Pond Water

Frank Luna
www.moon-labs.com

Created on Sunday, February 12, 2006

This short paper presents an outline to the *Pond Water* demo (see Figure 1), which implements a popular water technique, described in [Vlachos02], using reflection and refraction maps. We assume the reader has studied the topics presented in [Luna06], as we use several techniques described therein; in particular, the reader should already be familiar with rendering to a texture, normal mapping, texture animation, projective texturing, and per pixel lighting.

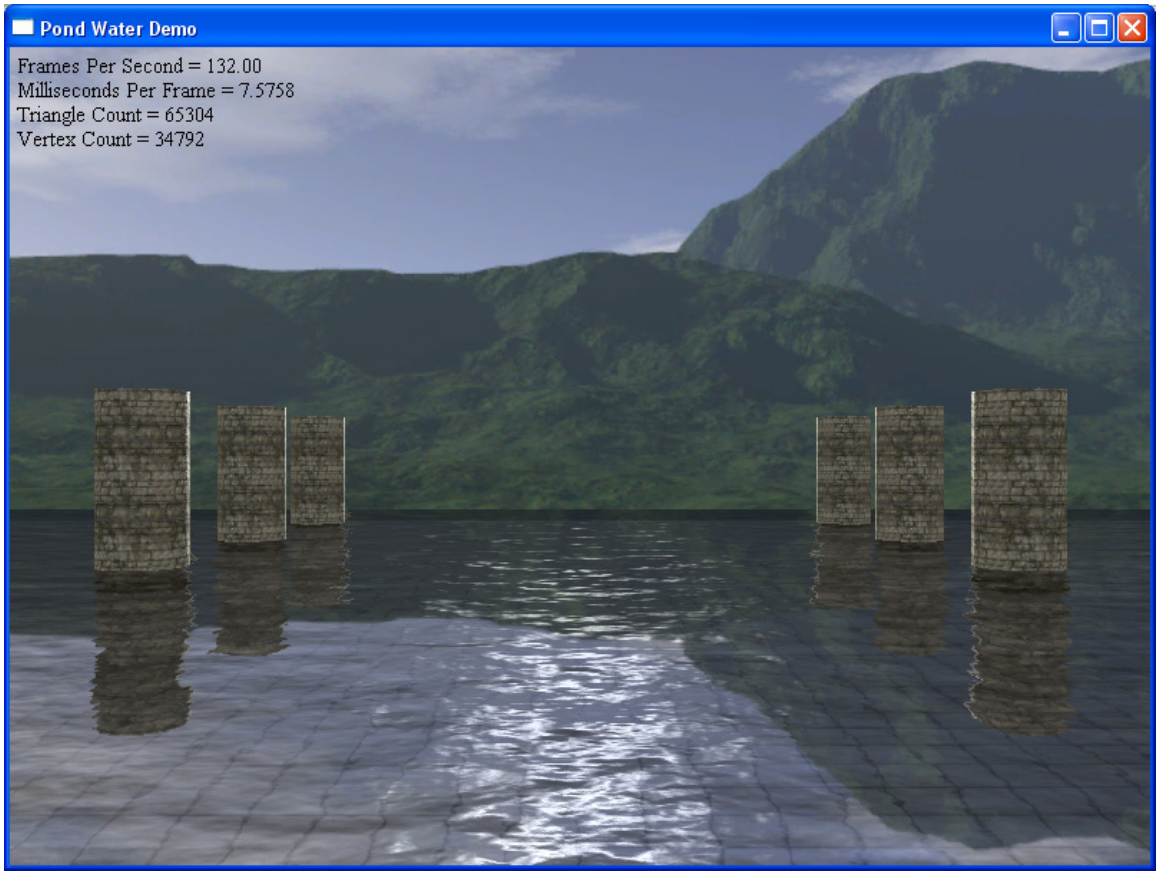

Figure 1: Screenshot of the technique.

