Rendering Ocean Water

John Isidoro, Alex Vlachos, and Chris Brennan

Introduction

In computer graphics, simulating water has always been a topic of much research. In particular, ocean water is especially difficult due to the shape and combination of multiple waves, in addition to the sun in the sky and the reflection of the clouds.

The shader in this article is meant to simulate the appearance of ocean water using vertex and pixel shaders. The interesting part of this shader is that it runs completely in hardware in a single pass on recent graphics cards (Radeon 8500). This has the advantage of leaving the CPU free for other calculations, as well as allowing for a courser tessellation of the input geometry that can be tessellated using N-Patches or other higher order surface schemes. The input geometry is a grid of quads with one set of texture coordinates and tangent space, though in theory only a position is actually needed if assumptions are made about the orientation of the up vector and the scale of the ocean waves in the shader.

This shader is best explained by separating the vertex shader and the pixel shader while keeping in mind the final result, shown in Figure 1.

Figure 1

Figure 2

Sinusoidal Perturbation in a Vertex Shader

The vertex shader is responsible for generating the combination of sine waves that perturb the position and the cosine waves that perturb the tangent space vectors for the vertex. A Taylor series approximation is used to generate sine and cosine functions within the shader. Due to the SIMD nature of vertex shaders, four sine and cosine waves are calculated in parallel, and the results are weighted and combined using a single dp4.

Each sine wave has fully adjustable direction, frequency, speed, and offset that is configured in the constant store.

The first step is to compute each wave's starting phase into the sine or cosine function. The texture coordinates are multiplied by the direction and frequency of the four waves in parallel. c14 and c15 are the frequencies of the wave relative to S and T, respectively.

```
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
```

Next, the time, which is stored in c16.x, is multiplied by the speed of the waves (in c13 and added to the wave offsets in c12):

```
mov r1, c16.x           //time...
mad r0, r1, c13, r0     //add scaled time to move bumps according to
                          frequency
add r0, r0, c12         //starting time offset
```

This computes the input to the cosine function. A Taylor approximation, however, is only accurate for the range it is created for, and more terms are needed the larger that range is. So for a repeating function like a cosine wave, the fractional portion of the wave phase can be extracted and then expanded to the – to range before calculating the Taylor series expansion.

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

Calculate the Taylor series expansion of sine (r4) and cosine (r5):

```
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!
```

The results are modulated by relative heights of each of the waves and the scaled sine wave is used to perturb the position along the normal. The new object space position is then transformed to compute the final position. The vertex input, v5.x, is used to allow artist control of how high the waves are in different parts of the ocean. This can be useful for shorelines where the ocean waves will be smaller than those farther out to sea:

```
sub r0, c0.z, v5.x              //... 1-wave scale
mul r4, r4, r0                  //scale sin
mul r5, r5, r0                  //scale cos

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

mul r0.xyz, v3, 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-Proj
                                  Matrix
```

The tangent and normal vectors are perturbed in a similar manner using the cosine wave instead of the sine wave. This is done because the cosine is the first derivative of the sine and therefore perturbs the tangent and normal vectors by the slope of the wave. The following code makes the assumption that the source art is a plane along the Z axis.

It is worth mentioning that this vertex perturbation technique can be extended to sinusoidally warp almost any geometry. See "Bubble Shader" for more details.

```
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, v3                   //starting normal
mad     r5.xy, r9, c10.y, r5     //warped normal move nx, ny according to
                                 //cos*wavedir*waveeheight
mov     r4, v8                   //tangent
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
```

The binormal is then calculated using a cross product of the warped normal and the warped tangent vector to create a tangent space basis matrix. This matrix will be used later to transform the bump map's tangent space normal into world space for cube-mapped environment-mapped bump mapping (CMEMBM).

```
mul     r3, r4.yzxw, r5.zxyw
mad     r3, r4.zxyw, -r5.yzxw, r3   //xprod to find binormal
```

CMEMBM needs the view vector to perform the reflection operation:

```
sub     r2, c8, r0                  //view vector
dp3     r10.x, r2, r2
rsq     r10.y, r10.x
mul     r2, r2, r10.y               //normalized view vector
```

The height map shown in Figure 3 is used to create a normal map. The incoming texture coordinates are used as a starting point to create two sets of coordinates that are rotated and scroll across each other based on time. These coordinates are used to scroll two bump maps past each other to produce the smaller ripples in the ocean. One interesting trick used in this shader is to swap the u and v coordinates for the second texture before compositing them. This eliminates the visual artifacts that occur when the scrolling textures align with each other exactly and the ripples appear to stop for a moment. Swapping the texture coordinates ensure that the maps never align with each other (unless they are radially symmetric).

