Natural Water Shader

Andrej Mihálik
Comenius University in Bratislava

Abstract

This work describes an implementation of optical phenomena on water surfaces. Despite high performance of current graphics hardware, shaders need essential simplifications and numerical approximation. Here we propose the implementation of common effects such as reflection, refraction and caustics. In order to have a simple and elegant implementation we have done coarse approximations to achieve real time animation, while still having a realistic appearance, which is important in real time simulation and games.

Categories: [Computer graphics]: Three-Dimensional Graphics and Realism

Keywords: refraction, reflection, caustics

1 Introduction

Ray tracing is fundamental problem that makes the photorealistic rendering of water surfaces so difficult. Particularly, to achieve real time frame rates. There are still some programming constrains in graphics hardware that need to be overrun. Computation of final color of our surface is performed in a shader program. We will use the OpenGL Shading Language to write the shaders, because it is a standard high level shading language in OpenGL. A fluid represented as a triangular mesh serves as an input to our shader that can be generated by the known fluid dynamic methods. Additional input data is the environment consisting of mesh and textures of objects (e.g. plants, shore, skybox). Generally, the object can be under the water surface, above the water or flowing on the surface. To handle all these cases we should focus on the known solution based on the environmental textures. The primary goal is to achieve realistic appearance using a simple straightforward procedure and deal with constrains. Our focus on photorealism may affect the frame rate.

2 Previous work

Several works in the past were focused on realistic rendering of water surfaces. Previous approaches mostly based on the ray tracing are computationally intensive and can not be used in real time. For this purpose we propose a simplified model of the optical phenomena. Authors in [1] and [2] use two pass rendering to achieve the phenomena like the reflection and the refraction. The essential problem is the color calculation described in [6]. Caustic rendering and additional effects are discussed in [3], and [5]. Paper [7] introduces environment mapping. Physical simulation of waves is proposed in [4] and [1] is out of scope of this research.

3 Refraction and Reflection texture

The effect of reflection and refraction is achieved by two pass rendering algorithm. In the first pass, we split the whole scene into the afloat part and the underwater part by the horizontal clipping plane.

Refraction texture. To obtain refraction texture we keep the camera in its original position and render just underwater part of environment, see Figure 1. Finally, we copy frame buffer into our refraction texture resulting in refraction texture consisting the image of lake bottom.

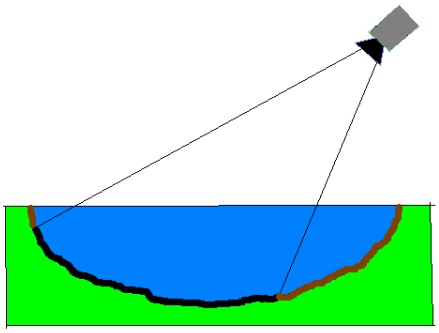

Figure 1. Refraction camera.

Consider the solid objects that are intersected by the plane. After splitting them and removing their afloat part, we may see back faces inside them, see Figure 2.

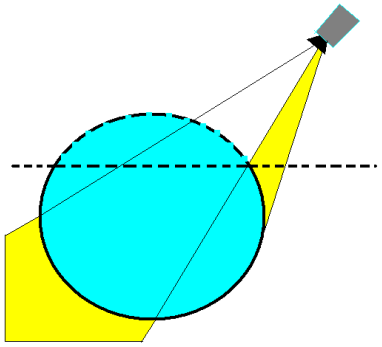

Figure 2. Back face culling

Note that appearance of back faces is not suitable. To avoid this artifact, we should enable back face culling. For future computation (e.g. deep) we may need depth texture of lake bottom. Therefore, it is time to store the depth buffer to a texture at this step.

Once we have finished generating refraction texture, we can start to generate reflection texture in the second pass.

Reflection texture. To obtain this texture we must put the camera upside-down to its mirror position. We render upper part of split environment, as shown in Figure 3. After the rendering it we store the frame buffer to the reflection texture.