1 Reflection and Refraction

We color our water by combining the effects of: 1) reflection—water is reflective and can act as a mirror; 2) refraction—water is transparent and the light rays bend as they pass between the air and water interfaces; see *refraction* in any introductory physics book; 3)

impurities—the particles suspended in the water affect the appearance of the water (e.g., dirty pond water looks much different from clean pool water); and 4) lighting.

2 Implementation

2.1 Refraction

For each rendering frame, we first render the scene (except the water plane) as usual to a texture. This forms the refractive map. Later, when we render the water plane, we will project this texture onto the water plane (the camera will be the projector). The part of the refraction map that gets projected onto the water plane will correspond to the pixels seen through the water due to transparency (i.e., exactly the pixels we need for refraction). Figure 2 illustrates.

```
DrawableTex2D* refractMap = mWater->mRefractMap;

refractMap->beginScene();

// Draw the scene, except the water plane to the refraction map.
drawAll(0);

refractMap->endScene();
```

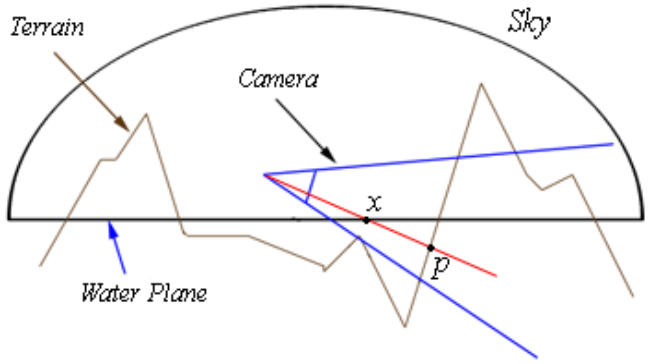

Figure 2: We render the scene as usual to a refractive map, and then project it onto the water surface. Observe that the pixel *p* on the refractive map that is projected onto the pixel *x* of the water surface corresponds to the pixel directly behind the water surface from the camera's viewpoint (i.e., it is the pixel the viewer would see behind the water surface due to transparency).

2.2 Reflection

For each rendering frame, we also render the scene (except the water plane*) reflected about the water* plane into a texture. This forms the reflective map. Later, when we render the water plane, we will also project this texture onto the water plane (the original camera will be the projector). The part of the reflection map that gets projected onto the water plane will correspond to the reflected scene above the water plane, which is exactly what we need to render reflections on the water plane.

The code to build the reflection map is as follows:

```
DrawableTex2D* reflectMap = mWater->mReflectMap;

HR(gd3dDevice->SetRenderState(D3DRS_CLIPPLANEENABLE, 1));
HR(gd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW));
reflectMap->beginScene();
gd3dDevice->Clear(0, 0, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
     0x00ffffff, 1.0f, 0);

// Reflection plane in local space.
D3DXPLANE waterPlaneL(0.0f, -1.0f, 0.0f, 0.0f);

// Reflection plane in world space.
D3DXMATRIX WInvTrans;
D3DXMatrixInverse(&WInvTrans, 0, &(mWaterWorld));
D3DXMatrixTranspose(&WInvTrans, &WInvTrans);
D3DXPLANE waterPlaneW;
D3DXPlaneTransform(&waterPlaneW, &waterPlaneL, &WInvTrans);

// Reflection plane in homogeneous clip space.
D3DXMATRIX WVPInvTrans;
D3DXMatrixInverse(&WVPInvTrans, 0, &(mWaterWorld*gCamera->viewProj()));
D3DXMatrixTranspose(&WVPInvTrans, &WVPInvTrans);
D3DXPLANE waterPlaneH;
D3DXPlaneTransform(&waterPlaneH, &waterPlaneL, &WVPInvTrans);
HR(gd3dDevice->SetClipPlane(0, (float*)waterPlaneH));

// Draw the scene, but reflected across the water plane in world space.
drawAll(&waterPlaneW);

reflectMap->endScene();
HR(gd3dDevice->SetRenderState(D3DRS_CLIPPLANEENABLE, 0));
HR(gd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW));
```

Here we have used a user-defined clip plane to clip reflected geometry above the water plane. Recall that the graphics card clips geometry against the six frustum planes. In addition, we can set user-defined clip planes, which the graphics card will also clip against. User-defined clip planes must be specified in homogeneous clip space. If we do not set a clip plane in this way, then geometry may be rendered into the reflection map that obscures the reflected geometry we want to see; see Figure 3.

