Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master in Computing

Master of Science Thesis

FLUIDS REAL-TIME RENDERING

Xavier de la Fuente Caballé

Advisor: Antonio Susín Sánchez

January 2011

Contents

1 Introduction

Rendering of a fluid in real-time computer graphics is highly dependent on the demands on realism. In the beginning of real-time graphics most applications treated fluid surfaces (normally water) as strictly planar surfaces which had artist generated textures applied to them. The fluid in these applications did not look very realistic, but neither did the rest of the graphics so it was not particularly problematic. But as the appearance of the applications increase with the available processing power and better computer graphic techniques, the look of the fluid is becoming increasingly important.

Fluid rendering differ significantly from rendering of most other objects. Fluids themselves have several properties that we have to consider when rendering them in a realistic manner:

- They are dynamic objects. Unless the fluid is supposed to be totally at rest it will have to be updated each frame. Wave interactions that give the surface its shape are immensely complex and probably have to be approximated.
- The surface can be huge. When rendering a large fluid body such as an ocean it will often span all the way to the horizon.
- Its look is to a large extent based on reflected and refracted light. The ratio between the reflected and refracted light vary depending on the angle the surface viewed at.

It is necessary to limit the realism of fluid rendering in order to make a real-time implementation feasible. In most applications it is common to limit the span of the surface to small lakes and ponds. Wave interactions are also immensely simplified. As mentioned in the introductory example, the reflections and refractions are usually ignored, the surface is just rendered with a texture that is supposed to look like water. Neither is it uncommon that water still is treated as a planar surface without any height differences whatsoever. But the state of computer generated fluids is rapidly improving

and although most aspects of the rendering still are crude approximations they are better looking approximations than the ones they replace.

1.1 Project overview

In this thesis the existing methods for realistic visualization of fluids in real-time are reviewed. The correct handling of the interaction of light with a fluid surface can highly increase the realism of the rendering, therefore method for physically accurate rendering of reflections and refractions will be used. The light-fluid interaction does not stop at the surface, but continues inside the fluid volume, causing caustics and beams of light. The simulation of fluids require extremely time-consuming processes to achieve physical accuracy and will not be explored, although the main concepts will be given.

Therefore, the main goals of this work are:

- Study and review the existing methods for rendering fluids in real-time.
- Find a simplified physical model of light interaction, because a complete physically correct model would not achieve real-time.
- Develop an application that uses the found methods and the light interaction model.

1.2 Organization of this document

This master thesis is organized as follows:

- Chapter 2 introduces the basics of optics and simulation algorithms that are relevant for physically-based computer graphics research.
- Chapter 3 introduces the existing rendering techniques that can be used for fluids and explains the analytical version of the light interaction model.
- Chapter 4 focuses on the implementation details using OpenGL. It also explains a developed method to distinguish between inside and outside fluid rendering.
- Chapter 5 presents the results of this work and summarizes the conclusions.

2 Physics background

2.1 Physics of light

Light is the portion of electromagnetic radiation that is visible to humans. Here we discuss the interaction of light and matter, however, complete electromagnetic descriptions of light are often difficult to apply in practice. Practical optics is usually done using simplified models. The most common of these, treats light as a collection of rays that travel in straight lines and bend when they pass through or reflect from surfaces.

2.1.1 Reflection and refraction

If a light ray strikes the surface of the fluid, it is split into reflected and transmitted (refracted) rays. The intensity of each of these two rays is diminished by reflectivity and transmissivity coefficients. Here we discuss the directions of the two outgoing rays, in the next subsection the coefficients are discussed.

The equation for reflection is well known. We can see on figure 2.1 the geometry of a reflected ray. For an eye vector I (i.e. the ray from the eye to the given point) and the surface normal N, the reflected ray is

$$R = 2(I \cdot N)N - I \tag{2.1}$$

As seen on figure 2.2, the direction of the refracted ray is given by Snell's law

$$n_1 \sin\theta_i = n_2 \sin\theta_t \tag{2.2}$$

where θ_i and θ_t are angles with the facet normal for incident and transmitted rays, respectively and n_1, n_2 are real indices of refraction for the corresponding media. Snell's law shows that for a sufficiently oblique ray going from water to air it is possible to have total internal reflection when only reflected ray is present.

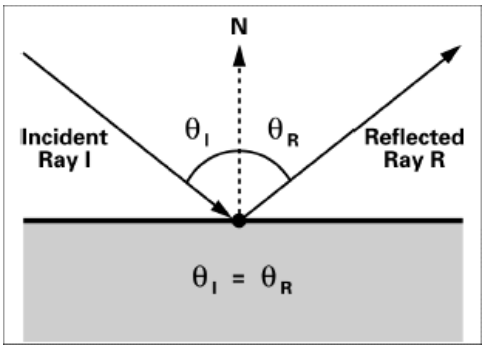

Figure 2.1: Calculating reflected ray

2.1.2 Fresnel reflectivity and transmissivity

The efficiency of the reflection and refraction through the surface is described by the reflectivity R and the transmissivity T, which relation is constrained by the law of conservation of energy

$$R + T = 1.$$

The derivation of the expressions for R and T is based on the electromagnetic theory of dielectrics. From [Glassner, 1994] we have two basic solutions

$$R_s = \left[\frac{n_i \cos\theta_i - n_t \sqrt{1 - (\frac{n_i}{n_t} \sin\theta_i)^2}}{n_i \cos\theta_i + n_t \sqrt{1 - (\frac{n_i}{n_t} \sin\theta_i)^2}} \right]^2$$

$$R_p = \left[\frac{n_i \sqrt{1 - (\frac{n_i}{n_t} \sin\theta_i)^2} - n_t \cos\theta_i}{n_i \sqrt{1 - (\frac{n_i}{n_t} \sin\theta_i)^2} + n_t \cos\theta_i} \right]^2$$

Assuming that the light is not polarized we get $R = \frac{R_s + R_p}{2}$. We can see on figure 2.3 a plot of the reflectivity for rays of light traveling down onto a water surface as a function of the angle of incidence to the surface, and how it changes if the incident ray comes from below the surface. In this case, reflectivity increases rapidly, so there is no transmissivity of light through the surface.

2.2 Optical properties of fluids

To generate realistic images of fluids one must consider in some detail the interaction of light with the fluid body, or participating media.

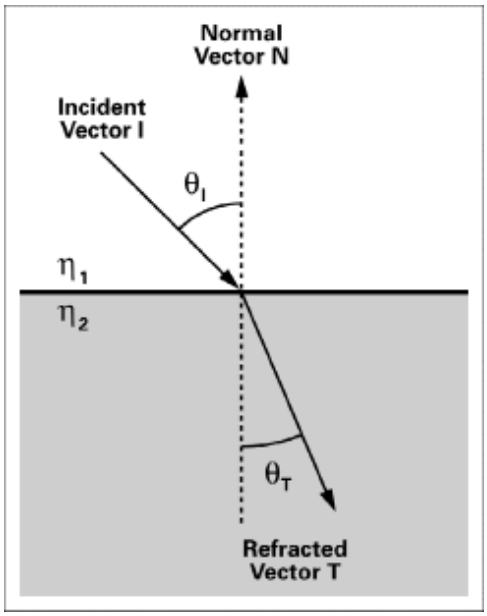

Figure 2.2: Snell's law

2.2.1 Participating media

Participating media are those media which alter in some way the electromagnetic radiation that traverses them. The propagation of light in participating media is described by the radiative transport equation (RTE) [Chandrasekhar, 1960]

$$\frac{dL(\vec{\omega_o})}{dl} = L_e(\vec{\omega_o}) - (\sigma_a + \sigma_s)L(\vec{\omega_o}) + \sigma_s \int_{4\pi} p(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i}) d\vec{\omega_i} \qquad (2.3)$$

which shows that radiance L gets affected at every differential point along the optical path l. $\int_{4\pi}$ refers to the integral of all the differential directions along a sphere. There can be light emission ($L_e(\vec{\omega_o})$), extinction due to absorption or out-scattering ($-(\sigma_a+\sigma_s)L(\vec{\omega_o})$), or radiance increase coming from in-scattering ($\sigma_s \int_{4\pi} p(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i})$). The absorption and scattering coefficients (σ_a and σ_s, respectively) are often added up to another coefficient named *extinction coefficient* $\sigma_k = \sigma_a + \sigma_s$. The phase function p specifies the normalized distribution of the scattered light, taking the form $p = \frac{1}{4\pi}$ if the medium is isotropic. It can be defined as

$$p(\vec{\omega_i}, \vec{\omega_o}) = \frac{dL_o(\vec{\omega_o})}{dL_i(\vec{\omega_i})}. \qquad (2.4)$$

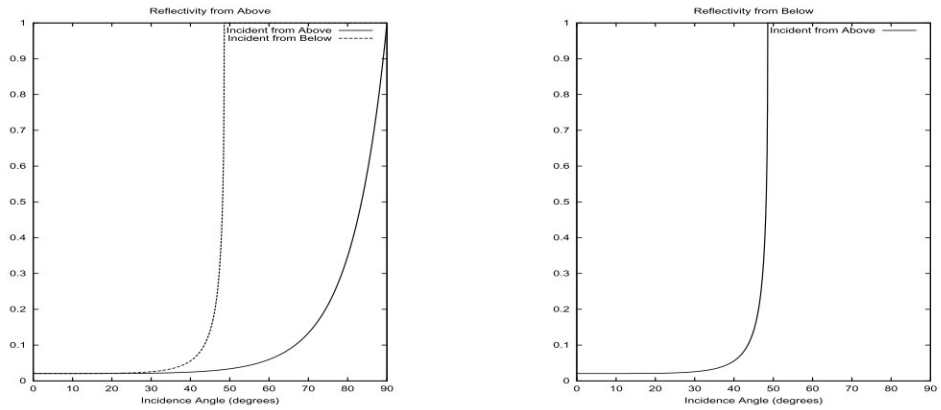

Figure 2.3: Reflectivity for light coming from the air down to water surface and from below the surface, respectively. From [Tessendorf, 1999]

2.2.2 Absorption

When light traverses a medium with some conductivity then the electromagnetic energy is taken up by the medium and transformed into other forms of energy, such as heat [Born and Wolf, 1999, Bohren and Huffman, 1983]. As light traverses the absorbing medium, its irradiance E gets diminished at every differential point along the path l

$$\frac{dE}{dl} = -\sigma_a E \tag{2.5}$$

The absorption coefficient is related to the attenuation index κ:

$$\sigma_a = \frac{4\pi}{\lambda}\kappa \tag{2.6}$$

The attenuation index is related to the amount of power absorbed by a medium as light traverses it. It depends on wavelength, so the absorption at some wavelengths can be different from other ones. If σ_a is constant through the medium, the attenuation of an irradiance E_0 for a path d becomes

$$E = E_0 e^{d\sigma_a} \tag{2.7}$$

which means that intensity decays by a factor of e^{-1} when the light has traveled a distance $\sigma_a^{-1} = \lambda(4\pi\kappa)^{-1}$.

2.2.3 Scattering

Scattering in fluids is caused by interactions of light at molecular level and with particles [Mobley, 1994]. It can be classified in two broad categories: *elastic* and *inelastic* scattering, depending on whether the scattered

photon maintains or changes its energy in the process. The inelastic scattering events can be further sub-classified according to the nature of the energy transfer: Stokes scattering, when a molecule of the medium absorbs the photon and re-emits it with a lower energy, and anti-Stokes scattering, when the re-emitted photon has a higher energy. The process implies an energy transfer from wavelength λ' to λ, with λ' being the excitation wavelength and λ the re-emitted wavelength.

Depending on the point of view, scattering can be classified into:

- *Out-scattering* From one side, light that is traversing a specific path is scattered.
- *In-scattering* From the other side, light that is traversing a specific path receives the radiance scattered from other parts of light.

Out-scattering is expressed as follows:

$$\frac{dL}{dl} = -\sigma_s L \tag{2.8}$$

The outgoing radiance from in-scattering is

$$\frac{dL_o(\vec{\omega_o})}{dl} = \sigma_s \int_{4\pi} p(\vec{\omega_i}, \vec{\omega_o}) L_i(\vec{\omega_i}) d\vec{\omega_i} \tag{2.9}$$

The scattering coefficient can depend on the wavelength. If the incoming radiance comes from a light source, is often called *single scattering*, while if it comes from a previous interaction in the medium is called *multiple scattering*.

