Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava

Algorithm Design for Real-Time Rendering of the Natural Waters

2007/2008

Andrej Mihálik

Affidavid

I hereby attest that I am a sole author of this thesis and it has been written based on my research and references listed herein.

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

Acknowledgements

The author wishes to thank his advisor Roman Ďurikovič for the help during the course of this work. Frame rate measurements were done by Michal Červeňanský and Juraj Starinský on their hardware. Special thanks to Juraj Starinský for his help with the implementaton. This research was supported by a Marie Curie International Reintegration Grant within the 6th European Community Framework Programme EU-FP6-MC-040681-APCOCOS. This work was presented in CESCG 2008 and ŠVK FMFI 2008.

Contents

1. Overview. 9

2. Introductions. 11

3. Related work. 12

4. Software specification. 17
4.1 Development environment. 17
4.2 Data structures. 18
4.3 Purpose of Program. 19
4.4 Operating description. 19

5. Refraction and Reflection texture. 20
5.1 Refraction texture. 20
5.2 Reflection texture. 22

6. Colouring. 24

7. Pixel Shading. 25
7.1 Intersection algorithm. 25
7.2 Refracted colour from texel. 29
7.3 Reflected colour from texel. 31
7.4 Fragment colour. 33
7.5 Underwater camera. 33

8. Caustics mapping. 37
8.1 CPU approach. 37
8.1 GPU approach. 38

9 Implementation. 40
9.1 Refraction and reflection. 40
9.2 Caustics. 43
9.3 Glow. 44

9.4 Constans and parameters. 45

10. Results. **46**

11. Conclusion. **48**

Glossary. **49**

References. **51**

List of Figures

1. Clipping pane. 20
2. Refraction camera. 21
3. Back face culling. 22
4. Reflection camera. 22
5. Missing texture parts. 23
6. Colour of the water. 24
7. First approximation. 25
8. Second approximation. 26
9. Third approximation. 27
10. Vector R do not point between A_3 and A_4. 27
11. Fourth approximation. 28
12. Z coordinate of $\boldsymbol{A_{j+1}}$ is closer to camera then Z coordinate of $\boldsymbol{W_j}$. 28
13. Refraction vector. 30
14. Reflection vector. 32
15. Reflection. 33
16. Underwater camera. 35
17. Mapping the photons from the orthogonal camera. 38
18. Demonstration application. 48
19. Glow effect. 53
20. Reflection of the tree. 53
21. View from underwater camera. 54
22. Refraction. 54

List of Tables

1. Constants. 45
2. Framerates. 46
3. Texture resolutions. 47

Chapter 1

Overview

Chapter 2 of this thesis gives an outline of the problem and discusses aims of this work.

Chapter 3 briefly discusses state of the art and common techniques used in this area.

Chapter 4 describes application programming interface and specification of our implementation.

Chapter 5 is devoted to obtaining textures needed to environmental mapping.

Chapter 6 discusses an explanation of how is colour of the water determined.

Chapter 7 will elaborate algorithms for environmental mapping and water surface shading.

Chapter 8 is devoted to creating caustic texture on CPU and GPU.

Chapter 9 provides implementation of proposed algorithms in the OpenGL Shading Language.

Chapter 10 discusses achieved results with proposed implementation.

Chapter 11 summarizes all our efforts and concludes this thesis and discusses possible future directions.

Chapter 2

Introductions

Ray tracing is a fundamental problem that makes the photo-realistic 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 colour of our surface is performed in a shader program. We will use the OpenGL Shading Language to write the shaders. A fluid represented as a triangular mesh serves as an input to our shader that can be generated by the known fluid dynamic methods. Although fluid movement simulation is out of scope of this work, there is plenty of literature devoted to this problem. 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 following 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 to deal with constrains.

Chapter 3

Related work

Several works in the past were focused on realistic rendering of water surfaces. Previous approaches mostly based on 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.

Optical effects like reflection and refraction are mostly done by environment mapping techniques.

First work devoted to environment mapping was [Bli76] published in seventies of twentieth century.

Newer approach based on GPU is proposed in paper [YYM05].

Work [Bel03] is devoted to creation of the triangular water surface and implementation of fundamental optics phenomena on featureless graphics hardware. Creation of water surface in [Bel03] is based on results of oceanography research. There are presented two models for imitating optics effects called "reflection and refraction" and "sky reflection".

The idea of the first model is that the whole scene (except water) is rendered into main camera forming "refraction" texture. Camera in the mirror position under the water plane, produces "reflection" texture. Then using projection mapping, the textures are mapped onto the water surface and mixed, using Fresnel coefficient. This technique works fine for a plane water. For oscillating water is need to take into account the normal change to get the correct result. It can be done in such way: in every grid point of surface is calculated the reflection ray and intersect it in with the plane raised above the water plane; intersection point is then projected into camera to form texture coordinate for the grid point. Texture coordinates for "refraction" are calculated in [Bel03] in

the same way as "reflection" except for three things: plane is lowered, refraction vector is used and intersection point is projected to main camera.

Second model can be used in case of invisible or nearly invisible bottom, for the open water (or the water that does not reflect anything except sky). It is based on per-pixel cubic environment mapping. So this approach need bump and cubic environment texture. Environment texture should contain sky (or other things to be reflected). Bump is generated from $\boldsymbol{N}$ x $\boldsymbol{N}$ normals matrix.

Proposed rendering technique is mostly depending on the way of constructing mesh representing water surface. In work [Bel03] four requirements appeared during the construction of surface. By the first requirement, final triangular representation of water surface must allow fast culling of invisible (out of the camera frustum) triangles. Of course, no calculations should be performed for vertices of the invisible triangles. The second requirement appeared because of artefacts in the final image when using translucency. The problem is in triangle rendering order (it is very important when dealing with translucent objects and z-buffer). When the triangle order was back-to-front, the distant triangles were seen through near ones, thus creating those artefacts. So, in a resulting mesh all triangles should be sorted front-to-back. To produce water surface of acceptable quality the grid step should be small enough. So the third requirement says mesh building algorithm must provide enough details near the camera while overall triangles number should be not very big. The fourth requirement is simple: water surface mesh should be easily stripifiable. That is for improving rendering speed.

Although proposed work covers fundamentals of water visualization, it does not involve features like light transporting model and caustics. This work also do not involve current hardware features like the fragment shader which could performs per-pixel lighting and shading of the water surface.

Works devoted to refraction usually consider one transparent object surrounded by an infinitely far environment represented by an environment map. This assumption can not be used in the case of the refraction of a ground surface through a water volume. Computing the refraction of nearby objects can be done with memory costly light-field precomputations [HLCS99].

The essential problem is also the colour calculation. As explained by [PA01], the physical study of light-water interactions is a full-fledged research field with a vast literature. For example Belyaev in [Bel04] proposed conversion from the energy-spectral representation to RGB colour space.

Tessendorf in [Tes02] is discuss radiosity of the ocean environment. The ocean environment consists of only four components: The water surface, the air, the sun, and the water volume below the surface.

The light seen by a camera is dependent on the flow of light energy from the source (i.e. the sun and sky) to the surface and into the camera. In addition to specular reflection of direct sunlight and skylight from the surface, some fraction of the incident light is transmitted through the surface. Ultimately, a fraction of the transmitted light is scattered by the water volume back up through the interface and into the air. Some of the light that is reflected or refracted at the surface may strike the surface a second time, producing more reflection and refraction events. Under some viewing conditions, multiple reflections and refractions can have a noticeable impact on images. Tessendorf ignore more than one reflection or refraction from the surface at a time. This not only makes his algorithms and computation easier and faster, but also is reasonably accurate in most viewing conditions and produces visually realistic imagery.

General part of [Tes02] is devoted to the mathematical description of the ocean waves. This is out of scope of our research.

Authors in [GC06] also present a real-time technique to render

realistic water volumes. But water volumes are represented as the space enclosed between a ground heightfield and an animable water surface heightfield. Method in [GC06] is a simplified raytracing approach which handles reflections and refractions and allows to render complex effects such as light absorption, refracted shadows and refracted caustics. They represent a water volume with two heightfields, the ground surface and the water surface, both encoded as 2D textures. In they approach is fragment shader used to perform ray-tracing. During this, they only consider one level of recursivity: for each viewing ray, they found the intersection with the water surface, then the intersection of the refracted ray with the ground. This single level recursion assumption is correct if the ground is non-reflective, and if no light-ray crosses the water surface more than once.

Caustics are caused by the convergence of light due to reflections and refractions. Computing caustics is hard to carry out in a forward raytracing approach because it requires to count the amount of incoming light at each rendered point. That is why most existing caustics algorithms are based on backward raytracing: rays are cast from the light source instead of the view point. In [GC06] is used a two-pass photon-mapping-like algorithm, involving GPU/CPU transfers of textures. The first pass renders the water surface from the light source into a photon texture. The texels of this texture record the coordinates of where the corresponding light rays hits the ground. Since the ground is a heightfield, they needed only recording the (x, y) coordinates, which leaves the third channel of the texture available to store the photon contribution, based on the Fresnel transmittance, the traveled distance, the incident angle and a ray sampling weight. Then the photon texture is transfered to CPU where it was processed to construct an illumination texture. The illumination texture is then transferred to the GPU where it is used for lighting in the final render pass.