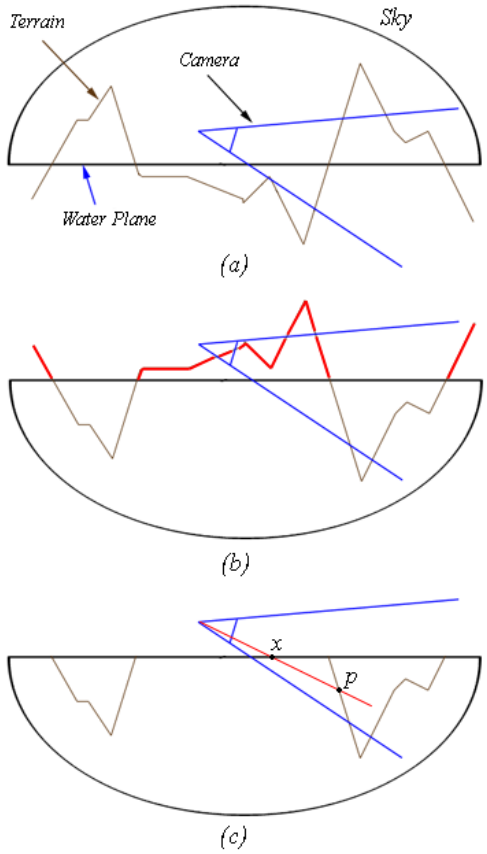

Figure 3: (a) The normal scene. (b) The scene after the terrain has been reflected across the water plane. Observe that there is reflected geometry above the water plane that would make it into the reflection map. This is incorrect because only geometry initially above the water plane should be reflected and rendered into the reflection map. (c) The scene after reflection, with geometry above the water plane clipped. Now, only geometry initially above the water plane is reflected and rendered into the reflection map, which is correct.

To summarize, after we have built the refraction and reflection maps and projected them onto the water plane, for each water plane pixel, there is a corresponding refraction and reflection texel associated with it; see Figure 4. We can then blend these refraction and reflection texels together at each pixel, along with material and lighting results, to produce the final pixel colors for the water.

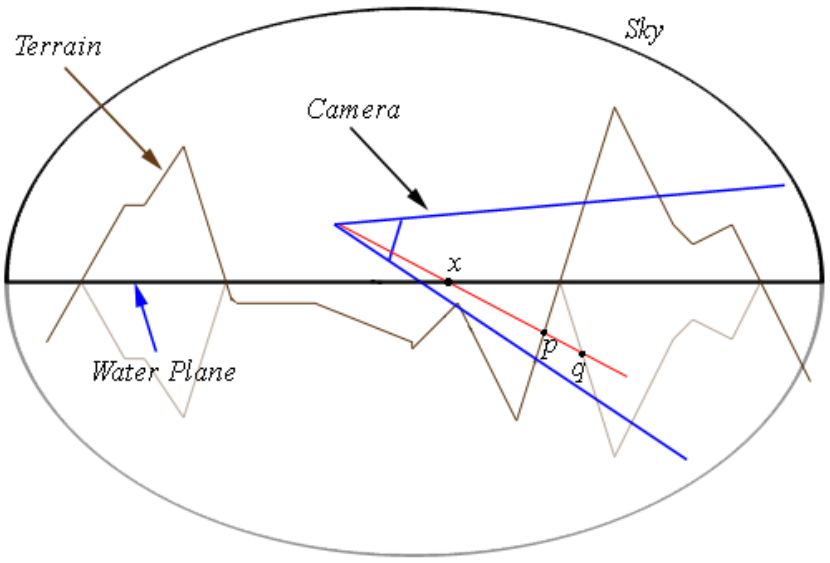

Figure 4: The reflection and refraction scenes superimposed. After projecting the refraction and reflection map onto the water plane, observe that for each water plane pixel *x*, a refracted pixel *p* and a reflected pixel *q* are projected onto it. We can color *x* by blending *p* and *q* in various ways.

