwise or counter clockwise depending on how your application specifies back/front faces. This is accomplished by setting the Direct3D render state `D3DRS_CULLMODE` appropriately. The code for doing this is shown below:

```
d3dDevice->SetRenderState (D3DRS_CULLMODE, D3DCULL_CCW);
```

The front faces are drawn by setting the opposite cull mode and drawing the model again.

```
d3dDevice->SetRenderState (D3DRS_CULLMODE, D3DCULL_CW);
```

For both passes, alpha blending is enabled in order to give the transparent effect. The code for setting this up is shown below:

```
d3dDevice->SetRenderState (D3DRS_ALPHABLENDENABLE, TRUE);
d3dDevice->SetRenderState (D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
d3dDevice->SetRenderState (D3DRS_DSTBLEND, D3DBLEND_INVSRCALPHA);
```

Now that we have set these general render states, we will explore the specifics of the crystal/candy vertex and pixel shaders.

Vertex Shader

The vertex shader for this effect is centered on computing the view direction. Additionally it will pass the tangent/texture space vectors along so that the bump map normal can be transformed into object space in the pixel shader. The first section of the vertex shader computes the transformed vertex position. The vertex shader code to perform this transformation follows:

```
// c0 - (0.125, 0.5, 1.0, 2.0)
// c2 - World Space Camera Position
// c4-c7- World/View/Projection Matrix
// v0 - Vertex Position
// v3 - Vertex Normal
// v7 - Vertex Texture Data u,v
// v8 - Vertex tangent Vector
vs.1.1
m4x4 oPos, v0, c4 // OutPos = ObjSpacePos * WVP Matrix
mov oT0, v7 // output basemap texcoords
```

The next section of the vertex shader computes the view vector in world space. This is accomplished by subtracting the world space camera position from the world space vertex position and then normalizing the result. Note that in this example the world transform is identity and the input vertex position is used directly. If this is not the case in your application, you will need to transform the input position by the world matrix before performing this calculation.

```
sub r8, c2, v0 // viewer direction
dp3 r10.x, r8, r8 // magnitude
rsq r10.y, r10.x // normalize
mul r8, r8, r10.y // V normalized view vector
mov oT1, r8 // world space view vector
```

The next section passes along the tangent/texture space basis matrix. In this example we compute the binormal by taking the cross product of the tangent and the

normal. Note that this is not a general solution it relies upon the fact that the texture coordinates for this model are not reversed on any triangles.

```

mov   r5, v3           // normal (tZ)
mov   oT4, r5
mov   r3, v8           // tangent (tY)
mov   oT2, r3
mul   r4, r5.yzxw, r3.zxyw // compute binormal (tX) as being
mad   r4, r5.zxyw, -r3.yzxw, r4 // perpendicular to normal
mov   oT3, r4

```

The last section of this vertex shader transforms the view vector into tangent space. It also scales and biases the vector so that it can be passed to the pixel shader in one of the color interpolators.

```

m3x3  r6, r8, r3           // xform view vec into tangent space
mad   oD0, r6, c0.y, c0.y // tan space V vector

```

Now that the per-vertex computations are described, we can look at what happens at the pixel level to create the crystal look.

Pixel Shader

The pixel shader for this effect is accomplished in two shader phases. The first phase computes the reflection vector used for the cubic environment map while the second phase composites the color from the dependent read with other maps. The full shader is shown below and a more detailed description of the various parts follows.

```

// c0 - (0.0, 0.5, 1.0, 0.75)
// c1 - base color (0.5, 0.0, 0.5, 1.0)
// c2 - reflection color (1.0, 1.0, 1.0, 1.0)
// c7 - (0.0, 0.0, 1.0, 0.4)
ps.1.4
texld r0, t0           // bump map
texcrd r1.rgb, t1      // view vec
texcrd r2.rgb, t2      // tan x
texcrd r3.rgb, t3      // tan y
texcrd r4.rgb, t4      // tan z

    lrp r0.rgb, c7.a, r0, c7 // weaken bump map by lerping with 0,0,1
    mul r5.rgb, r0.x, r2      // move bump normal into obj space
    mad r5.rgb, r0.y, r3, r5  // move bump normal into obj space
    mad r5.rgb, r0.z, r4, r5  // move bump normal into obj space

    dp3 r3.rgb, r1, r5        // V.N
    mad r2.rgb, r3, r5_x2, -r1 // 2N(V.N)-V

phase
texld r1, t0           // base map
texld r2, r2           // fetch from environment map

    dp3 r0.a, v0_bx2, r0     // (N.V)fresnel term
    mul r1.rgb, r1, c1       // base map * base color
    +mov r0.a, 1-r0.a        // 1-(N.V)
    mul r2.rgb, r2, c2       // reflection map * reflection color

