REAL-TIME RENDERING

Realistic Large Bodies of Water

Samuel Luc Gauthier

Bern University of Applied Sciences

BTI 7302 Project 2

supervised by
Prof. Marcus Hudritsch

January 20, 2018

Real-Time Rendering

Realistic Large Bodies of Water

Samuel Luc Gauthier

Bern University of Applied Sciences
gauts1@bfh.ch

January 20, 2018

Abstract

The realistic rendering of ocean water constitutes an open problem in computer graphics. It is a complex topic, as it involves many physical properties. In this project report we present some solutions from the litterature, and implement the *Gerstner Wave* model, a deep ocean model, into a demonstration application. The visual result is not conviencing but provides a good start and confirms our intuitons. For further work we will implement the *fast Fourier transform* model and use a derivative of the *Projected Grid* concept. If there is time left we will handle foam, caustics and interactions with objects.

Keywords: computer graphics, real-time simulation, water rendering, waves, ocean, GPU algorithms

Contents

1 Introduction

Although water covers most of the earth's surface[1] it still remains an open problem in both mathematics and physics. There is no single solution available and approximations have to be made. The topic is broad and complex involving subjects ranging form fast Fourier transforms to Lagrangian and Eulerian simulations. The purpose of this report is to present interesting water models we found in the literature and explain how we implemented one. This document also gives our vision for further work, namely which model should be used or avoided. In order to read this document the reader should be familiar with the rendering pipeline and its programmable parts. More than enough background information can be found in (Akenine-Möller, Haines, and Hoffman, 2008).

We begin this report by presenting the different water models and candidate applications in section 2. Then we explain how we implemented the *Gerstner Wave* model in section 3 and provide performance and visual results in section 4. These are analyzed and discussed in section 5. Finally we conclude this report and give our vision for further work in section 6.

Figure 1 shows an example of an unknown real-time water model implemented in the upcoming game *Anno 1800*, developed by Blue Byte.

Figure 1: Harbour view from *Anno 1800*. Source: `forum.quartertothree.com`

2 Water Models

We start this section by presenting the physically accurate description of fluids, Navier-Stokes Equations, in subsection 2.1. Water models are separated into groups depending on the water depth: deep, intermediate or shallow. This classification takes its root from oceanography (Darles et al., 2011). We will present the deep water group in subsection 2.2 and the shallow one in subsection 2.3 (we include the intermediate models directly into the shallow group). For each of them we will present selected relevant approaches from the literature. These two groups describe only the *shape* of the water. In subsection 2.4

[1] Approximately 71% (Perlman, 2016)

we discuss the methods to produce realistic water surfaces. This includes caustics, foam, light scattering and more. Finally in subsection 2.6 we talk about two applications in need of a real-time water solution. Readers who desire to get a broader picture about water rendering should read the survey from Darles et al. (Darles et al., 2011).

2.1 Navier-Stokes Equations

Navier-Stokes Equations (or NSE) describe the motion of a fluid in any dimension $\mathbb{R}^n$. They are named after Sir George Gabriel Stokes[2], an Irish mathematician and physicist and Henri Navier, a French engineer, mathematician and economist. Both worked on, and enhanced the Euler equations. Navier proposed in 1820 to add a term to them for the heat dissipation. In 1845 Stokes also suggested introducing a term for the energy dissipation in form of heat (Gallagher, 2010). The equations are shown below only for the curious reader as we will not further discuss them. The Navier-Stokes equation of an incompressible fluid flow is:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{\nabla P}{\rho} + \nu \nabla^2 \mathbf{u} \tag{2.1.1}$$

where ν is the kinematic viscosity, $\mathbf{u}$ is the velocity of the fluid parcel, P is the pressure, ρ is the fluid density and t the time (Weisstein, 2018).

Solving the Navier-Stokes equations in three dimensions remains an open problem in mathematics and physics. It is one of the seven "Millennium Problems" defined by the Clay Mathematics Institute.