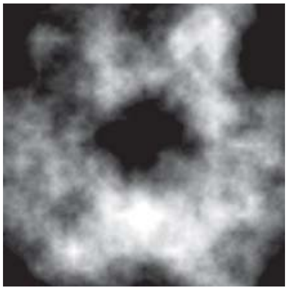

Figure 3: Height map used to create the normal map for the ocean shader

```
mov     r0, c16.x
mul     r0, r0, c17.xyxy
frc     r0.xy, r0              //frc of incoming time
add     r0, v7, r0             //add time to tex coords
mov     oT0, r0                //distorted tex coord 0

mov     r0, c16.x
mul     r0, r0, c17.zwzw
frc     r0.xy, r0              //frc of incoming time
add     r0, v7, r0             //add time to tex coords
mov     oT1, r0.yxzw           //distorted tex coord 1
```

The vertex shader is completed by the output of the remaining vectors used by the pixel shader. The pixel and vertex shader for the ocean water effect can be found in its entirety at the end of this article.

```
mov     oT2, r2                //pass in view vector (worldspace)
mov     oT3, r3                //tangent
mov     oT4, r4                //binormal
mov     oT5, r5                //normal
```

CMEMBM Pixel Shader with Fresnel Term

Once the vertex shader has completed, the pixel shader is responsible for producing the bump-mapped reflective ocean surface.

First, the pixel shader averages the two scrolling RGB normal bump maps to generate a composite normal. In this particular case, the bumps are softened further by dividing the x and y components in half. Next, it transforms the tangent space composite normal into world space and calculates a per-pixel reflection vector. The reflection vector is used to sample a skybox cubic environment map (Figure 4). The shader also calculates 2*N·V and uses it to sample a Fresnel 1D texture (Figure 5). This Fresnel map gives the water a more greenish appearance when looking straight down into it and a more bluish appearance when looking edge on. The scale by two is used to expand the range of the Fresnel map.

```
texld r0, t0                          //bump map 0
texld r1, t1                          //sample bump map 1
texcrd r2.rgb, t2                     //View vector
texcrd r3.rgb, t3                     //Tangent
texcrd r4.rgb, t4                     //Binormal
texcrd r5.rgb, t5                     //Normal

   add_d4  r0.xy, r0_bx2, r1_bx2  //Scaled Average of 2 bumpmaps' xy
                                    offsets

   mul r1.rgb, r0.x, r3
   mad r1.rgb, r0.y, r4, r1
   mad r1.rgb, r0.z, r5, r1       //Transform bumpmap normal into world
                                    space

   dp3 r0.rgb, r1, r2             //V.N
   mad r2.rgb, r1, r0_x2, -r2     //R = 2N(V.N)-V

   mov_sat r1, r0_x2              //2 * V.N  (sample over range of 1d
                                    map!)
```

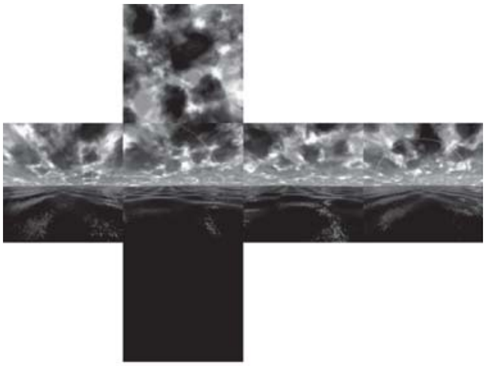

Figure 4: Cubic environment map used for ocean water reflections

Figure 5: 1D texture used for the water color addressed by 1–N·V

The second phase composites the water color from the Fresnel map, the environment map, and other specular highlights extracted from the environment map. One trick we use is to square the environment map color values to make the colors brighter and to enhance the contrast for compositing. The advantage to doing this in the pixel shader instead of as a preprocessing step is so the same skybox environment map can be used for other objects in the scene. To get the specular light sparkles in the water, a specular component is derived from the green channel of the environment map. For this example, the choice is based on the environment map artwork. The desired effect is to have the highlights in the water correspond to bright spots in the sky, and in this case, the green channel seemed to work best. To make sure the specular peaks were only generated from the brightest areas of the environment map, the specular value extracted from the green channel was raised to the eighth power. This has the effect of darkening all but the brightest areas of the image.

