5.5

Animation and Rendering of Underwater God Rays

Stefano Lanza

God rays are a truly beautiful phenomenon in underwater scenes (see Figure 5.5.1 for an example). They are beams of sunlight that filter through water and then dance underwater in many directions. You can admire them in many underwater documentaries and movies, or better, by diving in deep water and then looking up toward the sun.

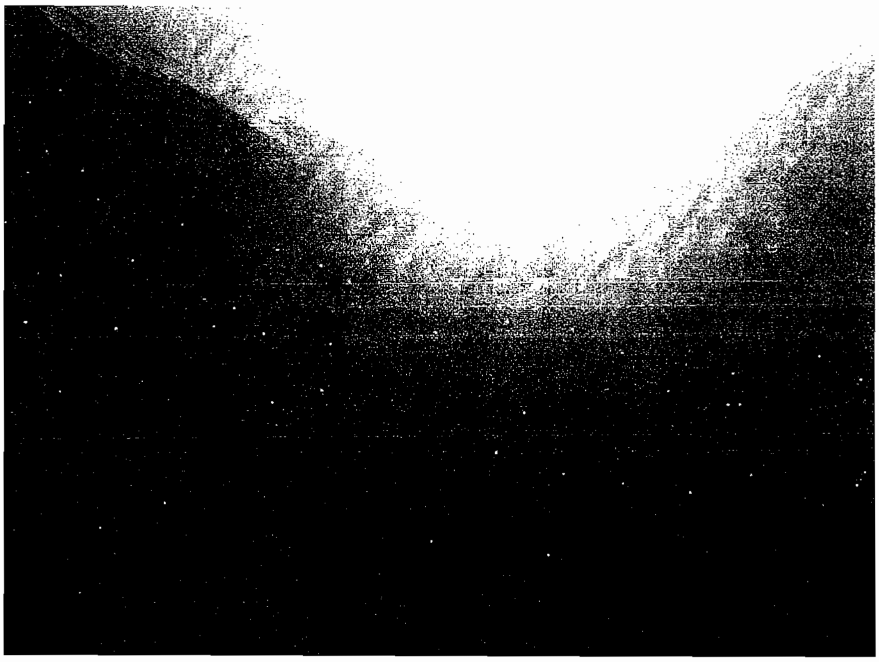

FIGURE 5.5.1 A sample screenshot of our god ray–rendering technique.

This article describes an inexpensive yet visually convincing technique to animate and render underwater god rays in real-time. It requires a graphics card supporting DirectX 9 Shader Model 2. This article also provides HLSL shaders to allow games or simulators to easily adopt this technique.

The Physics of Underwater God Rays

The physical explanation of god rays is clear: beams of sunlight initially refract at the air-water interface and then travel underwater in different directions. Light underwater is partly attenuated and partly scattered by particles or other suspended substances before it eventually reaches an observer. The amount of light reaching an observer depends mainly on the water characteristics and the viewing direction. We ignore second-order physical effects such as multiple scattering and the contribution of skylight. We also ignore the conservation of energy when light shafts diverge or converge, which maintains a constant intensity area over cross sections of a light beam.

Mathematical models accurately describe each of these physical phenomena [Deepocean] [Premoze00]. For refractions, the following formula returns the refracted vector **T** given the incident vector **I**, surface normal **N**, and the refraction index η_1 for the first medium (1.05 for air) and η_2 for the second medium (1.3333 for water).

$$T=\eta I+\left(-\eta I\cdot N-\sqrt{1-\eta^2\left(1-\left(I\cdot N\right)^2\right)}\right)N, \tag{5.5.1}$$

where $\eta = \eta_1/\eta_2$.

The amount of light $T(\theta)$ transmitted from air to water depends on the angle between the incident direction and the water normal; we use Schlick's approximation [Schlick94] to calculate this term (called the Fresnel term):

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5, \tag{5.5.2a}$$

$$T(\theta)=1\text{-}R(\theta) \tag{5.5.2b}$$

where R_0 is 0.0204 for water and $\cos(\Omega)$ is the dot product between the water normal and the incident vector.