2.3 Fluid simulation

For simulation of fluids sophisticated physical models are needed. As stated above this work will not cover the simulation part, but this section will cover the basic approaches which are used by the most common simulation methods. Here are briefly presented only two methods to simulate the animation of fluids, but those who want to go deeper into the subject can get more information on [Foster and Fedkiw, 2001, Stam, 1999, Foster and Metaxas, 1997].

2.3.1 Fast Fourier Transform

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse, i.e. Fourier transformation that samples the input at regularly placed points [Mastin et al., 1987]. The FFT is used to compute model which is not based on any physics models, but instead uses statistical models based on observations of the real sea. In this

statistical model of the sea, wave height is a random variable of horizontal position and time, h(X,t). It decomposes the wave height field into a set of sinus waves with different amplitudes and phases. FFT allows us to quickly evaluate the following sum:

$$h(X,t) = \sum_K \tilde{h}(K,t)e^{iK\cdot X} \tag{2.10}$$

where K is a 2D vector with components (k_x, k_y), $k_x = 2\pi n/L_x$, $k_y = 2\pi m/L_y$ and n and m are integers with bounds $-N/2 \leq n < N/2$ and $-M/2 \leq m < M/2$. The FFT will generate a height field at discrete points $x = (nL_x/N, mL_y/M)$ which will be used to construct the ocean surface.

2.3.2 Navier-Stokes equations

The Navier-Stokes equations are the fundamental partial differentials equations that describe the flow of incompressible fluids and consist of two parts [Batchelor, 2000]. The first, enforces incompressibility by saying that mass should always be conserved,

$$\nabla \cdot \vec{u} = 0 \tag{2.11}$$

where u is the liquid velocity field, and

$$\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$$

is the gradient operator. The second equation couples the velocity and pressure fields and relates them through the conservation of momentum,

$$\frac{\partial \vec{u}}{\partial t} + \vec{u}\cdot\nabla\vec{u} + \frac{1}{\rho}\nabla p = \vec{g} + \nu\nabla\cdot\nabla\vec{u}. \tag{2.12}$$

This equation models the changes in the velocity field over time due to the effects of viscosity (ν), density (ρ), pressure (p) and gravity (g). By solving 2.11 and 2.12 over time, we can model the behavior of a volume of liquid, which leads to a physically accurate but computationally intensive simulation. After that, we have a volume model which has to be rendered in a different way than the FFT approach, such as marching cubes [Lorensen and Cline, 1987].

3 Rendering techniques

Realistic looking fluids can be achieved by using special rendering techniques. This section surveys global illumination algorithms for environments including participating media.

The generation of physically accurate images of participating media is an extremely challenging computational problem. Some difficulties arise when treating light propagation in this type of environments. For example, interaction phenomena take place not only in the medium boundaries but within any point of the medium. Therefore, optical properties in each point of the medium have to be known, and not only radiances on surfaces, but source radiances throughout the space have to be computed. Phenomena like fluorescence or phosphorescence, which imply a transfer of energy from one wavelength to another are not significant in the wavelengths corresponding to the visible range of the light spectrum, so the emission of light is not considered.

This chapter is divided into five sections, the first explains the most used methods to render surfaces. The second section explains the approximation to the physical fresnel term. The third section gets into absorption and scattering models, and simplifying them to achieve real-time. The fourth and fifth sections describe rendering methods to generate surface and volume caustics, respectively.

3.1 Surface rendering

Many rendering techniques have been researched to obtain a final image. Nowadays, three families of surface rendering techniques coexist:

- Rasterization, which geometrically projects objects in the scene to an image plane.
- Ray casting, which considers the scene as observed from a specific point-of-view, calculating the observed image based only on geometry and very basic optical laws of reflection.

- Ray tracing, which is similar to ray casting, but employs more advanced optical simulation, and usually uses Monte Carlo techniques to obtain more realistic results at a speed that is often orders of magnitude slower.

In this thesis we will consider only rasterization algorithms, in order to get real-time, but in the next section a brief description of a ray tracing approach is given. After that, the main rasterization approaches are presented.

3.1.1 Ray tracing

Ray tracing [Whitted, 1980] is a general rendering technique in computer graphics. It is considered as a global illumination technique because addresses all the type of light interactions that occur in a real world scenario [Adabala and Manohar, 2002].

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. For calculation on the computer this approach is too expensive to achieve real-time, because it is hardly achievable to calculate all the necessary light rays. 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. However, a real time water volume ray tracer with strict limitations was presented by [Baboud and Décoret, 2006]. It is a field still in research, as seen in the work of [Jiménez et al., 2005, Gutierrez et al., 2008].

3.1.2 Environment mapping

Environment mapping [Blinn and Newell, 1976] is an old technique that that simulates the results of ray-tracing. Because environment mapping is performed using texture mapping hardware, it can obtain global reflection and lighting results in real-time. Essentially, it consists of pre-computing a texture map of the distant environment surrounding and then sampling texels from this texture during the rendering of a model. With this method it is possible to map any kind of image onto any type of geometry. Fig. 3.1

shows an unfolded cube map, which is the most extended form of environment mapping.

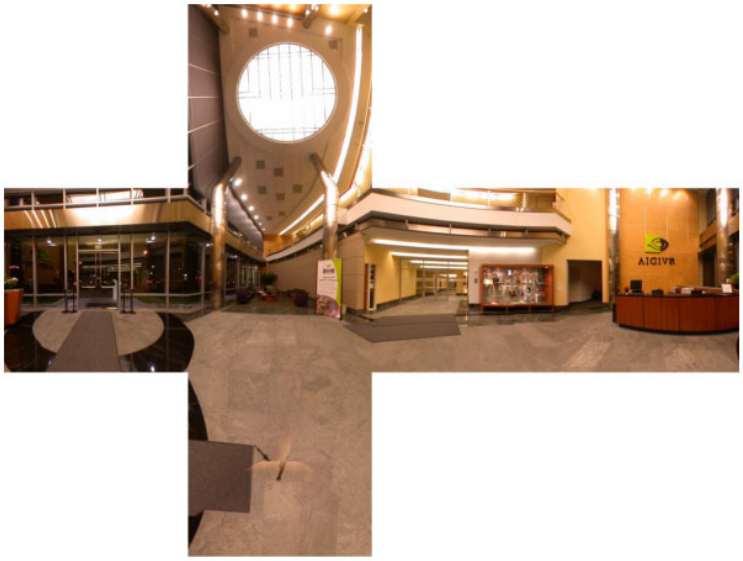

Figure 3.1: Unfolded cubemap texture [NVIDIA, 2010]

The environment of a reflective object is rendered from a point, called reference point, that is in the vicinity of the reflector. Figure 3.2 illustrates in 2D an object, an eye position, and a cube map texture that captures the environment surrounding the object. The incident ray I goes from the eye to the object's surface. When I reaches the surface, it is reflected in the direction R based on the surface normal N. This second ray is the reflected ray. During the rendering of the reflective object, the radiance of the reflected ray is looked up from the cubemap texture using only the 3D direction vector of the ray but ignoring its origin and neglecting self reflections. In other words, environment mapping assumes that the reflected points are very (infinitely) far, and thus the hit points of the rays become independent of the ray origin. This makes environment mapping to be not view dependent, that means that the view point does not influence the way the texture is mapped to the surface, so it's not very suitable for local reflections.

While global reflections are a good representation for objects that are part of the environment, they are not a good representation for object within the scene. Local reflections are defined as reflections from objects that cannot be considered to be infinitely far away. Put differently, reflections where the position on the reflecting surface in fact does matter, and the reflection are not only dependent on the angle from where it reflects. A cube map is capable of representing every possible angle of incoming light in a sin-

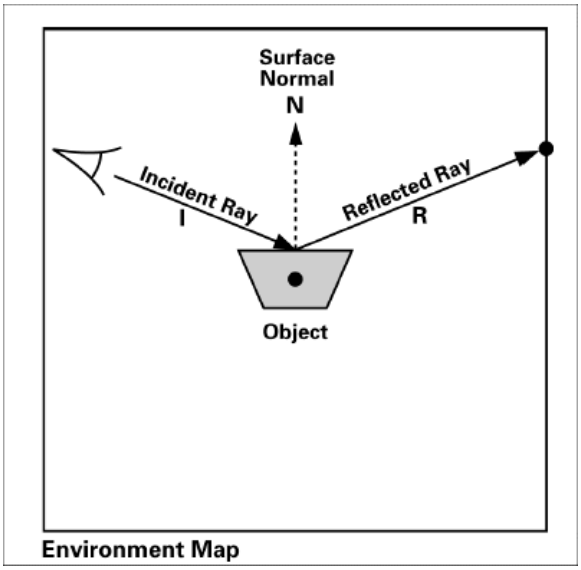

Figure 3.2: Environment mapping

gle point, but the reflection across an entire surface cannot be represented correctly using it.

[Brennan, 2002] describes a way to overcome this problem by adjusting the use of the environment map such that it has an finite radius that is close to the real proximity of the objects. The most precise way of doing this is by intersecting the reflection or refraction ray with a proxy geometry of the environment map (e.g. a sphere or a cube), and then use the vector from the center of the map to the intersection point to index the map, as seen on figure 3.3. For a fixed and simple proxy geometry, the ray intersection calculation can be executed by the GPU. However, the assumption of a simple and constant environment geometry creates visible artifacts that make the proxy geometry apparent during animation.

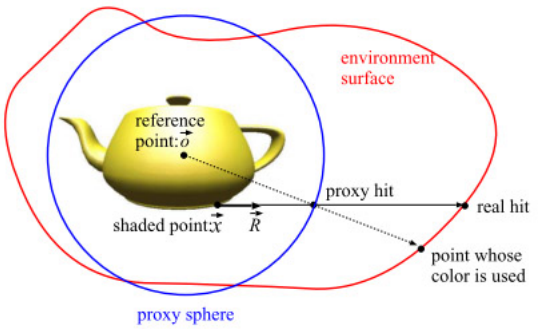

Figure 3.3: Use of a proxy sphere, from [Szirmay-Kalos et al., 2009]

To reflect a dynamically changing environment, the environment map texture must be regenerated each time the environment changes. A dynamic cube map texture is generated by first render the scene six times from the point of view of the reflective object. Each view is a different orthogonal 90 degree view frustum corresponding to one of the six faces of the cube map.

3.1.3 Projective texturing mapping

To allow for local reflections on our scene, the most common used approach is projective texture mapping. It is a technique for generating texture coordinates dynamically via a projection of 3D geometry into a texture map. In the same way that screen coordinates are generated by projecting 3D geometry onto your 2D screen, 3D geometry can be projected onto a texture map. The basic algorithm generates reflections for flat surfaces ([Kilgard, 2001]), but for a water surface is better to use the modification done by [Jensen and Goliáš, 2001].

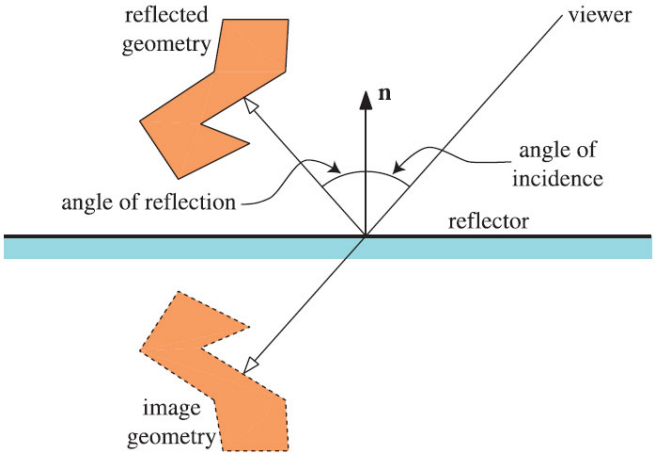

Figure 3.4: Reflection in a plane, showing the reflected geometry and its reflection. Source: [Akenine-Möller et al., 2008]

Figure 3.4 shows that the flat surface reflection acts like a mirror, where the reflected image of the object is simply the object itself, physically reflected through the plane. To render a scene, the objects to be reflected have to be drawn first, but mirrored along the desired plane. Objects that are behind the reflector plane should not be reflected, and thus a user-defined clipping plane is placed so it coincides with the reflecting plane. Then, the scene is rendered to a texture from the cameras point-of-view. When rendering the mirror plane, the texture can be used to provide the reflection by using a texture lookup with the actual screen coordinates as texture coordinates. Instead, we add an offset to the texture coordinates that is based on the intersection of the reflected ray with a plane positioned slightly above

the surface. This gives a more realistic result, but as the surface can reflect more of the scene than a flat mirror, there are parts which the reflected color is unknown. Typically, when rendering to the scene, the camera's field-of-view is set slightly higher than for the normal camera, but this only covers the image's margins. Go to section 4.4 to see a more detailed explanation of this method, as it has been implemeted.