Proposed work is devoted to heightfield rendering and has the problem of geometry-based objects that penetrate the water. This issue could be solved by dynamic projection of the geometry onto the heightfields.

Caustics mapping is also discussed in [MKP05], however this algorithm requires texture access in the vertex shader for intersection estimation in the caustic map generation step.

Chapter 4

Software specification

The goal of this work is to create algorithm providing real time visualization with photorealistic results. Real time rendering requires solution based on the GPU. One of possible way is to use OpenGL application programming interface.

4.1 Development environment

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. At its most basic level, OpenGL is a specification, meaning it is simply a document that describes a set of functions and the precise behaviors that they must perform. From this specification, hardware vendors create implementations, libraries of functions created to match the functions stated in the OpenGL specification, making use of hardware acceleration where possible.

With the recent advancements in graphics cards, new features have been added to allow for increased flexibility in the rendering pipeline at the vertex and fragment level. Programmability at this level is achieved with the use of fragment and vertex shaders.

To program this fragment and vertex shaders in OpenGL is most common way to use the GLSL, because it is a standard high level shading language in OpenGL. GLSL shaders themselves are simply a set of strings that are passed to the hardware vendor’s driver for compilation from within an application using the

OpenGL API's entry points. Shaders can be created on the fly from within an application or read in as text files, but must be sent to the driver in the form of a string.

TyphoonLabs ShaderDesigner is an integrated development environment which helps to developers creating vertex and fragment shader programs in GLSL.

Due to goal of creating algorithms providing just visualization, it is no need to build sophistic application. For testing results is enough a simple window which displays rendered scene from the specific position and direction of the camera controlled by arrow keys and mouse. For this purpose is optimal solution based on GLUT.

GLUT is designed for constructing small to medium sized OpenGL programs. While GLUT is well-suited to learning OpenGL and developing simple OpenGL applications, GLUT is not a full-featured toolkit so large applications requiring sophisticated user interfaces are better off using native window system toolkits, but it is simple, easy, and small.

To work with features like Frame Buffer Object, fragment and vertex shaders or another extensions in OpenGL is use of GLEW library suitable.

While OpenGL Shading Language is well for programming GPU, C++ language is suitable choice for creating fast programs. C++ uses a relatively straightforward compiler, provide low-level access to memory, provide language constructs that map efficiently to machine instructions, and require minimal run-time support.

4.2 Data structures

Data structures using during visualization are often textures. This textures are useful for example to perform environment mapping. Another structures are vertex arrays which store geometry data. In OpenGL Shading Language are typical data structures

vectors. Vectors are used for storing coordinates or colour information.

4.3 Purpose of Program

Purpose of programming demonstration application is to prove explored algorithms. In order to evaluate the results is necessary display the scene with animated water surface and the frame rates per second.

4.4 Operating description

Program loads environment scene geometry and textures. The scene consists of the lake without water and its geometry is stored in Wavefront OBJ files. In the program after loading the scene, animated water surface (triangular mesh generated by sine function) is created and added into the scene. Main part of the program is devoted to creation optical effects on this surface.

Chapter 5

Refraction and Reflection texture

The effect of reflection and refraction in our approach similar to [Bel03] and [Sou05] 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.

5.1 Refraction texture

To obtain refraction texture we put clipping plane little bit above the water plane (approximation of the water surface) to store little bit more geometry.

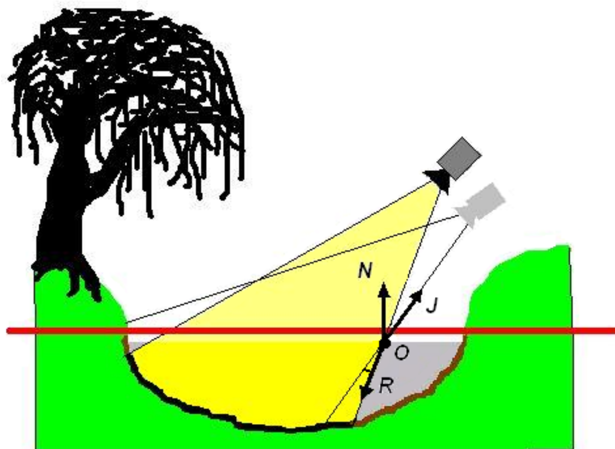

Figure 1. Clipping pane.

Then we move main camera to more suitable position (we assume here afloat camera in this example, we will discuss underwater camera later). We compute vector $\boldsymbol{J}$ (vector from bottom

center point of the screen to the camera) with directiom same as ray in bottom plane of view frustum. We find intersection point $\boldsymbol{O}$ of this ray with the water plane. Intersection is given by

$$\boldsymbol{C_1} - \frac{\boldsymbol{N.(C_1-V)}}{\boldsymbol{N.J}}\,\boldsymbol{J} \qquad (1)$$

where $\boldsymbol{N}$ is normal of clipping plane, $\boldsymbol{C_1}$ is position of main camera and $\boldsymbol{V}$ is arbitrary point in the water plane. See Figure 1. Then we compute refraction vector $\boldsymbol{R}$ by formula

$$\boldsymbol{k} = 1 - \boldsymbol{n_{12}}^2(1 - (\boldsymbol{N.I})^2)$$
$$\boldsymbol{R} = \boldsymbol{n_{12}I} - \boldsymbol{N}(\,\boldsymbol{n_{12}}\,\boldsymbol{N.I} + \sqrt{\boldsymbol{k}}\,) \qquad (2)$$

where $\boldsymbol{n_{12}} = \boldsymbol{n_1}/\boldsymbol{n_2}$; $\boldsymbol{n_1}$, $\boldsymbol{n_2}$ are indices of refraction and $\boldsymbol{I}$ is normalized -$\boldsymbol{J}$. We rotate camera around intersection point $\boldsymbol{O}$ about same angle as between $\boldsymbol{I}$ and $\boldsymbol{R}$. See Figure 1.

Then we render from this refraction camera just the underwater part of the environment (i.e. bottom part of the lake). See Figure 2. Finally, we copy frame buffer into our refraction texture resulting in refraction texture consisting of the image of the lake bottom. For future computation (e.g. deep) we may need depth texture of the underwater part of environment. Therefore, it is time to store the depth buffer to a texture at this step.

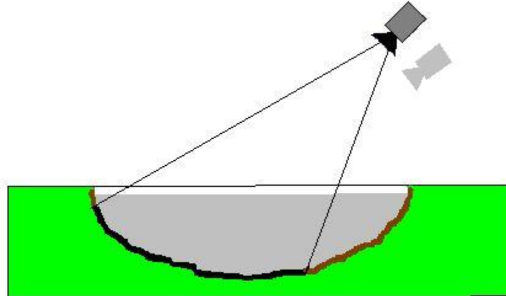

Figure 2. Refraction camera.

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 3.

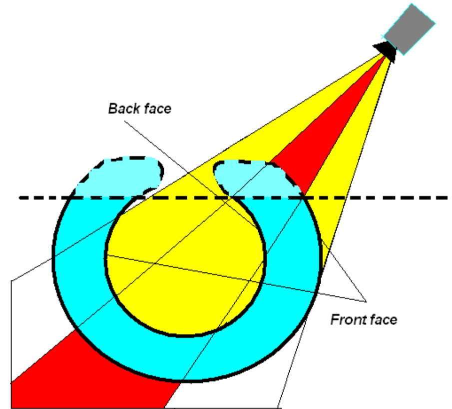

Figure 3. Back face culling.

Note that appearance of back faces is not suitable. To avoid this artifact, we should enable back face culling before clipping.

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

5.2 Reflection texture

To obtain this texture we must put clipping plane little bit down and the main camera upside-down into its mirror position (just scaling with Y=-1 if water plane is in the axes origin and vertically to Y). We render the upper part of the split environment, as shown in Figure 4. After the rendering it, we store the frame buffer to the reflection texture.

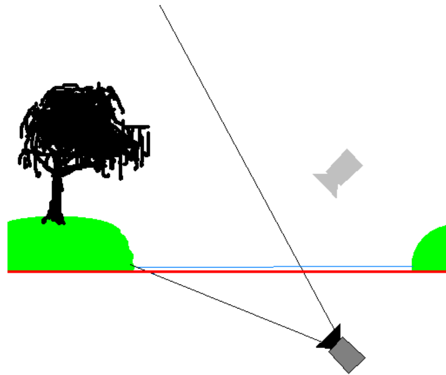

Figure 4. Reflection camera.

This approach has the problem of missed texels because rays changed their direction on the water surface due to refraction and reflection. This causes us to see on the surface a larger area than is actually stored in our refraction and reflection textures. See Figure 5. To restore the missing texture parts we extend the field of view before rendering of the reflection and the refraction texture. In our case extending the field of view about 50% with camera near the water surface was appropriate. When camera is farther from the water surface then extension could be 20%.

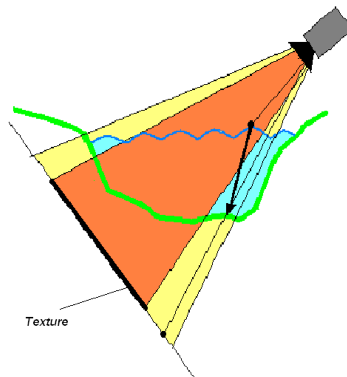

Figure 5. Missing texture parts.