We felt the need to mention them here because all the water models found in the literature are somehow linked to them. Computational Fluid Dynamics (CFD) is the field of finding numerical approximations to those equations.

2.2 Deep Water Models

Deep water models can further be classified into two groups: spatial and spectral (Darles et al., 2011). To describe the water surface the former uses a sum of sines and cosines and the latter an inverse fast Fourier transform (IFFT)[3]. Unfortunately both have downsides: spatial models produce water having a too rounded shape and spectral models are difficult to control (Darles et al., 2011). In order to benefit from both, a hybrid solution should be chosen (Darles et al., 2011). We present two models: one for each group.

2.2.1 Gerstner Waves

Observing the ocean from the atmosphere, as seen in figure Figure 2, a slightly repetitive pattern is noticeable. This resembles strongly to a hevily disturbed combination of sines and cosines. The combination of such curves, also called *Gerstner Waves*[4], was proposed

[2] Best known for *Stokes's law*, the expression of the frictional force of a fluid exerted onto a sphere.

[3] The FFT approach allows to use data from a buoy or from aerial water photographs.

[4] Gerstner Waves where first described in (Gerstner, 1802) and provide a solution to a certain type of Euler equations.

in (Max, 1981) for their ray-tracing procedural model. The height $y = h(x,z,t)$ for each point (x,z) at time t is

$$h(x,z,t) = -y_0 + \sum_{i=1}^{N_w} A_i \cos(k_{i_x} x + k_{i_y} y - \omega_i t) \tag{2.2.1}$$

where N_w is the total number of waves, A_i is the amplitude of wave i, (k_{i_x}, k_{i_y}) the wave vector and ω_i the angular frequency (Darles et al., 2011; Max, 1981). Figure 3 depicts the sum of four such waves rendered in 1981.

(a) Far view of the Barren Island

(b) Close view of the Barren Island

Figure 2: The Barren Island captured by the Earth Observing-1 satellite in 2007. The darker waves in Figure 2a are deep internal waves (source: `earthobservatory.nasa.gov`).

Another form of recommended Gerstner Waves, described in (Fernando, 2004, Chapter 1), displaces also the x and z directions. $\mathbf{P}$ is the position in space, and $\mathbf{N}$ is its

Figure 3: Ray-tracing of four waves using Equation 2.2.1. Source: (Max, 1981)

corresponding normal. See Equation 2.2.2 and Equation 2.2.3.

$$\mathbf{P}(x,y,t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x + \sum_i (Q_i A_i \cdot \mathbf{D}_i.x \cdot \cos(\omega_i \mathbf{D}_i \cdot (x,y) + \varphi_i t)) \\ y + \sum_i (Q_i A_i \cdot \mathbf{D}_i.y \cdot \cos(\omega_i \mathbf{D}_i \cdot (x,y) + \varphi_i t)) \\ \sum_i (A_i \cdot \sin(\omega_i \mathbf{D}_i \cdot (x,y) + \varphi_i t)) \end{bmatrix} \tag{2.2.2}$$

$$\mathbf{N}(x,y,t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -\sum_i (\mathbf{D}_i.x \cdot WA \cdot C(\cdot)) \\ -\sum_i (\mathbf{D}_i.y \cdot WA \cdot C(\cdot)) \\ 1 - \sum_i (Q_i \cdot WA \cdot S(\cdot)) \end{bmatrix} \tag{2.2.3}$$

With the substitutions $WA = \omega_i A_i$, $S(\cdot) = \sin(\omega_i \mathbf{D}_i \cdot \mathbf{P} + \varphi_i t)$, $C(\cdot) = \cos(\omega_i \mathbf{D}_i \cdot \mathbf{P} + \varphi_i t)$.