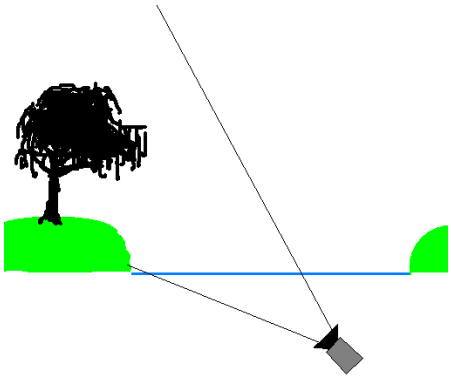

Figure 3. Reflection camera

This approach has the problem of missed texels because rays changed their direction on the water surface due to refraction. This causes that on the surface we see larger area than is actually stored in our refraction and reflection textures, see Figure 4. To restore the missing texture parts we extend field of view before rendering of the reflection and the refraction texture. In our case extending the field of view about 50% was appropriate.

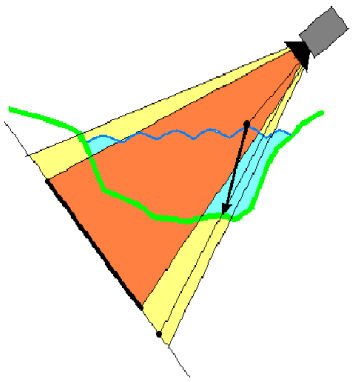

Figure 4. Missing texture parts.

4 Coloring

Let us suppose a lake under the blue sky during daylight. What attributes determine the ambient color of the water surface? It is the illumination and the water mixture. We simplify the outdoor scene to sunlight and sky illumination. The water consists of particles like green algae or other impurities giving

the water its inherent coloring, such as the commonly encountered blue and greenish tone. Once we have obtained the global color of our water, we should deal with its transparency. Density of the mentioned particles affects dirty appearance of the water volume. Scattering of the light in water volume causes its attenuation. Note the visibility of underwater objects is dependent on the amount of water between the object and the viewer. As a result the lake bottom is visible in shallow water but not in deep water, see Figure 5.

Figure 5. Color of the water.

5 Pixel shading

In this section we discuss how to obtain color of certain pixel on water surface. We assume the water surface represented by mesh as an input. Let's summarize that we have available surface refraction texture, depth texture of the bottom, surface reflection texture and global color of the water, at this moment. First, we render whole scene without water using the fixed pipeline. Then we render the surface and compute the color of pixels in fragment shader. Note that we have all the needed stuff is in the shader, now.

Refracted color from texel. When we are treating a certain pixel, we have got coordinates of its corresponding point on surface in camera coordinate system. We also have got surface normal N in this point. We can easily compute normalized eye vector E, because camera is in the origin of camera coordinate system. Once we have the eye and the normal vector, we can compute the normalized refraction vector R using air-water refraction index, see Figure 6. Direction of this refraction vector is conclusive for determining which part of the bottom is mapped to current fragment of the surface. Note that the functions computing reflection and refraction vector are defined in the OpenGL Shading Language. Since, we do not perform the ray casting, we are unable to find where the ray with direction R hits the bottom. We propose to estimate the hitting point by following formula:

$$W = U + dv*R \qquad (1)$$

where U is point on the surface corresponding to current fragment in eye space and dv is length of bold line from the surface to the bottom in Figure 6. The length dv is obtained by comparing Z coordinate of the surface and our depth texture. Large values of dv may cause some artifacts. Therefore, we should clamp and scale the value of dv.

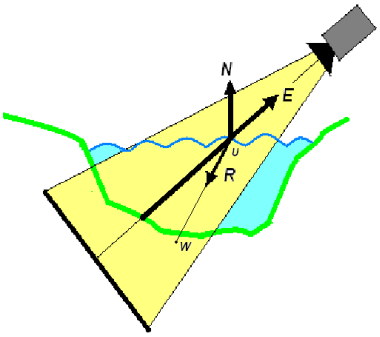

Figure 6. Refraction vector

Then we transform the coordinates of W to the texture coordinates of our refraction texture. Finally, we obtain the refracted color $C_{refract}$ from texel of the refraction texture at this coordinates. This transformation transforms coordinates of points in camera coordinate system to texture coordinates of correspondent points in the texture rendered from same camera. It just determines where the specific object with certain camera coordinates lies within the texture.