Chapter 6

Colouring

Let us suppose a lake under the blue sky during daylight. What attributes determine the ambient colour of the water surface? It is the illumination and the water mixture. We simplify the outdoor scene to include sunlight and sky illumination. The water consists of particles such as green algae or other impurities giving the water its inherent colouring, such as the commonly encountered blue and greenish tones. Once we have obtained the global colour 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 6.

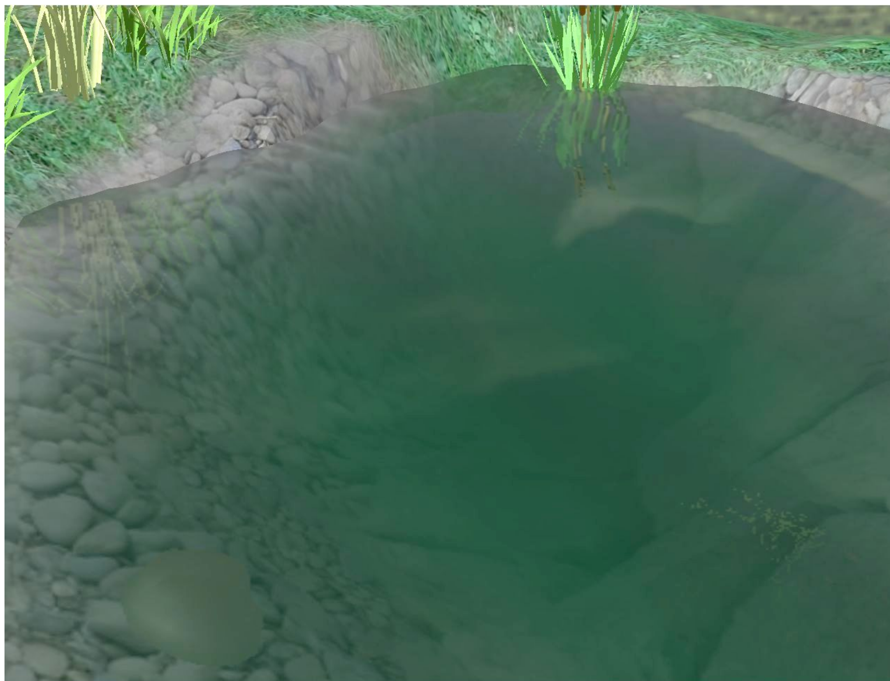

Figure 6. Colour of the water.

Chapter 7

Pixel Shading

In this section we discuss how to obtain the colour of certain pixels on water surfaces. We assume the water surface is represented by the mesh as an input. Let's summarize that we have available surface refraction texture, depth texture of the underwater part, surface reflection texture and global colour of the water at this moment. First, we render the whole scene without water using the fixed pipeline.

Then we render the surface and compute the colour of pixels in fragment shader. Note that we have all the necessary stuff in the shader now.

7.1 Intersection algorithm

Assume that we are in camera coordinates system. The afloat camera is looking at certain point $\boldsymbol{U}$ on the water surface. See Figure 7 . We have depth texture of the bottom rendered from this camera and vector $\boldsymbol{R}$ pointing to the bottom from $\boldsymbol{U}$. Our task is to estimate intersection of line from $\boldsymbol{U}$ in direction of $\boldsymbol{R}$ and the bottom. We denote this line as $\boldsymbol{p}$. Our first approximation is point $\boldsymbol{W_1}$.

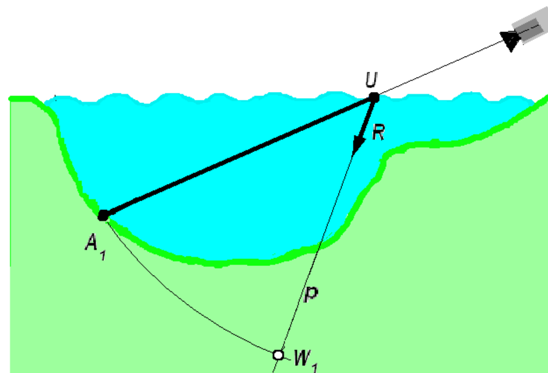

Figure 7. First approximation.

It is point on the line $\boldsymbol{p}$, where distance:$|\boldsymbol{A_1} - \boldsymbol{U}| = |\boldsymbol{W_1} - \boldsymbol{U}|$. Distance $|\boldsymbol{A_1} - \boldsymbol{U}|$ is obtained by comparing Z coordinate of the $\boldsymbol{A_1}$ and $\boldsymbol{U}$. Z coordinate of $\boldsymbol{A_1}$ is computed from depth texture. Next step is to obtain point $\boldsymbol{A_2}$ by transforming the coordinates of $\boldsymbol{W_1}$ to the texture space and obtaining Z of the bottom from depth texture. See Figure 8. We denote $\boldsymbol{T}$ as transformation from the camera space to the texture space. If $\boldsymbol{p}$ has intersection with line from $\boldsymbol{A_1}$ to $\boldsymbol{A_2}$ then we denote $\boldsymbol{W_2}$ as this intersection. See Figure 8. We get this intersection by formula:

$$\boldsymbol{W_2} = \boldsymbol{U} + \frac{((\boldsymbol{A_1} - \boldsymbol{U}) \times \boldsymbol{Q}).(\boldsymbol{R} \times \boldsymbol{Q})}{|\boldsymbol{R} \times \boldsymbol{Q}|^2} \boldsymbol{R} \qquad (3)$$

where $\boldsymbol{Q} = \boldsymbol{A_2} - \boldsymbol{A_1}$.

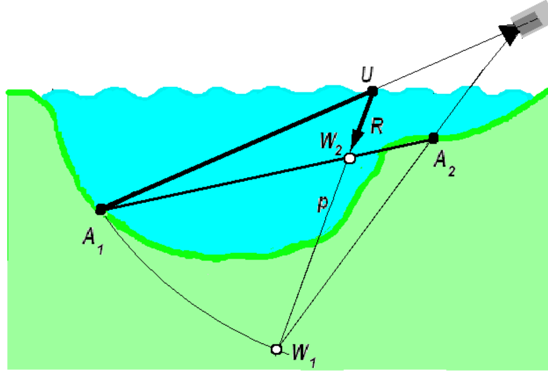

Figure 8. Second approximation.

Point $\boldsymbol{W_2}$ is between $\boldsymbol{A_1}$ and $\boldsymbol{A_2}$ iff $|\boldsymbol{W_2} - \boldsymbol{A_1}| + |\boldsymbol{A_2} - \boldsymbol{W_2}| = |\boldsymbol{A_2} - \boldsymbol{A_1}|$. Now we find intersection $\boldsymbol{A_3}$ of line from camera through $\boldsymbol{W_2}$ with the bottom. See Figure 9. This is computed by obtaining Z coordinate of this intersection $\boldsymbol{A_3}$. Z coordinate is obtained by transformation $\boldsymbol{T}$ of $\boldsymbol{W_2}$ coordinates to the texture coordinates and looking to depth texture at this coordinates. If $\boldsymbol{p}$ intersects line between $\boldsymbol{A_2}$ and $\boldsymbol{A_3}$ then we compute intersection point $\boldsymbol{W_3}$. See Figure 9. Now we compute intersection $\boldsymbol{A_4}$ of line from camera through $\boldsymbol{W_3}$ with the bottom using our depth

texture. Note that in the Figure 10 line $\boldsymbol{p}$ does not intersect the line segment between $\boldsymbol{A_3}$ and $\boldsymbol{A_4}$. In this case we have two configurations. In first configuration Z coordinate of $\boldsymbol{W_j}$ is closer to camera then Z coordinate of $\boldsymbol{A_{j+1}}$ (Z coordinate of camera is 0).

In Figure 10 is example for $\boldsymbol{W_3}$ and $\boldsymbol{A_4}$. In this case we compute $\boldsymbol{W_4}$ on line from $\boldsymbol{W_3}$ in direction $\boldsymbol{R}$ where length $|\boldsymbol{A_4} - \boldsymbol{W_3}| = |\boldsymbol{W_4} - \boldsymbol{W_3}|$. See Figure 11. In second configuration Z coordinate of $\boldsymbol{W_j}$ is farther from camera then Z coordinate of $\boldsymbol{A_{j+1}}$. See Figure 12. In this case we find $\boldsymbol{W_{j+1}}$ on line from point $\boldsymbol{W_j}$ up through point $\boldsymbol{U}$ where also $|\boldsymbol{A_{j+1}} - \boldsymbol{W_j}| = |\boldsymbol{W_{j+1}} - \boldsymbol{W_j}|$. See Figure 12.

We perform certain amount of steps of this algorithm. We get point $\boldsymbol{W_i}$ from step number $\boldsymbol{i}$ where for lengths stand: $|\boldsymbol{A_i} - \boldsymbol{W_i}| \leq |\boldsymbol{A_j} - \boldsymbol{W_j}|$ where $\boldsymbol{j}$ is number of arbitrary step. Then point $\boldsymbol{W_i}$ is supposed to be rough approximation of intersection of ray from $\boldsymbol{U}$ in direction $\boldsymbol{R}$ with the bottom.