We model light extinction, which includes losses owing to absorption and scattering, with an exponential attenuation:

$$I_L(\lambda)=I_0(\lambda)\exp(-c(\lambda)L). \tag{5.5.3}$$

Here λ is the light wavelength (red, green, or blue), $c(\lambda)$ is a wavelength-dependent extinction coefficient, and L is the total distance traveled by light in the attenuating medium. We can use hard-coded or artist-defined coefficients (less for blue, more for red and green) or derive them from the water characteristics, for example, using the physical model described in [Premoze00].

Light scattering is mathematically described by an isotropic phase function $F(\theta)$ that returns the amount of light that scatters toward the viewer, given the angle θ between the light direction and the viewing direction. Among the several functions proposed in the literature, we adopt the Henyey-Greenstein phase function [Henyey41] for our technique; this function is roughly wavelength independent and provides strong forward scattering, which is consistent with god rays being brighter when one is looking towards the sun. This function is:

$$\mathrm{F}(\theta)=\beta_M\frac{1-g^2}{(1+g^2-2g\cos(\theta))^{3/2}} \tag{5.5.4}$$

where β_M is the Mie coefficient and g is the eccentricity, which controls the power of forward scattering. For our purposes we can ignore β_M and use g equal to 0.5.

DirectX's HLSL includes all intrinsic functions necessary to reproduce these formulas in a shader. In particular, the refract instruction is an exact implementation of Equation 5.5.1.

Previous Work

Other work in computer graphics already addresses the problem of rendering god rays. Iwasaki [Iwasaki02] describes a physically based model for rendering underwater light shafts, making use of graphics hardware to accelerate the computation. The technique models each god ray as a parallelepiped that is subdivided into subvolumes. It accurately calculates the light intensity for each subvolume and accumulates the results in the frame buffer by using hardware color blending. It also simulates shadows via shadow mapping. The accuracy of this technique comes at the cost of prohibitively high rendering times.

NVIDIA simulates god rays with an image filter in the demo *Nalu* [NVIDIA04]. The *Nalu* demo first renders the bright region corresponding to the refracted sun to a render target, then renders shadow casters and subtracts them from the bright pixels. Finally, it radially blurs the image, creating an illusion of god rays. This technique is quite simple but only works when the sun is visible. It is also impossible to control the animation of god rays, as it indirectly depends on the rendering of the refracted sun disk.

Mitchell [Mitchell04] and Dobashi [Dobashi02] render light shafts by slicing the view frustum into a set of parallel planes. For each plane, they discretize the integrals of the mathematical simulation of god rays and accumulate the resulting intensity into the frame buffer. Shadows are simulated with shadow mapping. This technique is rather expensive, as many planes are necessary to obtain high-quality visual results that are free from severe aliasing. Apart from performance issues, this technique also cannot directly control the animation of light shafts.

Jensen [Jensen01] simulates underwater god rays by slicing the view volume into many (e.g., 32 in the paper) planes and projecting an animated caustics texture onto each plane, assuming that this texture represents the intensity of god rays for each plane.

Our Technique

Our technique is loosely inspired by [Iwasaki02] and works as follows: some light shafts, geometrically modeled as parallelepipeds, are animated in a vertex shader by refracting their supporting vectors against a procedurally animated water surface. They then render to a texture with additive alpha-blending that takes their depth attenuation and other physical phenomena into account. We simulate shadows via standard shadow mapping [Everitt00]. The resulting texture accumulates the approximate intensity of god rays in the scene. In a second pass we additively blend a full-screen quad to the frame buffer; for each pixel we calculate the amount of light scattered toward the viewer and multiply this result with the light intensity fetched from the first texture, obtaining an approximation of the god rays' intensity at that pixel. The following paragraphs describe each step in detail.

Geometrical Model

Each light shaft is modeled as a parallelepiped, as shown in Figure 5.5.2. The top face lies on the water surface, and the lateral edges correspond to refracted sun rays extending underwater to a certain depth.