To achieve more foggy appearance of water we must mix in the global color of water. First we define the attenuation coeficient :

$$a := e^{(-d * k)} \quad (5)$$

where k is a suitable constant which determines transparency and d is length of the ray from bottom to surface which is computed from our depth texture by comparing distance of surface from camera in Z direction and distance from bottom to camera. Now the new refraction color is updated by :

$$C_{newrefract} := C_{water} + a*(C_{refract} - C_{water}) \quad (6)$$

where C_{water} is global color of water. Figure 5 shows variance of the color in the dependence of the deep.

Reflected color from texel. Let us consider reflection case. In this case we have to compute reflection vector, see Figure 7. Let R is normalized reflection vector. We determine reflection color $C_{reflect}$ by obtaining a texel form reflection texture. This texel lies at the texture coordinates obtained by transformation from coordinates of following point:

$$W = U + c*R \quad (2)$$

where U is point on the surface corespondents to current fragment in camera coordinate system and c is a suitable scaling factor.

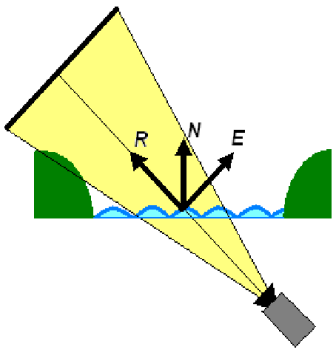

Figure 7. Reflection vector

This approach is very coarse approximation, however, visual result is plausible, see Figure 8.

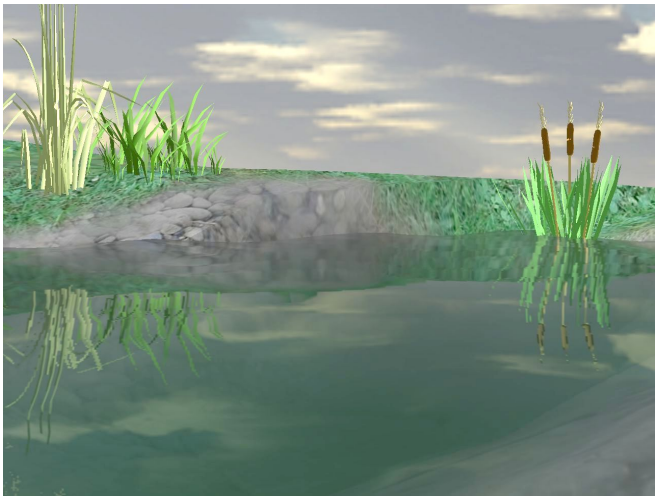

Figure 8. Reflection.

Final color of the fragment is combination of new refraction color $C_{newrefract}$ computed above and reflection color $C_{refract}$ obtained from texel in the reflection texture. To add Fresnel phenomenon we evaluate coefficient F.

$$F := (1 - (E.N))^{q} \quad (3)$$

Where q is suitable positive constant and $E.N$ is dot product of normal and eye vector. Final color is calculated by the following formula :

$$C := C_{newrefract} + F*(C_{reflect} - C_{newrefract}) \quad (4)$$

where C is final color, $C_{newrefract}$ is new refraction color and $C_{reflect}$ is reflection color.

6 Caustics mapping

Caustics are observed as brightness, increase due to many light paths hitting a bottom at the same position. We will map caustics to the bottom during rendering the refraction texture. For this reason we

create the caustics texture and then we map this texture using shadow mapping technique to the bottom. First, we set up orthogonal camera in to the light direction, see Figure 9. We use orthogonal projection because the sun rays are almost parallel. Then we render bottom from light position and store the z buffer to the depth texture. Creating the caustics texture is performed by casting of the photons to the surface. We just render the water surface from our orthogonal camera with specific shader program. In this shader we compute *dv* from our depth texture in the same way as in the refraction case. We approximate position where the photon hits the bottom by position of the point W. Position of this point is computed by (1), where R and U have same meaning as in the refraction case. Then we transform coordinates of W in camera space to texture coordinates and store this coordinates to the fragment color in RG components. Next we estimate photon contribution, based on Fresnel transmittance and the traveled distance. This contribution is obtained in same way as color was obtained, just refraction color is white and others are black. We store the result into B component of the fragment color. Then we store the frame buffer obtained by this shader to array in main memory. Finally, we create the caustics texture by searching this array on CPU and adding the stored amount of light (B component) to stored coordinates (RG components). The algorithm outline of searching is the following:

```
for x = 0 to frustum_width
  for y = 0 to frustum_height
    u_coord = array[x][y].r;
    v_coord = array[x][y].g;
    illum   = array[x][y].b;
    caustics_texture(u_coord,v_coord) += illum;
```

