Rippling Reflective and Refractive Water

Alex Vlachos
ATI Research

John Isidoro
ATI Research

Chris Oat
ATI Research

One of the classic challenges of real-time computer graphics is to generate realistic looking water. Because water can look very different depending on the context of the scene, it is important to define a few different categories of water effects. For example, techniques used to render ocean water may not produce realistic looking puddle water. With this in mind, the shaders presented in this chapter will focus on highly realistic per-pixel lighting effects used to render real-time lake water. Unlike ocean water, lake water does not have large rolling swells. Instead, the surface geometry consists of high frequency, non-uniform perturbations that give the surface a subtle choppiness resulting in slightly distorted reflections and refractions. Because of these surface characteristics, shallow water reflections and refractions as shown in Figure 1 can be reasonably approximated with planar reflections and renderable textures.

Renderable textures allow the programmer to distort the reflection and refraction textures using a sum of scrolling bump maps to simulate the water surface ripples. One of the nice things about distorting these textures with a bump map is that the renderable textures can be of a much lower resolution than the screen resolution and the end result will still look very compelling. The example below was rendered using two 512×256 renderable textures. Since the reflected and refracted geometry is drawn once for each renderable texture, rendering to these maps at a lower resolution provides a significant performance boost.

Figure 1 – This scene was drawn using the reflection and refraction techniques described in this chapter.

Generating Reflection & Refraction Maps

When dealing with fairly flat water surfaces such as lake water, river water, or puddle water reflections can be convincingly approximated with planar reflections. A reflection map is created by reflecting the scene geometry over the plane of the water's surface and rendering to a texture. It is important to utilize user clip planes in order to clip any geometry that is already below the plane of reflection as this geometry is underwater and should not be included in the reflection map, see Figure 2.

Figure 2 – Example of a reflection map created by reflecting scene geometry about the plane of the water's surface.

As illustrated in Figure 3 the refraction map is generated by clipping all scene geometry above the plane of the water's surface. As with the reflection map, user clips planes are used to cull the geometry above the water's surface.

Figure 3 – Example of a refraction map created by drawing only the geometry below the water's surface.

Vertex Shader

A vertex shader is used to approximate sine and cosine waves to generate rippling water geometry as well as perturbing the tangent space vectors per vertex. This vertex shader is very similar to the vertex shader presented in. The vertex shader is presented below along with a description of the changes necessary to produce less drastic ripples.

```
vs.1.1
// v0    - Vertex Position
// v7    - Vertex Texture Data u,v
// v8    - Vertex Tangent (v direction)
//
// c0    - { 0.0,  0.5, 1.0, 2.0}
// c1    - { 4.0, .5pi, pi, 2pi}
// c2    - {1, -1/3!, 1/5!, -1/7!  }  //for sin
// c3    - {1/2!, -1/4!, 1/6!, -1/8!  }  //for cos
// c4-7  - Composite World-View-Projection Matrix
// c8    - ModelSpace Camera Position
// c9    - ModelSpace Light Position
// c10   - {fixup factor for taylor series imprecision, }
// c11   - {waveHeight0, waveHeight1, waveHeight2, waveHeight3}
// c12   - {waveOffset0, waveOffset1, waveOffset2, waveOffset3}
// c13   - {waveSpeed0, waveSpeed1, waveSpeed2, waveSpeed3}
// c14   - {waveDirX0, waveDirX1, waveDirX2, waveDirX3}
// c15   - {waveDirY0, waveDirY1, waveDirY2, waveDirY3}
// c16   - {time, sin(time)}
// c17   - {basetexcoord distortion x0, y0, x1, y1}

mul r0, c14, v7.x          // use tex coords as inputs to sinusoidal warp
mad r0, c15, v7.y, r0      // use tex coords as inputs to sinusoidal warp

mov r1, c16.x              // time...
mad r0, r1, c13, r0        // add scaled time to move bumps by frequency
add r0, r0, c12
frc r0.xy, r0              // take frac of all 4 components
frc r1.xy, r0.zwzw         //
mov r0.zw, r1.xyxy         //

mul r0, r0, c10.x          // multiply by fixup factor (fix taylor series inaccuracy)
```

```
sub r0, r0, c0.y           // subtract .5
mul r0, r0, c1.w           // mult tex coords by 2pi  coords range from(-pi to pi)

mul r5, r0, r0             // (wave vec)^2
mul r1, r5, r0             // (wave vec)^3
mul r6, r1, r0             // (wave vec)^4
mul r2, r6, r0             // (wave vec)^5
mul r7, r2, r0             // (wave vec)^6
mul r3, r7, r0             // (wave vec)^7
mul r8, r3, r0             // (wave vec)^8

mad r4, r1, c2.y, r0       // (wave vec) - ((wave vec)^3)/3!
mad r4, r2, c2.z, r4       //  + ((wave vec)^5)/5!
mad r4, r3, c2.w, r4       //  - ((wave vec)^7)/7!

mov r0, c0.z               // 1
mad r5, r5, c3.x ,r0       // -(wave vec)^2/2!
mad r5, r6, c3.y, r5       // +(wave vec)^4/4!
mad r5, r7, c3.z, r5       // -(wave vec)^6/6!
mad r5, r8, c3.w, r5       // +(wave vec)^8/8!

dp4 r0, r4, c11            // multiply wave heights by waves

mul r0.xyz, c0.xxzx, r0    // multiply wave magnitude at this vertex by normal
add r0.xyz, r0, v0         // add to position
mov r0.w, c0.z             // homogenous component

m4x4 oPos, r0, c4          // OutPos = ObjSpacePos * World-View-Projection Matrix
mul r1, r5, c11            // cos* waveheight
dp4 r9.x, -r1, c14         // normal x offset
dp4 r9.yzw, -r1, c15       // normal y offset and tangent offset
mov r5, c0.xxzx
mad r5.xy, r9, c10.y, r5   // warped normal move according to cos*wavedir*waveeheight
mov r4, v9
mad r4.z, -r9.x, c10.y, r4.z // warped tangent vector
dp3 r10.x, r5, r5
rsq r10.y, r10.x
mul r5, r5, r10.y          // normalize normal


dp3 r10.x, r4, r4
rsq r10.y, r10.x
mul r4, r4, r10.y          // normalize tangent
mul r3, r4.yzxw, r5.zxyw
mad r3, r4.zxyw, -r5.yzxw, r3 // cross product to find binormal
sub r1, c9, r0             // light vector
sub r2, c8, r0             // view vector
dp3 r10.x, r1, r1          // normalize light vector
rsq r10.y, r10.x
mul r1, r1, r10.y
dp3 r6.x, r1, r3
dp3 r6.y, r1, r4
dp3 r6.z, r1, r5   // transform light vector into tangent space
dp3 r10.x, r2, r2
rsq r10.y, r10.x
mul r2, r2, r10.y  // normalized view vector

dp3 r7.x, r2, r3
dp3 r7.y, r2, r4
dp3 r7.z, r2, r5   // put view vector in tangent space

mov r0, c16.x
mul r0, r0, c24
frc r0.xy, r0
mul r1, v7, c26

add oT0, r1, r0    // bump map coord1
mov r0, c16.x
mul r0, r0, c25
frc r0.xy, r0
mul r1, v7, c27
```

```
add oT1, r1, r0    // bump map coord 2
dp4 r0.x, v0, c20
dp4 r0.y, v0, c21
dp4 r0.zw, v0, c22

mov oT2, r0       // projective tex coords for reflection/ refreaction maps
mov oT3, r7       // tan space view vec
mov oT4, v7       // base map coords
mov oT5, r6       // tan space light vec
```