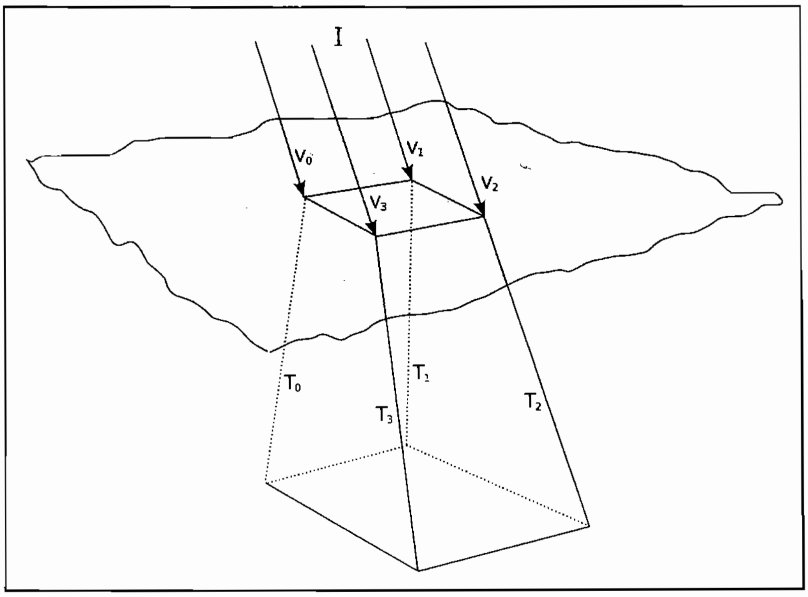

FIGURE 5.5.2 Geometrical model of a god ray. $\mathbf{V}_i$ stand for vertices, $\mathbf{T}_i$ for the corresponding refracted vectors, and **I** is the incoming sunlight, assumed constant for all god rays.

We render many god rays with a single indexed primitive. Two sets of vertices represent the god rays' geometry, one set for the top faces and one for the bottom faces. These vertices are arranged on a regular square grid located on the water surface; the top vertices are assigned a depth of 0, while the bottom vertices are assigned a maximum depth (e.g., 30 m). This depth value is used in the vertex shader to move the bottom vertices down along the direction of the refracted sunlight, creating an extruded parallelepiped for each god ray.

A static vertex buffer stores all vertices centered around the point (0, 0, 0) with a spacing of 1. A vertex shader translates and scales these vertices according to the current viewer's position and depth to optimize the distribution of light beams over the visible water surface. The verrices are first recentered around the image of the sun on the water surface, which corresponds to the point $\hat{\mathbf{P}}$ given as the intersection between the water plane and a ray traveling from the viewer in the direction of the refracted sunlight, assuming flat water (see Figure 5.5.3). It equals

$$\hat{\mathrm{P}} = \mathrm{E} + \mathrm{T}\left(h_0 - \mathrm{E}_y\right)/\mathrm{T}_y, \tag{5.5.5a}$$

$$\mathbf{T} = \mathrm{refract}(\mathbf{I},\mathbf{N},1/1.333), \tag{5.5.5b}$$

where $\mathbf{I}$ is the incident sunlight, $\mathbf{N} = (0,1,0)$ is the normal for flat water, $\mathbf{E}$ is the viewer position, and h_0 is the water altitude.

God rays are arranged optimally when they completely cover the visible water surface. Assuming that the viewer looks upward, the spacing between god rays should thus be linearly proportional to the viewer's depth, because the area of the visible water surface over which the god rays are arranged increases as the viewer moves deeper. The following formula computes the optimal spacing:

$$S = K\frac{2d\tan\left(fov/2\right)}{N} \tag{5.5.6}$$

where d is the viewer's depth, fov is the field of view, N is the number of god ray vertices along the grid and K is an adjustment factor (around 0.8).

Animation

We mentioned in the previous section that the parallelepipeds representing god rays extrude in the direction of sunlight refracted by an animating water surface.

How to animate water is outside the scope of this article. For example, many recent games use a statistical model evaluated with an FFT [Jensen01]. This method generates heights and normals for a realistic simulation of water but comes at a high cost for our simple needs and does not work well on Shader Model 2 GPUs, where vertex shaders cannot access textures.