3.2 Fresnel approximation

The computation of the exact Fresnel term is quite expensive even on the graphics hardware. In real time applications, we need its approximation, which is much cheaper to evaluate, but is accurate enough not to destroy image quality.

[Schlick, 1994] gives an approximation of Fresnel reflectance that is fairly accurate for most substances:

$$R_F(\theta_i) = R_F(0) + (1 - R_F(0))(1 - \cos\theta_i)^5 \tag{3.1}$$

This equation gives reasonably accurate results for most materials, as seen in figure 3.5, with a cost of 5 multiplications. With this approximation, $R_F(0)$ is the only parameter that controls the reflectance. $R_F(0)$ is available for many real-world materials and its value varies between 0 and 1. It is common to assume the refractive index n_1 to be 1, as is the index of air. Using n instead of n_2 gives the following equation:

$$R_F(0) = \left(\frac{n-1}{n+1}\right)^2 \tag{3.2}$$

Other approximations like [Lazányi and Szirmay-Kalos, 2005] give better results for certain materials, but using a more complex formula which requires more computational cost.

Although external reflection is the most commonly encountered case, internal reflection is important as well. Total internal reflection can occur when a light ray traveling in a transparent material encounters an interface with another transparent, but less optically dense material, that is $n_1 > n_2$ in figure 3.6. The Schlick approximation form equation 3.1 is correct for external reflection, and it can be used for internal reflection if the transmission angle θ_t is substituted for θ_i.

3.3 Modeling absorption and scattering

For the spectral absorption and scattering functions of fluids we will be using ocean water models [Preisendorfer, 1976, Smith and Baker, 1981, Mobley, 1994]. Apart from the water molecules themselves, natural water

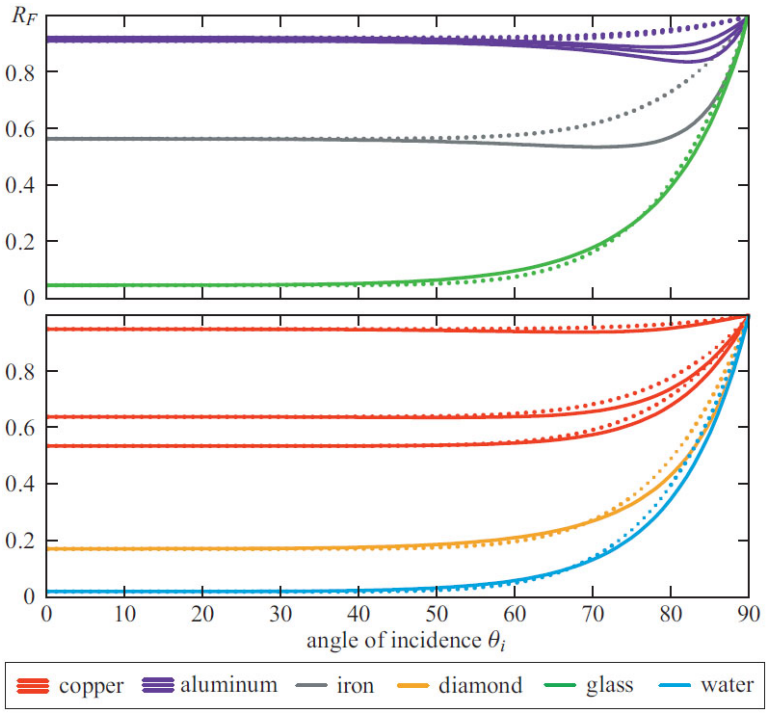

Figure 3.5: Schlick's approximation of Fresnel reflectance compared to the correct values for external reflection from a variety of substances. The solid lines were computed from the full Fresnel equations and the dotted lines are the result of Schlick's approximation. Source: [Akenine-Möller et al., 2008]

is made up of a high variety of optically influent constituents which can determine the optical properties of water. For fluid compatibility, we will only consider inherent optical properties, which depend only on the constituents of water.

To proceed analytically it is assumed that the medium is throughout homogeneous, in other words $\sigma_k = \text{const}$.

Single scattering happens when radiation is only scattered by one localized scattering center. It is very common that scattering centers are grouped together, and in those cases the radiation may scatter many times, which is known as multiple scattering. Single scattering represents a high simplification with respect to multiple scattering. When the participating medium is optically thin (i.e. the transmittance through the entire medium is nearly one) or has low albedo, then the source radiance can be simplified to ignore multiple scattering within the medium. The albeldo can be written

$$\Lambda = \frac{\sigma_s}{\sigma_k}.$$

As shown on figure 3.7, assuming single scattering, the ray with direction

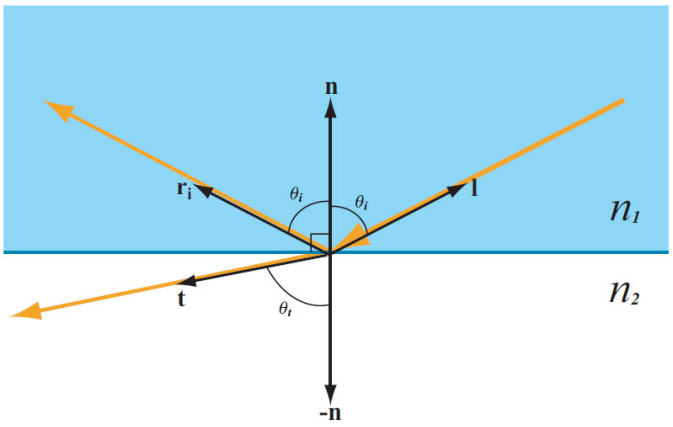

Figure 3.6: Internal reflection. Source: [Akenine-Möller et al., 2008]

$\vec{\omega}$ through x the media only receives light from the refracted path defined by $\vec{\omega_p}$ to the light source, then the integral over $\Omega_{4\pi}$ collapses to the single direction $\vec{\omega_p}$. Then, the light reaching the viewpoint from the fluid is calculated using the following equation

$$L(\omega) = \int_0^s e^{(-x\sigma_k)}\sigma_s(x)p(x,\vec{\omega},\vec{\omega_p})L_i(x,\vec{\omega_p})dx \tag{3.3}$$

Also, the incoming radiance L_i coming from the direction $\vec{\omega_p}$ through the homogeneous media can be evaluated as:

$$L_i(x,\vec{\omega_p}) = L_o e^{-k(x)\sigma_s}. \tag{3.4}$$

where $k(x)$ is the distance from $-x\vec{\omega}$ to the surface of the media in the direction $\vec{\omega_p}$. L_o is the amount of radiance coming directly from the light source inside the fluid. At each position x in the media the distance to the surface toward the point light k will change accordingly. If the media is refractive the direction $\vec{w}_{pl}$ is the refracted direction from the light source, in which we have

$$k = \frac{\sin\alpha}{\sin\beta}x = cx \tag{3.5}$$

where c is a variable that depends on the index of refraction. If the normal at the intersection of the scattered light ray is assumed to be constant the relation c will be a constant.

If the light source is positioned far away from the ray which is scattering the light backwards to the eye, it is possible to assume that the direction toward the light source can be seen as independent of x, thereby having a constant direction $\vec{\omega_p}(x) = \vec{\omega_p}$. This assumption is valid as long as the length of integration is short compared to the distance to the light, which

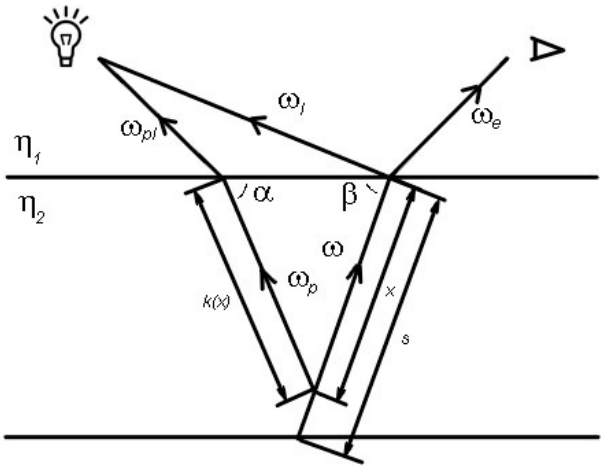

Figure 3.7: Calculation of light intensity reaching viewpoint outside the fluid

is the case of the sun. When the light source is positioned far away, it is also possible to assume that the radiance coming from the light source is constant along the eye ray $L_i = \text{const}$.

With all these simplifications and assumptions, according to [Gath, 2008], the equation 3.3 becomes

$$\begin{aligned} L(\vec{\omega_i}) &= -\sigma_k L(\vec{\omega}) + \sigma_s \int_{4\pi} p(\vec{\omega_p}, \vec{\omega}) L_i(\vec{\omega_p}) dx \\ &= -\sigma_k L(\vec{\omega}) + \sigma_s p(\vec{\omega}) L_i \int_0^s e^{-\sigma_k x(1+c)s} dx \\ &= -\sigma_k L(\vec{\omega}) + \Lambda \frac{p(\vec{\omega})}{1+c}(1 - e^{-\sigma_k(1+c)s} L_i) \end{aligned} \tag{3.6}$$

In the case where the viewpoint is inside the fluid, the equation comes from the situation depicted on figure 3.8, which becomes

$$L(\omega) = \int_0^s e^{(-x\sigma_k)} \sigma_s(x) p(x, \vec{\omega}, \vec{\omega_p}) L_i(x, \vec{\omega_p}) dx + L_o e^{-z\sigma_k} \tag{3.7}$$

This expression is very close to equation 3.3, so it can be simplified under the same assumptions, becoming

$$L(\vec{\omega_i}) = -\sigma_k L(\vec{\omega_o}) + \Lambda \frac{p(\vec{\omega})}{1+c}(1 - e^{-\sigma_k(1+c)s} L_i) + L_o e^{-z\sigma_k} \tag{3.8}$$

If the light is near the viewer we cannot assume that $\vec{\omega_i}(x) = \vec{\omega_i}$, and thus the integral along s cannot be avoided. However, a real-time rendering

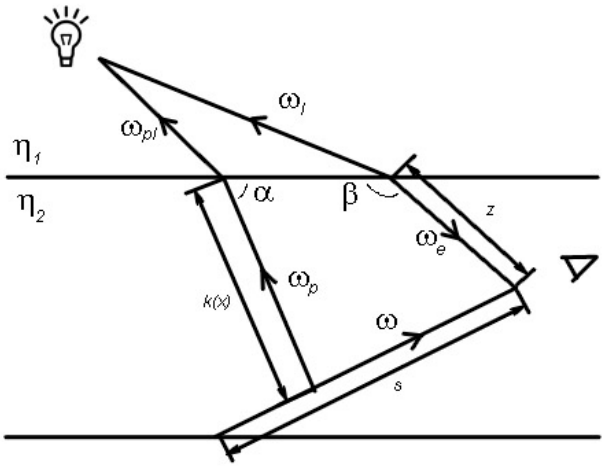

Figure 3.8: Calculation of light intensity reaching viewpoint inside the fluid

solution is still possible through work by [Sun et al., 2005], which describes the incoming radiance when the eye and light source both are present in the same media. To achieve it, they derive an expression by making some assumptions - isotropic light source, homogeneous media, single scattering and no volumetric shadows. They arrive to a solution that has no analytical expression, but which can be stored as a 2D table as it is independent of the physical parameters of the problem. Later, this work was expended in [Wyman and Ramsey, 2008] to include volumetric shadows.