Thus the caustics texture consists the values of light that reach a bottom from light position. We map the caustics texture to the lake bottom.

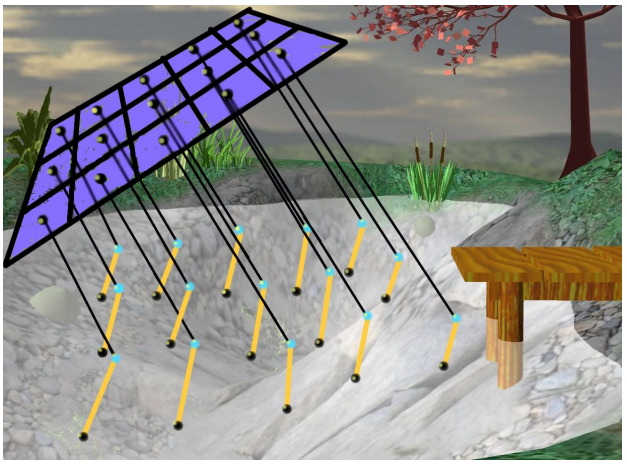

Figure 9. Mapping the photons from the orthogonal camera.

7 Implementation

As we mentioned earlier, we have decided to use OpenGL Shading Language. Let us describe the key parts of our fragment shader. Let E is normalized eye vector, N is surface normal and *dv* is length of bold line from surface to bottom in Figure 6. Selection of color of texel in our refraction texture noted by *refraTex* is given by the following code.

```
vec3 refr = refract(-E,N,0.75)*clamp(dv,0.0,2.0)/3.0;
vec4 refrvec = vec4(u+refr,1.0);
vec4 refrpos = gl_ProjectionMatrix * refrvec;
float tx2 = (refrpos.x/refrpos.w)/(2.0*FOV_ext)+0.5;
float ty2 = (refrpos.y/refrpos.w)/(2.0*FOV_ext)+0.5;
vec4 refrcol = texture2D(refraTex, vec2(tx2,ty2));
```

First, we compute the refraction vector and multiply by *dv*. Because *dv* is length of straight ray and not refracted ray, value *dv* is not suitable "as is". To minimize amount of artifacts we must decrease this value. We add this refraction vector to point u. Note that u is current point on the surface in camera coordinates. As a result the point is supposed to be on bottom, where refracted ray hit it. However, this is just a rough approximation. Next we convert coordinates of hit point to texture coordinates of our refraction texture. Note that *FOV_ext* is coefficient of extension of field of view. In our case its value is *1.5*, because we found suitable the *50%* extension. The rendering pass for reflection is simplification of above approach, and is described by the following code.

```
vec3 refl = c*reflect(-reflect(-E,N),-wn);
vec4 reflpos = gl_ProjectionMatrix * vec4(u+refl,1.0);
float tx = (reflpos.x/reflpos.w)/(2.0*FOV_ext)+0.5;
float ty = (reflpos.y/reflpos.w)/(2.0*FOV_ext)+0.5;
vec4 reflcol = texture2D(refleTex, vec2(tx,ty));
```

Here we consider the reflection texture noted by *refleTex* and determine the texture coordinates using reflection vector. Constant c is suitable scaling factor and should be 1. Note that *wn* is normal to our clipping plane. Once we have obtained color of both texels, we should mix them to create final fragment color. First, we add color of water to refracted color. We compute d, the length from u to bottom (length of tiny line in Figure 6).

```
float t = depth/refrvec.z;
float tmp = depth - u.z;
vec3 dvv = vec3(t*refrvec.x-u.x, t*refrvec.y-u.y, tmp);
float d = length(dvv);  //length from surface to bottom
```

The variable *depth* stores the Z value of estimated point of bottom in camera coordinates. This variable is computed from depth texture.