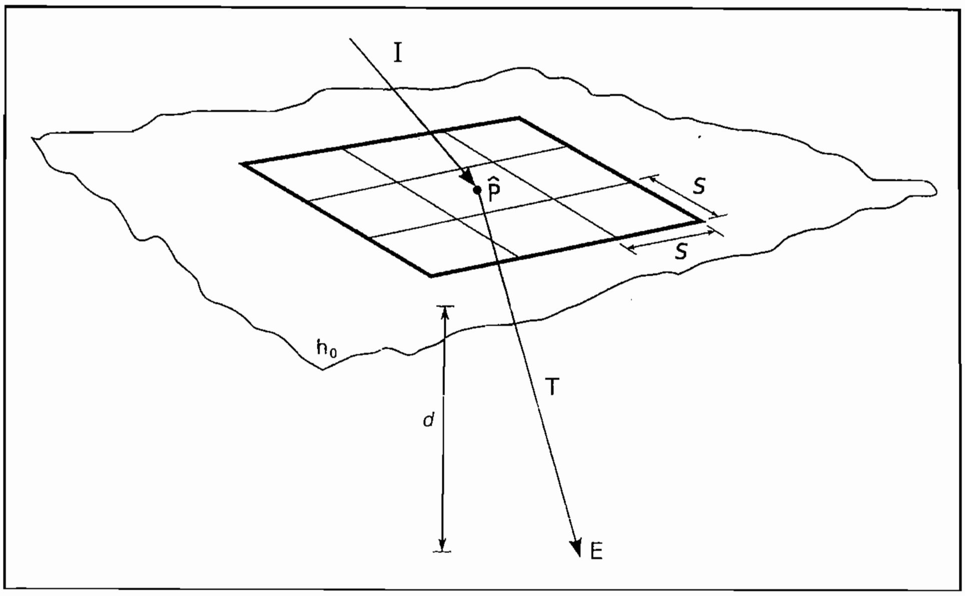

FIGURE 5.5.3 Central position and spacing of vertices on the water's surface.

We propose a procedural way to animate water entirely on the GPU. We model water as a sum of Gerstner waves [Hinsinger02]. Given a point $P = (x_0, y_0, z_0)$ on the water surface, this point displaces as follows:

$$\begin{cases} \varphi_i = K_i \cdot X_0 - w_i t + \varphi_i^0 \\ X = X_0 - \sum_{i=1}^{N} a_i \frac{K_i}{\|K_i\|} \sin(\varphi^i) \\ y = y_0 + \sum_{i=1}^{N} a_i \cos(\varphi^i) \end{cases} \tag{5.5.7}$$

where $X_0 = (x_0, z_0)$ is the location of the point at rest, y_0 is its altitude at rest and a_i, w_i, K_i, and φ_i^0 are the amplitude, frequency, wave number, and initial phase of the *i*th wave, respectively. The resulting shape is similar to a sinusoid for very small amplitudes, but is sharper otherwise.

From Equation 5.5.7, we obtain an analytical expression for the gradients **T** and **B** through differentiation:

$$\begin{cases} T_x = -\sum_{i=1}^{N} \frac{a_i (K_{ix})^2}{\|K_i\|} \cos(\varphi^i) \\ T_z = -\sum_{i=1}^{N} \frac{a_i K_{ix} K_{iy}}{\|K_i\|} \cos(\varphi^i) \\ T_y = -\sum_{i=1}^{N} a_i K_{ix} \sin(\varphi^i) \\ B_x = -\sum_{i=1}^{N} \frac{a_i K_{ix} K_{iy}}{\|K_i\|} \cos(\varphi^i) \\ B_z = -\sum_{i=1}^{N} \frac{a_i (K_{iy})^2}{\|K_i\|} \cos(\varphi^i) \\ B_y = -\sum_{i=1}^{N} a_i K_{iy} \sin(\varphi^i) \end{cases} \tag{5.5.8}$$

The water normal is the normalized cross product of the two gradients. The following HLSL function optimally computes the water normal by considering four waves at a time.

```
// Waves constants
// kx, ky, A/sqr(kx*kx + ky*ky), A, wt+phi0
float4   waves[NUM_WAVES/4 * 5];

float3 CalculateWaterNormal(float x0, float y0) {
  float3 t1 = float3(1,0,0);
  float3 t2 = float3(0,0,1);

  for (int i = 0, j = 0; i < NUM_WAVES/4; i++, j += 5) {
    float4 kx    = waves[j];
    float4 ky    = waves[j+1];
    float4 Ainvk = waves[j+2];
    float4 A     = waves[j+3];
    float4 wt    = waves[j+4];
    float4 phase = (kx*x0 + ky*y0 - wt);
    float4 sinp, cosp;
    sincos(phase, sinp, cosp);

    // Update tangent vector along x0
    t1.x -= dot(Ainvk, kx*cosp*kx);
    t1.z -= dot(Ainvk, ky*cosp*kx);
    t1.y += dot(A, (-sinp)*(kx));

    // Update tangent vector along y0
    t2.x -= dot(Ainvk, kx*cosp*ky);
    t2.z -= dot(Ainvk, ky*cosp*ky);
    t2.y += dot(A, (-sinp)*(ky));
  }
```

```
    // Calculate and return normal
    return normalize( cross(t2, t1) );
  }
```