3.4 Surface Caustics

Caustics show up as beautiful patterns on diffuse surfaces, formed by light paths originating at light sources and visiting mirrors or refracting surfaces. Caustics are the concentration of light, which can "burn". The name caustic, in fact, derived from Greek "kaustiko", which means "burning". 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. A more formal definition is that whenever multiple rays of light converge on the same point on a surface, they cause that region to become relatively brighter than its surrounding regions.

There are two major classes of caustic rendering techniques: image space and object space. [Ernst et al., 2005] was one of the first object space algorithms, it used the idea of a warped caustic volume. Each of the objects of the scene is tagged as caustic generator, and/or as a caustic receiver. Caustics are computed in two passes. First, the receivers are rendered to a texture

Figure 3.9: Surface caustics from real world due to reflection and refraction. Source [Akenine-Möller et al., 2008]

in their world positions. The second pass consists of drawing a bounding volume for each caustic volume. Using the positions from the texture, point-in-volume tests are computed for every visited pixel by a fragment shader. For points inside the volume, caustic intensity is computed and accumulated in the framebuffer. This process results in light being added where it focuses. This method works well for the rendering of underwater scenes, but it introduces a significant geometry load and does not account for the blocking of caustics rays by a receiver.

[Wyman, 2008] is an image space algorithm that computes caustics in three passes. In the first pass, the scene is rendered from the light point of view. Each texel whit a reflective or refractive object will cause the light to divert and change its direction. This diverted illumination is tracked, once a diffuse object is hit, the texel location is recorded as having received illumination. This image with depth is called the *photon buffer*. The second pass treats each location that received light as a point object. By treating them as small spheres that drop in intensity (*splats*), are then transformed to the eye's viewpoint and rendered to a *caustic map*. Depending on the amount of convergence or divergence of the photons, splats can be represented smaller or larger. The caustic map is then projected onto the scene, like a shadow map. This technique extends previous ones [Szirmay-Kalos et al., 2005, Wyman and Davis, 2006, Shah et al., 2007] by using a mip-map based approach to treat the photons in a more efficient hierarchical manner.

Some of the caustic rendering techniques presented in the next section are also able to compute surface caustics, like [Krüger et al., 2006, Sun et al., 2008, Papadopoulos and Papaioannou, 2009].

3.5 Volume caustics

Volume caustics (also known as godrays) are caused by the same concentration of light that creates surface caustics. The difference comes from the fact that in the presence of participating media (fog, clouds, smoke, etc.), the light is affected by emission, scattering and absorption. This creates changes in the light's direction of propagation, and thus the shafts of light can arrive to an external viewer, who perceives them as *volume caustics*.

One of the first methods [Jensen and Christensen, 1998] to simulate this kind of phenomena introduced *volume photon maps,* which is only used to represent indirect illumination, that is, it only stores photons that have been reflected or transmitted by surfaces before interacting with the media, and photons that have been scattered at least once in the media. This ray casting approach creates very realistic results, although it's an off-line approach that requires several minutes to render one single image.

[Nishita and Nakamae, 1994] demonstrated that porting this effects to scan line algorithms was possible, proposing a light shaft structure based on triangles from the reflector/refractor object. This method generates photo-realistic images, but it also is very geometric intensive. This approach is similar to the concept employed after by [Ernst et al., 2005], although it still doesn't achieve real-time performance.

[Ihrke et al., 2007] pre-compute a volumetric representation of a specularly deformed wavefront, allowing the interactive rendering of specular effects including complex volumetric caustics. Unfortunately, significant pre-computation costs preclude application in dynamic scenes. A different approach is used in [Krüger et al., 2006], where the authors directly splat energy to screen pixels using refracted lines as querying primitives. They get approximate solutions in real time, at the cost of neglecting some of the effects of the light (like absorption).

The work by [Sun et al., 2008] is capable of rendering the effects of refraction, single scattering, and absorption at interactive rates, even as the lighting, material properties, geometry, and viewing parameters are changing. They also use lines between specular object and receiver as rendering primitives. Here, however, the lines are rasterized into an intermediate illumination volume which enables a correct evaluation using a second ray marching step. However, they results have limited resolution due to high memory consumption.

[Papadopoulos and Papaioannou, 2009] is another method that uses lines as rendering primitives, similar to [Krüger et al., 2006]. It casts photons

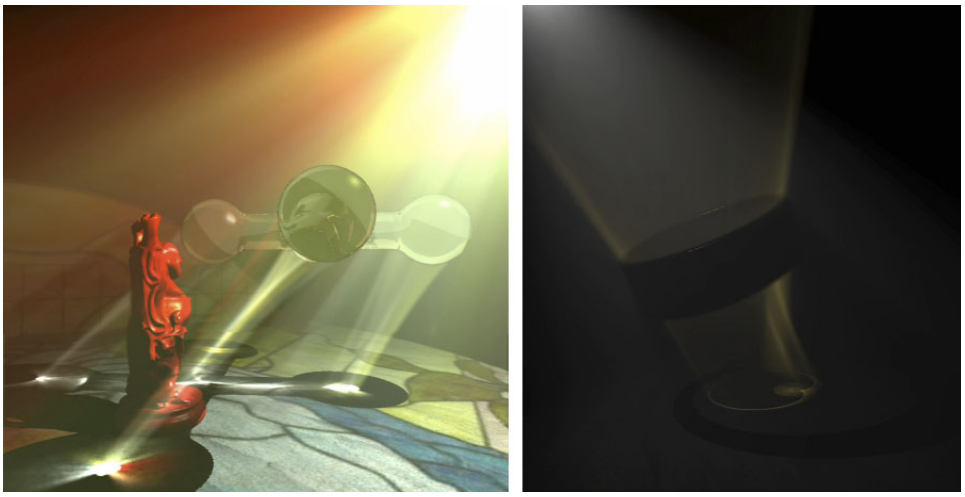

Figure 3.10: Volume and surface caustics from [Hu et al., 2010]

from the light source evenly distributed over a grid. These photons are intersected against the scene geometry, and line primitives are created between the intersection points. This method is capable of rendering both surface and volume caustics. They get very good performance and reasonably results, but only supports homogeneous media.

Recently, [Hu et al., 2010] presented an approach that can be seen as a combination of [Krüger et al., 2006] and [Sun et al., 2008]. Its screen-based approach gives high performance, which combined with physical-based accuracy gives good real-time results, but it still has some limitations, mainly due to its screen-based approach.

4 Implementation

4.1 Computer graphics using OpenGL

This section will give a brief introduction to OpenGL, a standard specification that defines a cross-platform API language for producing 2D and 3D graphics. In a more detailed vision, OpenGL is considered a state machine. The basic operation of OpenGL's state machine, that receives the name of Standard Graphics Pipeline, is to receive primitives points, lines and polygons and to convert them into pixels shown in the screen display. Most OpenGL commands either issue primitives to the graphics pipeline, or configure how the pipeline processes these primitives.

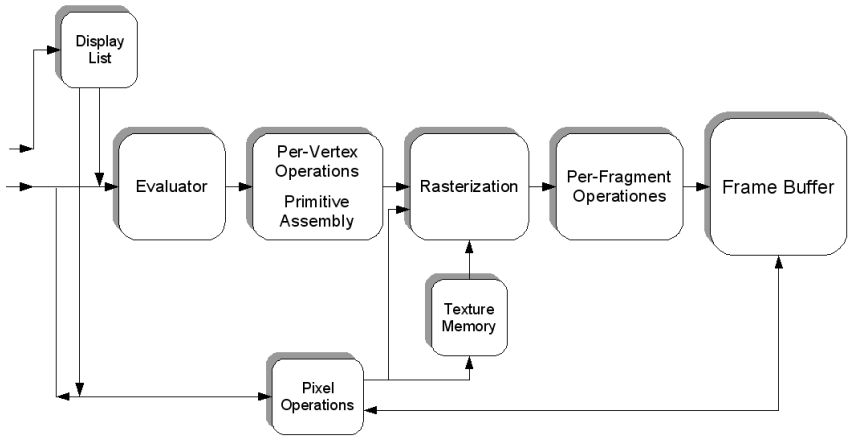

Figure 4.1: Simplified version of the Graphics Pipeline Process. Source [OpenGL, 2010]

A brief description of the process in the graphics pipeline is presented in 4.1, which consists of

1. Evaluation, if necessary, of the polynomial functions which define certain inputs, like NURBS surfaces, approximating curves and the sur-

face geometry.

2. Vertex operations, transforming and lighting them depending on their material. Also clipping non-visible parts of the scene in order to produce the viewing volume.

3. Rasterization or conversion of the previous information into pixels. The polygons are represented by the appropriate color by means of interpolation algorithms.

4. Per-fragment operations, like updating values depending on incoming and previously stored depth values, or color combinations, among others.

5. Finally, fragments are inserted into the frame buffer.

The purpose of the OpenGL pipeline is to convert three-dimensional objects into a two-dimensional image. To accomplish the transformation from three dimensions to two, OpenGL defines several coordinate spaces and transformations between those spaces. Each coordinate space has some properties that make it useful for some part of the rendering process.

Three-dimensional object attributes, such as vertex positions and surface normals, are defined in *object space*. This coordinate space is one that is convenient for describing the object that is being modeled. For example, the origin of this coordinate system (i.e., the point (0, 0, 0)) can be different for each object. In order to compose a scene that contains a variety of three-dimensional objects, each of which might be defined in its own unique object space, we need a common coordinate system. This common coordinate system is called *world space*. The units and the origin of this coordinate system is also arbitrary. After world space is defined, all of the objects in the scene must be transformed from their own unique object coordinates into world coordinates through the model transformation matrix.

After the scene has been defined, it is necessary to specify the viewing parameters. The camera position from which the scene will be viewed has to be set, as well as other viewing parameters such as the focus point and the up direction. The viewing parameters are combined in the vieweing matrix. When multiplied by this matrix, a coordinate is transformed from world space into *eye space*. By definition, the origin of this coordinate system is at the eye position.

Other 3D graphics APIs allow to specify the modeling matrix and the viewing matrix separately, but OpenGL combines them into a single matrix called the modelview matrix. This matrix transforms coordinates from object space into eye space, as seen in Figure 4.2.

The next step is to define a viewing volume. Is the region of the scene that will be visible in the image, and its described in the projection matrix. The projection matrix is a transformation that takes the objects in

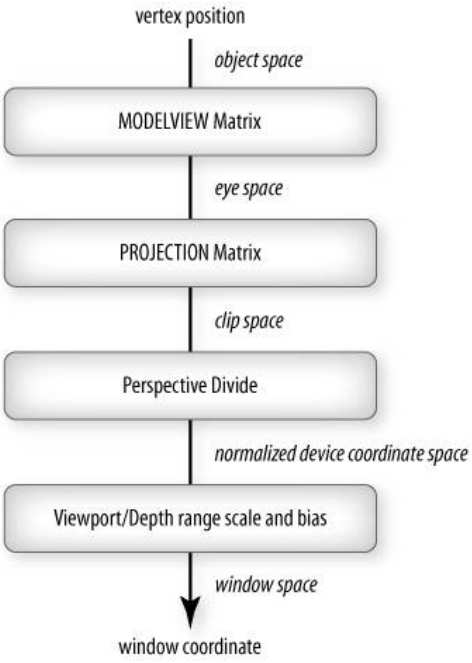

Figure 4.2: OpenGL coordinate spaces and transforms. Source [OpenGL, 2010]

the viewing volume into *clip space*, a coordinate space that is suitable for clipping.

The next stage in the transformation of vertex positions is the perspective divide. This operation divides each component of the clip space coordinate by the homogeneous coordinate w. The resulting x, y, and z components will range from [1,1]. This space is called the *normalized device coordinate space*.

Finally, to get pixel positions, the viewport transformation is needed. It specifies the mapping from normalized device coordinates into window coordinates, where x values range from 0 to the width of the window, and y values range from 0 to the height of the window.

4.1.1 Shader programs

Since version 2.0, OpenGL allowed the programmer to replace the fixed-function pipeline with vertex and fragment shaders, and later the geometry shaders were introduced.

A vertex shader is responsible of computing the final 3D vertex position, with the peculiarity of knowing nothing about other vertices. Geometry shaders are executed after vertex shaders, and their input is a whole primitive (that is, vertex-relation information, and possibly adjacency information

too). These shaders can emit zero or more primitives as output, which will be passed to the fragment shader. The fragment shader, also known as pixel shader, defines the final color of a fragment (a potential pixel) from illumination, material and texture properties.