Now we update the refracted color as we explained earlier. We write this in OpenGL Shading Language notation as

```
float a = exp(-d*k);
newrefrcol = watercol + a*(refrcol-watercol);
```

Following code will mix it with the reflected color.

```
float F= pow(1.0-max(dot(N,E),0.0), q);
vec4 col = newrefrcol + F*(newreflcol-refrcol);
```

This adds Fresnel phenomena to final appearance of our surface. To add the caustics we map caustics texture to bottom while we render the refraction texture using shader which works similar to the shadow mapping. To create caustics texture we perform technique similar to photon mapping. Note that sun produces the directional light, for this reason we set up orthogonal projection. We put the orthogonal camera above the water surface, to render the bottom from light position. Now we render depth texture. This texture is used to compute *dv* (length of bold line in Figure 6) in following fragment shader.

```
vec3 refr = refract(-E,N,0.75)*dv;
vec4 refrvec = vec4(u+refr,1.0);
vec4 refrpos = gl_ProjectionMatrix * refrvec;
float tx2 = (refrpos.x/refrpos.w)/2.0+0.5;
float ty2 = (refrpos.y/refrpos.w)/2.0+0.5;
if (tx2>=0.0 && tx2<=1.0 && ty2>=0.0 && ty2<=1.0 && dv>=0.0){
   float fFresnel= pow(1.0-max(dot(N,E),0.0), 4.5);
   float blend = exp(-dv*k); // depends on water transparency
   float illum = 1.0 - fFresnel;
   illum = blend * illum;
   gl_FragColor = vec4(tx2, ty2, illum, 1.0);
 } else {
   gl_FragColor = vec4(0.0,0.0,0.0,0.0);
 }
```

First, part of shader code is similar to one where we choose the texel from the refraction texture. We are computing the texture coordinates, but they are not referred to any texture yet. At this point, we estimate where the photon casted from camera is hitting the bottom, see Figure 9. Second, part of shader code is similar to computation of a color, but here we compute just intensity noted by *illum*. It represents amount of not scattered or reflected energy in the place where the photon hits the bottom. The shader stores coordinates where photon hits the bottom into RG component of fragment color. It stores *illum* in B component, and A component is allocated for a binary information if the refracted photon is in or out of the light frustum. We store the frame buffer rendered by this shader to the array in main memory. Finally, we can create the caustics texture by searching this array and adding *illum* value into texel in caustics texture on the corresponding texture coordinates stored in the RG components.

Next table shows values of constants we found suitable in our example.

c	1.0f
k	0.5f
watercol	vec4(0.0, 0.2, 0.1, 1.0)
q	4.5f

Table 1. Constants.

8 Results

We tested the proposed algorithm on multiple machines. Our lake scene (see Figure 10) consist of 131437 vertices and 85409 faces. The water surface was represented by the mesh with 2401 vertices and 4608 faces. The animation of the surface was performed using sine function. As the caustics and the surface mesh are performed on CPU, the framerates are not very high, see Table 2.

Referenced machine	caustics	
	on	off
Intel P4 3.0GHz, 1GB ram, nvidia 6600 (256MB)	10fps	19fps
Intel P4 3.0Ghz, 1GB ram, nvidia 7600GT (256MB)	10fps	18fps
AMD Sempron 2600+ (1,6Ghz), 1GB ram, nvidia 7950GT (512MB)	6fps	10fps
Intel Core2Duo 2.6Ghz, 2GB ram, nvidia 7950GT (512MB)	17fps	25fps
AMD Athlon 64 dual core 4400+ geforce 800GT (512MB)	9fps	14.12fps

Table 2. Framerates.