Another approach for encoding specular highlights in an environment map is to have the artists specify a glow map as an alpha channel of the environment map. See "Bubble Shader" for more details.

```
texcrd r0.rgb, r0
texld r2, r2                  //cubic env map
texld r3, r1                  //Index fresnel map using 2*V.N

mul r2.rgb, r2, r2            //Square the environment map
+mul r2.a, r2.g, r2.g         //use green channel of env map as
                                specular
mul r2.rgb, r2, 1-r0.r        //Fresnel Term
+mul r2.a, r2.a, r2.a         //Specular highlight ^4

add_d4_sat r2.rgb, r2, r3_x2  //+= Water color
+mul r2.a, r2.a, r2.a         //Specular highlight ^8

mad_sat r0, r2.a, c1, r2      //+= Specular highlight * highlight
                                color
```

Ocean Water Shader Source Code

```
DefineParam texture rgbNormalBumpMap NULL
SetParamEnum rgbNormalBumpMap EP_TEX0

DefineParam texture waterGradientMap NULL
SetParamEnum waterGradientMap EP_TEX1

DefineParam texture cubeEnvMap NULL
SetParamEnum cubeEnvMap EP_TEX2

//Constant store
DefineParam vector4 commonConst (0.0, 0.5, 1.0, 2.0)

DefineParam vector4 appConst (0.0, 0.0, 0.0, 0.0) //Time, 1.0/lightFalloffDist
SetParamEnum appConst EP_VECTOR3

DefineParam vector4 worldSpaceCamPos (0, 0, 0, 0)
BindParamToState worldSpaceCamPos STATE_VECTOR_CAMERA_POSITION 0 WORLD_SPACE

DefineParam vector4 worldSpaceLightPos (-10000, -25000, 2000, 1)
SetParamEnum worldSpaceLightPos EP_VECTOR0

DefineParam matrix4x4 wvp [(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)]
BindParamToState wvp STATE_MATRIX_PVW

//=============================================================================
//commonly used constants
//heights for waves 4 different fronts
DefineParam vector4 waveHeights (80.0, 100.0, 5.0, 5.0)

//offset in sine wave.. (ranges 0 to 1)
DefineParam vector4 waveOffset (0.0, 0.2, 0.0, 0.0)

//freqency of the waves (e.g., waves per unit time..)
DefineParam vector4 waveSpeed (0.2, 0.15, 0.4, 0.4)

//diection of waves in tangent space (also controls frequency in space)
DefineParam vector4 waveDirx (0.25, 0.0, -0.7, -0.8)
DefineParam vector4 waveDiry (0.0, 0.15, -0.7, 0.1)

//scale factor for distortion of base map coords
```

```
//bump map scroll speed
DefineParam vector4 bumpSpeed (0.031, 0.04, -0.03, 0.02)

DefineParam vector4 piVector (4.0, 1.57079632, 3.14159265, 6.28318530)

//Vectors for taylor's series expansion of sin and cos
DefineParam vector4 sin7 (1, -0.16161616, 0.0083333, -0.00019841)
DefineParam vector4 cos8 (-0.5, 0.041666666, -0.0013888889, 0.000024801587)

//frcFixup.x is a fixup to make the edges of the clamped sin wave match up again
due to // numerical inaccuracy
//frcFixup.y should be equal to the average of du/dx and dv/dy for the base texture
// coords.. this scales the warping of the normal

DefineParam vector4 frcFixup (1.02, 0.003, 0, 0)

DefineParam vector4 psCommonConst (0, 0.5, 1, 0.25)
DefineParam vector4 highlightColor (0.8, 0.76, 0.62, 1)

DefineParam vector4 waterColor (0.50, 0.6, 0.7, 1)

//==============================================================================
// 1 Pass
//==============================================================================
StartShader
   Requirement VERTEXSHADERVERSION 1.1
   Requirement PIXELSHADERVERSION 1.4
   StartPass
      SetTexture 0 rgbNormalBumpMap
      SetTextureFilter 0 BILINEAR
      SetTextureStageState 0 MIPMAPLODBIAS -1.0

      SetTexture 1 rgbNormalBumpMap
      SetTextureFilter 1 BILINEAR
      SetTextureStageState 1 MIPMAPLODBIAS -1.0

      SetTexture 2 cubeEnvMap
      SetTextureWrap 2 CLAMP CLAMP CLAMP
      SetTextureFilter 2 BILINEAR
      SetTextureStageState 2 MIPMAPLODBIAS 0.0

      SetTexture 3 waterGradientMap
      SetTextureWrap 3 CLAMP CLAMP CLAMP
      SetTextureFilter 3 LINEAR

      SetVertexShaderConstant 0 commonConst
      SetVertexShaderConstant 1 piVector
      SetVertexShaderConstant 2 sin7
      SetVertexShaderConstant 3 cos8

      SetVertexShaderConstant 4 wvp
      SetVertexShaderConstant 8 worldSpaceCamPos
      SetVertexShaderConstant 9 worldSpaceLightPos