2.3 Impurities and Lighting

The effects of dirt, algae, etc., suspended in the water can be taken into account by tweaking the water's material properties or with separate color texture layers. We light the water surface as we did in Chapter 20 of [Luna06], that is, by scrolling two wave normal maps at different frequencies across the water plane and doing per pixel lighting. Displacement mapping, as discussed in Chapter 21 of [Luna06] can also be added to model waves on a geometric level.

3 Waves/Ripples

We have colored the surface of the water by combining the pixels of reflected and refracted geometry. However, this does not look like water yet, since we have not modeled waves and ripples. To simulate waves and ripples, we use the animated *xz*-coordinates of the scrolling normal maps used in lighting to perturb the projective texture coordinates used to sample the reflection and refraction maps. This creates a rippling effect, which distorts the images shown on the water plane; see Figure 5.

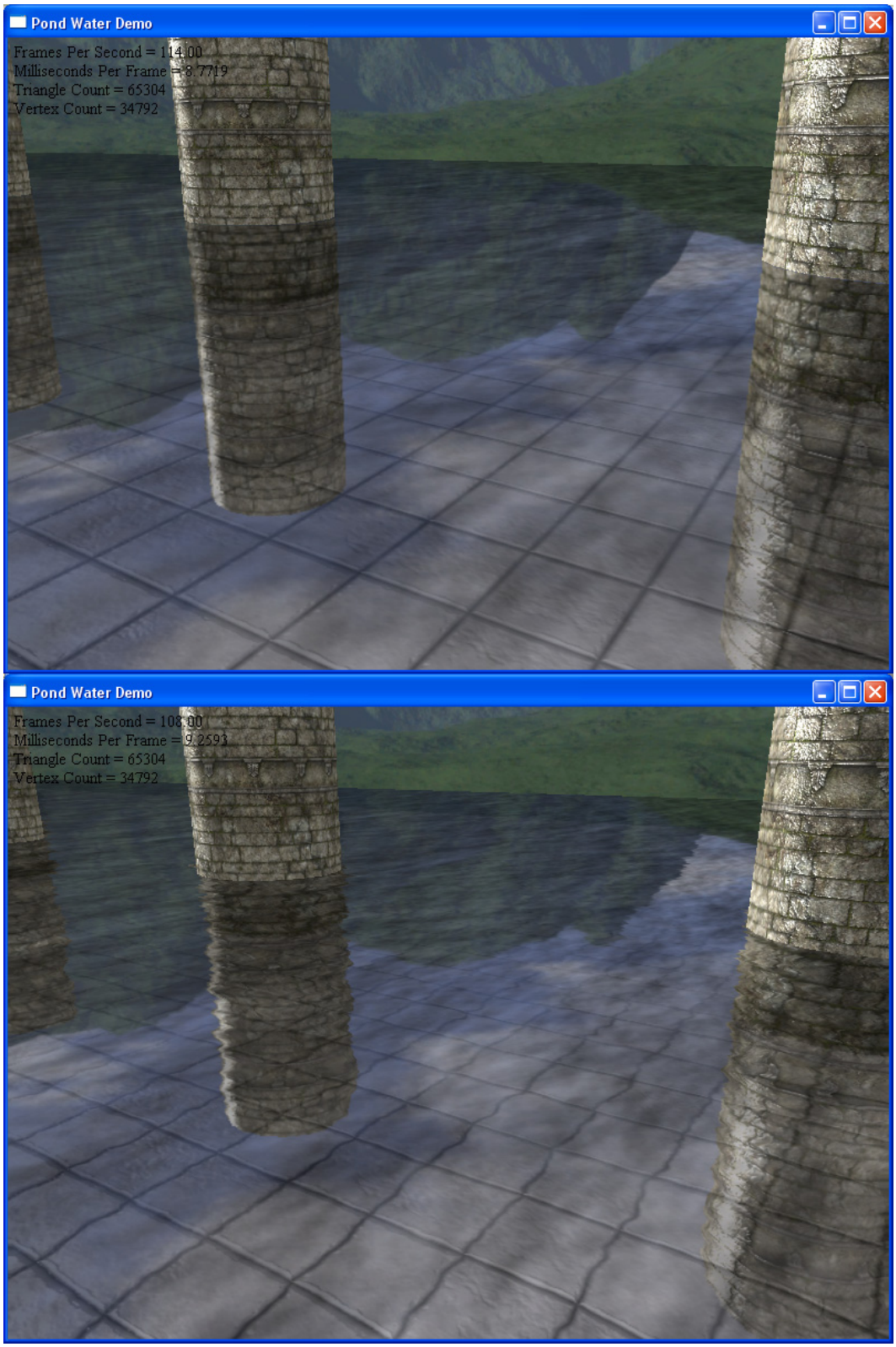

Figure 5: On the top, the texture coordinates are not perturbed and the reflection and refraction is very clean. On the bottom, the texture coordinates are perturbed, which causes wavy distortions to simulate water waves and ripples.

The following code fragment shows how the texture perturbation is done in the pixel shader:

```
// Average the two vectors.
float3 normalT = normalize(0.5f*(normalT0 + normalT1));

...

// Project the texture coordinates and scale/offset to [0,1].
projTexC.xy /= projTexC.w;
projTexC.x =  0.5f*projTexC.x + 0.5f;
projTexC.y = -0.5f*projTexC.y + 0.5f;

// Sample reflect/refract maps and perturb texture coordinates.
// Perturb more along u-axis than v-axis.
float3 reflectCol = tex2D(ReflectMapS,
      projTexC.xy+(normalT.xz*gRippleScale)).rgb;