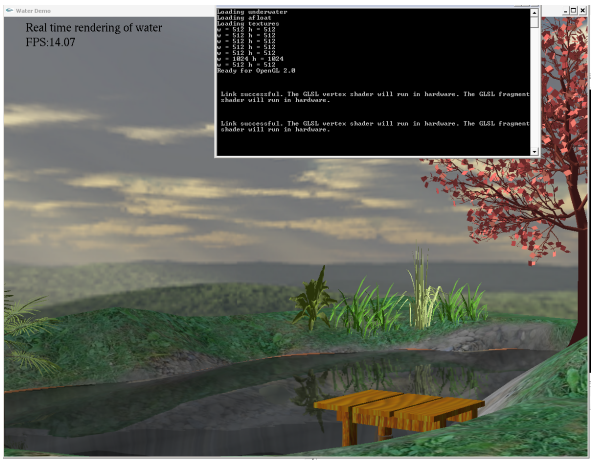

Figure 10. Demonstration application.

9 Conclusion

This work insists on the simplicity of implementation. Although the visual aspect of generated images are plausible, there are still some performance issues. The results are obtained from demonstration application implementing the above procedures in OpenGL, see Figure 10. For the reflection and the refraction textures the resolution is set to 512x512. Extending the field of view we may loose the resolution, which starts to be notable in case of 512x512. For caustics texture the resolution of 256x256 with enabled filtering was used. Some artifacts may appear in the water near shore that can be fixed by putting the clipping plane little bit above and adjusting the position of the bottom before the rendering of refraction texture. Rendering of the reflection texture is done by moving the clipping plane a little bit under the ware surface.

Currently we are implementing the other ways to extend this work by adding underwater camera, see Figure 11. It could be implemented by inversing the reflection and the refraction approach. Another visual improvement could be achieved by adding HDR effects.

Acknowledgments

Author wishes to thank his advisor Roman Durikovic for the help during the course of this work. Frame rate measurements were done by Michal Červeňanský, Filip Zigo on their hardware. This research was supported by a Marie Curie International Reintegration Grant within the 6*th* European Community Framework Programme EU-FP6-MC-040681-APCOCOS.

References

[1] Belyaev V., Real-time simulation of water surface. GraphiCon-2003, Conference Proceedinks, pp. 131-138.

[2] Sousa T., Generic Refraction Simulation. GPU Gems 2, NVIDIA Corporation 2005, pp. 295-305. ISBN-10:0321335597. ISBN-13:978-0321335593.

[3] Galin, E., Chiba, N., Realistic Water Volume in Real-Time. Eurographics Workshop on Natural Phenomena (2006), pp. 1-8.

[4] Tessendorf, J., Simulating Ocean Water, SIGGRAPH 2002 Course Notes #9 (Simulating Nature: Realistic and Interactive Techniques), ACM Press.

[5] Musawir, S., Konttinen J., Pattaniak, S., Caustics Mapping: An Image-space Technique for Real-time Caustics IEEE Transactions On Visualization And Computer graphics.

[6] Premože, S., Ashikhmin M., Rendering Natural Waters. Computer Graphics forum. Volume 20, number 4 (2001), pp189-199.

[7] Blinn, J., Texture and Reflection In Computer Generated Imges, CACM, 19(10), October 1976, pp 542-547.

Figure 12. Refraction.

Figure 11. First underwater camera experiments.

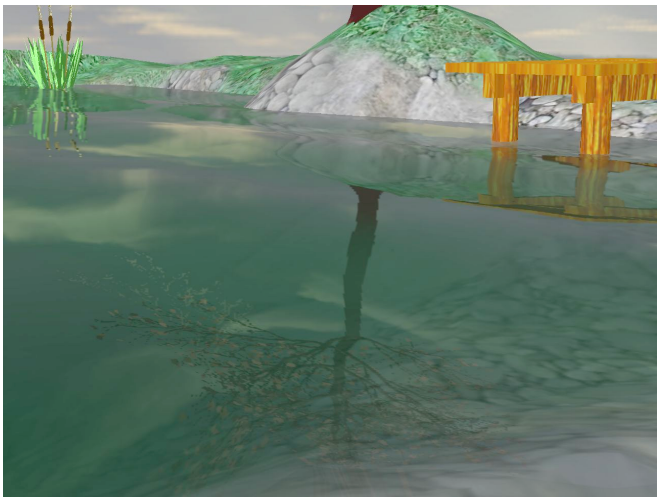

Figure 13. Reflection of the tree.