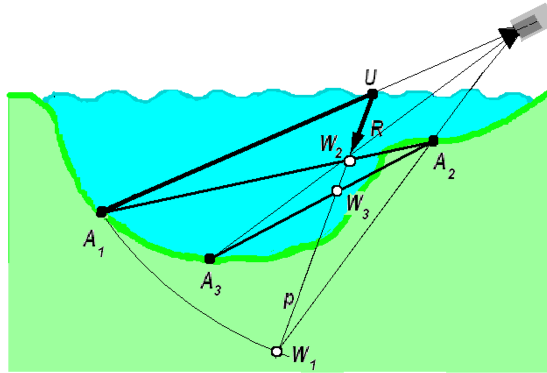

Figure 9. Third approximation.

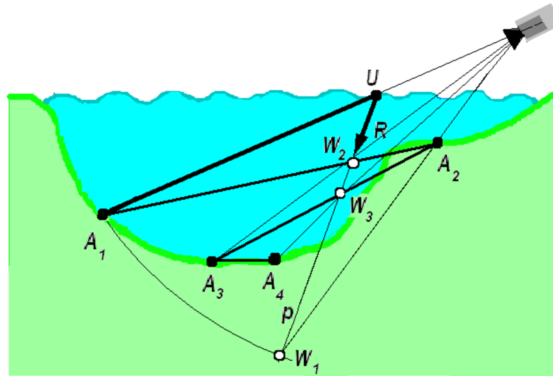

Figure 10. Vector $\boldsymbol{R}$ do not point between $\boldsymbol{A_3}$ and $\boldsymbol{A_4}$.

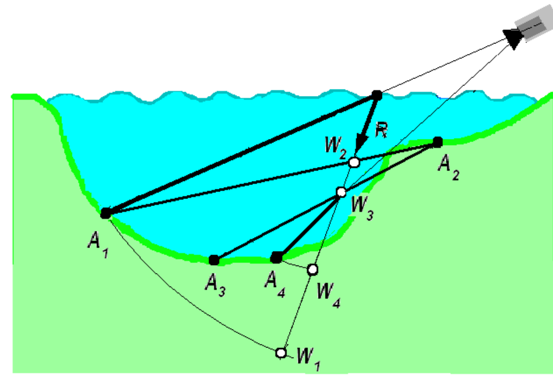

Figure 11. Fourth approximation.

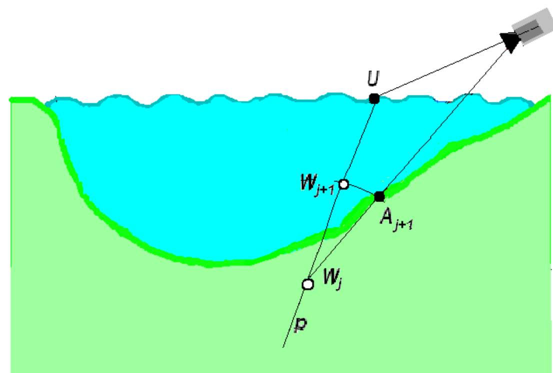

Figure 12. Z coordinate of $\boldsymbol{A_{j+1}}$ is closer to camera then Z coordinate of $\boldsymbol{W_j}$.

Next follows code of this algorithm. First function gives intersection of two lines.

```
vec3 RayIntersec(vec3 P1, vec3 P2, vec3 v1, vec3 v2)
{
     vec3 c = P2-P1;
     float m = dot(cross(c,v2),cross(v1,v2));
     float n = pow(length(cross(v1,v2)),2.0);
     return P1 + (m/n)*v1;
}

W_0 = A_0 = U;
dist = ∞;
for(int i = 0; i < n; i++)
{
      uv = CameraSpaceToTextureSpace(W_i);
      Z = DepthToZ(texture2D(depthTex, uv));
      A_{i+1} = vec3(W_i.x * Z/W_i.z, W_i.y * Z/W_i.z, Z);
      if(|A_{i+1}-W_i| < dist){
         W = W_i;
         A = A_{i+1};
         dist = |A_{i+1}-W_i|;
      }

      W_{i+1} = RayIntersec(W_i, U, A_{i+1}-A_i, R);

      zn = 1;
      if(|A_{i+1}-A_i| < |W_{i+1}-A_i|+|A_{i+1}-W_{i+1}| OR i = 0){
        if(|W_i.z| > |A_{i+1}.z|) zn = -1;
        W_{i+1} = W_i + zn*|A_{i+1}-W_i|*R;
      }
}
```

After n iterations we have as result $\boldsymbol{W}$ one point from $\boldsymbol{U} = \boldsymbol{W}_{0}, \boldsymbol{W}_{1}, \ldots$ $\ldots, \boldsymbol{W}_{n-1}$.

7.2 Refracted colour from texel

When we are treating a certain pixel, we have got coordinates of its corresponding point on the surface in camera coordinate system. We have also got surface normal $\boldsymbol{N}$ at this point. We can

easily compute normalized eye vector $\boldsymbol{E}$, because the camera position is the origin of camera coordinate system. Once we have the eye and the normal vector, we can compute the normalized refraction vector $\boldsymbol{R}$ using air-water refraction index $\boldsymbol{n_{12}}$. Refraction vector formula is (1) where $\boldsymbol{I}$ is -$\boldsymbol{E}$. The direction of this refraction vector is essential for determining which part of the underwater part of the scene is mapped to current fragment of the surface. Note that the functions computing reflection and refraction vectors 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 $\boldsymbol{R}$ hits the bottom. See Figure 13. We propose to estimate the hitting point by performing mentioned intersection algorithm.

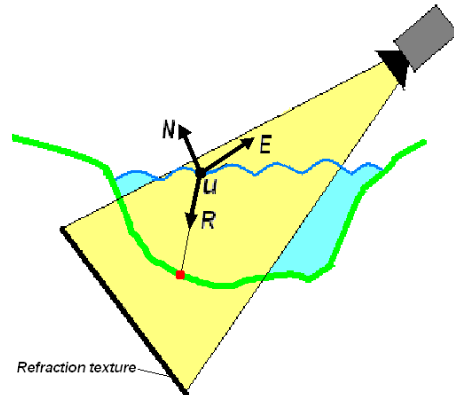

Figure 13. Refraction vector.

If we have obtained some approximation $\boldsymbol{W}$ by this algorithm, we transform the coordinates of $\boldsymbol{W}$ to the texture coordinates of our stored refraction texture by transformation $\boldsymbol{T}$. Finally we obtain the refracted colour $\boldsymbol{C_{refract}}$ from the texel of the refraction texture at this coordinate. This transformation $\boldsymbol{T}$ transforms coordinates of points in camera coordinate system to texture coordinates of corresponding points in the texture rendered from the

same camera. $\boldsymbol{T}$ just determines where the specific object with certain camera coordinates lies within the texture. Since refraction and main camera are not the same, we need trnsformation $\boldsymbol{M}$ from main camera to refraction camera. When we are looking up for texture coordinates of certain point in main camera coordinates, we need first use transformation $\boldsymbol{M}$ to get coordinates in refraction camera space and then apply transformation $\boldsymbol{T}$ to get texture coordinates.

To achieve more foggy appearance of water we must mix in the global colour of water in to refracted colour. First we define the attenuation coefficient:

$$\boldsymbol{a} := \boldsymbol{e}^{(-\,\boldsymbol{kd})} \qquad (4)$$

where $\boldsymbol{k}$ is a suitable constant which determines transparency and $\boldsymbol{d}$ is the length $|\boldsymbol{A} - \boldsymbol{U}|$ of the ray from surface to bottom where $\boldsymbol{A}$ is computed in intersection algorithm. Now the new refraction colour is updated by:

$$\boldsymbol{C}_{\boldsymbol{newrefract}} := \boldsymbol{C}_{\boldsymbol{water}} + \boldsymbol{a}(\,\boldsymbol{C}_{\boldsymbol{refract}} - \boldsymbol{C}_{\boldsymbol{water}}\,) \qquad (5)$$

where $\boldsymbol{C}_{\boldsymbol{water}}$ is global colour of water. Figure 6 shows variance of the colour in the dependence of the deep.

7.3 Reflected colour from texel

Let us consider reflection. In this case we have to compute the reflection vector. See Figure 14. Letter $\boldsymbol{R}$ denotes normalized reflection vector. We determine reflection colour $\boldsymbol{C}_{\boldsymbol{reflect}}$ by obtaining a texel form reflection texture. This texel lies at the texture coordinates obtained by transformation $\boldsymbol{T}$ from coordinates (in reflection camera space) of the following point (see Figure 14):

$$\boldsymbol{W} = \boldsymbol{U} + \boldsymbol{cR} \qquad (6)$$

where $\boldsymbol{U}$ is a point on the surface that corresponds to the current fragment in the camera coordinate system and $\boldsymbol{c}$ is a suitable scaling factor. Factor $\boldsymbol{c}$ depends on scale range of the model. Since we are using 1 meter, factor $\boldsymbol{c}$ equal 1 is suitable.

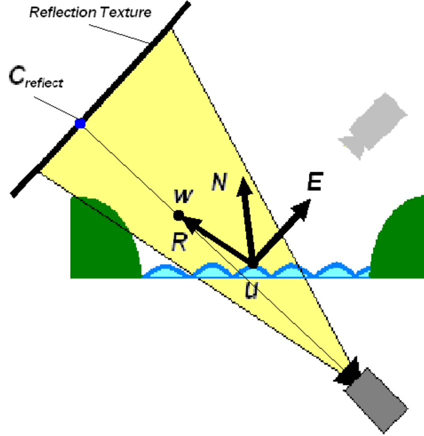

Figure 14. Reflection vector.