The CPU has to generate the parameters a_i, $w_i t + \varphi_i^0$, and $\mathbf{K}_i$ for some number of waves (eight typically suffices) and pass them to vertex shader constants in the appropriate format. The code on the accompanying CD-ROM provides the data structures and utility functions.

ON THE CD

The vertex shader processing the god rays' vertices calculates a new position in world space for each vertex. First, the spacing and position of vertices on the water surface is adjusted. Next, for the bottom vertices, we refract the incoming sunlight direction against the water normal using the HLSL instruction refract. We then obtain the final vertex position in world space by multiplying the refracted vector by the extrusion length stored in the vertex structure passed to the shader. This position is finally transformed to clip space and passed as output. The shader also calculates the total light attenuation from the water surface to the vertex position and from the vertex to the viewer, as well as the Fresnel term for the amount of light transmitted from air to water. The rasterizer then linearly interpolates the resulting light intensity over the polygon and passes it to the pixel shader. Using vertex computations and subsequent interpolation is cheaper than calculating the intensity in the pixel shader, although that would be more correct. The final HLSL vertex shader is:

```
// Constants
float3   origin;           // central position of vertices
float    spacing;          // new spacing between vertices
float3   extinction_c;     // extinction coefficient
float3   viewer;           // viewer in world space
float3   sunDir;           // sunlight direction
float4x4 shadowmapMatrix;  // projective shadowmap matrix


// Input and output vertex structure
struct VS_INPUT {
  float4 pos       : POSITION;
  float  length    : TEXCOORD0;
};

struct VS_OUTPUT {
  float4 pos          : POSITION;
  float3 intensity    : TEXCOORD0;
  float4 shadowmapPos : TEXCOORD1;
};


// Fresnel approximation, formula 2
float FastFresnel(float3 I, float3 N, float R0) {
  return R0 + (1-R0)*pow(1-dot(I, N), 5);
}
```

```
VS_OUTPUT VS(const VS_INPUT input) {
  VS_OUTPUT output = (VS_OUTPUT)0;

  // Scale and translate the vertex on the water surface
  float3 worldPos = input.pos.xyz*float3(spacing,1,spacing);
  worldPos += origin;

  // Calculate the water normal at this point
  float3 normal = CalculateWaterNormal(worldPos.x, worldPos.z);

  // Extrude bottom vertices along the direction of the refracted
  // sunlight
  if (input.length > 0) {
    // Calculate refraction vector and extrude polygon
    float3 refr = refract(sunDir, normal, 1./1.333);
    worldPos += refr*input.length;
  }

  // Calculate transmittance
  float tr = 1-FastFresnel(-sunDir, normal);

  // Calculate god ray intensity
  float totalDist = input.length + length(worldPos-viewer);
  output.intensity = exp(-totalDist*extinction_c)*tr;

  // Calculate position in light space for shadowmapping
  output.shadowmapPos = mul(shadowmapMatrix,
  float4(worldPos,1) );

  // Transform position from world to clip space
  output.pos = mul(viewProjMatrix, float4(worldPos, 1) );
  // Tweak z position not to clip shafts very close to the viewer
  output.pos.z = 0.01;

  return output;
}
```

With reference to the above code, `extinction_c` is the attenuation coefficient from Equation 5.5.3, while `shadowmapMatrix` is a matrix that transforms the vertices to light space as part of shadow mapping (see the following "Rendering" section).

