Real-Time Rendering of Water in Computer Graphics

Bernhard Fleck*
E0325551
186.162 Seminar (mit Bachelorarbeit) WS, 4h

Abstract

The simulation and rendering of realistic looking water is a very difficult task in computer graphics, due to the fact that everybody knows how it should behave and look like. This work will focus on rendering realistic looking water in real time. The simulation of water will not be described. Therefore this paper can be seen as a survey of current rendering methods for water bodies. Though simulation will not be mentioned data structures which are used by the most common simulation methods are described, because they can directly be used as input for the later presented surface extraction and rendering techniques. The correct handling of the interaction of light with a water surface can highly increase the perceived realism of the rendering. Therefore methods for physically exact rendering of reflections and refractions will be shown, using Snell's law and the Fresnel equations. The light water interaction does not stop with the water surface, but continues inside the water volume, causing caustics and beams of light. Rendering techniques for these phenomena will be described as well as bubbles and foam.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

Keywords: water, rendering, real-time, reflection, refraction, foam, bubbles, fresnel, caustics

1 Introduction

One important part in computer graphics since the last two decades is the realistic simulation of natural phenomena. Among them water may be the most challenging one. At rest, large water bodies can easily represented as flat surfaces, but that can change rapidly, because water is also a fluid which can move in a very complex way. Even if we except the simplification that water can be represented as a flat surface, a realistic looking rendering can not be achieved easily because of complex optical effects, caused by reflection and refraction. If we move below the water surface things stay as complicated as above. In fact the complexity increases due to light scattering effects. Given these statements the problem of representing the natural phenomena water in modern computer graphics can be separated into two parts:

- Simulation of the complex motion of water, which includes time dependant equations for physically correct behaviour.

*e-mail: bernhard@fleck.cc

Figure 1: Rendering of ocean (Image courtesy of J. Tessendorf).

- Rendering of water, which includes the rendering of complex light water interactions.

This work will only deal with the second part, the realistic rendering of water, with the focus on real time rendering techniques. For an introduction to the simulation aspect see [Schuster 2007] or [Bridson and Müller-Fischer 2007].

This paper can further be seen as a survey to the whole water rendering process. First the necessary basics will be presented to mediate the needed back ground knowledge for the later sections. The basics will also shortly mention the two most common simulation approaches and their impact on rendering techniques due to different data structures. After the basics, a section about rendering techniques shows a few methods how water can be rendered on today's graphics hardware. As mentioned above, the plain and simple rendering of a water surface will not look very convincing, therefore the last section is about the question how we can improve the perceived realism. For an example rendering using described techniques in this paper see Fig. 1.

2 Basics

For simulation and rendering of water sophisticated physical models are needed. As stated above this work will not cover the simulation part, but this section will cover the basic data structures which are used by the most common simulation methods. In general the motion of fluids is described by the Navier-Stokes equations. Until now there are two different approaches to track the motion of a fluid, the Lagrangian and the Eulerian. The Lagrangian approach uses a particle system to track the fluid. Each particle represents a small potion of the fluid with individual parameters like position $\vec{x}$ and velocity $\vec{v}$. One could say one particle represents one molecule of the fluid. In contrast the Eulerian approach traces the motion of fluids at fixed points, e.g. with a fixed grid structure. On these

fixed points (grid nodes) the change of fluid parameters are monitored. These parameters could be density, pressure or temperature. This section will therefore describe the data structures used in both approaches, namely particle systems and grids.

Almost every method to increase the realism of a rendered water surface needs background knowledge about the physically correct behaviour of light in connection with water. Therefore the optical basics are also covered in this section. Heightfields are also described, which can be used to represent water volumes at very little memory costs. At last the cube map texturing technique will be presented, which can be used for very fast and efficient reflection and refraction rendering.

2.1 Optics

For realistic water renderings it is essential to handle the interaction between the water surface and light correctly. This realism can be achieved if reflections and refractions are computed and if the Fresnel equations are used to calculate the intensity of the reflected and refracted light rays. Another important point is under water light absorption, e.g. the behaviour of light travel in water volumes. This section will therefore cover the methods to calculate basic ray reflections and refractions in vector form. The Fresnel equations will also be covered with focus on air - water interaction. Finally the necessary equations for under water light absorption are presented.

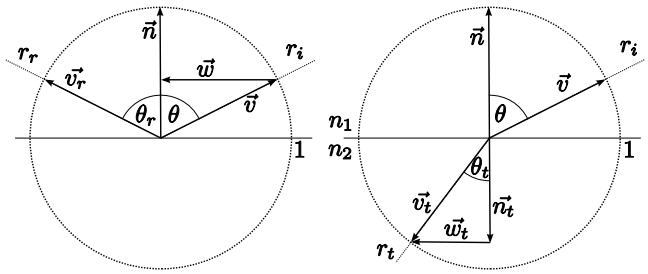

Figure 2: Left: Reflection. Right: Refraction.

2.1.1 Reflection

Reflection is the change in direction of light, or more general of a wave, on the interface of two different substances so that the light returns into the media it comes from. There exist two types of reflection:

- Specular Reflection
- Diffuse Reflection

Only specular reflection is covered in this section, because for diffuse reflection multiple outgoing light rays have to be calculated and this can not be achieved in real time. The law of reflection states, that the angle θ between the incoming light ray r_i and the surface normal $\vec{n}$ is equal to the angle θ_r between the reflected light ray r_r and the surface normal. See Fig. 2 Left.

Let $\vec{v}$ be the inverse direction vector of the incoming light ray and $\vec{v_r}$ the direction vector of the outgoing light ray, then the direction of the outgoing ray can be calculated in vector form as followed, assuming that $\vec{n}$ and $\vec{v}$ are normalized:

$$\cos\theta = \vec{n}\cdot\vec{v} \tag{1}$$
$$\vec{w} = \vec{n}\cos\theta - \vec{v} \tag{2}$$
$$\vec{v_r} = \vec{v} + 2\vec{w} \tag{3}$$

The intensity of the reflected ray has to be calculated with the Fresnel equations which are described later on in this section.

2.1.2 Refraction

Refraction is the change in direction of a wave, such as light, in relation to a change in its speed. Such changes in speed happen if the media in which the wave travels changes. The most common known example is the refraction of light on a water surface.

Snell's law describes this behaviour and states that the angle θ between the incoming ray r_i and the surface normal $\vec{n}$ is related to the angle θ_t between the refracted light ray r_t and the inverse normal $\vec{n_t}$. See Fig. 2 Right. This relation is given as followed:

$$\frac{\sin\theta}{\sin\theta_t} = \frac{v_1}{v_2} = \frac{n_2}{n_1} \tag{4}$$

where v_1 and v_2 are the wave velocities in the corresponding media and n_1 and n_2 are the indices of refraction depending on the media. To get the refracted angle θ_t we can use Snell's law:

$$\cos^2\theta_t = 1 - \sin^2\theta_t \qquad \text{Pythagorean identity} \tag{5}$$
$$= 1 - \eta^2\sin^2\theta \qquad \text{Snell's law, where} \eta = \frac{n_1}{n_2} \tag{6}$$
$$= 1 - \left(\eta^2 - \eta^2\cos^2\theta\right) \tag{7}$$

The direction of the refracted light ray r_t with its direction vector $\vec{v_t}$ can be calculated as followed:

$$\cos\theta_t = \sqrt{\cos^2\theta_t} \tag{8}$$
$$\vec{n_t} = -\vec{n}\cos\theta_t \tag{9}$$
$$\vec{v_t} = \eta\vec{w} + \vec{n_t} \tag{10}$$

The last equation is due to $\vec{w_t} = \frac{\vec{w}}{|\vec{w}|}\sin\theta_t = \vec{w}\frac{\sin\theta_t}{\sin\theta} = \eta\vec{w}$.

2.1.3 Fresnel Equations

The intensities of reflected and refracted light rays depend on the incident angle θ_i and on the refraction indices n_1 and n_2. With the Fresnel Equations the corresponding coefficients can be calculated. The coefficient is dependent of the polarization of the incoming light ray. For s-polarized light the reflection coefficient R_s is given by:

$$R_s = \left[\frac{\sin(\theta_t - \theta_i)}{\sin(\theta_t + \theta_i)}\right]^2 = \left(\frac{n_1\cos\theta_i - n_2\cos\theta_t}{n_1\cos\theta_i + n_2\cos\theta_t}\right)^2 \tag{11}$$

For p-polarized light the coefficient R_p is given by:

$$R_p = \left[\frac{\tan(\theta_t - \theta_i)}{\tan(\theta_t + \theta_i)}\right]^2 = \left(\frac{n_1\cos\theta_t - n_2\cos\theta_i}{n_1\cos\theta_t + n_2\cos\theta_i}\right)^2 \tag{12}$$

The refraction (transmission) coefficients are given by: $T_s = 1 - R_s$ and $T_p = 1 - R_p$. For unpolarized light containing an equal mix of p- and s-polarized light the coefficients are given by:

$$R = \frac{R_s + R_p}{2} \qquad T = \frac{T_s + T_p}{2} \tag{13}$$

Fig. 3 shows the reflection and refraction coefficients for an air to water transition at angles form 0° to 90°.