float3 refractCol = tex2D(RefractMapS,
      projTexC.xy+(normalT.xz*gRippleScale)).rgb;
```

Here, `gRippleScale` is a `float2` effect parameter that specifies the magnitude of the texture coordinate perturbation along the *u*- and *v*-axes.

4 Fresnel Effect

The magnitude of reflection and refraction is viewpoint dependent (i.e., they depend on where the viewer is). For example, when the angle between the view vector (i.e., the vector from a point on the surface to the camera) and surface normal is near, say, $90°$, the water will appear almost 100% reflective; on the other hand, if the angle between the view vector and surface normal is near $0°$, the water will appear almost 100% refractive. The variation of reflection and refraction with the view angle is called the *fresnel effect*. Although we will not model the fresnel effect is a physically accurate way, a simple hack achieves visually satisfactory results.

We start with a formula reminiscent to the diffuse lighting calculation in which the intensity of diffuse light a pixel received was a function of the angle between the light vector and the normal vector. In our case, the amount of refraction is a function of the view vector and the normal vector:

$$w = w(\theta) = \max(\cos\theta, 0) = \max(\vec{n}\cdot\vec{v}, 0).$$

Note that $0 \le w(\theta) \le 1$. The graph of this function is shown in Figure 6. The angle θ is the angle between the unit view vector $\vec{v}$ (vector from surface point to the camera) and the unit normal vector $\vec{n}$. We define the amount of reflection by $(1-w)$, which means that as w (the amount of refraction) decreases the amount of reflection increases, and conversely. Thus, from Figure 6, as θ varies from $0°$ to $90°$, the amount of refraction decreases, which, in turn, causes the amount of reflection to increase.