The authors derive the binormal and tangent vectors from the position. They can be used with the normal vector from Equation 2.2.3 to compute the transformation from the tangent space into the local space.

$$\mathbf{B}(x,y,t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 - \sum_i (Q_i \cdot \mathbf{D}_i.x^2 \cdot WA \cdot S(\cdot)) \\ -\sum_i (Q_i \cdot \mathbf{D}_i.x \cdot \mathbf{D}_i.y \cdot WA \cdot S(\cdot)) \\ \sum_i (\mathbf{D}_i.x \cdot WA \cdot C(\cdot)) \end{bmatrix} \tag{2.2.4}$$

$$\mathbf{T}(x,y,t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -\sum_i (Q_i \cdot \mathbf{D}_i.x \cdot \mathbf{D}_i.y \cdot WA \cdot S(\cdot)) \\ 1 - \sum_i (Q_i \cdot \mathbf{D}_i.y^2 \cdot WA \cdot S(\cdot)) \\ \sum_i (\mathbf{D}_i.y \cdot WA \cdot C(\cdot)) \end{bmatrix} \tag{2.2.5}$$

The Gerstner wave approach is easy to implement and understand but produces too rounded shapes. On one hand computing only four waves produces a too repetitive result. On the other hand adding more of them reduces the performance. To obtain a reasonable result it should be combined with another model.

2.2.2 Fast Fourier Transform (FFT)

The fast Fourier transform approach was introduced in 1987 by Mastin, Watterberg, and Mareda (Mastin, Watterberg, and Mareda, 1987). The representation of the wave height

h given a horizontal position $\mathbf{x} = (x, z)$ and the time t is given by

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.2.6}$$

where $\mathbf{k}$ is the wave vector and $\tilde{h}(\mathbf{k}, t)$ is the amplitude and phase of the $\mathbf{k}$th wave at time t. The wave vector $\mathbf{k}$ is a two dimensional vector equal to $(k_x, k_z) = (2\pi n/s, 2\pi m/s)$. n and m are bounded by the resolution r of the grid in the following manner: $-r/2 \leq n, m \leq r/2$. s is the dimension of the animated area (Darles et al., 2011; Jensen and Golias, 2001).

In order to compute $\tilde{h}$ its value at time $t = 0$ is needed, which is denoted by $\tilde{h}_0(\mathbf{k})$. Then $\tilde{h}(\mathbf{k}, t)$ can be expressed as

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) e^{i\omega(k)t} + \tilde{h}_0^*(\mathbf{k}) e^{i\omega(k)t} \tag{2.2.7}$$

where $\tilde{h}_0^*(\mathbf{k})$ is the complex conjugate s.t. $\tilde{h}(\mathbf{-k}) = \tilde{h}^*(\mathbf{k})$ and $\omega(k) = gk$ with g being the gravitational constant.

Mastin, Watterberg, and Mareda take for $\tilde{h}_0$ the following:

$$\tilde{h}_0(\mathbf{k}) = \frac{\alpha g^2}{(2\pi)^4 f^5} e^{-\frac{5}{4}\left(\frac{f_m}{f}\right)^4}, \quad f_m = \frac{0.13}{u_{10}} \tag{2.2.8}$$

f is the frequency in Hz, f_m is the peak frequency of the wind speed u_{10}, measured at ten meters above the ocean surface, $\alpha = 0.0081$ is Phillips' constant and g is again the gravitational constant.

Tessendorf et al. proposed another amplitude function $\tilde{h}_0$,

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})} \tag{2.2.9}$$

where ξ_r and ξ_i are constants and $P_h(\mathbf{k})$, the spectrum, is:

$$P_h(\mathbf{k}) = A \frac{e^{\frac{-1}{(kL)^2}}}{k^4} |\mathbf{k} \cdot \mathbf{w}| \tag{2.2.10}$$

A is a constant, $L = V^2/g$, with V being the wind speed, g the gravitational constant and finally $\mathbf{w}$ the wind direction.