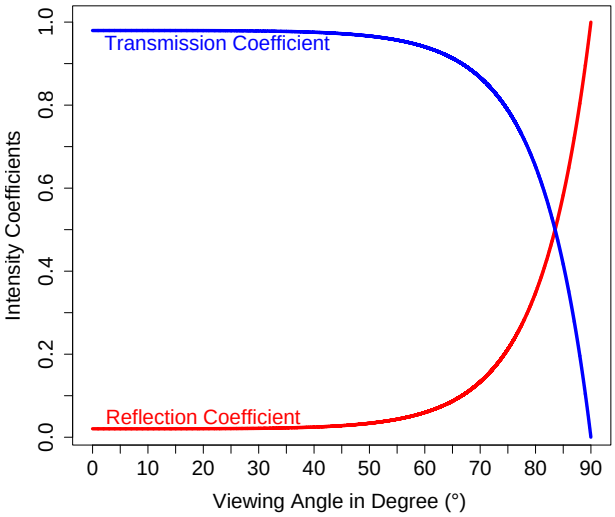

Figure 3: Reflection and Refraction coefficients for an air water surface at angles 0° to 90°.

2.1.4 Underwater light absorption

When photons enter a water volume they are scattered and absorbed in such a complex manor, that computing this phenomena is very difficult. The following will present a simplified model presented by [Baboud and Décoret 2006], which is based on [Premŏze and Ashikhmin 2001]. The model describes the transmitted radiance from one point p_w under water to a point on the water surface p_s for a given wavelength λ:

$$L_\lambda(p_s,\vec{\omega}) = \underbrace{\alpha_\lambda(d,0)\,L_\lambda(p_w,\vec{\omega})}_{\text{radiance coming from } p_w} + \underbrace{(1-\alpha_\lambda(d,z))\,L_{d\lambda}}_{\text{diffuse scattering}} \tag{14}$$

Where $\vec{\omega}$ is the direction from p_w to p_s, $L_\lambda(p,\vec{\omega})$ is the outgoing radiance at p in direction $\vec{\omega}$, z is the depth of p_w, d is the distance between p_w and p_s and $\alpha_\lambda(d,z)$ is an exponential attenuation factor dependant of depth and distance:

$$\alpha_\lambda(d,z) = e^{-a_\lambda d - b_\lambda z} \tag{15}$$

where a_λ and b_λ are attenuation coefficients depending on the water properties itself.

In nature light attenuation is dependant on wavelength, therefore the computations should be done per wavelength. A simplification would be to just use the 3 components of the RGB colour space and do the computations component wise:

$$\alpha(d,z) = (\alpha_R(d,z), \alpha_G(d,z), \alpha_B(d,z)) \tag{16}$$

The equation can further be simplified by taking the observation into account, that the influence of depth is minimal, e.g. the b_λ term. By simply dropping it the exponential attenuation factor can be reduced to:

$$\alpha_\lambda(d) = e^{-a_\lambda d} \tag{17}$$

This yields to the fact that $L(p_s,\vec{w})$ is simply a linear blending between $L(p_w,\vec{\omega})$ and L_d with respect to $\alpha_\lambda(d)$.

2.2 MAC-Grid

The Marker and Cell method is used to discretize space and was first mentioned by [Harlow and Welch 1965] for solving incompressible flow problems. They introduced a new grid structure. Now it is one of the most popular methods for fluid simulation. Space is divided into small cells with a given edge length h. Each cell contains certain values needed for the simulation, like pressure and density. These values are stored at the centre of the cell. For each cell velocity is also stored, not in the centre of the cell, but in the centre of the edges of the cell. See Fig. 4.

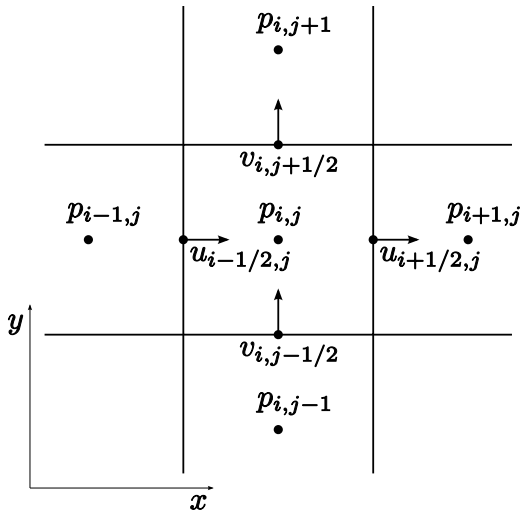

Figure 4: MAC-Grid in 2D.

This staggering of the variables makes the simulation more stable. Additionally for simulation marker particles are used. They move through the velocity field represented by the MAC-Grid. These marker particles determine which cell contains fluid, i.e. they determine changes in pressure and density.

2.3 Particle Systems

Particle systems are rendering techniques in computer graphics to simulate fuzzy phenomena like fire, smoke and explosions. A particle system consists of N particles $0 \leq i < N$ with at least values for position x_i and velocity v_i for each particle. Additional parameters could be: size, shape, colour or texture and for physical simulations: mass and accumulated external forces.

A particle system is usually controlled by so called emitters. Emitters create new particles at a user given rate (new particles per time step) and they describe the behaviour parameters for particles, e.g. they set the initial position and velocities for particles. It is common to set the values for a particle to a central value given by the emitter with a certain random variation. Particles also have a lifetime set by the emitter. If the lifetime exceeds the particle either fades out smoothly or just vanishes.

The basic steps of a particle system algorithm can be divided into two steps: simulation and rendering. During simulation new particles are created according to the emitter, particles with exceeded lifetimes are destroyed and the attributes of existing particles are updated. During the rendering step all particles with current attributes are rendered. There are several rendering methods for particles, but the easiest way would be to just render them as points

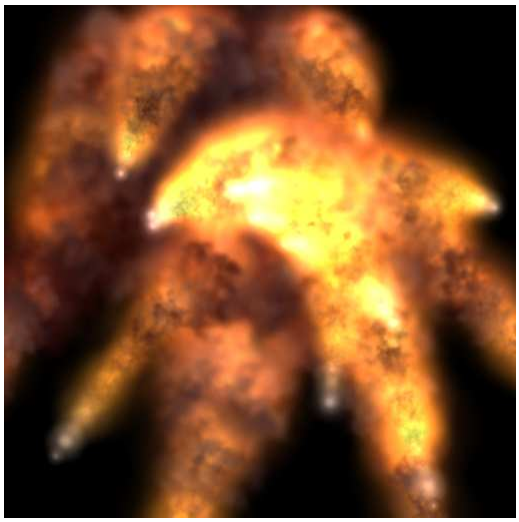

Figure 5: Example particle system.

with a given size. The most common approach is to render them as oriented, alpha blended and textured billboards. Billboards are small quads oriented so that they always face the camera. See Fig. 5 for an example.

2.4 Heightfields

A heightfield is a raster image which represents surface elevation (height) and is therefore also called heightmap. The most common application for heightfields is terrain rendering, but they can also represent water surfaces (as shown in Section 3.3). See Fig. 6 Left for an example heightmap. Black colour values in the image represent low elevation, while white values represent high elevation. Any common texture format can store a heightmap, but 8bit greyscale formats are mostly used. With an 8bit greyscale image 256 different height values can be represented. But also 16bit (65536 height values) or 32bit (16M height values) images can be used depending on the needed detail.

For rendering we first construct a regular grid in the xz-plane, with N_x nodes along the x-axis and N_z nodes along the z-axis. The values for N_x and N_z are given by the resolution of the heightmap, e.g. $N_x =$ width and $N_z =$ height of the image. A user parameter h determines the space between nodes. The total size of the resulting grid is therefore $N_x \cdot h$ along the x-axis and $N_z \cdot h$ along the z-axis. The y values for each grid point is calculated from the heightfield: $y_{i,j} = \text{heightfield}_{i,j}$ where i and j represent pixel positions in the heightfield image, with $0 \leq i < N_x$ and $0 \leq j < N_z$. See Fig. 6 Right for an example rendering.

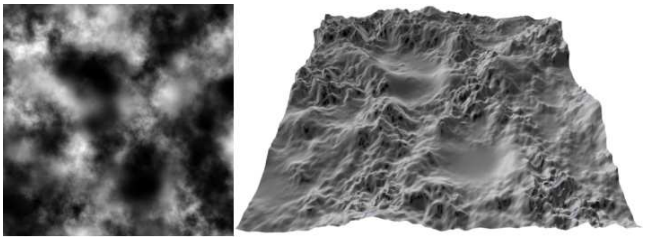

Figure 6: Left: Sample Heightfield. Right: Rendering of Heightfield

2.5 Cube Mapping

Cube mapping is a special texture mapping technique. In normal texture mapping a two dimensional image (the texture) is applied onto an object's surface. Each vertex of the object surface is assigned a 2D texture coordinate which represents the position in the texture applied to that vertex. With this method it is possible to map any kind of image onto any type op geometry. In practice it is most common to map 2D textures to triangular meshes. This mapping method is not view dependant, that means that the view point does not influence the way the texture is mapped to the surface.