Shaders are written in GLSL (OpenGL Shading Language), a high level shading language based on the C programming language. The language has a rich set of types, including vector and matrix types to make code more concise for typical 3D operations. A special set of type qualifiers manages the unique forms of input and output needed by the shaders. The *attibute*, *uniform* and *varying* qualifiers specify what type of input or output a variable serves. Attribute variables communicate frequently changing values from the application to a vertex shader, uniform variables comunicate infrequently changing values from the application to any shader, and varying variables communicate interpolated values between shaders. GLSL also adds some built-in variables that begin with the reserved prefix "gl_" in order to access existing OpenGL state and to communicate with the fixed functionality. For instance, shader can access built-in uniform variables that contain state values that are readily available within the current rendering context, like `gl_ModelViewMatrix` for obtaining the current modelview matrix. The vertex shader must write the variable `gl_Position` in order to provide necessary information to the fixed functionality stages. A geometry shader has to call the functions `EmitVertex()` and `EndPrimitive()` depending on the number and type of primitives it outputs. The fragment shader typically writes `gl_FragColor` and/or `gl_FragDepth`, or `gl_FragData[i]` in case of multiple render targets.

To use a shader, shader source code is first loaded into a shader object and then compiled. One or more shader objects are then attached to a program object. A program object is then linked, which generates executable code from all the compiled shader objects attached to the program. When a linked program object is used as the current program object, the executable code for the vertex shaders it contains is used to process vertices, and fragment shaders affect the processing of fragments during rasterization. If the program object does not have one of the types of shaders, or no program object is currently in use, the fixed-function method for that type of shader is used instead.

4.1.2 Framebuffer Objects

Framebuffers objects, known as FBOs, is an extension to OpenGL that allows to divert the rendering away from the window's framebuffer to one or more created offscreen framebuffers. Offscreen means that the content of the framebuffer is not visible until is first copied into the original window. Is similar to rendering to the back buffer, which is not visible until swapped.

FBOs have special characteristics that makes them interesting during

the rendering:

- FBOs are not limited to the window resolution.
- Textures can be attached to FBOs, allowing direct rendering to textures without an explicit copy.
- FBOs can contain multiple color buffers, which can be written to simultaneously from a fragment shader, known as multiple render targets.

The FBOs have two main uses: the post-processing of rendered images, and composition between different scenes.

4.2 Above/under fluid division

To the best of our knowledge, the division of a scene to separate fluid rendering between above and under it has never been explicitly proposed or evaluated. Previous methods only consider one of these situations, while we want to be able to change between media. To solve this problem, we have developed a technique that uses the stencil buffer, similar to *stencil shadow volumes*, in order to differentiate between above and under the fluid. Shadow volumes [Crow, 1977] is an old technique to add shadows into a rendered scene, it divides the virtual world in two: areas that are in shadow and areas that are not.

The stencil buffer [Neider et al., 1997] is an extra buffer of the computer graphics hardware, besides the color buffer and the depth buffer. The buffer is per pixel, and works on integer values, usually with a depth of one byte per pixel. The stencil function controls whether a fragment is discarded or not by the stencil test, and the stencil operation determines how the stencil buffer is updated as a result of that test, so it can be used to limit the area of rendering. Stencil buffer actions are part of OpenGL's fragment operations, it occurs immediately after the alpha test, and immediately before the depth test. Whether a stencil operation for a given fragment passes or fails has nothing to do with the color or depth value of the fragment. The stencil operation is a comparison between the value in the stencil buffer for the fragment's destination pixel and the stencil reference value. The value in the stencil planes is bitwise AND-ed with mask and with the reference value before the comparison is applied. The reference value, the comparison function, and the comparison mask are set by `glStencilFunc()`. The available comparison functions are listed in Table 4.1.

If the stencil test fails, the fragment is discarded (the color and depth values for that pixel remain unchanged) and the stencil operation associated with the stencil test failing is applied to that stencil value. If the stencil test passes, then the depth test is applied. If the depth test passes (or if

Comparison	Comparison test between reference and stencil value
`GL_NEVER`	Always fails
`GL_ALWAYS`	Always passes
`GL_LESS`	Passes if reference value is less than stencil buffer
`GL_LEQUAL`	Passes if reference value is less than or equal to the stencil buffer
`GL_EQUAL`	Passes if reference value is equal to the stencil buffer
`GL_GEQUAL`	Passes if reference value is greater than or equal to the stencil buffer
`GL_GREATER`	Passes if reference value is greater than the stencil buffer
`GL_NOTEQUAL`	Passes if reference value is not equal to the stencil buffer

Table 4.1: Stencil buffer comparisons

depth testing is disabled or if the visual does not have a depth buffer), the fragment continues on through the pixel pipeline, and the stencil operation corresponding to both stencil and depth passing is applied to the stencil value for that pixel. If the depth test fails, the stencil operation set for stencil passing but depth failing is applied to the pixel's stencil value. Thus, the stencil test controls which fragments continue towards the framebuffer, and the stencil operation controls how the stencil buffer is updated by the results of both the stencil test and the depth test. The available stencil operations are presented in Table 4.2, which are set by `glStencilOp()`.

`GL_KEEP`	Keeps stencil value unchanged
`GL_ZERO`	Sets stencil value to 0
`GL_REPLACE`	Replaces stencil value by stencil reference value
`GL_INCR`	Increments the current stencil buffer value
`GL_DECR`	Decrements the current stencil buffer value. Clamps to 0
`GL_INVERT`	Bitwise inverts the current stencil buffer value

Table 4.2: Stencil buffer operations

Writes to the stencil buffer can be disabled and enabled per bit by `glStencilMask()`. This allows an application to apply stencil tests without the results affecting the stencil values.

The developed technique uses the stencil buffer to divide the scene in two with an accuracy to the pixel. First, we assume that the fluid has its normals pointing outside the volume. A first solution would be just to use

the facing to determine the division, being inside the volume when back-facing, and outside when front-facing. But if we have other objects inside the fluid volume this approach is not valid, as it would classify as outside the volume the pixels within the object. Another solution consists of simply rendering a color mask depending on the facing when rendering the volume. This allows to separate the scene, but we implemented a similar technique that permits to separate the rendering into different stages to get a clearer code, not for performance reasons. Our solution makes two passes of the fluid volume, and thus are not affected by the other objects of the scene. As in shadow volumes, we render the front and back surfaces of the volume in separate passes. If we are looking at the interior of the volume all the faces will be backwards, or forward if we are looking at the exterior. Then, we need several rendering passes of the scene to complete the process:

1. First pass, increment value if it is the internal part of the volume.
2. Second pass, decrement value if it is the external part of the volume.
3. Third pass divided into two:
    - Render scene outside the volume.
    - Render scene inside the volume.

In a more detailed view, the steps of each pass are the presented. First pass:

1. Disable writing to the depth and color buffers.
2. Enable stencil buffer.
3. Use front-face culling.
4. Set the stencil operation to increment on depth pass. Stencil function is set to always pass.
5. Render fluid volume

Second pass:

1. Use back-face culling.
2. Set the stencil operation to decrement on depth pass. Stencil function is set to always pass.
3. Render fluid volume

After this is accomplished, the stencil buffer has 0 in the positions looking outside the volume, and 1 where looking inside. Now that we have the division of the scene we can render it normally. Third pass:

1. Enable writing to the depth and color buffers.
2. Set stencil function to pass if value is 0. Stencil operation keeps the values.
3. Render objects outside the volume.
4. Set stencil function to pass if value is 1. Stencil operation keeps the values.
5. Render objects inside the volume.

During the third pass it is possible that we do not know if an object is inside or outside the volume, or even it might be on both. In this case, we have to render twice the object, which increases the rendering time.

Although this approach requires several rendering passes, they are not as costly as full-rendering passes. The first two passes only render the fluid volume, not the whole scene, and they have disabled writing to the depth and color buffers, which also improves efficiency.

4.3 Light rendering

As mentioned in section 2.1.2, the Fresnel reflectance term is essential to achieve realistic shading of water. It is recommended to use the Fresnel term on a per-fragment basis as it can change quite rapidly. The Fresnel term is calculated using the formula 3.1, implemented on a fragment shader with the following function

```
void fresnel(in vec3 incom, in vec3 normal, in bool internal, in float index,
     out float reflectance, out float refracted_angle)
{
  float eta;
  if(internal)
  {
    eta = index;
  }
  else
  {
    eta = 1.0/index;
  }
  float rf = pow((index - 1.0)/(index + 1.0),2.0);
  float cos_theta1 = dot(normal, incom);
  float cos_theta2 = sqrt(1.0 - (pow(eta,2.0) * ( 1.0 - pow(cos_theta1,2.0))));
  float cos_theta_slope = dot(normal,vec3(0,0,1));
  refracted_angle = 3.141592 + (acos(cos_theta2) - acos(cos_theta_slope));

  reflectance = rf + (1.0 - rf) * pow((1.0 - abs(cos_theta1)),5.0);
}
```

where the incoming ray and the normal have to be in the same transformation space.

Light attenuation can vary with wavelength, producing visually compelling chromatic scales. Ideally, the color spectrum has to be discretized and computations must be done per wavelength. Using only the three components of the RGB color space gives an appropriate approximation. This is done by computing the attenuation color per wavelength:

$$L(d,z) = (L_R(d,z), L_G(d,z), L_B(d,z))$$

and combining it component-wise with the incoming colors. Absorption and scattering coefficients per wavelength can be found on oceanographic bibliography like [Preisendorfer, 1976, Mobley, 1994], if we are simulating water. In the case of other fluids, the parameters can be found on related papers.

The following fragment shader gets the reflected and refracted colors from the scene, and using the equation 3.6 and the Fresnel function obtains the final color.

```
void main(void)
{
    vec4 refraction_color;
    float fresnelR, angle, depth;

    vec3 attenuation = _underwaterAbsortion + _underwaterScattering;
    vec3 albeldo = _underwaterScattering / attenuation;

    vec4 reflection_color = computeReflectionColor();
    computeRefractionColor(refraction_color,depth);

    //calculate fresnel
    fresnel(view, normal, false, index, fresnelR, angle);

    refraction_color.rgb = -attenuation * refraction_color.rgb +
        (albeldo / 2.0) * (1.0 - light_in * exp(-attenuation*2.0*
         depth-gl_FragCoord.z*zFar));

    vec4 final_color = mix(refraction_color,reflection_color,fresnelR);

    gl_FragColor = final_color;
}
```

4.4 Surface rendering

The implemented projective texture mapping first stores the reflected and refracted environment in different framebuffer objects. The refracted FBO also stores the depth in order to retrieve the distance from the bottom. In this passes, all the objects from the scene excluding the fluid itself are rendered. During the reflection pass, all the objects are mirrored so they appear upside down in the reflection.

Once we have the textures rendered, we have to send to the GPU the reflective surface and all the parameters that the shader needs. The resulting vertex shader is the following:

```
void rayIntersection(in vec4 plane, in vec3 origin,
  in vec3 dir, out vec3 intersection)
{
    float dNV = dot(plane.xyz, dir);

    float t =  -(dot(plane.xyz, origin) + plane.z) / dNV;
    intersection = origin + (dir * t);
}

void main(void)
{
  mat4 bias = mat4(0.5, 0.0, 0.0, 0.0,
                   0.0, 0.5, 0.0, 0.0,
                   0.0, 0.0, 0.5, 0.0,
                   0.5, 0.5, 0.5, 1.0);

    vec3 intersection;
    vec3 dir = normalize(gl_Vertex.xyz - _eyePosition.xzy);

    normal = normalize(gl_NormalMatrix * gl_Normal);

    vec3 reflection = reflect(dir, normalize(gl_Normal));
    vec3 refraction = refract(dir, normalize(gl_Normal), eta);

    rayIntersection(plane_under, gl_Vertex.xyz, refraction, intersection);
    waterRefrTex = bias * gl_ModelViewProjectionMatrix *
        vec4(intersection.xyz,gl_Vertex.w);

    rayIntersection(plane_above, gl_Vertex.xyz,
        vec3(reflection.xy,refraction.z), intersection);
    waterReflTex = bias * gl_ModelViewProjectionMatrix *
        vec4(intersection.xyz,gl_Vertex.w);

    view = normalize(vec3(gl_ModelViewMatrix * gl_Vertex));

    gl_Position = ftransform();
}
```

The texture coordinates are computed and passed to the fragment shader via *varying*. The reflection texture coordinate is computed from the intersection of the reflected ray in world space with a plane situated above the surface. The refracted texture coordinate is computed in a similar way, but with a plane situated under the surface. The *bias* matrix is needed in order to change the coordinates from the range [-1,1] to the range [0,1] used by the textures. Other computed values are passed to the fragment shader for its own calculations.