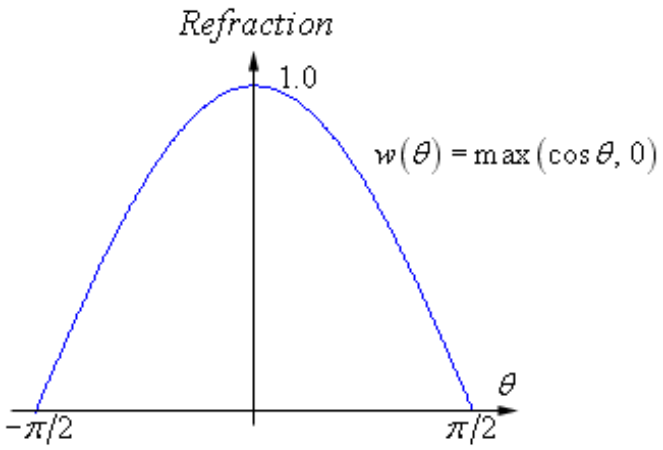

Figure 6: A simple function that controls the amount of refraction based on the angle between the normal vector and view vector.

The combined color is a weighted average between the refracted color and reflected color:

$$\begin{aligned} combinedColor &= w \cdot refractedColor + (1-w) \cdot reflectedColor \\ &= lerp(reflectedColor, refractedColor, w) \end{aligned}$$

Example

If $\theta = 45°$, then

$$\begin{aligned} combinedColor &= (0.707) \cdot refractedColor + (0.293) \cdot reflectedColor \\ &= lerp(reflectedColor, refractedColor, 0.707) \end{aligned}$$

That is, the combined color for $\theta = 45°$ is approximately given by 70% of the refracted color and 30% of the reflected color.

If $\theta = 0°$, then

$$\begin{aligned} combinedColor &= (1.0) \cdot refractedColor + (0.0) \cdot reflectedColor \\ &= lerp(reflectedColor, refractedColor, 1.0) \\ &= refractedColor \end{aligned}$$

That is, the combined color for $\theta = 0°$ is given by 100% of the refracted color and 0% of the reflected color.

If $\theta = 90°$, then

$$\begin{aligned} combinedColor &= (0.0)\cdot refractedColor + (1.0)\cdot reflectedColor \\ &= lerp(reflectedColor, refractedColor, 0.0) \\ &= reflectedColor \end{aligned}$$

That is, the combined color for $\theta = 90°$ is given by 0% of the refracted color and 100% of the reflected color.

The formula given previously for $w(\theta)$ has the general properties we desire, but it is limiting in that we have no artistic control over the function. For instance, for all angles except $90°$, the refraction weight has some nonzero value. We may wish to have the refraction weight falloff to zero for smaller angles than $90°$; we can achieve this by raising the cosine function to a power p. In addition, we may wish to always include at least a constant amount of refraction; we can achieve this by adding a biasing term b (can be positive or negative depending on which direction you want to bias). The new equation looks like this, and its graph is shown in Figure 7 for some fixed p and b:

$$w = w(\theta) = saturate\left(b + \max^{p}(\vec{n}\cdot\vec{v}, 0)\right).$$

An artistic can now tweak the constants p and b to better customize the $w(\theta)$ function in order to get the desired result.

Note: Because of the bias term, we could get a value outside the zero to one range; therefore, we must now clamp the result with the `saturate` function.

Note: The above formula uses a uniform bias term. A bias term that varies per pixel can be achieved with a grayscale map, where each pixel stores a biasing term, which can be normalized to the [0, 1] range; this map is then stretched over the water plane. In this way, you could make the water near the shoreline more refractive than where the water depth is deeper.