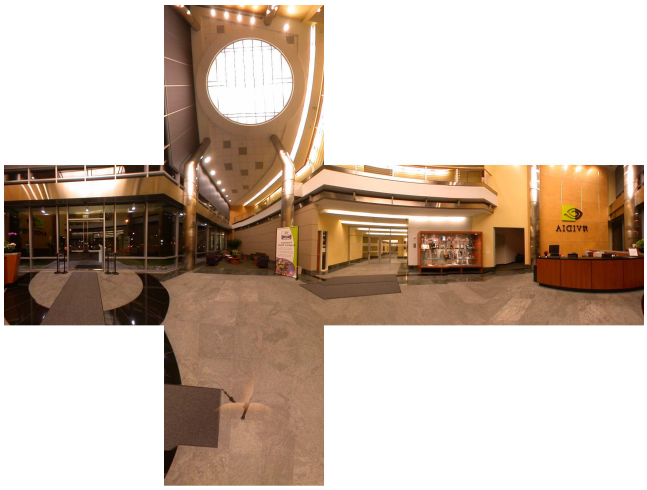

Figure 7: Unfolded cube map texture

This texturing approach is not applicable for reflective surfaces like water, because as described in Section 2.1 reflection is dependant on the incoming light ray which is view dependant (The ray from the object to the viewer can also be seen as light ray). For reflections it is needed to map the environment onto the object surface with respect to the reflection direction.

This problem, of how to map a direction for a given surface point to a texture can not easily be done with a normal 2D texture. The environment we want to map can be seen as omnidirectional picture centred at the current position. This again can not be represented as one simple 2D texture. Cube maps are a solution for this problem. With a cube map the whole environment around an object can be stored. Cube map texturing is a texturing technique which uses a 3D direction vector to access six different 2D textures arranged on the faces of a cube. Fig. 7 shows an unfolded cube map. A cube map can be build by generating six images each rotated by 90° from each other at a fixed position in space. Cube map texturing is well supported in hardware since DirectX 7 and in OpenGL with the `EXT_texture_cube_map` extension.

Cube maps are not necessarily be pre calculated, but they can also be dynamically created during the rendering. This is essential, because if the environment changes the cube map has to be updated. To generate a cube map dynamically the scene is rendered as seen from a fixed point, e.g. as seen from a reflective object, six times. For each time one of the following directions is used: positive x-, negative x-, positive y-, negative y, positive z- and negative z-axis direction. It is important to set the field of view of the camera to 90° to get an orthogonal 90° viewing frustum. Each viewing frustum corresponds to on side of the cube map. The generated renderings need to be stored as final faces in the cube map. With current graphics hardware it is possible to render directly to a texture which eliminates bottlenecks in copying the frame buffer to the cube map

texture. Afterwards the projection matrix can be set as usual and the scene rendered with the newly created cube map. It is also possible to render multiple reflective objects with cube maps. This would require a cube map for each reflective object and multiple recursive cube map updates to gain visually appealing results.

3 Rendering Techniques

So far we only have data structures which are either filled by physically correct simulations or by approximations. One could now just simply use point splatting techniques or render the particle system as mentioned in Sec. 2.3. But that would result in not very realistic looking renderings, because no additional effects like reflections and refractions are possible. For these phenomena we have to extract surfaces from the data representations. At least surfaces (meshes) are needed if we want to take advantage of current graphics hardware acceleration techniques.

This section will therefore present the marching cubes algorithm and screen space meshes, which are both surface extraction techniques. Additionally a real time ray tracing approach will be presented.

3.1 Marching Cubes

The marching cubes algorithm extracts high resolution 3D surfaces. It was first developed by [Lorensen and Cline 1987]. Their research included fast visualization of medical scans such as computed tomography (CT) and magnetic resonance (MR).

The algorithm uses underlying data structures like voxel grids or 3D arrays consisting of values like pressure or density. The result of the marching cubes algorithm is a polygonal mesh with constant density. Polygonal meshes can be rendered very quickly with current graphics hardware.

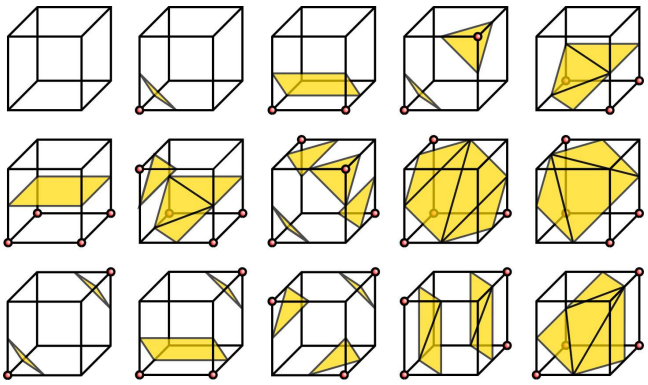

Figure 8: The 15 triangulation cases.

The algorithm works as follows. Surface triangulation and normals are calculated via a divide and conquer approach. First a cube is constructed out of 8 vertices of the underlying 3D data structure. For each cube the intersection points of the resulting surface with the cube is calculated. The resulting triangles and normals are stored in an output-list. Then the algorithm continues with the next cube. An example cube is given in Fig. 9 Left.

For the surface intersection we need a user defined value to determine which values of the 3D grid are inside, outside or on the surface. If the value of the vertex is greater or equal than the user defined value, then the vertex is inside. Otherwise the vertex is outside the surface. So we can mark all vertices with one which are inside or on the surface and with zero which are outside the surface.

The surface intersects an edge of the cube where one vertex of the edge is inside and the other vertex outside. We have 8 vertices and each vertex has 2 states (inside or outside), that gives us $2^8 = 256$ cases a surface can intersect a cube. To lookup the edges which intersect with the surface we can use a pre calculated table of the 256 intersection cases. Due to complementary and symmetry the 256 cases can be reduced to 15 shown in Figure 8.

An index for the 256 cases can be calculated as followed: The state of each vertex is either zero or one depending if the vertex is inside or outside the surface. All eight vertices form a 8 bit number which represents the index to the case table. An example cube is given in Figure 9.

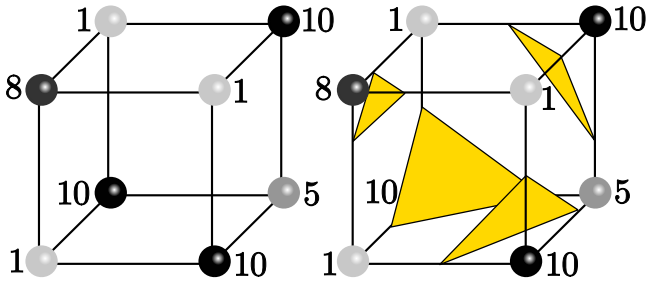

Figure 9: Left: Example cube. Vertices with values ≥ 9 are on or inside the surface. Right: Cube with triangulation.

Now that we know which edges intersect with the surface, we can calculate the intersection points. All the intersection points lie on the edges therefore we get the intersection points by linear interpolation of the vertex values.

The normals of the resulting triangles are calculated by linear interpolation of the cube vertices. These cube vertex normals are computed using central differences of the underlying 3D data structure along the three coordinate axes:

$$N_x(i,j,k) = \frac{V(i+1,j,k) - V(i-1,j,k)}{\Delta x} \tag{18}$$

$$N_y(i,j,k) = \frac{V(i,j+1,k) - V(i,j-1,k)}{\Delta y} \tag{19}$$

$$N_z(i,j,k) = \frac{V(i,j,k+1) - V(i,j,k-1)}{\Delta z} \tag{20}$$

Where $N(i,j,k)$ is the cube vertex normal, $V(i,j,k)$ the value of the 3D array at (i,j,k) and $\Delta x, \Delta y, \Delta z$ the length of the cube edges. Fig. 10 shows how important it is to calculate per vertex normals for visual quality.

In summary the marching cubes algorithm works as follows:

1. Read 3D array representing the model we want to render.
2. Create a cube out of four values forming a quad from slice A_i and four values forming a quad from slice A_{i+1}.
3. Calculate an index for the cube by comparing the values of each vertex of the cube with the user given constant.
4. Look up the index in a pre calculated edge table to get a list of edges.
5. Calculate the surface intersection by linear interpolation.

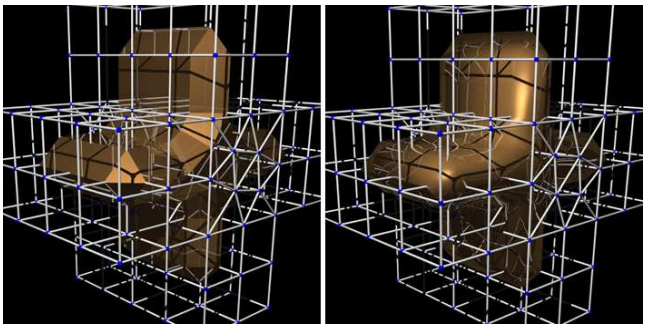

Figure 10: Surface generated using the marching cubes algorithm. Left: without per vertex normals. Right: with per vertex normals (Image courtesy of P. Bourke).

6. Calculate vertex normals.
7. Output the triangle vertices and vertex normals.

To enhance efficiency of the presented algorithm we can take advantage of the fact, that in most cases a newly constructed cube has already at least four neighbouring cubes. Therefore we only have to look at three edges of the cube and interpolate triangle vertices only at that edges.