The code which computes the reflected and refracted colors consists of

```
vec4 computeReflectionColor()
{
    //get projective texcoords
    vec4 tmp = vec4(1.0 / waterReflTex.w);
    vec4 projCoord = waterReflTex * tmp;
    projCoord = clamp(projCoord, 0.001, 0.999);

    //load reflection
    vec4 refl = texture2DProj(water_reflection, projCoord);
    return refl;
```

```
}

void computeRefractionColor(out vec4 refr, out float depth)
{
    //get projective texcoords
    vec4 tmp = vec4(1.0 / waterRefrTex.w);
    vec4 projCoord = waterRefrTex * tmp;
    projCoord = clamp(projCoord, 0.001, 0.999);

    //load refraction and depth texture
    depth = texture2DProj(_depthScene, projCoord).x;
    refr = texture2DProj(water_refraction, projCoord);
}
```

4.5 Surface caustics

Recent research developments have introduced a new image-space technique for rendering caustics in real-time entitled *"Realistic Real-time Underwater Caustics and Godrays"*. We have decided to implement this technique because it gets better performance results than previous ones and its easier to implement. It only supports homogeneous media, but our fluids are in fact homogeneous. This section explains the technique and a stepwise guide of how has been implemented. The presented steps are different from those of the paper, as it has been modified to allow for a non-planar refractive geometry, because the original paper [Papadopoulos and Papaioannou, 2009] works with a planar surface and simulates water movement generating normals through a noise function.

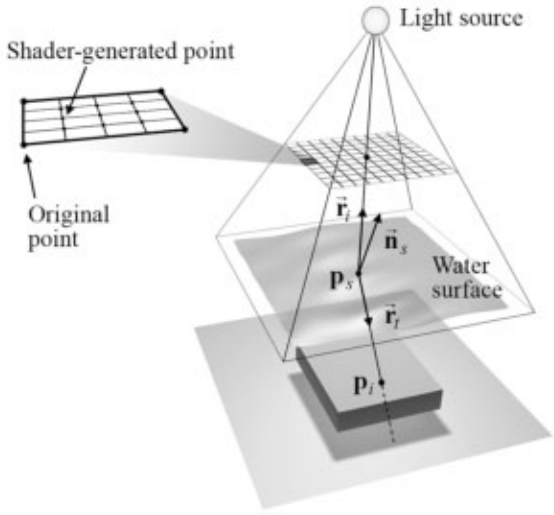

Figure 4.3: The process followed to cast photons. Source [Papadopoulos and Papaioannou, 2009]

The algorithm is conceptually quite simple and intuitive. It starts by casting photons from the light source evenly distributed over a grid. These photons are intersected against the refractive object to find the refracted photon. The intersection point of the refracted ray with the scene geometry is computed, and point primitives of variable size are created at the final position. The emitted point primitives are then rasterized and added to the final image. Figure 4.3 shows a schematic of the algorithm that finds the intersection points. Following is a high-level overview of the rendering pipeline:

1. Obtain depth and surface normals of the refractive object. The refractive object is rendered to texture from the lights view. The surface normals in world space and depth are output for each pixel. These textures are used with a grid of vertices of equal resolution, such that each vertex maps to a pixel on the texture. The vertex grid is used for the remainder of the algorithm in place of the refractive object.

2. Obtain 3D positions of the receiver geometry. The receiver geometry is rendered to a positions texture from the lights view. 3D world coordinates are output for each pixel instead of color. This positions texture is used for ray-intersection estimation in the next step.

3. Cast a regular photon grid from the light's view. An array of points is sent to the GPU in the light's canonical screen space to simulate the emission of photons.

4. A geometry shader tesselates each grid cell to produce the desired number of points, ensuring a dense photon distribution. The geometry shader also performs the refraction and emitts the final points.

5. The emitted point primitives are then rasterized and filtered to avoid aliasing in the low intensity areas.

Steps 1 and 2 are pretty straightforward, consisting only on FBO operations and simple shaders programs which output the interpolated 3D world positions instead of color. In step 3, the photon grid is created and initializated with a low resolution of points in the light's screen space, with coordinates ranging from -1.0 to 1.0. Then, using the geometry shader, each grid cell is subdivided to increment the number of photons. Now that the photon vertex grid has been setup, we need to determine the positions where the light rays intersect with the refractive object. We can do that by using the depth texture from step 1, transforming a point in light's screen space to world space. Using the surface normals texture we can also retrieve the normal in world space for that surface point, which allow us to perform the

refraction of the emitted light ray. Next, we need to compute the intersection points of the refracted light rays with the scene geometry to determine where the caustics will form.

The used method to find the intersection point is a variant of the Newton-Raphson method proposed in [Shah et al., 2007], which converges faster than others. The intersection estimation algorithm utilizes the positions texture rendered in the second step. A schematic illustration of the procedure is shown in Figure 4.4. Let v be the position of the surface intersection and $\vec{r}$ the refracted light vector. Points along the refracted ray are thus defined as $P = v + d \cdot \vec{r}$. An initial value of 1 is assigned to d and a new position, P_1, is computed. P_1 is then projected into the lights view space and used to look up the positions texture. The distance, d, between v and the looked up position is used as an estimate value for d to obtain a new point, P_2. Finally, P_2 is projected in to the lights view space and the positions texture is looked up once more to obtain the estimated intersection point.

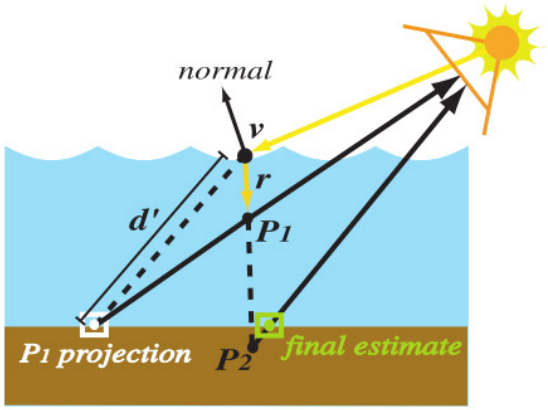

Figure 4.4: Diagram of the estimation intersection alghorithm. Source: [Shah et al., 2007]

The input parameters to the algorithm are respectively: *pos*, the origin of the refracted ray; *dir*, the direction of the refracted ray; *mVP*, the lights View-Projection matrix; *posTex*, the receiver geometry positions texture; *num_iter*, the desired number of iterations.

```
vec2 getTC(mat4 mVP, vec3 pos)
{
    vec4 texpos = mVP * vec4(pos,1.0);
    texpos = texpos / texpos.w;
    vec2 tc = vec2(0.5*(texpos.xy)+vec2(0.5,0.5));
    return tc;
}

float rayGeoNP(vec3 pos, vec3 dir, mat4 mVP, sampler2D posTex, int num_iter)
{
    float eps = 0.1;
```

```
    vec2 tc = vec2(0.0,0.0);
    // initial guess
    float x_k = 1.0;
    for(int i=0;i<num_iter;i++)
    {
        // f(x_k)
        vec3 pos_p = pos + dir*x_k;
        tc = getTC(mVP, pos_p);
        vec4 newPos1 = texture2D(posTex, tc);
        newPos1 = newPos1 / newPos1.w;
        x_k = distance(newPos1.xyz, pos_p);
    }
    return x_k;
}
```

Generally, each successive iteration increases the accuracy of the approximation. However, for the purpose of rendering caustics just a single iteration is sufficient. Once the intersection point is found, it is transformed into camera eye space coordinates. They are rendered using additive blending with their size corresponding to their screen space coverage, so when several refracted light rays intersect at the same point, multiple light deposits will be accumulated in that region causing it to get brighter and thus forming caustics. If all points emitted have a constant size, then the projection of the distant point primitives would overlap on the view plane, resulting in much brighter caustics than the ones close to the camera. The solution to this issue is to regulate the size of the points based on their distance from the camera, with smaller points close to the camera and greater for distant ones.

The code of the geometry shader that perform all these operations is the following:

```
void main()
{
    float wDepth = shadow2D(_waterDepth,
    vec3(0.5*(gl_PositionIn[0].xy)+vec2(0.5,0.5),1.0)).x;
    if(wDepth < 1.0)
    {
        float s = 2;
        for(int i = 0; i < s; i++)
        {
            for(int j = 0; j < s; j++)
            {
                vec4 position = gl_PositionIn[0];
                position.x += i * _photonDist/s;
                position.y += j * _photonDist/s;

                //to world space
                vec4 ri = normalize(gl_ModelViewProjectionMatrixInverse * position);
                vec4 water_intersection = gl_ModelViewProjectionMatrixInverse
                  * vec4(position.xy,wDepth*2-1,1.0);
                water_intersection = water_intersection / water_intersection.w;

                //find normal in world space
                vec4 normal = texture2D(_waterNormals, vec2(0.5*(position.xy)
                +vec2(0.5,0.5)));
```

```
                //refraction
                vec3 rs = refract(ri.xyz,normalize(normal.xzy),eta);
                rs = normalize(rs);

                //estimated intersection point
                float d = rayGeoNP(water_intersection.xyz,
                rs,gl_ModelViewProjectionMatrix,_scenePos,1);

                //scene intersection
                vec4 pi = vec4(water_intersection.xyz + rs * d,1.0);
                vec2 tc = getTC(gl_ModelViewProjectionMatrix, pi.xyz);
                vec4 final_pos = texture2D(_scenePos, tc);
                final_pos = final_pos / final_pos.w;

                //point size regulation
                float smin = 2.0;
                float smax = 55.0;

                float a = smax - (zFar * (smax - smin)) / (zFar - zNear);
                float b = (zNear * zFar * (smax - smin)) / (zFar - zNear);
                float distPointViewer = distance(pi.xyz,_cameraPosition);

                //to camera screen space
                gl_PointSize = a + b / distPointViewer;
                vec4 final_position = _cameraProjectionMatrix * _cameraViewMatrix
                * final_pos;
                gl_Position = final_position;
                EmitVertex();
                EndPrimitive();
            }
        }
    }
}
```

Finally, the fragment shader checks if the rasterized point is visible from the camera's view, discarding it if not. It also computes the final intensity value of the photon according to the formula

$$L_i = L_o e^{-\sigma_k \cdot d} \tag{4.1}$$

where d is the distance of the photon from the water surface. It doesn't take into account the distance travelled to the viewer, as it is already handled via the point size regulation.

After these steps, the resulting image can display some aliasing in the areas with a low light intensity. A low pass filter, a gaussian blur, is applied to avoid it. The filtering requires 9 texture samples per pixel:

```
vec4 blur(void)
{
    vec4 sample[9];

    vec2 pixelSize = 1.0/_texSize;
    vec2 texCoord = gl_TexCoord[0].st;

    sample[0] = texture2D(_caustics, texCoord  + vec2(-pixelSize.x,-pixelSize.y));
    sample[1] = texture2D(_caustics, texCoord  + vec2(-pixelSize.x,0.0));
    sample[2] = texture2D(_caustics, texCoord  + vec2(-pixelSize.x,pixelSize.y));
```

```
    sample[3] = texture2D(_caustics, texCoord  + vec2(0.0,-pixelSize.y));
    sample[4] = texture2D(_caustics, texCoord  + vec2(0.0,0.0));
    sample[5] = texture2D(_caustics, texCoord  + vec2(0.0,pixelSize.y));

    sample[6] = texture2D(_caustics, texCoord  + vec2(pixelSize.x,-pixelSize.y));
    sample[7] = texture2D(_caustics, texCoord  + vec2(pixelSize.x,0.0));
    sample[8] = texture2D(_caustics, texCoord  + vec2(pixelSize.x,pixelSize.y));

    // Gaussian kernel
    //   1 2 1
    //   2 4 2   / 16
    //   1 2 1

    return (sample[0] + (2.0*sample[1]) + sample[2] +
        (2.0*sample[3]) + (4.0*sample[4]) + (2.0*sample[5]) +
        sample[6] + (2.0*sample[7]) + sample[8]) / 16.0;
}
```

4.6 Volume caustics

The technique of volume caustics uses most of the work done in the previous section, as the two implementations are very close and are part of the same work [Papadopoulos and Papaioannou, 2009]. All the needed buffers and computations are exactly equal than in the case of surface caustics, except for the primitives emitted by the geometry shader. Instead of emitting point primitives, line primitives are created from the fluid intersection point to the scene intersection point. The main difference lies in the formula that produces the final intensity value per godray fragment, which is

$$L_i = L_o e^{-\sigma_k \cdot d} e^{-\sigma_k \cdot s} p(\theta) \tag{4.2}$$

where $p(\theta)$ is the phase function, d is the distance of the photon from the water surface, and s is the distance from the camera.