This approach is a very coarse approximation. However, a visual result is plausible. See Figure 15.

Final colour of the fragment is a combination of the new refraction colour $\boldsymbol{C}_{newrefract}$ computed above and the reflection colour $\boldsymbol{C}_{reflect}$ obtained from texel in the reflection texture. To add Fresnel phenomenon we make the simplifying assumptions of the Fresnel equations, because they are relatively complex. An approximation for the ratio between reflected light and refracted light according to [Ros06] is

$$\boldsymbol{F} := \boldsymbol{f} + (1 - \boldsymbol{f})\ (1 - (\ \boldsymbol{E}.\boldsymbol{N}\ ))^{\ \boldsymbol{q}} \tag{7}$$

In this equation, $\boldsymbol{q}$ is a suitable positive constant, $\boldsymbol{E}.\boldsymbol{N}$ is a dot product of normal and eye vector and $\boldsymbol{f}$ is the reflectance of the material

$$\boldsymbol{f} = \left(\frac{1-\frac{n_1}{n_2}}{1+\frac{n_1}{n_2}}\right)^2 \qquad (8)$$

where $\boldsymbol{n_1}$ and $\boldsymbol{n_2}$ are the indices of refraction for air and water.

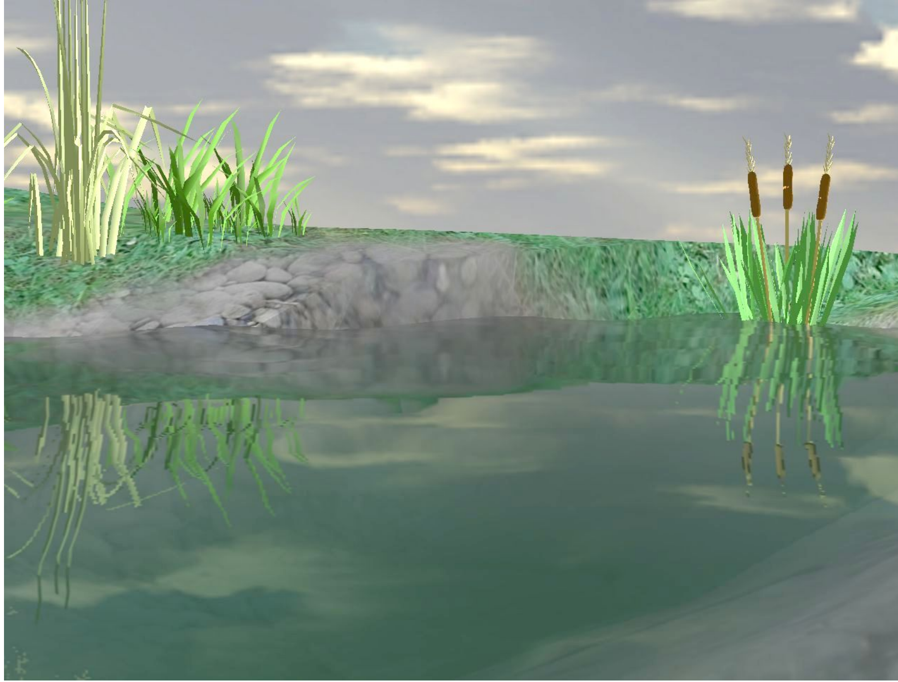

Figure 15. Reflection.

7.4 Fragment colour

Final colour is calculated by the following formula:

$$\boldsymbol{C := C_{newrefract} + F(C_{reflect} - C_{newrefract})} \qquad (9)$$

where $\boldsymbol{C}$ is the final colour, $\boldsymbol{C_{newrefract}}$ is new refraction colour and $\boldsymbol{C_{reflect}}$ is the reflection colour. Some artifacts may appear in the water near the shore that can be fixed by moving camera a little bit and adjusting the position of the bottom before the rendering of refraction texture.

7.5 Underwater camera

Proposed approach works fine for camera with afloat position. Now we will discuss how to deal with camera underwater. When we are underwater then we are surrounded just by the underwater objects (including bottom) and the water surface. See Figure 16. We are mixing in fragment shader global water colour into every object in the scene except water surface. We render the scene with underwater objects without the water surface (we will render water surface later with another shader). Then we blend colour of objects in the water volume by:

$$\boldsymbol{C} := \boldsymbol{C}_{\boldsymbol{water}} + \boldsymbol{a}(\ \boldsymbol{C}_{\boldsymbol{obj}} - \boldsymbol{C}_{\boldsymbol{water}}\ ) \tag{10}$$

where $\boldsymbol{C}$ is final colour, $\boldsymbol{C}_{\boldsymbol{obj}}$ is colour of objects under water including bottom, $\boldsymbol{d}$ in attenuation coefficient (4) is equal to distance from camera to object ($|A_1 - C_1|$ in Figure 16, where A_1 is point correspondent to treated fragment in shader). Let us now focus on rendering of the water surface from underwater camera. Refraction texture is obtained from main camera C_1. Reflection texture is obtained from camera C_2 in mirror position. During rendering of reflection texture we mix (in fragmnet shader) global water colour into the texture by formula (10). In attenuation coefficient (4) parameter $\boldsymbol{d}$ is equal to $|A_2 - C_2|$ (attenuation along $A_2 - C_2$ is involved, where A_2 is point correspondent to treated fragment in shader).

Now we assume certain point $\boldsymbol{U}$ on the water surface (see Figure 16). Note that $\boldsymbol{N}$ is surface normal (normal direction is from water surface up toward sky) and $\boldsymbol{E}$ is eye vector. According to [Ros06] when:

$$1 - 1{,}33^2\,(1 - (\boldsymbol{N}.\boldsymbol{E})) < 0 \tag{11}$$

function refract() in GLSL returns 0. For this reason may occurs some artifacts (no refraction appears) when is camera near

the water surface. To avoid this we do not compute refraction vector by function refract(), but we are using approximation:

$$\boldsymbol{R_1} = 1{,}33\,(-\boldsymbol{N}-\boldsymbol{E}) + \boldsymbol{N} \qquad (12)$$

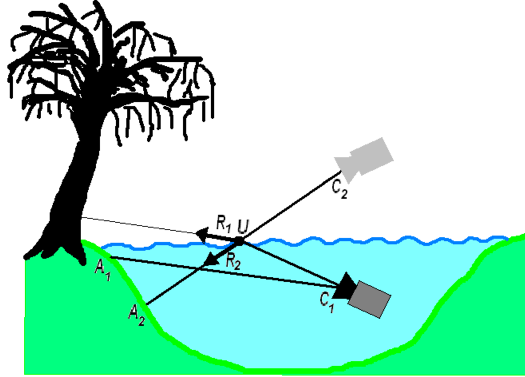

Figure 16. Underwater camera.

Now we obtain $\boldsymbol{W_1}$ by the formula:

$$\boldsymbol{W_1} = \boldsymbol{U} + \boldsymbol{m}\,\boldsymbol{R_1} \qquad (13)$$

where $\boldsymbol{m}$ is suitable scaling parameter (in our example we use 1.5). We transform $\boldsymbol{W_1}$ to texture coordinate space and obtain colour $\boldsymbol{C_{refract}}$ from texel in refraction texture at this coordinates.

Reflection vector $\boldsymbol{R_2}$ is obtained by standard formula:

$$\boldsymbol{R_2} = 2(\boldsymbol{N}.\boldsymbol{E})\boldsymbol{N} - \boldsymbol{E} \qquad (14)$$

where $\boldsymbol{N}$ is surface normal and $\boldsymbol{E}$ is eye vector. We obtain $\boldsymbol{W_2}$ similar by formula:

$$\boldsymbol{W_2} = \boldsymbol{U} + \boldsymbol{m}\,\boldsymbol{R_2} \qquad (15)$$

We transform $\boldsymbol{W_2}$ to texture coordinate space and obtain colour $\boldsymbol{C_{reflect}}$ from texel in reflection texture at this coordinates.

To assign colour to fragment corresponding to point $\boldsymbol{U}$ on water surface, we need mix refracted and reflected colour. We use formula (5) to add global water colour to refracted colour and get $\boldsymbol{C_{newrefract}}$. In attenuation coefficient (4) $\boldsymbol{d}$ is equal to $|\boldsymbol{U}–\mathrm{C}_1|$. Now we mix this $\boldsymbol{C_{newrefract}}$ with $\boldsymbol{C_{reflect}}$ by (9) (in underwater camera case is $\boldsymbol{n_1}$ index of refraction for water and $\boldsymbol{n_2}$ for air). Note that when we are blending between new reflected and refracted colour we always assume attenuation along $\boldsymbol{U} - \mathrm{C}_1$ because it is involved in $\boldsymbol{C_{newrefract}}$. In $\boldsymbol{C_{reflect}}$ we have attenuation along $\boldsymbol{U} - \mathrm{C}_2$ (it was added during rendering of reflection texture), but $|\boldsymbol{U} - \mathrm{C}_2|$ is close approximation of $|\boldsymbol{U} - \mathrm{C}_1|$. Therefor its close approximation is involved in $\boldsymbol{C_{reflect}}$. In $\boldsymbol{C_{reflect}}$ we have also involved attenuation along $\mathrm{A}_2 - \boldsymbol{U}$ (also added during rendering of texture) which appears just in reflection case because the ray is redirected back into the water volume.