The original marching cubes algorithm has flaws – in some cases the resulting polygon mesh could have holes. [Nielson and Hamann 1991] solved this issue by using other triangulations in certain cube on cube combinations.

Recent developments in graphics hardware made GPU implementations of the marching cubes algorithm possible, which are about 5–10 times faster than software implementations.

3.2 Screen Space Meshes

Screen space meshes are a new approach for generating and rendering surfaces described by a 3D point cloud, like particle systems, but without the need to embed them in regular grids. The following algorithm was first described by [Müller et al. 2007]. The basic idea is to transform every point of the point cloud to screen space, set up a depth map with these points, generate the silhouette and construct a 2D triangle mesh using a marching squares like technique. This 2D mesh is then transformed back to world space, for the calculation of reflections, refractions and occlusions. For an example see Figure 11.

The field of surface generation is well known and there is much related work about it. The main problem is, that other approaches can not be rendered in real time. Therefore they are not suitable for real time applications. Screen space meshes are significantly faster, because only the front most layer of the surface is constructed and rendered. With help of fake refractions and reflections the artefacts of this simplification can be reduced to a minimum.

The main advantages of this method are:

- View dependent level of detail comes for free, because of the nature of this approach. Regions near the look-at point of the camera get a higher triangle density.
- The mesh is constructed in 2D therefore a marching squares approach can be used and is naturally very fast.

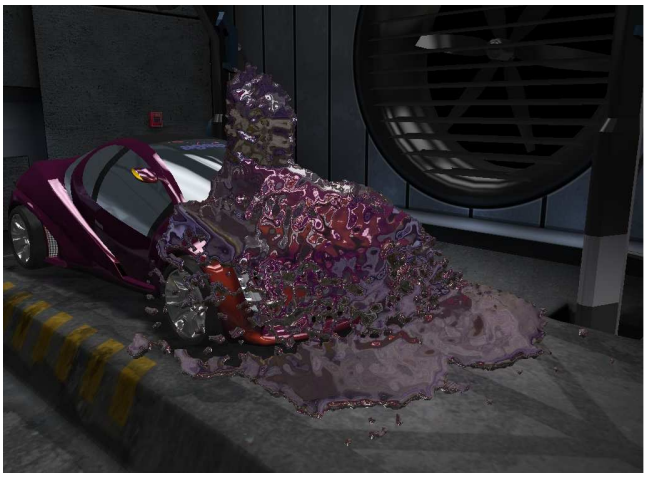

Figure 11: Example of liquid rendered using screen space meshes (Image courtesy of M. Müller et al. 2007).

- In contrast to ray tracing or point splatting which are methods with similar aspects, we can take advantage of current standard rendering hardware.
- The mesh can easily be smoothed in screen space, using depth and silhouette smoothing, resulting in better images.

The input for the algorithm is a set of 3D points. Further the projection matrix $\mathbf{P} \in \mathbb{R}^{4\times4}$, and the parameters in Table 1 are needed. The main steps of the algorithm are:

1. Transformation of points to screen space
2. Setup of depth map
3. Find silhouette
4. Smooth the depth map
5. Mesh generation, using a marching squares like approach
6. Smoothing of silhouettes
7. Transformation of constructed mesh back to world space
8. Render mesh

The smoothing of the depth map and the silhouette is optional and an extension to the algorithm and are not necessary steps to make the algorithm work. Therefore smoothing is described after the main algorithm.

Parameter	Description	Range
h	screen spacing	$1-10$
r	particle size	≥ 1
n_{filter}	filter size for depth smoothing	$0-10$
n_{iters}	silhouette smoothing iterations	$0-10$
$z_{\max}$	depth connection threshold	$> r_z$

Table 1: Parameters used for Algorithm

3.2.1 Transformation to Screen Space

First we need to transform the given set of 3D particles to screen space. For each particle let $\mathbf{x} = [x,y,z,1]^T$ be the hommogenous

coordinates. We use projection matrix **P** to get

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{21}$$

If the projection matrix is defined as in OpenGL or DirectX, the resulting coordinates of the above transformation are between -1 and 1. Therefore we need the width (W) and height (H) of our screen size in pixels to calculate the coordinates in relation to our screen window:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} W \cdot \left(\frac{1}{2} + \frac{1}{2} \cdot x'/w\right) \\ H \cdot \left(\frac{1}{2} + \frac{1}{2} \cdot y'/w\right) \\ z' \end{bmatrix} \tag{22}$$

This results in coordinates $x_p \in [0 \ldots W]$, $y_p \in [0 \ldots H]$ and z_p is the distance to the camera. The radii in screen space are calculated as followed:

$$\begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} rW\sqrt{p_{1,1}^2 + p_{1,2}^2 + p_{1,3}^2}/w \\ rH\sqrt{p_{2,1}^2 + p_{2,2}^2 + p_{2,3}^2}/w \\ r\sqrt{p_{3,1}^2 + p_{3,2}^2 + p_{3,3}^2} \end{bmatrix} \tag{23}$$

where $p_{i,j}$ are the entries of the projection matrix **P**. When using a projection matrix like the ones in OpenGL or DirectX $\sqrt{p_{3,1}^2 + p_{3,2}^2 + p_{3,3}^2} = 1$ and therefore $r_z = r$. If the aspect ratio of the projection is equal to the viewport (W/H) then the projected radii result in a circle in screen space with $r_p = r_x = r_y$.

3.2.2 Depth Map Setup

The size of the depth map depends on the width (W) and height (H) of the screen size. A user given parameter $h \in \mathbb{R}$, the screen spacing, determines the resolution of the depth map. The screen spacing parameter divides the depth map into grid cells with cell size h. This gives us a depth map resolution of $N_x = \left\lceil \frac{W}{h} + 1 \right\rceil$ nodes horizontally and $N_y = \left\lceil \frac{H}{h} + 1 \right\rceil$ nodes vertically. The depth map stores depth values $z_{i,j}$ at each node of the grid and is first initialized with ∞.

Then the algorithm iterates through all N particles two times. In the first round the depth values for all the areas which are covered with particles are set. In the second round additional values and nodes are calculated where needed for silhouettes (See Fig. 12).

The first iteration updates all depth values $z_{i,j}$ where $\left(ih - x_p\right)^2 + \left(jh - y_p\right)^2 \leq r_p^2$ with

$$z_{i,j} \leftarrow \min\left(z_{i,j},\ z_p - r_z h_{i,j}\right) \tag{24}$$

where $h_{i,j} = \sqrt{1 - \frac{(ih-x_p)^2 + (jh-y_p)^2}{r_p^2}}$. In most cases the results are sufficient even if we let go of the square root in the above equation. After that the particles are roughly sampled among the grid nodes.

3.2.3 Silhouette Detection

The second iteration over all particles is for silhouette detection. In this iteration only edges of the depth map with adjacent nodes which differ more than $z_{\max}$ are considered (see Fig. 12). These

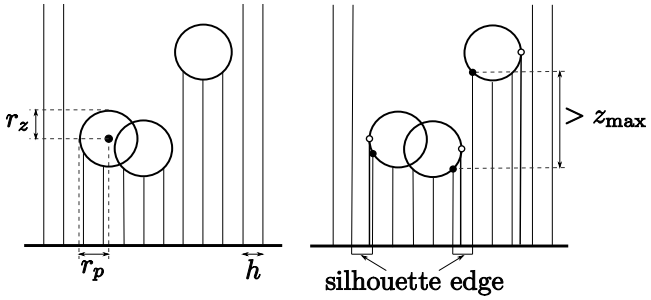

Figure 12: Left: Side view of depth map with three particles. Right: Between adjacent nodes at most one additional node (white dot) is created for the silhouette.

edges are called silhouette edges. The goal for this iteration is to get at least one new node for each silhouette edge, which is called silhouette node. A silhouette node lies on the silhouette edge with the depth value of the front layer. The front layer is the layer of the particle next to the silhouette node and nearest to the camera.

To get a new silhouette node the cut between the silhouette edge and each circle around a particle with radius r_p is been calculated. The new depth value of this newly calculated silhouette node z_p is only stored if

- the newly calculated z_p is nearer to the front layer than the back layer. This means that z_p is smaller than the average z of the silhouette edge.
- the calculated cut is further away along the silhouette edge from the particle belonging to the front layer as a previously stored silhouette node. See Fig. 13.

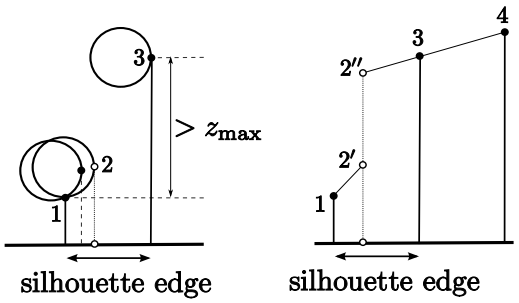

Figure 13: Left: Two cuts are generated on the silhouette edge by the two lower left particles. But only the right most is stored, because it is furthest away from the node with the smaller depth value. Right: Two vertices with different depth values have to be calculated in that case.