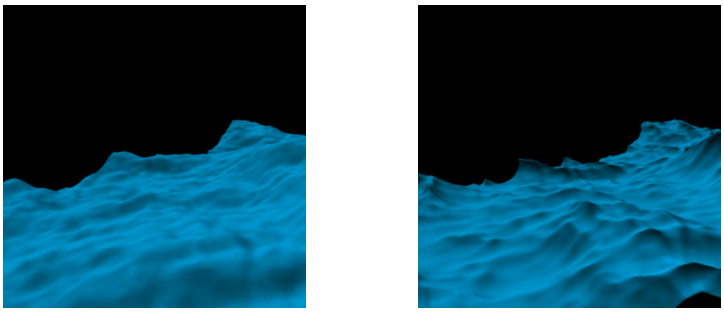
As seen in figure Figure 4a, the method only produces rounded waves. In order to get the choppy effect of a stormy ocean, the position can be displaced locally in the horizontal direction. This can be done with

$$\mathbf{x} = \mathbf{x} + \lambda \mathbf{D}(\mathbf{x}, t), \quad \mathbf{D} = \sum_{\mathbf{k}} -i\frac{\mathbf{k}}{k} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.2.11}$$

Using this method, it is possible to compute two height fields of different resolution. The first one can be used as a displacement map and the second one as a bump map. As already mentioned before, the frequency spectrum is hard to control for an artist and the gird needs to be large to display most of the frequencies (Gonzalez-Ochoa, 2016).

2.3 Shallow Water Models

Shallow water models try to discretize the Navier-Stokes equations (described in Equation 2.1.1) in two ways (Darles et al., 2011). The first one uses a two or three dimensional



(a) Without local displacement (b) With local displacement

Figure 4: Result from the use of the fast Fourier transform to compute the ocean's shape. Source: (Tessendorf et al., 2001).

grid to represent the fluid moving from cell to cell. It is called the Eulerian approach. The second one makes the assumption that particles transport a small fluid volume. This one is called the Lagrangian approach. Both share similar problems and advantages (Darles et al., 2011). The two can reproduce many different phenomena but are constrained by the grid size and the particle amount. They also become expensive when simulating finer details such as foam and spray as they need a lot of cells or particles (Darles et al., 2011).

We will present only one hybrid method because a derivation of it has been used for an industry production as described in section 2.5.1.

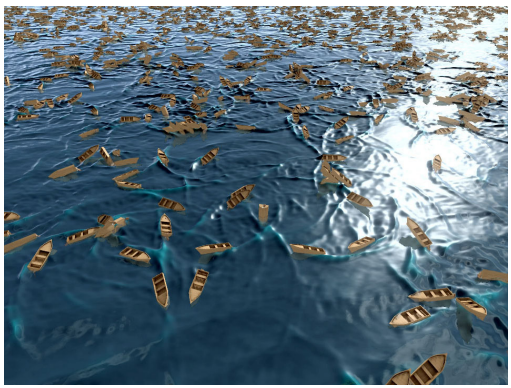
2.3.1 Wave Particles

The *Wave Particles* method as described in (Yuksel, House, and Keyser, 2007) is a hybrid shallow water model. It uses an Eulerian approach to compute the shape of the water and a Lagrangian one for the interaction with objects. Each *particle* is subdivided into three smaller ones as the wave expands. If the *particle* is too small it dies and is not displayed. When a *particle* encounters a fixed boundary it is reflected back. To compute the water volume moved by an object, Yuksel, House, and Keyser find the volume displaced by each face of the object. This is done with the help of a low resolution silhouette in form of a texture. They also compute the buoyant, drag and lift forces acting on the object (Yuksel, House, and Keyser, 2007). The result can be seen in figs. 5a and 5b.

2.4 Rendering of Ocean Details

Up until now we only presented methods which describe the shape of the water. The most important property of water is its reflection and refraction. We will discuss those two in section 2.4.1. Closely linked to them are caustics (section 2.4.2). Another one which can be observed is foam, presented in section 2.4.3. Finally the color of the water varies depending on the environment as explained in section 2.4.2.