Chapter 8
Caustics mapping

Caustics are observed as brightness; an increase due to many light paths hitting a bottom at the same position. We will map caustics to the bottom during rendering of the refraction texture. For this reason we create the caustics texture and then we map this texture using shadow mapping technique to the bottom. Firstly, we set up an orthogonal camera in the light direction. See Figure 17. We use orthogonal projection because the sun rays are almost parallel. Then we render underwater part of the scene from light position and store the Z buffer to the depth texture.

8.1 CPU approach

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 approximate the position where the photon hits the bottom or underwater object by position of the point $\boldsymbol{W}$ in the same way as intersection in the refraction case. Then we transform coordinates of $\boldsymbol{W}$ in camera space to texture coordinates and store these uv coordinates to the fragment colour in RG components. Next we estimate photon contribution, based on Fresnel transmittance and the traveled distance. This contribution is obtained in the same way as the colour was obtained; the refraction colour is now white and the others are black. We store the result into the B component of the fragment colour. Then we store the frame buffer obtained by this shader to array in the 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 = 1 to viewport_width
  for y = 1 to viewport_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 of the values of light that reach a bottom or underwater object from light a position. We map the caustics texture to the lake bottom.

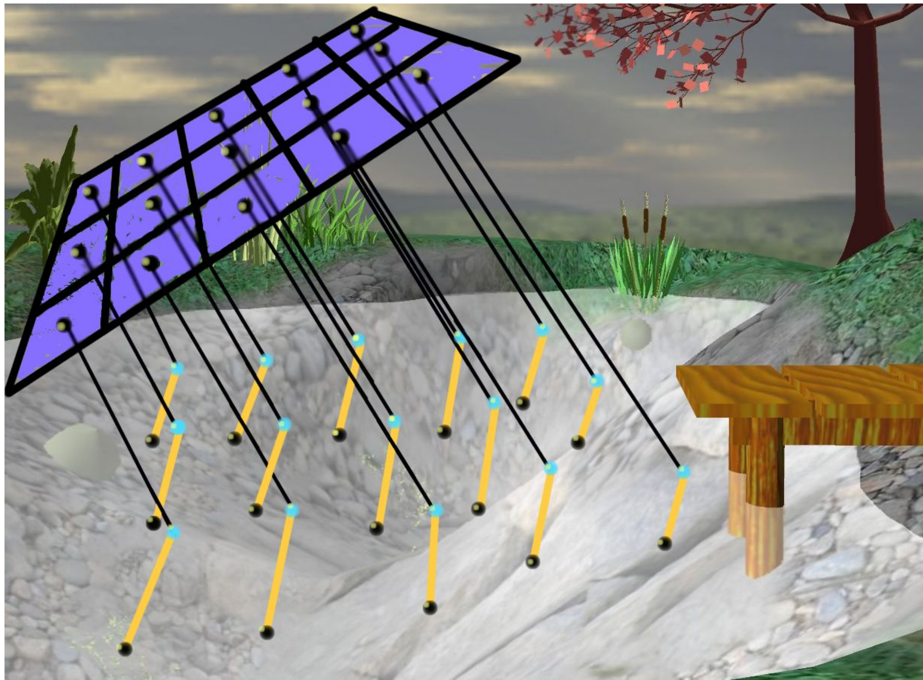

Figure 17. Mapping the photons from the orthogonal camera.

8.1 GPU approach

OpenGL 2.1 adds backwards compatible enhancements to OpenGL's advanced programmable pipeline including: Pixel Buffer Objects

for fast texture and pixel copies between frame buffer and buffer objects in GPU memory; texture images specified in standard sRGB color space for enhanced application color management flexibility. Suggestion [Sta08] was based on usage of Pixel Buffer Object. First we attach float texture to Frame Buffer Object and render water surface by mentioned fragment shader which stores coordinates of intersection to RG and intensity to B component. Then we copy this to Vertex Buffer Object. If we draw this Vertex Buffer Object we will get vertices with coordinates: X = R, Y = G and Z = B. We can now treat this vertices in the vertex shader. Note that we have uv coordinates of photon in X and Y coordinates of vertex and its intensity in Z coordinate. Before rendering to caustics texture we setup orthogonal projection where X and Y are from 0 to 1 (because uv coordinates are from 0 to 1 and then we do not need to transform it in vertex program). Then we render this vertices as points and we define its color in vertex program as photon intensity (Z coordinate of vertex). To achieve effect of caustics we need to increase intensity by every point which hits the same position. This is done by enabling blending with function which increase color in framebuffer by incoming color (addition of both colours). Then we store this to caustics texture.

If graphics card does not support Pixel Buffer Object, we can (same as in CPU approach) store fragment buffer (with uv coordinates and photon intensities) into array in main memory. We can draw this array as array of vertices. We process this vertices in vertex program and blend them as points in the same way as was mentioned.

Chapter 9

Implementation

As we mentioned earlier, we have decided to use OpenGL Shading Language. Let us describe the key parts of our fragment shader.

9.1 Refraction and reflection

Let $\boldsymbol{E}$ denotes normalized eye vector, $\boldsymbol{N}$ is surface normal in main camera coordinate space. Now we assume refraction camera. We denote $\boldsymbol{d_v}$ as the length from current point to the bottom ($|\boldsymbol{A_1}\text{-}\boldsymbol{U}|$ in Figure 7). Following code performs proposed intersection algorithm.

```
vec2 TexCoor(vec3 ve){
  vec4 pos = gl_ProjectionMatrix * vec4(ve, 0.0);
  float s = (pos.x/pos.w)/(2.0*FOV_ext) + 0.5;
  float t = (pos.y/pos.w)/(2.0*FOV_ext) + 0.5;
  return vec2(s, t);
}

vec3 RayIntersec(vec3 P1, vec3 P2, vec3 v1, vec3 v2){
  vec3 c = P2-P1;
  float m = dot(cross(c,v2),cross(v1,v2));
  float n = pow(length(cross(v1,v2)),2.0);
  return P1 + (m/n)*v1;
}

vec3 toRefractionCamSpace(vec3 v){
  return vec3 (gl_TextureMatrix[2] * vec4(v, 0.0));
}

//refraction, step1 - computation A1 W1
vec3 refrvec = refract(-E,N,0.75);
vec3 R1 = toRefractionCamSpace(refrvec);
vec3 u1 = toRefractionCamSpace(u);
vec3 E1 = normalize(-u1);
vec3 A1 = u1 + dv*(-E1);
vec3 W1 = u1 + dv*R1;
```

```
//step 2 - computation A2 , W2
vec2 t_coor = TexCoor(W1);
depth = ConvertDepth(texture2D(depthTex, t_coor).z);
vec3 A2 = -vec3(W1.x*depth/W1.z, W1.y*depth/W1.z, depth);
W = W1;
dist = length(A2-W1);
A = A2;
vec3 W2 = RayIntersec(A1,u1,A2-A1,R1);
float len1=length(A2-A1);
float len2=length(W2-A1);
float len3=length(A2-W2);
float zn = 1.0;
if(len1*1.1 < len2+len3){
 if(abs(W1.z)> abs(A2.z)) zn = -1.0;
 W2 = W1 + zn*length(A2-W1)*refrvec;
}
//step 3- computation A3, W3
t_coor = TexCoor(W2);
depth = ConvertDepth(texture2D(depthTex, t_coor).z);
vec3 A3 = -vec3(W2.x*depth/W2.z, W2.y*depth/W2.z, depth);
if(length(A3-W2) < dist){
 W = W2;
 dist = length(A3-W2);
 A = A3;
}
```

Because of hardware constrains we are not able to perform more then just three steps of our intersection algorithm. Note that $\boldsymbol{u}$ is the current point (correspondent to treated fragment) on the surface in main camera coordinates. As a result the point $\boldsymbol{W}$ is supposed to be on the bottom, where the refracted ray hits it. We get ***u1*** and ***R1*** by transformation $\boldsymbol{M}$ from $\boldsymbol{u}$ and refraction vector. $\boldsymbol{M}$ is transformation from the main camera coordinate space to refraction camera coordinate space (see Section 7.2) performed by ***toRefractionCamSpace()*** (in matrix ***gl_ TextureMatrix[2]*** is transformation from main camera modelview to refraction camera modelview).

Function ***ConvertDepth()*** converts depth buffer values to Z coordinetes. To get texture coordinates we perform transformation ***T*** (from camera coordinates to texture coordinates) by function ***TexCoor()***. Since we have extended field of view (see Figure 5) during rendering of textures, we assume coefficient of extension of field of view ***FOV_ext***. Selection of colour of texel in our refraction texture noted by ***refraTex*** is then given by the following two lines.

```
t_coor = TexCoor(W);
vec4 refrcol = texture2D(refraTex, t_coor);
```

The rendering pass for reflection is simpler then the above approach, and is described by the following code.

```
vec3 refl = c*reflect(-reflect(-E,N),-wn);
t_coor = TexCoor(ur + refl);
vec4 reflcol = texture2D(refleTex, t_coor);
```

Here we consider the reflection texture noted by ***refleTex*** and determine the texture coordinates using reflection vector. Constant ***c*** is a suitable scaling factor. Note that ***wn*** is normal to our clipping plane used to transform reflection vector into the reflection camera space. Also ***ur*** is mirror position of ***u*** on the other side of the water plane. Then we obtain colour of texel from ***refleTex***.