3.2.4 Mesh Generation

Each initialized grid node of the depth map (each node with a value of $z_{i,j} \neq \infty$) creates exactly one vertex. Extra care has to be taken on silhouette edges. On silhouette edges with one adjacent initialized node one extra vertex for the outer silhouette has to be generated with the values from the silhouette node for this silhouette edge. This is the normal case. On silhouette edges with two adjacent initialized nodes two vertices have to be generated. The first one is generated as for the normal case, but the depth value for the second one has to be interpolated between adjacent nodes belonging to the back layer. See Fig. 13.

For the final triangulation a square is formed out of four adjacent grid nodes. Each of the edges of this square can be a silhouette edge, which leaves 16 triangulation cases. See Fig. 14. If a square contains a non initialized node, all triangles sharing this node are simply dropped. The above is done for each square on the grid.

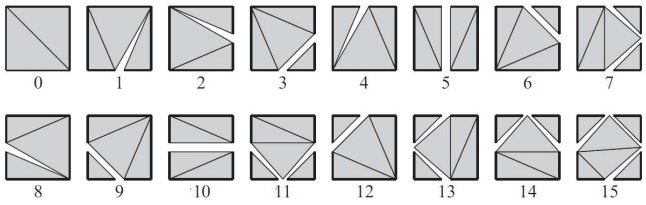

Figure 14: The 16 triangulation cases (Image courtesy of M. Müller et al. 2007).

3.2.5 Transformation back to World Space

The resulting mesh (with coordinates like $[x_p, y_p, z_p]^T$) is in screen space and therefore has to be transformed back to world space for rendering. To get the world space coordinates we need the inverse transformation for Equations 21 and 22. Let $\mathbf{Q} \in \mathbb{R}^{4\times4}$ be the inverse of the projection matrix $\mathbf{P}$ ($\mathbf{Q} = \mathbf{P}^{-1}$). To get $[x, y, z]^T$ we calculate

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{Q} \begin{bmatrix} \left(-1 + 2x_p/W\right) w \\ \left(-1 + 2y_p/H\right) w \\ z_p \\ w \end{bmatrix} \tag{25}$$

To get w from only known parameters we calculate

$$w = \frac{1 - q_{4,3} z_p}{q_{4,1}\left(-1 + 2x_p/W\right) + q_{4,2}\left(-1 + 2y_p/H\right) + q_{4,4}} \tag{26}$$

where $q_{i,j}$ are the entries from the transformation matrix $\mathbf{Q}$. After the transformation to world space, per vertex normals for the triangle mesh can be calculated.

3.2.6 Smoothing

Without smoothing the resulting mesh can be very bumpy. This is because of the depth map. To circumvent this a binominal filter with a used defined parameter n_{filter} is used. The filter is first applied horizontally and then vertically. The filter is shown in Fig. 15.

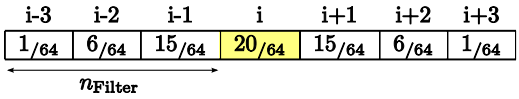

Figure 15: The filter used for smoothing the depth map, with half length $n_{\text{Filter}} = 3$.

Silhouette smoothing is also needed, because depth smoothing does not alter the silhouette. To smooth the silhouette the $[x_p, y_p]$ coordinates of the nodes in screen space are altered. Each node coordinate is replaced by the average of itself with neighbouring nodes which have a depth value of $z_p \neq \infty$. Internal mesh nodes are excluded from smoothing. The count of smoothing intervals is given by parameter n_{iters}. Smoothing of the silhouette results in shrinking of the mesh. In most cases this is a desired effect, because each particle has a certain size, which can lead to rather odd looking too large blobs. With silhouette smoothing this effect can be reduced and the rendered image looks more natural.

3.3 Ray tracing

Ray tracing is a general rendering technique in computer graphics. In nature light rays are shot from light sources, like the sun or lamps, which interact with the environment causing new light rays to be created. This process is iteratively continued for each newly created light ray. We actually see because of these light rays.

For calculation on the computer this approach would be far too expensive, because it is hardly achievable to calculate all the necessary light rays. Even light rays which does not hit the eye would be calculated. Therefore the basic idea is to shoot rays not from the light sources but from the viewer into the scene. Rays are shot from the view point through each pixel of the screen. If the ray hits scene objects new rays are cast or not, depending on the depth of recursion.

Ray tracing methods are usually slower than scan line algorithms, which use data coherence to share computations between adjacent pixels. For ray tracing such an approach can not work because for each ray the calculations start from the beginning. Depending on the used geometry in the scene the ray - object intersection calculations can be very expensive, therefore ray tracing is hardly achievable in real time. Though a real time ray tracer with strict limitations as presented by [Baboud and Décoret 2006] is shortly described.

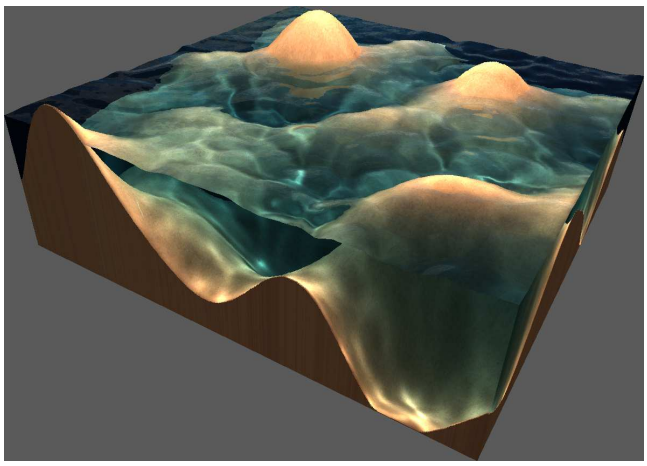

Figure 16: Reflections, Refractions and Caustics rendered with a GPU raytracer (Image courtesy of L. Baboud and X. Décoret).

3.3.1 Real Time Ray Tracing Approach

The real time ray tracing method presented by [Baboud and Décoret 2006] is based on efficient ray - heightfield intersection calculations, which can be implemented on today's GPUs. The basic algorithm for ray - heightfield intersection works as followed: The heightfield texture is sampled along each viewing ray, e.g. by fixed horizontal planes. Then a binary search can be performed to find the exact intersection point. This basic method can cause stairs. With precomputed information the stairs effect can be reduced and the ray sampled optimally.

To get interactive frame rates all ray traced scene objects have to be represented as heightfields. For water surfaces which are almost flat this works perfectly. Terrain can also be modelled as displacement over a flat plane. This results in two heightfields, ground and water surface, defining the water volume. Both heightfields can be stored as 2D textures. The water texture can be the output of a real physical simulation or any other water simulation method. The terrain texture can contain real terrain data or can be procedurally generated at an initialization step.

For rendering, the bounding box of the combined water and ground volume is rendered, and a fragment program (pixel shader) is used for ray tracing. An important simplification is, that only first order rays are calculated, e.g. the ground surface has to be of diffuse material. This is because currently there exists no hardware support for recursive function calls. The basic steps of the algorithm are as followed:

1. Calculate the intersection point from the viewing ray with the water surface.
2. Calculate the reflected and refracted rays using Snell's law.
3. Intersect reflected and refracted rays with either the ground surface or an environment map.
4. Calculate corresponding colour values for the intersection points.
5. Blend the two colour values with respect to the Fresnel equations.

The different interactions for the viewing ray with the water volume is shown in Fig. 17. If the viewing ray hits the ground surface

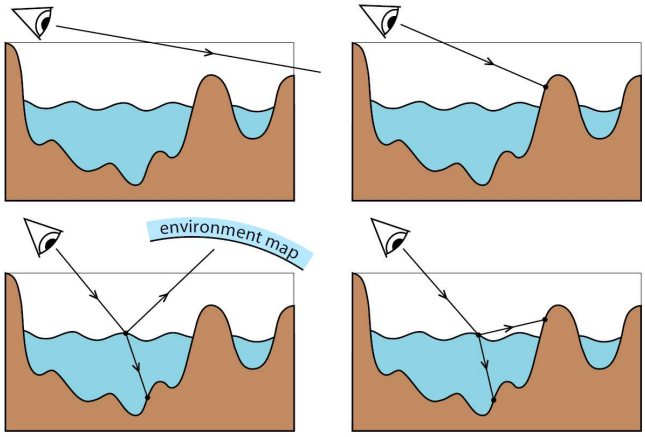

Figure 17: Four cases how the viewing ray can interact with the water volume (Image courtesy of L. Baboud and X. Decoret).

before the water surface, then no reflection and refraction rays are calculated, and the colour value for the intersection point is directly computed. For reflected rays three different kind of intersections can happen. Intersection with:

1. Environment Map
2. Local Objects, which are not handled in this approach.
3. Ground Surface.

The correct blending of the colour contributions for the reflected and refracted rays is done by the Fresnel equations. For fixed refraction indexes n_1 and n_2 the reflection and refraction coefficients only depend on the viewing angle θ. These values are precomputed and stored in a 1D texture.

For the colouring of the ground special care has to be taken due to under water light absorption. The details of the used model are given in Section 2.1. After simplifications the under water light absorption is only dependant of the travelled distance and can be precomputed and stored, as for the reflection and refraction coefficients, in a 1D texture.