```

Crystal / Candy Shader

```
mad r0.rgb, r2, r0.a, r1 // (1-(N.V)) * reflection map * reflection color
                          // + base map * base color
+add_sat r0.a, r0.a, c0.y // boost opacity: (1-(N.V)) + 0.5
```

At the core of the shader is the environment mapping which is sampled as a dependent read at the top of the second phase. The following lines from the pixel shader compute the per-pixel reflection vector and sample the cube map:

```
dp3 r3.rgb, r1, r5 // V.N
mad r2.rgb, r3, r5_x2, -r1 // 2N(V.N)-V

phase
texld r2, r2 // fetch from env map
```

If the shader were changed to just do this computation, the result would look like the following figure. One thing to note is that in this picture the top portion of the model has intentionally faceted normals to give the appearance of some kind of cut crystal while the bottom part of the model has smooth normals.



Figure 2 - Just environment mapping

To this effect we want to add bumpiness based on the normal map. In order for this to look correct we take the sampled normal (r_0) from the bump map and transform it into object space using the tangent space matrix that was computed in the vertex shader and interpolated across the polygon. The following lines show the part of the pixel shader where this computation takes place.

```
mul r5.rgb, r0.x, r2 // move bump normal into obj space
mad r5.rgb, r0.y, r3, r5 // move bump normal into obj space
mad r5.rgb, r0.z, r4, r5 // move bump normal into obj space
```

If this is added to the environment mapping you get something that looks like the following figure:



Figure 3 - Bumpy Environment Mapped

Since this looks a little too bumpy for the effect we want, a small term is linearly interpolated with the bump map prior to doing the bump calculation. The following line of code from the shader performs this task:

```
lerp r0.rgb, c7.a, r0, c7 // weaken bump map by lerping with 0,0,1
```

The figure below shows what this looks like on the above model.



Figure 3 - Weaker Bumpy Environment Mapped

This basically wraps up the computation that occurs in the first phase of the pixel shader. In order to make the inner parts of the model (where you would be seeing through more of the actual object) seem more opaque and the outer edges seem more transparent, a Fresnel term is used. (For more uses of the Fresnel term, see Chapter [Brennan].) The computation takes a few additional instructions, with one hidden in a co-

Excerpted from *ShaderX: Vertex and Pixel Shader Tips and Tricks*

issue alpha instruction. The following lines of code compute the Fresnel term used to give the impression of thicker crystal:

```
dp3 r0.a, v0_bx2, r0      // (N.V)fresnel term
+mov r0.a, 1-r0.a        // 1-(N.V)
```

The next figure shows what the model looks like with this term added to the shader.



Figure 4 - Fresnel term added

Since we are going for a more solid looking effect, the Fresnel term is tweaked a bit to make the model look more opaque. This is also done in a co-issued alpha instruction, shown below:

```
+add_sat r0.a, r0.a, c0.y // boost opacity: (1-(N.V)) + 0.5
```

The following figure shows what the model looks like with this term applied.

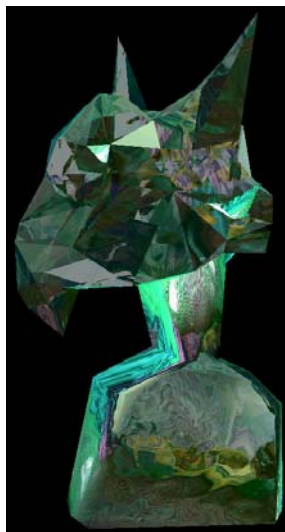


Figure 5 - Tweaked Fresnel term

Excerpted from *ShaderX: Vertex and Pixel Shader Tips and Tricks*

Now we want to add some color to the model using the base map and base color. The following line in the pixel shader computes this contribution.

```
mul r1.rgb, r1, c1          // base map * base color
```

The following figure shows what this would look like on its own.



Figure 6 - Base Map and Base Color

The final step is to combine the terms together to create the final image. The following code performs this computation:

```
mul r2.rgb, r2, c2          // reflection map * reflection color  
mad r0.rgb, r2, r0.a, r1    // (1-(N.V)) * reflection map * reflection color  
                             // + base map * base color
```

This is what the final image looks like.



Figure 6 - Putting it all together

The next image shows what this looks like within an environment from the ATI *Sushi* engine.



Figure 7 - The shader within a scene

Summary

This section has shown a method for creating a semi-transparent crystal/candy look on an object. It used cubic environment mapping for reflections and a per-pixel bump mapping. Hopefully this gives you some ideas and techniques that you can use within your applications.