Rendering

Underwater god rays extend in many directions and usually cover the entire view volume. As a result, their rendering involves processing every screen pixel. When light shafts overlap, screen pixels are processed multiple times. To achieve high frame rates, it is thus imperative to adopt approximations that reduce the cost of rendering god rays. For this purpose, we break their rendering into two steps. The first step blends shadowed light shafts into a temporary render target using a relatively inexpensive pixel shader. The second step operates in image space; it reads the previous step's render target for each screen pixel and modulates the result with a phase function that improves the realism of the rendered scene. The resulting color is blended into the frame buffer, adding the light shafts on top of a previously rendered scene.

Overall, this approach greatly reduces fill rate, which tends to be the bottleneck when rendering god rays. By moving most of the calculations from the pixel to the vertex shader, we sacrifice physical accuracy in favor of rendering performance. We now discuss this approach in more detail.

Since we adopt shadow mapping to simulate shadows, the first step implies rendering the shadow casters to a texture from the point of view of the light source $\hat{P}$ (Equation 5.5.5a), storing the distance to the nearest caster for each pixel.

Next, a number of underwater light beams (e.g., 100) are rendered to a temporary render target with additive blending and z-buffering disabled, using the vertex shader described in the previous section. This step accumulates an approximate light intensity in the render target without relying on complex physical models. Because we use additive blending, the render target contains high values where many light beams overlap, and low values where few or no beams exist.

For performance reasons, it is convenient to use a render target smaller than the current frame buffer, because rendering many alpha-blended polygons stresses the fill rate and bandwidth of graphics cards, especially older ones. We find a quarter of the frame buffer to be a good compromise between rendering performance and quality. The format of this texture (D3DFMT_A8R8G8B8 in DirectX) has 8-bit precision per channel, resulting in a range of 0 to 255 for each color, providing enough precision to represent the average number of god rays overlapping at each pixel. Furthermore, all graphics cards support alpha blending on render targets of this format.

The pixel shader that renders the light shafts is simple and efficient. It first determines whether the pixel is in shadow or not (by comparing its distance from the light source to the corresponding distance stored in the shadow map). We do not filter the shadow map, as that degrades performance without noticeable quality improvements. The pixel shader then multiplies the result of the shadow comparison with the interpolated per-vertex color. The pixel shader is

```
texture shadowmap;

sampler2D shadowmapSam = sampler_state {
  Texture = <shadowmap>;
  MinFilter = Point;
  MagFilter = Point;
  MipFilter = Point;
  AddressU  = Clamp;
  AddressV  = Clamp;
};


float4 PS(VS_OUTPUT input) : COLOR {

  // Calculate distance to light source
  float z = input.shadowmapPos.z/input.shadowmapPos.w;
```

```
    // Fetch shadowmap
    float sz = tex2Dproj(shadowmapSam, input.shadowmapPos).x;

    float lit = (z < sz)? 1 : 0;
    return float4(input.outScattering*lit, 1);
  }
```

A second rendering pass then completes the rendering of god rays. We render a quad covering the entire screen with additive alpha-blending turned on. For each pixel, we read four bilinear samples from the previously rendered texture and average them. This averaging blurs the god rays' intensities. This averaged intensity is then modulated with the percentage of light scattered toward the viewer. The phase function takes as input two arguments: the view direction and the light direction. We calculate the former for each pixel by performing an inverse projection from screen to view space. The light direction theoretically differs from shaft to shaft, but this information is unavailable. We therefore use a constant light direction calculated according to Equation 5.5.5b. As a final touch, we add a small bias to the god rays' intensities that greatly minimizes their aliasing where scattering is strong (see the pixel shader below). The complete vertex and pixel shaders for the second pass follow.

```
// Constants
float4x4 invViewProjMatrix; // matrix from clip to world space
float3   HGg;               // packed Mie parameters
float3   refrSunDir;        // refracted sun direction
float3   intensity;         // god rays intensity
float2   invTexWidth;       // inverse of texture width, height
float3   viewer;            // viewer in world space

// Input and output vertex structure
struct VS_INPUT {
  float4 pos        : POSITION;
  float2 uv         : TEXCOORD0;
};

struct VS_OUTPUT {
  float4 pos        : POSITION;
  float4 uv[2]      : TEXCOORD0;
  float3 viewVector : TEXCOORD2;
};


// Mie phase function
float ComputeMie(float3 viewDir, float3 sunDir) {
  float den = (HGg.y - HGg.z*dot(sunDir, viewDir));
  den = rsqrt(den);
  float phase = HGg.x * (den*den*den);
  return phase;
}
```

```
VS_OUTPUT VS(const VS_INPUT input) {
  VS_OUTPUT output = (VS_OUTPUT)0;

  output.pos = input.pos;

  // Calculate texture coordinates for four samples
  output.uv[0].xy = input.uv;
  output.uv[0].zw = input.uv + float2(invTexWidth.x, 0);
  output.uv[1].xy = input.uv + invTexWidth;
  output.uv[1].zw = input.uv + float2(0, invTexWidth.y);

  // Transform input position from clip to world space
  output.viewVector = mul(invViewProjMatrix, input.pos)-viewer;

  return output;
}

float4 PS(VS_OUTPUT input) : COLOR {
  // Read godrays intensity, averaging four samples to
  // reduce aliasing
  float4 shafts = 0;
  for (int i = 0; i < 2; i++) {
    shafts += tex2D(godraysSam, input.uv[i].xy);
    shafts += tex2D(godraysSam, input.uv[i].zw);
  }
  shafts /= 4;

  float3 viewVector = normalize(input.viewVector);
  float phase = ComputeMie(-viewVector, refrSunDir);

  // Calculate final color, adding a little bias (0.15 here)
  // to hide aliasing
  float3 color = (0.15 + intensity*shafts.xyz)*phase;
  return float4(color, 1);
}
```