5 Results

5.1 Evaluation

The final tests were performed on a system with an Intel Core 2 Duo processor running at 1.86GHz, with 2 gigabytes of RAM and NVidia GeForce 8800 GTX GPU with 128 stream processors. The main window viewport resolution was 800 × 600, the caustic and godray buffers were 400 × 300, and the grid for caustics and godrays were 700 × 700 × 4 and 200 × 200 × 4 photons respectively. The volumetric nature of the godray effect requires a smaller amount of cast photons when compared to the caustic effect in order to achieve satisfactory detail.

The rendered scene consists of a surface with a resolution of 128 × 128 vertices simulated using FFT, an underwater object (a dolphin) and limiting walls. With this configuration, the programs achieves a frame rate of 35 fps. The above and under fluid surface rendering takes 3,086 and 2,444 *milliseconds* respectively, with a combined frame rate of 160 fps. Surface caustics rendering needs 14,492 *ms*, while volume caustics require 5,882 *ms*. Using all caustic rendering effects only, the application achieves 50 frames per second.

The simulated fluid uses absorption and scattering coefficients for pure water extracted from [Premoze and Ashikhmin, 2001]. Although naturally clear, water can present other particles suspended on it that change its appearance. Parameters such as phytoplankton concentration are not handled by our implementation, and thus our results for water can differ from real natural water. Physical simplifications made on section 3.3 like single scattering neither help towards realistic appearance, but we still get good results about water color as seen on figure 5.1.

A comparison with different attenuation coefficients is available in figure 5.2. The absorption parameters for pure water at 720, 550 and 450 *nanometers* (Red, green and blue, respectively) are (1.169,0.0638,0.0150). The parameters for the scattering coefficient at the same wavelengths are presented in table 5.1.

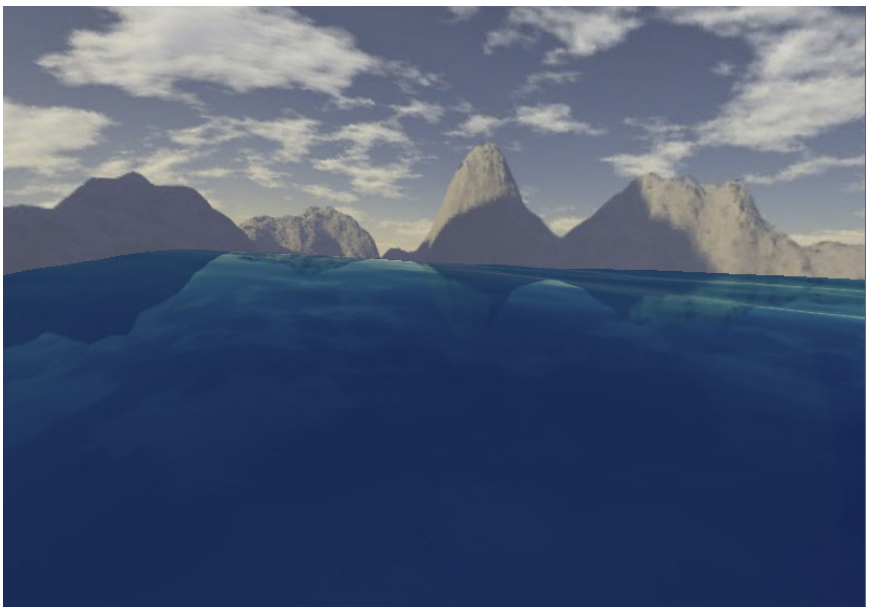

Figure 5.1: Render of a clear ocean surface.

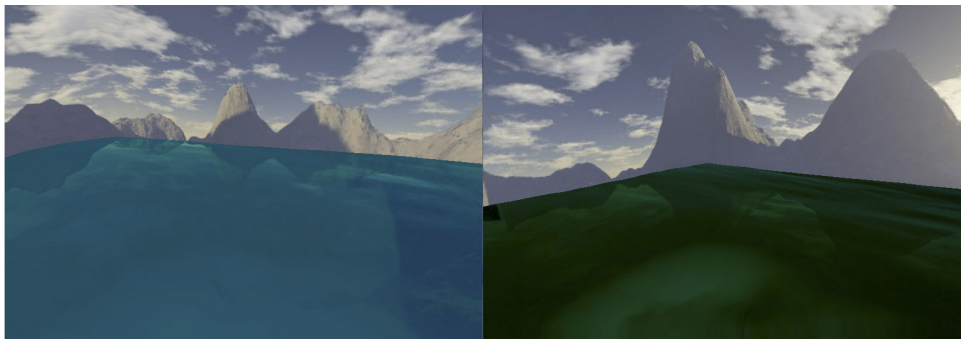

Figure 5.2: Render of coastal ocean and turbid harbor, respectively.

Clear ocean	(0.028, 0.035, 0.039)
Coastal ocean	(0.17, 0.20, 0.234)
Turbid harbor	(0.71, 0.82, 0.96)

Table 5.1: Different scattering coefficients used

Reflection and refraction colors can be observed in figures 5.3 and 5.4. Notice how the dolphin is visible in the second image. The method works as expected, changing the texture coordinates accordingly with the water movement. But it still has some issues that need to be addressed, as it doesn't handle precisely colors at the border of the viewport, specially when the camera is situated near the surface. This problem comes from the fact that the reflected or refracted color may not be available from the rendered textures.

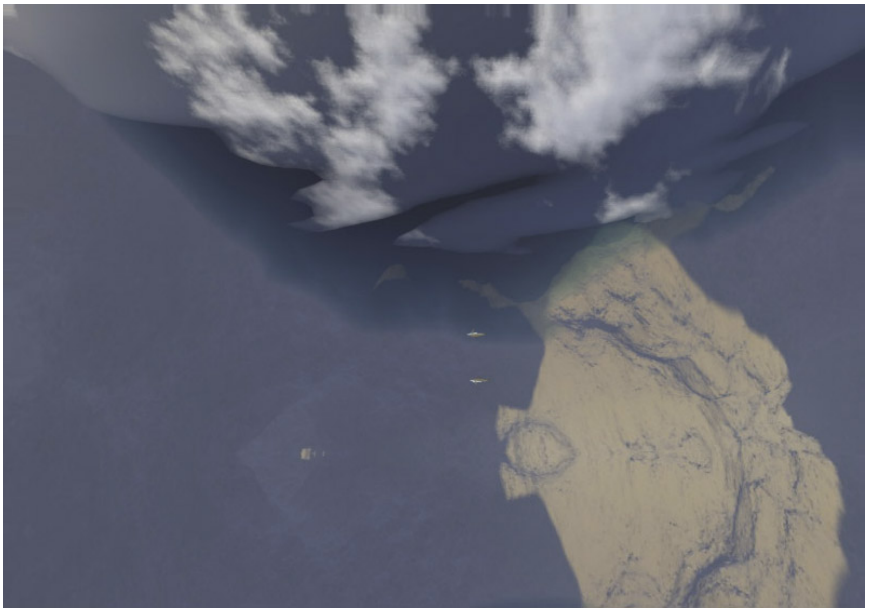

Figure 5.3: Render of underwater reflection and refraction. Attenuation is not calculated for color clarity.

Scene division can be seen on figure 5.5, where volumetric caustics can also be observed. Figure 5.6 shows an underwater image where volume and surface caustics are more visible, as well as light attenuation. Although it cannot be observed in a static image, light decreases as deeper the viewer goes. Caustics don't show up as very realistic, but it's expected from the physical simplicity of the employed method. Caustic patterns are not very sharpened, mainly due to a low variance of the surface normals.

We can compare our application with other available water rendering ap-

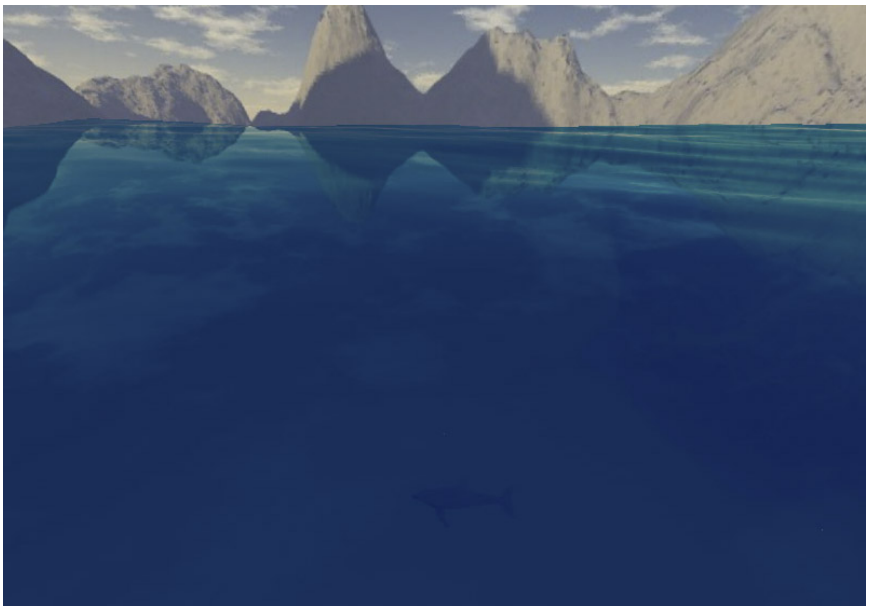

Figure 5.4: Reflection and refraction are mixed according to fresnel term.

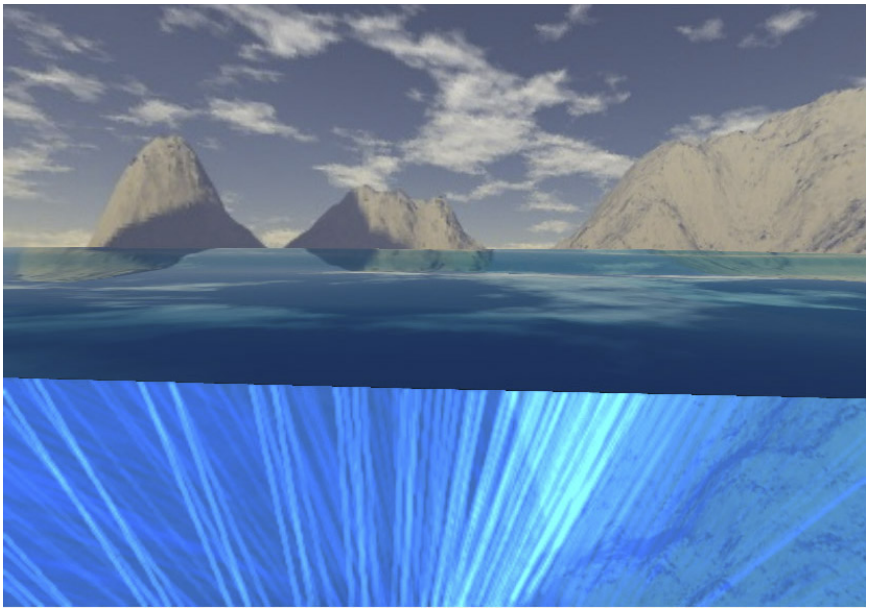

Figure 5.5: Simultaneous rendering of above and under water.