Integration with other objects which are no heightfields is also possible, but with certain constraints. Objects outside the bounding box of the water volume are no problem, because the z buffer is correctly set for all rendered fragments. Therefore that objects can be rendered using the standard rendering pipeline. For objects which lie partially or total inside the water volume the benefit of the fast ray tracing algorithm for heightfields is lost. Rendering such objects with the standard rendering pipeline would result in false looking reflections and refractions.

Caustics can also be simulated, but are hard to compute in a forward ray tracer, therefore [Baboud and Décoret 2006] use a two pass photon mapping approach. First the water surface is rendered from the light source into a texture, storing the positions where the refracted light rays hit the ground surface. The ground surface is a heightfield, therefore only the (x,y) coordinates need to be saved and can be stored in the first two channels of the texture. The third channel is used to store the photon contribution, based on the fresnel equations for the transmittance coefficient, the travelled distance and a ray sampling weight.

By gathering the photons of this texture an illumination texture is generated. This is done by looking at each texel of the photon texture, extracting position and intensity of the related photon and adding this intensity to that stored at the corresponding position in the illumination texture. The illumination texture can be very noise, because only a limited number of photons can be cast to achieve real time frame rates. Therefore the illumination texture has to be filtered to improve visual quality. A benefit for this method is, that shadows cast by the ground on itself are also generated. As above the integration with other objects would break with the restrictions and must be separately handled, loosing the performance advantage of the fast ray heightfield intersection method.

4 Adding Realism

At that point we only have a polygonal mesh representing the water surface and per vertex normals. Even if we would render this mesh with appropriate water textures it would not look very realistic. Realistic looking water can be achieved by using special rendering techniques. This section will focus on these techniques and how they impact the realism of the simulation.

4.1 Reflection and Refraction

Reflections and refractions contribute the most to the perceived realism of the simulation of water surfaces. When a ray hits the water surface part of it is reflected back in the upward direction and part of it is refracted inside the water volume. The reflected ray can further hit other objects causing reflective caustics. The refracted scattered ray can also cause caustics on diffuse objects like the ground, and it is also responsible for god rays.

The basic calculations for reflection and refraction were presented in Section 2.1. This section will focus on rendering techniques for

reflections and refractions. First a general approach using environment cube maps and projective textures for both reflections and refractions is presented (see Fig.19). Then it is shown how this approach can be implemented on today's graphics hardware. With use of the GPU for the reflection and reflection calculations rendering times in real time can be achieved. The last part of this section will focus on a GPU accelerated ray tracing approach specialized for water surfaces represented as height fields.

To render reflections we first need the reflected ray of an incoming ray. We further need the surface normal. See Section 2.1 for a detailed description of the calculations. The reflected ray is then used for a look up in a cube environment map. This works fine for non moving objects far away because this can be pre calculated and stored in an environment map. For near local moving objects a different method has to be used. For relatively flat water surfaces like ocean, ponds or pools a method presented by [Jensen and Goliáš 2001] can be used. It is based on the basic algorithm of reflections on a flat plane.

The reflection on a flat plane works as followed: First the scene without the reflection plane is rendered. Then the reflecting plane is rendered into the colour buffer and into an auxiliary buffer, like the stencil buffer. The depth buffer is set to the maximum value for the area covered by the plane. Then the whole scene is mirrored by this plane and rendered. Updates to the colour buffer are only done if the values of the corresponding auxiliary buffer positions were earlier set by the reflection plane. (See [Kilgard 1999] for details).

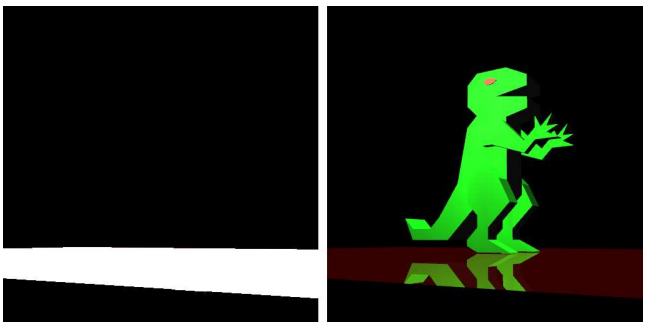

Figure 18: Left: Stencil buffer. Right: Rendering.

In contrast the algorithm for a water surface is slightly different. For simplifications the scene is not directly reflected by the water surface but by a plane placed at the average height of the water surface. Then the scene is rendered as seen from the water surface into a texture. With the use of projective textures the reflection could simply be rendered onto the water surface, but without taking the rays reflected by the water surface into consideration, which would result in false looking reflections. To improve this the assumption is made, that the whole scene lies on a plane slightly above the water surface (scene plane). Then the intersections with the rays reflected by the water surface and this scene plane are calculated. These intersection points on the scene plane are then fed into the computations for the projective texture.

During rendering to the texture the field of view of the camera has to be slightly higher than for the normal scene because the water surface can reflect more than a flat plane.

For refraction again a cube environment map can be used to render the global underwater environment. For local refractions [Jensen and Goliáš 2001] use a similar approach like above: The only difference is that the plane which intersects the the refracted rays is located beneath the water surface.

For refraction the colour of the water has also to be taken into account. In very deep water only refractions near the water surface should be rendered because of the light absorption of water. Even this shallow refractions must be attenuated with respect to the correct water colour and depth.

[Nishita and Nakamae 1994] describe light scattering and absorption under water. With certain simplifications (the water surface is a flat plane and no light scattering takes place) the water colour is only depending on the viewing angle and the water matter. The various colour values can then be pre calculated and stored into a cube map.

The light of an underwater object that reaches the surface above is absorbed exponentionally, depending on the depth and the properties of the water itself.

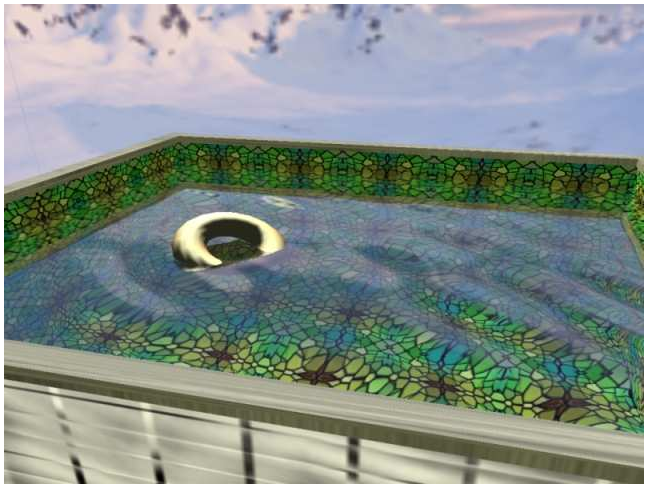

Figure 19: Reflection and Refraction using Environment Maps (Image courtesy of B. Goldlücke and M. A. Magnor).

An important part is the physically correct blending of the reflection and refraction. The Fresnel equation defines this weight for blending. Without correct blending the results are looking very plastic. The exact calculation of the Fresnel term is described in Sec. 2.1. The Fresnel term depends on the angle between the incoming light ray and the surface normal, and on the indices of refraction from Snell's law. In most cases the indices of refraction are constant, e.g. only refractions between air and water volumes are considered. Therefore the Fresnel term only depends on the angle of incident and can be pre calculated and stored for various angles. Another method to speed things up is to approximate the Fresnel equation. [Jensen and Goliáš 2001] have shown that using

$$f(\cos\theta) = \frac{1}{(1+\cos\theta)^8} \tag{27}$$

as approximation gives good results.

[Goldlücke and Magnor 2004] show how to implement reflections and refractions with cube maps on the GPU. The reflection and refraction rays are calculated per vertex in vertex programs. The resulting rays are stored as texture coordinates in separate texture units for later combining. The Fresnel term is also calculated using GPU instructions. They either use exact precomputed values of the Fresnel equation stored in another texture unit as an alpha texture or they us the approximated Fresnel equation given in Eq. 27 computed also on the GPU. Depth attenuation is calculated per vertex and stored in the primary alpha channel. The colour of the water is

stored in the primary colour channel. At last the primary colour is blended with the texture unit storing the cube map for the refraction with respect to the primary alpha channel. The result gets blended with the texture unit containing the cube map for reflection, with respect to the texture unit containing the alpha texture (The Fresnel term).

The results show that with this method a simulation can run at a resolution of 1024x786x32 in real time. The bottleneck is not the rendering of reflections and refractions but the simulation of the water surface, which involved FFTs for a 64x64 height field. An example of the described method is shown in Fig. 19.

4.2 Caustics

Reflections, refractions and the scattering of light inside the water volume cause the focusing of light known as caustics. Caustics are an example of indirect lighting effects and usually hard to render in real time.

As described earlier caustics can be rendered in a ray tracer using backward ray tracing and photon mapping. This section will focus on rendering caustics using only today's standard graphics primitives, assuming all rendered primitives are polygonal meshes. The following method is presented by [Jensen and Goliáš 2001].

To get results in real time certain constraints have to be set. Only first order rays are considered. That means if the reflected and refracted light rays hit an object no more outgoing light rays from that object are generated. It is further assumed that the surface the caustics are rendered on (i.e. the bottom of the ocean) is at constant depth.