Pixel Shader

The pixel shader used to render the water has a few novel features. When sampling from the renderable textures, the texture coordinates must be interpolated linearly in screen space. The reason for this is that the scene is already rendered from the point of view of the camera and thus contents of the renderable texture are already perspective correct. To achieve this, the projection matrix must be altered in the following way:

```
Matrix M = { 0.5f,  0.0f, 0.0f, 0.0f,
             0.0f, -0.5f, 0.0f, 0.0f,
             0.0f,  0.0f, 0.0f, 0.0f,
             0.5f,  0.5f, 1.0f, 0.0f };

// row major element (1,4) is set to zero
projectionMatrix[11] = 0.0f;

// projection matrix uses scale and bias to get coordinates into
// [0.0, 1.0] range
projectionMatrix = M * projectionMatrix;
```

Now the projection matrix will move vertices into a normalized texture space for indexing into the pre-projected reflection and refraction maps. Given the linearly interpolated texture coordinates, the perturbations for the reflection and refraction maps can be performed on a per pixel basis by simply adding a scaled and rotated version of the xy offsets from the scrolling bump maps. It is important to scale and rotate these offsets so that as the bump map scrolls, the perturbations move in the direction that the water is flowing. Another interesting trick is to sample the refraction map using the swapped texture coordinates used to index the reflection map (for example, index the reflection map using coordinates <u,v> and then sample the refraction map using coordinates <v,u>). Swapping the coordinates in this way prevents the two scrolling bump maps from aligning. After sampling, the refractions are modulated with the water color and the perturbed reflections are attenuated with a per pixel fresnel term and modulated with a reflection color. The per pixel fresnel term scales the reflections such that the strength of the reflection is dependant on the angle of the view vector with respect to the water surface normal. The full shader is shown below along with further comments.

```
ps.1.4

// T0 : Bump map 1 Coordinates
// T1 : Bump map 2 Coordinates
// T2 : Projective Texture Coordinates for Reflection/Refraction Maps
// T3 : Tangent Space View Vector
```

```
// T5 : Tangent Space Light Vector
texld r0, t0
texld r1, t1

texcrd r2.xy, t2_dw.xyw   // renderable textures
texcrd r4.xyz, t3         // tan space V
texcrd r5.xyz, t5         // tan space L

add_d2 r1.rgb, r0_bx2, r1_bx2

mov r3.xy, r2

dp3 r0.r, r4, r1             // V.N
dp3 r0.g, r1, r5             // L.N

mad_sat r5.rg, r1, c2, r3    // perturb refraction
mad_sat r4.rg, r1, c1, r3    // perturb reflection

phase

texcrd r0.rgb, r0    // V.N, L.N, R.L

texld r2, r4           // Reflection
texld r3, r5           // Refraction

mul_sat r2.rgb, r2, c3 // reflection color
+mul r2.a, r0.g, r0.g

mul_sat r3.rgb, r3, c4    // refraction color
+mul_sat r2.a, r2.a, r2.a

mad_sat r0.rgb, 1-r0.r, r2, r3
+mul_sat r2.a, r2.a, r2.a

mad_sat r0.rgb, r2.a, c6, r0   // add specular highlights to reflection
+mov r0.a, r0.r
```

Conclusion

This paper focused on using vertex and pixel shaders to render realistic rippling water with realistic reflections and refractions by building on techniques described in other sections. The shaders presented here demonstrate the use of Taylor Series approximations to compute sine and cosine waves in a vertex shader as well as the use of renderable textures to create rippling reflections and refractions.

The shaders described in this section as well as the accompanying screenshots were taken from ATI's Nature Demo. This real time interactive graphics demo shows rippling reflective and refractive water in a complex natural environment.