(a) Pool test scene, source (Yuksel, House, and Keyser, 2007)



(b) Boat armada test scene, source (Yuksel, House, and Keyser, 2007)

Figure 5: Result of the demonstration application from (Yuksel, House, and Keyser, 2007)

2.4.1 Reflection and Refraction

Reflection can occur on any kind of surface. The most common one encountered every day is the mirror. The law of reflection states that the angle of incidence between the normal and an incident ray is equal to the angle between the normal and the reflected ray. The reflection vector can be computed using Equation 2.4.1. A visual representation can be seen in Figure 6.

$$\mathbf{r}_i = \mathbf{l} - 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} \tag{2.4.1}$$

Refraction depends on the two media the incident ray interacts with, as shown in Figure 7. It follows Snell's law from Equation 2.4.2.

$$\frac{\sin(\theta_i)}{\sin(\theta_t)} = \frac{n_t}{n_i} \tag{2.4.2}$$

To get the refraction vector, we need to use Equation 2.4.2 and split up $\mathbf{t}$ into its

tangent and normal part. The result is shown in Equation 2.4.3.

$$\mathbf{t} = \frac{1}{n}\mathbf{l} + \left(\frac{1}{n}\cos(\theta_i) + \sqrt{1 - \frac{1}{n^2}\left(1 - \cos^2(\theta_i)\right)}\right)\mathbf{n} \tag{2.4.3}$$

Other surfaces than perfect mirrors do not entirely reflect the incoming ray. Only one part is reflected and the other refracted. The Fresnel equations give the ratio between the reflected and refracted light. As they are too expensive to compute, Schlick proposed the following approximation:

$$F_0 = \frac{n-1}{n+1}^2 \tag{2.4.4}$$

$$F(\theta_i) = F_0 + (1 - F_0)(1 - \cos(\theta_i))^5 \tag{2.4.5}$$

where n is the refractive index of the material. Here we assume that the incident ray comes from the air medium.

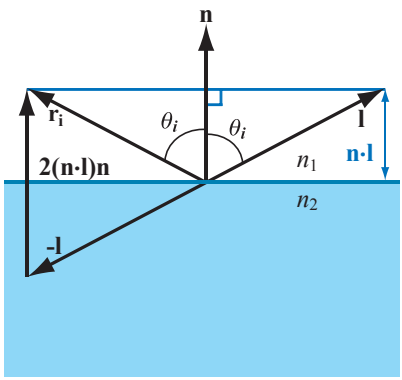


Figure 6: Computation of the reflection vector $\boldsymbol{r_i}$. Source: (Akenine-Möller, Haines, and Hoffman, 2008).

2.4.2 Caustics

Caustics only appear in shallow water and can be observed for example in rivers, pools, near hulls or at the beach. A caustic is formed by reflected and refracted light rays originating from a curved surface. For real-time applications, approximations have to be made. In (Fernando, 2004, Chapter 2), Guardado and Sánchez-Crespo propose a technique based on photon mapping using a simplified backward Monte Carlo ray tracing method. This technique is displayed in algorithm 1.

Other techniques exist (Darles et al., 2011) to compute caustics such as in (Arvo, 1986) where they are stored into an illumination map. Iwasaki, Dobashi, and Nishita treat light rays as "beams"(Heckbert and Hanrahan, 1984; Watt, 1990). They use the Z-buffer and stencil buffer to compute the intersection between "illumination volumes" and objects. However, we think that these techniques are too costly for a real-time application and should be simplified and computed beforehand.