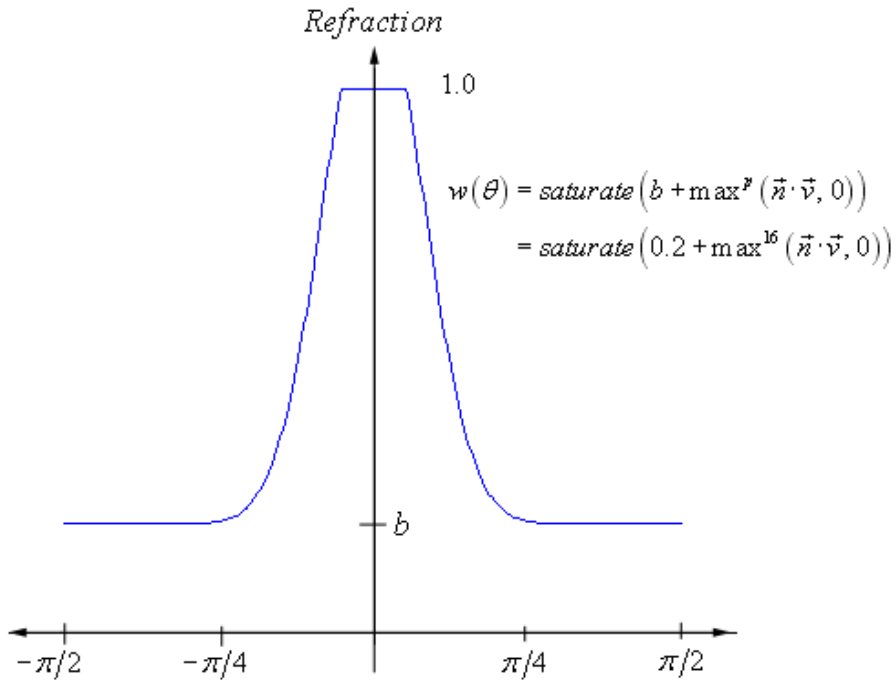

Figure 7: A more complicated function that controls the amount of refraction based on the angle between the normal vector and view vector.

The following code fragment shows how the fresnel effect is approximated in the pixel shader:

```
// To avoid clamping artifacts near the bottom screen edge, we
// scale the perturbation magnitude of the v-coordinate so that
// when v is near the bottom edge of the texture (i.e., v near 1.0),
// it doesn't cause much distortion.  The following power function
// scales v very little until it gets near 1.0.
// (Plot this function to see how it looks.)
float vPerturbMod = -pow(projTexC.y, 10.0f) + 1.0f;

// Sample reflect/refract maps and perturb texture coordinates.
float2 perturbVec = normalT.xz*gRippleScale;
perturbVec.y *= vPerturbMod;
float3 reflectCol = tex2D(ReflectMapS, projTexC.xy+perturbVec).rgb;
float3 refractCol = tex2D(RefractMapS, projTexC.xy+perturbVec).rgb;

// Refract based on view angle.
float refractWt = saturate(gRefractBias+pow(max(
                    dot(toEyeT, normalT), 0.0f), gRefractPower));

// Weighted average between the reflected color and refracted
// color, modulated with the material.
float3 ambientMtrl = gMtrl.ambient*lerp(reflectCol, refractCol, refractWt);
float3 diffuseMtrl = gMtrl.diffuse*lerp(reflectCol, refractCol, refractWt);
```

Note: You may also want the refraction weight to also be a function of depth and distance. That is, as the water gets deeper, the less refraction occurs; similarly, the

further the viewer is from a point on the water's surface, the less refraction occurs. You can create a heightmap (depth map), which describes the water depth at each vertex.

5 Summary

1. We build refraction and reflection maps, which capture the scene image underneath the water surface and the scene image reflected across the water plane; we then project them onto the water plane. For each water plane pixel, there is a corresponding refraction and reflection texel associated with it. We can then blend these refraction and reflection texels together at each pixel, along with material and lighting results, to produce the final pixel colors for the water.

2. To simulate waves and ripples, we scroll two normal maps across the water plane at different velocities, where the averaged animated normals' *xz*-coordinates perturbs the projective texture coordinates used to sample the reflection and refraction maps. This creates a rippling effect, which distorts the images shown on the water plane

4. The Fresnel Effect refers to the amount of reflection and refraction observed based on the angle in which the surface is being viewed.

6 References

[Luna06] Luna, Frank D. *Introduction to 3D Game Programming with Direct X 9.0c: A Shader Approach*. Wordware Publishing Inc., 2006.

[Vlachos02] Vlachos, Alex, John Isidoro, and Chris Oat. "Rippling Reflective and Refractive Water," *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware Publishing Inc., 2002.