Once we have obtained the colour of both texels, we should mix them to create the final fragment colour. We add colour of water to refracted colour. We compute ***d***, the length from ***u*** to bottom ***A*** from intersection algorithm. Then we update the refraction colour as we explained earlier. We write this in OpenGL Shading Language notation as

```
float d = length(A-u); //length from surface to bottom
float a = exp(-d*k);
newrefrcol = watercol + a*(refrcol-watercol);
```

Note that the parameter $\boldsymbol{k}$ determines transparency of the water. The Following code will mix it with the reflected colour.

```
float f = ((1.0-0.75)*(1.0-0.75))/((1.0+0.75)*(1.0+0.75));
fFresnel= f + (1.0-f)*pow(1.0-max(dot(E,N),0.0), q);
vec4 col = newrefrcol + fFresnel*(newreflcol-refrcol);
```

This adds Fresnel phenomena to the final appearance of our surface.

9.2 Caustics

For caustics texture the resolution of 256x256 with enabled filtering is suitable. To add the caustics we map caustics texture to bottom using shader which works similar to the shadow mapping. Therefore we can add also shadows. To create caustics texture we perform a technique similar to photon mapping. Note that the sun produces the directional light; for this reason we set up the orthogonal projection. We put the orthogonal camera above the water surface to render the bottom from light position and store Z buffer to depth texture. This texture is used to compute length $\boldsymbol{d_v}=|\boldsymbol{A_1} - \boldsymbol{U}|$ and $\boldsymbol{d}=|\boldsymbol{A_2} - \boldsymbol{U}|$ from Figure 8 in the fragment shader.

```
vec3 W1 = u + refract(-E,N,0.75)*dv;
vec4 refrpos = gl_ProjectionMatrix * vec4(W1,1.0);
float tx2 = (refrpos.x/refrpos.w)/2.0+0.5;
float ty2 = (refrpos.y/refrpos.w)/2.0+0.5;
depth = ConvertDepth(texture2D(depthTex, vec2(tx2, ty2)).z);
vec3 A2 = -vec3(W1.x*depth/W1.z, W1.y*depth/W1.z, depth);
float d = length(A2-u);



float blend = exp(-d*k);
float illum = 1.0 - fFresnel;
illum = blend * illum;
gl_FragColor = vec4(tx2, ty2, illum, 1.0);
```

First part of this shader code is actually one iteration of our intersection algorithm. We are here computing the texture coordinates, but they are not referred to in any texture yet. At this point, we estimate where the photon cast from camera is hitting the bottom. See Figure 17. Second part of this shader code is similar to the computation of a colour, but here we compute just the intensity noted by ***illum***. It represents the amount of "not-scattered" or reflected energy in the place where the photon hits the bottom. The shader stores coordinates where the photon hits the bottom into RG component of fragment colour. It stores ***illum*** in B component. We store the frame buffer rendered by this shader to the array in the 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 on CPU. We can bypass searching of this array on CPU by drawing it as vertex array and threat it in vertex program. We use transformation from orthogonal camera to main camera to map caustics texture. Since we have stored depth texture we can easily add shadows.

9.3 Glow

To achieve this HDR effect we render only water surface specular highlight into low resolution viewport by following fragment shader:

```
float RE = dot(R,E);
vec4 sun_color = vec4(1.0,1.0,0.8,1.0);
gl_FragColor = pow(max(0.0, RE),200.0) * sun_color;
```

where $\boldsymbol{N}$ is surface normal and $\boldsymbol{E}$ is eye vector. Then we store the framebuffer into the array in main memory. Next we apply following filter to this rendered image and store it to glow texture:

```
for x = 1 to viewport_width
  for y = 1 to viewport_height
   color = array[x][y] / 13;
   for k = -3 to 3
    for l = -3 to 3
     if (k<>0 AND l<>0)
      color += array[x][y] / 13;
    glow_texture(x,y) = color;
```

We are dividing each intensity by 13 because there are 13 elements on diagonal of grid 7x7. We are finally blending this texture with glow into rendered images.

9.4 Constans and parameters

We have examined variety of parameters. The following 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	5.0f
FOV_ext	1,5f

Table 1. Constants.

Chapter 10

Results

The results are obtained from application where are above algorithms implemented in OpenGL. See Figure 18. Our lake scene consist of 131437 vertices and 85409 faces. To accelerate rendering we draw scene using Vertex Buffer Object. The water surface was represented by the mesh with 2401 vertices and 4608 faces. The animation of the surface was performed using sine function. We tested the application on multiple machines, see Table 2.

Referenced machine	caustics	
	on	off
Intel P4 3.0Ghz ATI Radeon X800 256MB	27fps	75fps
Intel Core2Duo 2.66GHz GeForce 7950GT 512MB	130fps	170fps
Intel P4 2.80Ghz GeForce 6800 128MB	57fps	69fps

Table 2. Framerates.

We create textures using Frame Buffer Object. We tested multiple resolutions of reflection and refraction texture. By extending the field of view we may loose the resolution, which starts to be notable in case of 512x512. We use 16 bit depth buffer in Frame Buffer Object on ATI Radeon. Since it can cause on some nVidia cards problems it is suitable to use 24 bit depth buffer on this hardware.

Next table shows performance in dependence of reflection, refraction and caustics texture resolutions. This result was obtained from Intel P4 3.0Ghz, 1GB ram, ATI RADEON X800 (256MB).

texture resolution	256x256	512x512	1024x1024
framerates	31fps	28fps	17fps

Table 3. Texture resolutions.

Chapter 11

Conclusion

We have proposed algorithm for creting realistic refraction (in deep water refraction is more distorted then in shallow). We have also discussed implementation of caustics on GPU. We simplified model of light transport inside the water volume to combination of refraction with caustics and global water colour. We have adapted this ideas to underwater camera case. We have also improved visual aspect of reflection by addition of glow effect to specular highlight. However, there are still other ways to extend this work by adding more realistic wave model. Another visual improvement could be achieved by adding effects like foam or god rays.

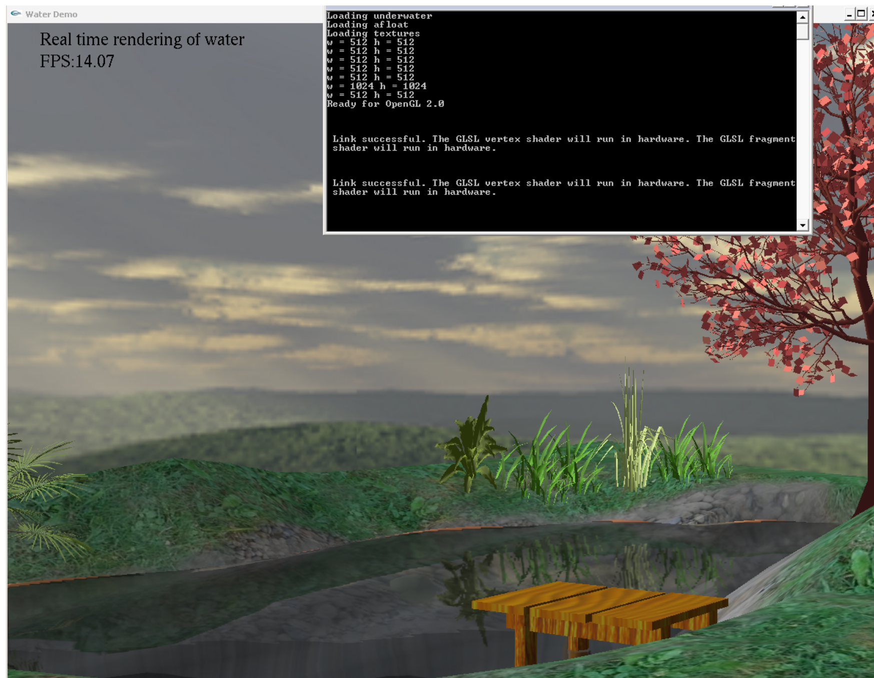

Figure 18. Demonstration application.

Glossary

GLUT

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so we can write a single OpenGL program that works across all PC and workstation OS platforms.

GLEW

GLEW is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

GLSL

OpenGL Shading Language also known as GLslang is a high level shading language based on the C programming language. It was created by the OpenGL Architecture Review Board to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages.

Fresnel equations

The Fresnel equations describe the reflection and refraction that occur at a material boundary as a function of the angle of incidence, the polarization and wavelength of the light, and the indices of refraction of the materials involved. It turns out that many materials exhibit a higher degree of reflectivity at extremely shallow (grazing) angles. Even a material such as nonglossy paper exhibits this phenomenon. For instance, if we hold a sheet of paper so that we are looking at a page at a grazing angle and looking towards a light source. We will see a specular (mirrorlike)

reflection from the paper, something we wouldn't see at steeper angles.

Caustics

Caustics are complex patterns of shimmering light that can be seen on surfaces in presence of reflective or refractive objects, for example those formed on the floor of a swimming pool in sunlight. Caustics occur when light rays from a source, such as the sun, get refracted, or reflected, and converge at a single point on a non-shiny surface. This creates the non-uniform distribution of bright and dark areas.