In the code above `invViewProjMatrix` is the inverse of the transformation matrix from world to clip space, `HGg` is the vector $(1-g^2, 1+g^2, 2g)$, where g is the constant from Equation 5.5.4, intensity is a user-defined multiplication factor, and `invTexWidth` is the inverse of the width and height of the render target storing the shafts' intensities.

Conclusion

This article describes an inexpensive way of simulating underwater god rays on common graphics hardware. The proposed technique takes advantage of reasonable approximations to greatly accelerate the rendering of god rays and makes it feasible for interactive applications such as games or simulators. Our technique works entirely on the GPU by exploiting programmable vertex and pixel shaders and render-to-texture capabilities. Although our implementation works with DirectX and HLSL, the described algorithms and shaders easily port to other rendering APIs, for example, OpenGL. The demo on the companion CD-ROM features an underwater scene powered by the

ON THE CD

Typhoon engine developed by the author, with caustics, underwater scattering, and god rays simulated using the described technique.

References

[Deepocean] Available online at *http://www.deepocean.net.*

[Dobashi02] Dobashi, Y., T. Yamamoto, and T. Nishita. "Interactive Rendering of Atmospheric Scattering Effects using Graphics Hardware." *Proc. Graphics Hardware* (2002): pp. 99–108.

[Everitt00] Everitt, C., A. Rege, and C. Cebenoyan. "Hardware Shadow Mapping." Tech. rep., NVIDIA Corp., 2000. Available online at *http://www.nvidia.com.*

[Henyey41] Henyey, L. G. and J. L. Greenstein. "Diffuse Radiation in the Galaxy." *Astrophysics Journal 93* (1941): pp. 70–83.

[Hinsinger02] Hinsinger, D., F. Neyret, and M. P. Cani. "Interactive Animation of Ocean Waves." *Proceedings of the ACM SIGGRAPH Symposium on Computer Animation.* 2002: pp. 161–166.

[Iwasaki02] Iwasaki, K., Y. Dobashi, and T. Nishita. "An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware." *Computer Graphics Forum*, Volume *21*(4) (November 2002).

[Jensen01] Jensen, L.S. and R. Golias. "Deep Water Animation and Rendering." Available online at *http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm.*

[Mitchell04] Mitchell, J. "Light Shafts Rendering Shadows in Participating Media." Available online at *http://www.ati.com/developer/gdc/Mitchell_LightShafts.pdf.*

[NVIDIA04] NVIDIA's *Nalu* Demo. Available online at *http://www.nzone.com/object/nzone_nalu_home.html.*

[Premoze00] Premoze, S. and M. Ashikhmin. "Rendering Natural Waters." *Proceedings of Pacific Graphics.* 2000, pp. 23–30.

[Schlick94] Schlick, C. "An Inexpensive BDRF Model for Physically-Based Models." *Computer Graphics Forum*, 1994.