First for each triangle of the water surface a ray from the light source to the vertices of the triangle is calculated (light beam). These light rays get refracted by the water surface using Snell's law (see Sec.2.1). These refracted light rays are then intersected with the xz-plane at a given depth. This xz-plane represents the bottom of the water volume. This way the surface triangles are projected onto the xz-plane, resulting in transformed may overlapping triangles. See Fig.20 for an example.

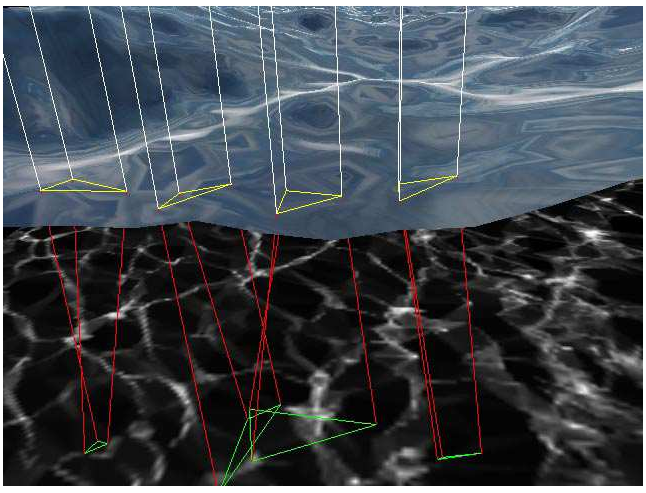

Figure 20: Refracted light beams and their corresponding triangles on the ocean ground (Image courtesy of L. S. Jensen and R. Goliáš).

The intensity of the resulting refracted light beams at the vertices of the triangles on the xz-plane can be calculated as followed

$$I_c = N_s \cdot L \left(\frac{a_s}{a_c} \right) \tag{28}$$

Where N is the normal of the water surface triangle and L is the vector from the surface triangle vertex to the light source, a_s is the area of the surface triangle and a_c is the area of the projected triangle.

The resulting triangles all lie in the xz-plane and can therefore be easily rasterized into a texture for further rendering. In order to achieve visually appealing results the texture has to be filtered to reduce aliasing artefacts. This can be done by four rendering passes of the same texture and perturbate the texture coordinates for each pass slightly, which yields to the effect of a 2x2 super sampling.

For applying the caustic texture on to under water objects it gets parallel projected from the height of the water surface in the direction of the light ray. Additionally the dot product between the surface normal and the inverted light ray can be used as intensity for the applied texture.

4.3 Godrays

Godrays are somewhat connected with caustics. Both visual phenomena are caused by the focusing and defocusing of the light ray while travelling through the water surface. The light rays are further scattered inside the water volume by small particles like plankton or dirt. This scattering makes the light rays visible causing light beams, also known as god rays. An example rendering with godrays is shown in Fig. 21. This section will present two different methods to handle godrays. Both approaches are closely related to caustic generation and light beams.

Figure 21: Sample scene rendered with Godrays (Image courtesy of L. S. Jensen and R. Goliáš).

For physically correct renderings of godrays all the light scattering and absorption effects would have to be considered and afterwards they could be rendered using volumetric rendering. In practice this is hardly achievable because of the amount of computations necessary for the correct light transport within the water volume, e.g. a light ray is scattered, causing new light rays which probably also have to be scattered, and so on.

With an already generated caustics texture is is relatively simple to simulate godrays with a volume rendering like algorithm. The

caustic texture already represents the intensity and shape of the resulting godrays, but only at a specific depth, namely the bottom of the water volume. The basic idea is to use this caustic texture as representation for the light intensity of the whole volume. Several slices are created depending on the position of the camera, with a high density distribution of the slices near the camera and a low distribution far away from it. A high density distribution near the camera is important to reduce visual artefacts. Finally the slices are rendered into the completed scene with adaptive alpha blending enabled. A further improvement in quality can be achieved if more slices are generated and rendered at once using multi texturing capabilities of graphics hardware. Fig. 21 shows an example rendered with the above described technique.

Another approach using light beams which are called illumination volumes were presented by [Iwasaki et al. 2002]. Their approach takes direct sunlight and skylight as ambient light into account. The resulting intensities are also viewpoint dependant. For their light equations the following model is used. The received light underwater at a certain viewpoint I_v from some point on the water surface (Q) is given by:

$$I_v(\lambda) = I_Q(\lambda)\, e^{-\alpha_\lambda \,\mathrm{dist}} + \int_0^{\mathrm{dist}} I_p(\lambda)\, e^{-\alpha_\lambda l}\, dl \qquad (29)$$

Where λ is the wavelength of light, I_Q is the light intensity just beneath the water surface and can be calculated using the fresnel equations, dist is the distance from Q to to viewpoint, $-\alpha_\lambda$ is the light attenuation coefficient within the water and I_p is the intensity of scattered light at a point between Q and the viewpoint. Therefore the integral term can be seen as scattered light contribution along the viewing ray.

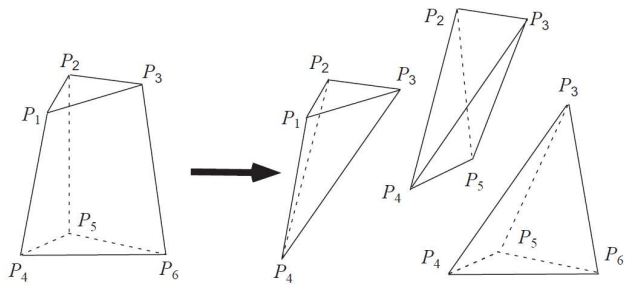

Figure 22: Illumination sub-volume and it's subdivided three tetrahedra (Image courtesy of K. Iwasaki et al. 2002).

[Iwasaki et al. 2002] use illumination volumes to approximate the calculations for the integral term in Eq. 29. They are later also used for rendering the light scattering effects. A illumination volume is created out of a water surface triangle and the refracted light rays at the corresponding triangle vertices (See Section 4.2). The front faces of the illumination volume could be used to render the godrays, but the intensities of each rendered pixel would have to be calculated per pixel, which is far to expensive. Therefore the illumination volume is further subdivided horizontally in sub-volumes. Due to the fact that light is absorbed exponentionally as it travels through water, the sizes of the illumination volumes are scaled exponentionally, with small volume sizes near the water surface. This way the intensities can be linearly approximated within each sub-volume, which can be done with hardware accelerated blending. With this simplification the scattered intensity of a given point I_p^s on a sub-volume can be calculated as followed:

$$I_p^s(\lambda) = \left(I_{\mathrm{sun}}(\lambda)\, T\, \frac{a_s}{a_c}\, \beta(\lambda,\phi)\, e^{-a_\lambda d} + I_a \right) \rho \qquad (30)$$

Where I_{sun} is the intensity of the sunlight on the water surface, T is the transmission coefficient from the fresnel equations, d is the distance the light had to travel under water, $\beta(\lambda,\phi)$ is the phase function, ρ the density, I_a the ambient light, a_s is the area of the original water surface triangle of the illumination volume and a_c is the area of current sub-volume triangle.

The sub-volumes itself are further divided into tetrahedra for rendering (see Fig. 22). For rendering it is important to weight the scattered intensities of the sub-volume vertices accordingly to the distance to the viewpoint. This is done by:

$$I_p(\lambda) = I_p^s(\lambda)\, e^{-a_\lambda \,\mathrm{dist}} \qquad (31)$$

Where dist is the distance between the point on the volume and the viewpoint. Finally the tetrahedra are rendered as followed: Each tetrahedra is intersected with the viewing ray on the thickest spot resulting in two intersection points A and B. The intensities for these points can be calculated by linearly interpolate the intensities of the tetrahedra vertices. The tetrahedra can be rendered as triangle fan with the intersection point nearest to the camera as centre vertex. The intensity for this centre vertex is calculated by $I_C = \frac{I_A + I_B}{2} \left|\overline{AB}\right|$. The intensities for the outer vertices are set to 0. It is important to activate adaptive blending to correctly accumulate the intensities of all illumination volumes.

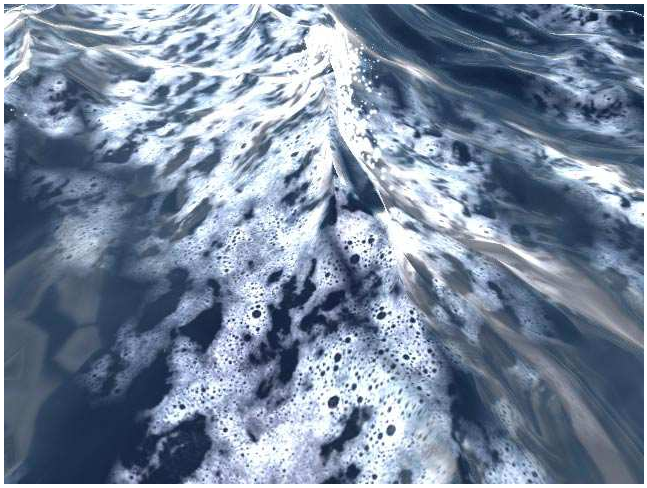

Figure 23: Foam rendered using an alpha blended texture (Image courtesy of L. S. Jensen and R. Goliáš).