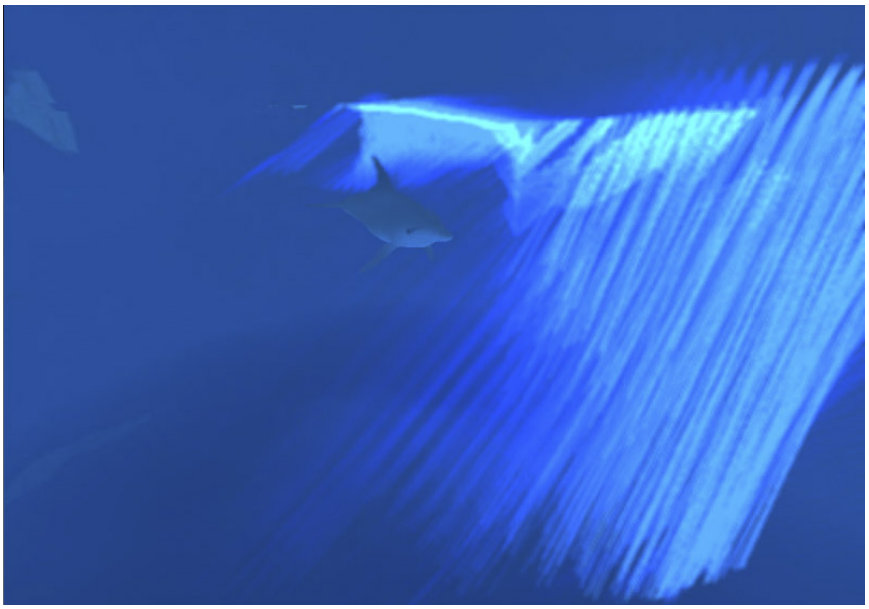

Figure 5.6: Underwater rendering with caustics.

plications, such as Hydrax [Hydrax, 2010] and osgOcean [osgOcean, 2009]. They are add-ons to open source 3D graphics engines, Ogre and OpenSceneGraph respectively. Although our application is not specifically oriented to ocean rendering, we can see on table 5.2 that similar techniques are used. Our techniques are almost the same as the ones used by Hydrax, but compared to osgOcean gets better results, as the latter doesn't take into account the movement of the surface in the reflections and refractions. It neither divides the scene, it just applies the underwater shader from a defined height, without taking into account the waves.

	Hydrax	osgOcean	Our application
Reflections and refractions	Projective textures	Planar textures	Projective textures
Fresnel term	Yes	Yes	Yes
Scene division	Per-pixel	Based on eye position	Per-pixel
Caustics	Yes	Yes	Yes
Others	Foam	Foam	-

Table 5.2: Techniques used according to the program.

5.2 Future work

Although the main goals have been achieved, there is still lots of work to do. Some of the areas that need improving are listed below.

- The used light interaction model has been derived from physical models, but it has been simplified in order to achieve an analytical expression. Use of stored textures with pre-computed values, as in [Sun et al., 2005], could improve the realism while maintaining interactive rates.
- We have not addressed the use of a bidirectional reflectance distribution function (BRDF) to reflect sunlight. A suitable approach in the case of ocean rendering can be found on [Bruneton et al., 2010], which presents an accurate analytical BRDF model.
- The implemented caustics methods could be improved to reach more realistic results, or use other techniques presented in sections 3.4 and 3.5.

5.3 Conclusions

In this thesis, many computer graphic and physical fields have been touched for the aim of building a real-time fluid rendering application:

- Basic light physics concepts have been reviewed.
- Fluid simulation techniques have been introduced.
- Nowadays rendering techniques have been researched.
- Physically-based rendering models have been studied.
- An application using all previous steps has been implemented.

Fluid rendering has become more interesting as newer generations of graphics hardware has made it possible to approximate the shading in an increasingly realistic manner. Fluids like water has often been troublesome as it did not directly apply to the standard lighting models that were traditionally used. Processing power and rendering techniques have improved to the degree that a surface no longer has to be approximated as a flat plane, it can be rendered with very high detail. It is feasible to consider per-fragment shading and it is viable to provide reasonably realistic reflections and refractions.

Bibliography

[Adabala and Manohar, 2002] Adabala, N. and Manohar, S. (2002). Techniques for realistic visualization of fluids: A survey. *Computer Graphics Forum*, 21:65–82.

[Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-time rendering. Third Edition.* Ak Peters Series. A.K. Peters.

[Baboud and Décoret, 2006] Baboud, L. and Décoret, X. (2006). Realistic water volumes in real-time. *Eurographics Workshop on Natural Phenomena.*

[Batchelor, 2000] Batchelor, G. K. (2000). *An Introduction to Fluid Dynamics.* Cambridge University Press.

[Blinn and Newell, 1976] Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Communications of the ACM 19*, 10:542–547.

[Bohren and Huffman, 1983] Bohren, C. F. and Huffman, D. R. (1983). *Absorption and scattering of light by small particles.* Wiley-Interscience, New York.

[Born and Wolf, 1999] Born, M. and Wolf, E. (1999). *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light.* Cambridge University Press.

[Brennan, 2002] Brennan, C. (2002). Accurate environment mapped reflections and refractions by adjusting for object distance. In *Shader X.* Engel W.

[Bruneton et al., 2010] Bruneton, E., Neyret, F., and Holzschuch, N. (2010). Real-time realistic ocean lighting using seamless transitions from geometry to brdf. *Computer Graphics Forum*, 29:487–496.

[Chandrasekhar, 1960] Chandrasekhar, S. (1960). *Radiative Transfer*. Dover Publications, Inc.

[Crow, 1977] Crow, F. C. (1977). Shadow algorithms for computer graphics. *SIGGRAPH Computer Graphics*, 11:242–248.

[Ernst et al., 2005] Ernst, M., Akenine-Möller, T., and Jensen, H. W. (2005). Interactive rendering of caustics using interpolated warped volumes. In *GI '05: Proceedings of Graphics Interface 2005*, pages 87–96, University of Waterloo, Ontario, Canada. Canadian Human-Computer Communications Society.

[Foster and Fedkiw, 2001] Foster, N. and Fedkiw, R. (2001). Practical animation of liquids. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 23–30, New York, NY, USA. ACM.

[Foster and Metaxas, 1997] Foster, N. and Metaxas, D. (1997). Controlling fluid animation. *Computer Graphics International Conference*, 0:178.

[Gath, 2008] Gath, J. (2008). Analytic approaches to single scattering in participating media. http://www.blacksmith-studios.dk.

[Glassner, 1994] Glassner, A. S. (1994). *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Gutierrez et al., 2008] Gutierrez, D., Seron, F. J., Muñoz, A., and Anson, O. (2008). Visualizing underwater ocean optics. *Computer Graphics Forum*, 27(2):547–556.

[Hu et al., 2010] Hu, W., Dong, Z., Ihrke, I., Grosch, T., Yuan, G., and Seidel, H.-P. (2010). Interactive volume caustics in single-scattering media. In *I3D '10: Proceedings of the 2010 ACM Siggraph symposium on Interactive 3D Graphics and Games*, pages 109–117, New York, NY, USA. ACM.

[Hydrax, 2010] Hydrax (2010). Hydrax web site. http://www.ogre3d.org/tikiwiki/Hydrax.

[Ihrke et al., 2007] Ihrke, I., Ziegler, G., Tevs, A., Theobalt, C., Magnor, M., and Seidel, H.-P. (2007). Eikonal rendering: efficient light transport in refractive objects. *ACM Trans. Graph.*, 26(3):59.

[Jensen and Christensen, 1998] Jensen, H. W. and Christensen, P. H. (1998). Efficient simulation of light transport in scenes with participating media using photon maps. In *Siggraph '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 311–320, New York, NY, USA. ACM.

[Jensen and Goliáš, 2001] Jensen, L. S. and Goliáš, R. (2001). Deep-water animation and rendering. *Proceedings of the Game Developer's Conference Europe*.

[Jiménez et al., 2005] Jiménez, J.-R., Myszkowski, K., and Pueyo, X. (2005). Interactive global illumination in dynamic participating media using selective photon tracing. In *Proceedings of the 21st spring conference on Computer graphics*, SCCG '05, pages 211–218, New York, NY, USA. ACM.

[Kilgard, 2001] Kilgard, M. J. (2001). Improving shadows and reflections via the stencil buffer. *nVidia white paper*.

[Krüger et al., 2006] Krüger, J., Bürger, K., and Westermann, R. (2006). Interactive screen-space accurate photon tracing on GPUs. In *Proc. of EGSR*, pages 319–329.

[Lazányi and Szirmay-Kalos, 2005] Lazányi, I. and Szirmay-Kalos, L. (2005). Fresnel term approximations for metals. *WSCG 2005, Short Papers*, pages 77–80.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169.

[Mastin et al., 1987] Mastin, G., Watterberg, P., and Mareda, J. (1987). Fourier synthesis of ocean scenes. *Computer Graphics and Applications, IEEE*, 7(3):16–23.

[Mobley, 1994] Mobley, C. D. (1994). *Light and Water: Radiative Transfer in Natural Waters*. Academic Press, Inc., San Diego.

[Neider et al., 1997] Neider, J., Davis, T., and Woo, M. (1997). *OpenGL Programming Guide*. Addison-Wesley.

[Nishita and Nakamae, 1994] Nishita, T. and Nakamae, E. (1994). Method of displaying optical effects within water using accumulation buffer. In *Siggraph '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 373–379, New York, NY, USA. ACM.

[NVIDIA, 2010] NVIDIA (2010). Nvidia developer web site. http://developer.nvidia.com/.

[OpenGL, 2010] OpenGL (2010). Opengl web site. http://www.opengl.org.

[osgOcean, 2009] osgOcean (2009). osgocean web site. http://code.google.com/p/osgocean.

[Papadopoulos and Papaioannou, 2009] Papadopoulos, C. and Papaioannou, G. (2009). Realistic real-time underwater caustics and godrays. In *Proc. GraphiCon '09*, pages 89–95.

[Preisendorfer, 1976] Preisendorfer, R. W. (1976). *Hydrologic Optics. Introduction, vol. 1.* National Technical Information Service, Springfield, IL.

[Premoze and Ashikhmin, 2001] Premoze, S. and Ashikhmin, M. (2001). Rendering natural waters. *Computer Graphics Forum*, 20(4):189–199.

[Schlick, 1994] Schlick, C. (1994). An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13:233–246.

[Shah et al., 2007] Shah, M. A., Konttinen, J., and Pattanaik, S. (2007). Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13:272–280.

[Smith and Baker, 1981] Smith, R. C. and Baker, K. S. (1981). Optical properties of the clearest natural waters (200-800 nm). *Applied optics*, 20(2):177–184.

[Stam, 1999] Stam, J. (1999). Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Sun et al., 2005] Sun, B., Ramamoorthi, R., Narasimhan, S. G., and Nayar, S. K. (2005). A practical analytic single scattering model for real time rendering. *ACM Trans. Graph.*, 24(3):1040–1049.

[Sun et al., 2008] Sun, X., Zhou, K., Stollnitz, E., Shi, J., and Guo, B. (2008). Interactive relighting of dynamic refractive objects. In *Siggraph '08: ACM Siggraph 2008 papers*, pages 1–9, New York, NY, USA. ACM.

[Szirmay-Kalos et al., 2005] Szirmay-Kalos, L., Aszódi, B., Lazányi, I., and Premecz, M. (2005). Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24:695–704.

[Szirmay-Kalos et al., 2009] Szirmay-Kalos, L., Umenhoffer, T., Patow, G., Szécsi, L., and Sbert, M. (2009). Specular effects on the gpu: State of the art. *Computer Graphics Forum*, 28:1586–1617.

[Tessendorf, 1999] Tessendorf, J. (1999). Simulating ocean water. In *Siggraph '99 Course Notes*.

[Whitted, 1980] Whitted, T. (1980). An improved illumination model for shaded display. *ACM*, 23:343–349.

[Wyman, 2008] Wyman, C. (2008). Hierarchical caustic maps. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 163–171, New York, NY, USA. ACM.

[Wyman and Davis, 2006] Wyman, C. and Davis, S. (2006). Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 153–160, New York, NY, USA. ACM.

[Wyman and Ramsey, 2008] Wyman, C. and Ramsey, S. (2008). Interactive volumetric shadows in participating media with single-scattering. In *Interactive Ray Tracing, 2008. IEEE Symposium*, pages 87–92.