HDR

Dynamic range refers to the range of brightness levels that exist in a particular scene – from darkest – before complete and featureless black, to lightest – before complete featureless white. High dynamic range rendering is the rendering of 3D computer graphics scenes by using lighting calculations done in a larger dynamic range. One of the primary features of HDR is that both dark and bright areas of a scene can be accurately represented.

Glow

Glow or light bloom is an effect used in high dynamic range rendering to reproduce an imaging artifact of real-world cameras. The effect produces fringes (or feathers) of light around very bright objects in an image.

References

[Bel03] Belyaev, V., Real-time Simulation of Water Surface. GraphiCon-2003, Conference Proceedinks, pp. 131-138.

[Sou05] Sousa, T., Generic Refraction Simulation. GPU Gems 2, NVIDIA Corporation 2005, pp. 295-305. ISBN-10:0321335597. ISBN-13:978-0321335593.

[GC06] Galin, E., Chiba, N., Realistic Water Volume in Real-time. Eurographics Workshop on Natural Phenomena (2006), pp. 1-8.

[Tes02] Tessendorf, J., Simulating Ocean Water, SIGGRAPH 2002 Course Notes #9 (Simulating Nature: Realistic and Interactive Techniques), ACM Press.

[MKP05] Musawir, S., Konttinen J., Pattaniak, S., Caustics Mapping: An Image-space Technique for Real-time Caustics. IEEE Transactions On Visualization And Computer graphics (2005).

[PA01] Premože, S., Ashikhmin M., Rendering Natural Waters. Computer Graphics forum. Volume 20, number 4 (2001), pp189-199.

[Bli76] Blinn, J., Texture and Reflection in Computer Generated Imges, CACM, 19(10), October 1976, pp 542-547.

[Bel04] Belyaev, V., Real-time rendering of shallow water. GraphiCon'2004, Conference Proceedings, 2004.

[YYM05] Yu J., Yang J., McMillan L., Real-time reflection mapping with parallax. In Symposium on Interactive 3D graphics and games (2005), pp. 133–138.

[HLCS99] Heidrich W., Lensch H., Cohen M. F., Seidel H.-P., Light field techniques for reflections and refractions. In Rendering Techniques '99 (Proc. EGWorkshop on Rendering)(June 1999), pp. 187–196.

[Sta08] Starinský J., 2008. Personal communication. Bratislava 2008.

[Ros06] Rost R. OpenGL(R) Shading Language. Addison Wesley 2006 ISBN:0321197895.

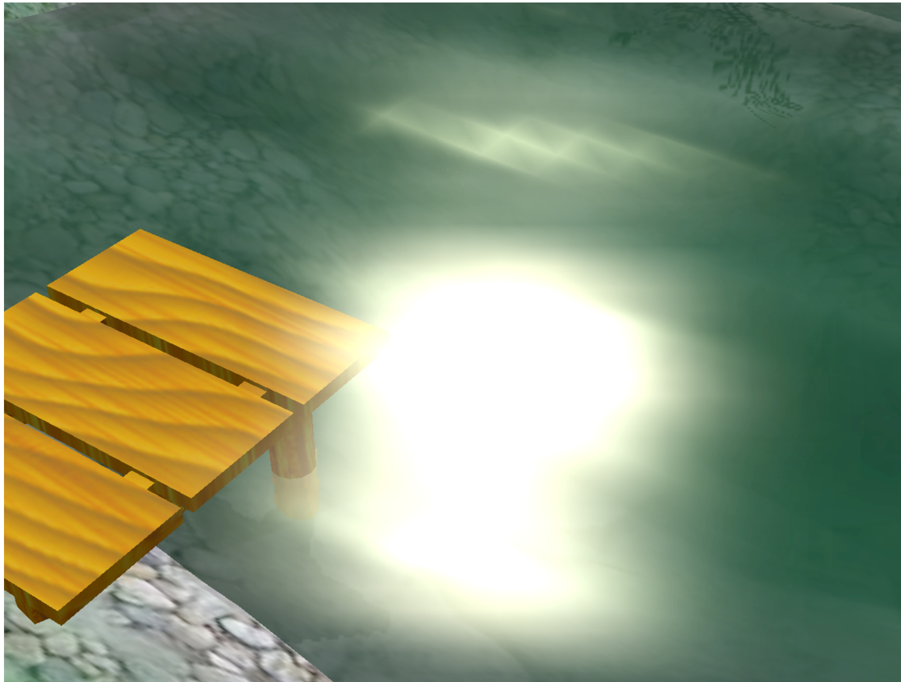

Figure 19. Glow effect.

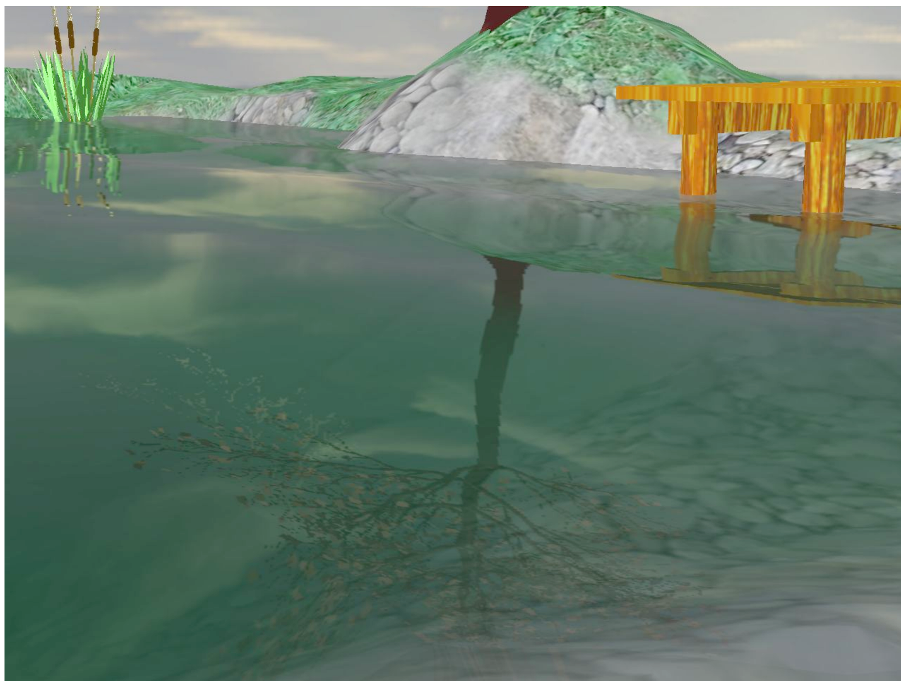

Figure 20. Reflection of the tree.

Figure 21. View from underwater camera.

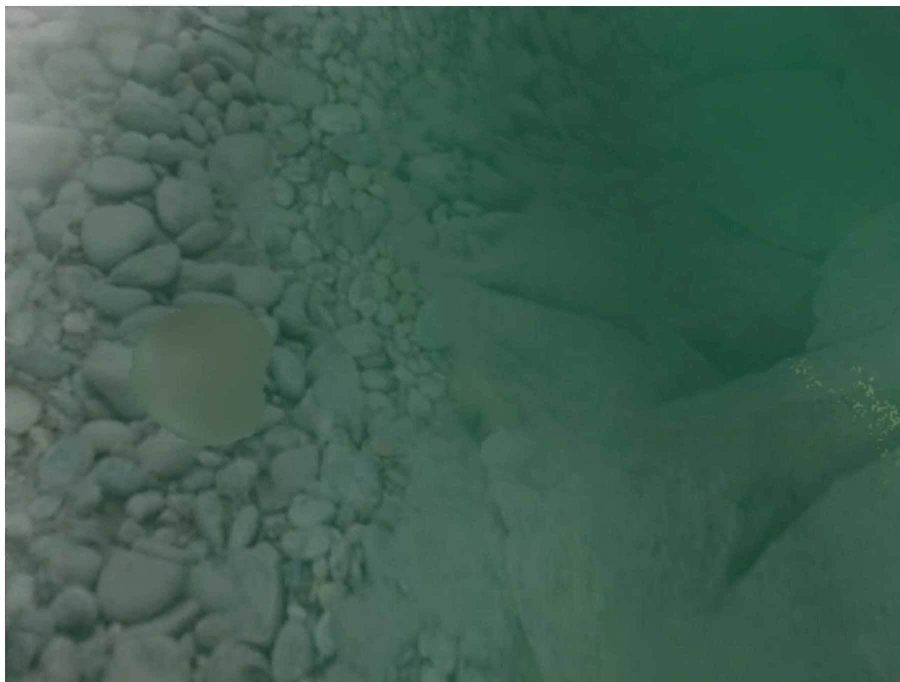

Figure 22. Refraction.

Abstrakt

Táto práca sa zaobera implementáciou optických javov na vodných hladinách. Aj napriek možnosťiam dnešného hardvéru, je aproximácia a imitácia stále podstatnou súčasťou grafiky v reálnom čase. Prezentované algoritmy su zamerané na zobrazenie známych efektov ako sú reflekcia, refrakcia a kaustiky. Napriek jednoduchosti algoritmov dokážeme dosiahnuť slušný vzhľad animácie, čo je dôležité v súčastných počitačových hrách a simuláciach v reálnom čase.

Kategórie: [Počitačová grafika] 3D grafika a realizmus

Kľúčové slová: refrakcia, reflekcia, kaustiky.