4.4 Foam

Foam is caused on water surfaces by rough sea, obstacles in the water breaking waves or the wind blowing. The best way to render foam may be to use a particle system with a fixed position on top of the water surface. [Jensen and Goliáš 2001] take advantage of the fact that foam stays always on top of the water surface and render foam as additional transparent texture onto the water surface. For each vertex of the water surface a variable stores the amount of foam associated with that point of the water surface. This variable is then used as alpha value for the foam texture at that point in the rendering stage. The amount of foam depends on the difference between the y coordinate of the vertex and two neighbouring vertices in the x and z directions. If that difference is less than a user given negative limit, the amount of foam is increased a bit. Otherwise the amount of foam is reduced by small amount. This way foam is generated near wave tops. The amount of foam is not limited

to the range $[0\ldots1]$ as the transparency value is, e.g. the amount of foam can get > 1. It is very important to increase and decrease the amount of foam slowly because it would look very odd if foam suddenly pops up out of nowhere. An example rendered with this method is shown in Fig. 23.

Another method to calculate the amount of foam depending on wind speed is given by [Monahan and Mac Niocaill 1986]. They present an empirical formula which describes the fractional amount of a water surface covered by foam:

$$f = 1.59 * 10^{-5} U^{2.55} e^{0.0861(T_w - T_a)} \tag{32}$$

Where U is wind speed, T_w is water temperature and T_a is air temperature.

4.5 Bubbles

Bubbles are, like foam important water phenomena. [Thürey et al. 2007] present a bubble simulation method which is integrated within a shallow water framework, e.g. in a shallow water simulation only the water surface is being simulated. One of the key aspects of a bubble simulation is, that as a bubble emerges to the water surface, the water around it gets perturbated, which is hard to compute. In their approach each bubble is treated as spherical particle with position p_i, velocity u_i and volume m_i as parameters. These particles interact with each other and the water surface. The movement of each bubble is calculated by Euler steps and the flow around a bubble is simulated by a spherical vortex, which is a flow field describing the irrational flow around the bubble. The velocity and position of other bubbles can get perturbated by this vortex if they are close enough. Their simulation method further allows joining of bubbles with a distance smaller than the sum of their radii. If two bubbles are to be joined, they are dropped and a new bubble is created with the following new parameters (assuming the indici i and j stand for the joining bubbles):

$$m_n = m_i + m_j \tag{33}$$

$$u_n = \frac{u_i \cdot m_i}{m_n} + \frac{u_j \cdot m_j}{m_n} \tag{34}$$

$$r_n = \frac{3}{4 \cdot \sqrt[3]{m_n}} \tag{35}$$

If a bubble reaches the surface it is turned into foam by a certain probability. If not, the bubble just vanishes and a surface wave is propagated from the last position of the bubble. Bubbles can be rendered as spheres or with any of the rendering methods for particle systems mentioned in Section 2.3. Fig. 24 shows an example rendering of the method described.

4.6 Splashes

If water collides with obstacles, or objects are thrown into the water, water splashes are produced. The best way to handle such situations is to integrate a rigid body simulation (for the correct physical behaviour of rigid objects) into a 3D water simulation. This way water splashes would be automatically generated during simulation. If that's not the case, e.g. a full 3D water simulation is too cost expensive, water splashes can also be faked using particle systems. [Jensen and Goliáš 2001] only simulate the water surface and use particle systems for splashes. The velocity for new splash particles is directly taken by the velocity of the water surface. During its lifetime each particle is subject to external forces like gravity, wind or other external forces. A sample rendering with the described particle approach is shown in Fig. 25.

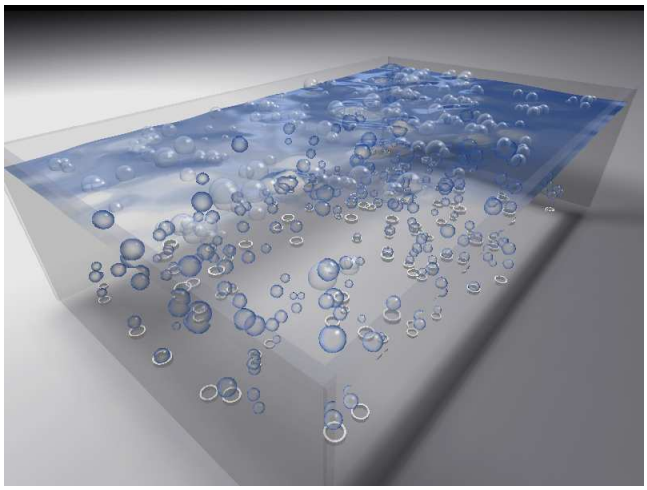

Figure 24: Bubble and Foam simulation within a shallow water framework (Image courtesy of N. Thürey et al.).

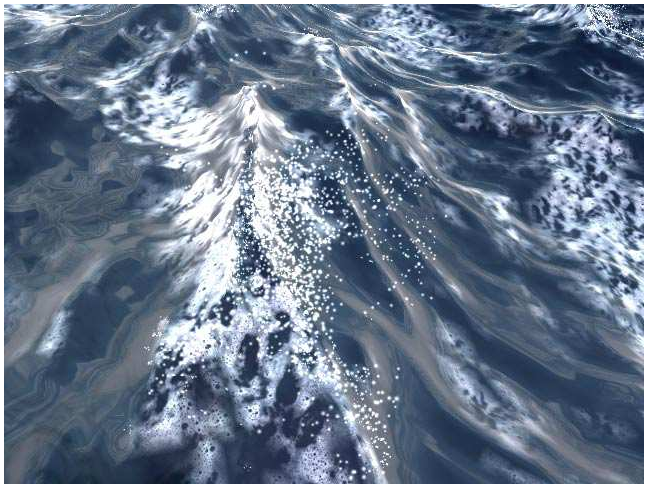

Figure 25: Rendering of foam and splashes (Image courtesy of L. S. Jensen and R. Goliáš).

5 Conclusion

This paper covered the main aspects needed for realistic water rendering. The basic data structures were presented as well as rendering algorithms and techniques to increase visual detail. It was shown that with current graphics hardware realistic looking water can be rendered in real time, even with such complex optical phenomena like reflection and refraction. Though these effects can only be achieved with simplifications they look very convincing and further hardware developments may change this rapidly.

An interesting development is the increased use of ray tracing methods in current real time applications. Even if the presented GPU ray tracing method is again restricted to certain simplifications upcoming approaches may circumvent these restrictions resulting in either dropping some restrictions, qualitative better renderings or both.

References

BABOUD, L., AND DÉCORET, X. 2006. Realistic water volumes in real-time. In *Eurographics Workshop on Natural Phenomena*.

BRIDSON, R., AND MÜLLER-FISCHER, M. 2007. Fluid simulation: Siggraph 2007 course notes. In *Proceedings of the 2007 ACM SIGGRAPH*, 1–81.

GOLDLÜCKE, B., AND MAGNOR, M. A. 2004. A vertex program for interactive rendering of realistic shallow water. Tech. rep., Max-Planck-Institut für Informatik.

HARLOW, F. H., AND WELCH, J. E. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids 8*, 12, 2182–2189.

IGLESIAS, A. 2004/11/1. Computer graphics for water modeling and rendering: a survey. *Future Generation Computer Systems 20*, 8, 1355–1374.

IWASAKI, K., DOBASHI, Y., AND NISHITA, T. 2002. An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. *Computer Graphics Forum 21*, 4, 701–711.

JENSEN, L. S., AND GOLIÁŠ, R. 2001. Deep-water animation and rendering. *Gamasutra*.

KILGARD, M. J. 1999. Improving shadows and reflections via the stencil buffer. Tech. rep., NVIDIA Corporation.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 1987 ACM SIGGRAPH*, 163–169.

MONAHAN, E., AND MAC NIOCAILL, G. 1986. *Oceanic Whitecaps and Their Role in Air-Sea Exchange Processes*. Springer.

MÜLLER, M., SCHIRM, S., AND DUTHALER, S. 2007. Screen space meshes. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 9–15.

NIELSON, G. M., AND HAMANN, B. 1991. The asymptotic decider: resolving the ambiguity in marching cubes. In *Visualization, 1991. Visualization '91, Proceedings., IEEE Conference on*, 83–91, 413.

NISHITA, T., AND NAKAMAE, E. 1994. Method of displaying optical effects within water using accumulation buffer. In *Proceedings of the 1994 ACM SIGGRAPH*, 373–379.

PREMŐZE, S., AND ASHIKHMIN, M. 2001. Rendering natural waters. *Computer Graphics Forum 20*, 4, 189–199.

REEVES, W. T. 1983. Particle systems - a technique for modeling a class of fuzzy objects. *ACM Trans. Graph. 2*, 2, 91–108.

SCHUSTER, R. 2007. Algorithms and data structures of fluids in computer graphics. Unpublished State of the Art Report.

TESSENDORF, J. 1999. Simulating ocean water. *Proceedings of the 1999 ACM SIGGRAPH 2*.

THÜREY, N., SADLO, F., SCHIRM, S., MÜLLER-FISCHER, M., AND GROSS, M. 2007. Real-time simulations of bubbles and foam within a shallow water framework. *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 191–198.