      SetVertexShaderConstant 10 frcFixup
      SetVertexShaderConstant 11 waveHeights
      SetVertexShaderConstant 12 waveOffset
      SetVertexShaderConstant 13 waveSpeed
      SetVertexShaderConstant 14 waveDirx
      SetVertexShaderConstant 15 waveDiry
      SetVertexShaderConstant 16 appConst
      SetVertexShaderConstant 17 bumpSpeed

      StartVertexShader
         // v0    - Vertex Position
         // v3    - Vertex Normal
```

```
// 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}

vs.1.1

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 according to
                          frequency
add r0, r0, c12         //starting time offset
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 (due to 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!

sub r0, c0.z, v5.x      //... 1-wave scale
mul r4, r4, r0          //scale sin
mul r5, r5, r0          //scale cos

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

mul r0.xyz, v3, 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, v3                   //starting normal
   mad     r5.xy, r9, c10.y, r5     //warped normal move nx, ny according to
                                    //cos*wavedir*waveeheight
   mov     r4, v8                   //tangent
   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 //xprod to find binormal

   sub     r2, c8, r0               //view vector
   dp3     r10.x, r2, r2
   rsq     r10.y, r10.x
   mul     r2, r2, r10.y            //normalized view vector

   mov     r0, c16.x
   mul     r0, r0, c17.xyxy
   frc     r0.xy, r0               //frc of incoming time
   add     r0, v7, r0              //add time to tex coords
   mov     oT0, r0                 //distorted tex coord 0

   mov     r0, c16.x
   mul     r0, r0, c17.zwzw
   frc     r0.xy, r0               //frc of incoming time
   add     r0, v7, r0              //add time to tex coords
   mov     oT1, r0.yxzw            //distorted tex coord 1

   mov     oT2, r2                 //pass in view vector (worldspace)
   mov     oT3, r3                 //tangent
   mov     oT4, r4                 //binormal
   mov     oT5, r5                 //normal
EndVertexShader

SetPixelShaderConstant 0 psCommonConst
SetPixelShaderConstant 1 highlightColor

StartPixelShader
   ps.1.4

   texld r0, t0                   //bump map 0
   texld r1, t1                   //sample bump map 1
   texcrd r2.rgb, t2              //View vector
   texcrd r3.rgb, t3              //Tangent
   texcrd r4.rgb, t4              //Binormal
   texcrd r5.rgb, t5              //Normal

      add_d4  r0.xy, r0_bx2, r1_bx2  //Scaled Average of 2 bumpmaps xy
                                       offsets

      mul r1.rgb, r0.x, r3
      mad r1.rgb, r0.y, r4, r1
      mad r1.rgb, r0.z, r5, r1       //Put bumpmap normal into world space

      dp3 r0.rgb, r1, r2             //V.N
```

```
            mad r2.rgb, r1, r0_x2, -r2     //R = 2N(V.N)-V

            mov_sat r1, r0_x2              //2 * V.N  (sample over range of 1d
                                                       map!)

         phase
            texcrd r0.rgb, r0
            texld r2, r2                   //cubic env map
            texld r3, r1                   //Index fresnel map using 2*V.N

            mul r2.rgb, r2, r2             //Square the environment map
    +mul r2.a, r2.g, r2.g                  //use green channel of env map as
                                             specular
     mul r2.rgb, r2, 1-r0.r                //Fresnel Term
            +mul r2.a, r2.a, r2.a          //Specular highlight ^4

            add_d4_sat r2.rgb, r2, r3_x2   //+= Water color
            +mul r2.a, r2.a, r2.a          //Specular highlight ^8

            mad_sat r0, r2.a, c1, r2       //+= Specular highlight * highlight
                                             color

      EndPixelShader

   EndPass
EndShader
```

Sample Applications

This shader can be seen in the Island demos (http://www.ati.com/na/pages/resource_centre/dev_rel/Demos.html) and the Ocean Screen Saver (http://www.ati.com/na/pages/resource_centre/dev_rel/screensavers.html), as well as on the companion CD.