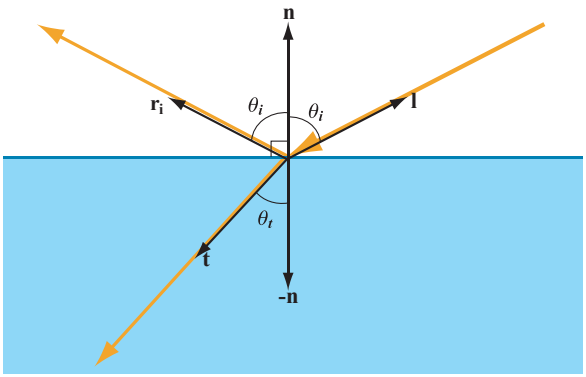


Figure 7: Reflection and refraction vectors $\boldsymbol{r_i}$ and $\boldsymbol{t}$ computed from the incoming light vector $\boldsymbol{l}$. Source: (Akenine-Möller, Haines, and Hoffman, 2008).

Algorithm 1: Pseudocode for real-time water caustics rendering (Fernando, 2004, Chapter 2)

```
Paint the ocean floor.
For each vertex in the fine mesh:
    Send a vertical ray.
    Collide the ray with the ocean's mesh.
    Compute the refracted ray using Snell's Law in reverse.
    Use the refracted ray to compute texture coordinates for the "Sun" map.
    Apply texture coordinates to vertices in the finer mesh.
Render the ocean surface.
```

2.4.3 Foam

Foam is only formed[5] when the wind velocity exceeds 13km/h (Darles et al., 2011; Munk, 1947). In (Jensen and Golias, 2001; Jeschke, Birkholz, and Schmann, 2003) the authors use the height of the vertex and the slope formed with the surrounding vertices to fade the foam in and out.

Particle systems also exist to represent foam but they are too expensive to render as many of them are needed to get realistic results (Darles et al., 2011).

2.4.4 Water Color

The color of the water depends on organic matter present in it, such as phytoplanktons, chlorophyll concentration and turbidity. (Darles et al., 2011). In (Gonzato and Le Saëc, 2000) the authors use a color lookup table taken from (Ivanoff, 1972) which contains the color of the seas and oceans based on the angle between the sun and the horizon. Premože and Ashikhmin use a more complicated approach: they compute absorption and

[5]The other kind of foam that can be observed at beaches is caused by the decaying of algal blooms or by pollutants and detergents (Oceanic and Administration, 2017).

backscattering coefficients based on a table containing the wave length of different water types (Premože and Ashikhmin, 2001). Unfortunately this method was used in a Monte Carlo path tracer and we doubt that it can be used like this in a real-time application.

2.5 Methods Used in Industry Productions

Many different water models are used in industry productions. One that has been applied extensively in movies (such as "Titanic", "13 days", "The World Is Not Enough") and most of the games is the one from Tessendorf (Tessendorf et al., 2001) as seen in section 2.2.2. During our research we came across two solutions which where presented at the Game Developer Conference or at the SIGGRAPH conference. They are implemented in Uncharted and World of Warships.

2.5.1 Uncharted

A modified version of the *Wave Particle* approach from section 2.3.1 has been used in *Uncharted 3* (2011) and *4* (2016) to render rapids and the ocean. Developers chose this technique because it is more intuitive for artists to control, it presents no tiling artifacts, it is deterministic in time and fast to compute (Gonzalez-Ochoa and Holder, 2012). For the level of detail they use a "Continuous Distant-Dependent Level of Detail" from (Strugar, 2009). This technique stitches quads of different size together. The quads become bigger the further away they are from the camera. The result can be seen in Figure 8.

Figure 8: View of the ocean from *Uncharted 4*. Source: (Gonzalez-Ochoa, 2016)

2.5.2 World of Warships

The developers at *Wargaming* choose to superpose three different height maps, as described in (Pharr and Fernando, 2005, Chapter 18), for their *World of Warships* (2015) game (Kryachko, 2016). For the calm zones they use an additional one. To approximate the geometry they use the "Projected Grid" concept from (Claes, 2004), which works as follows:

Algorithm 2: Projected Grid. Source: (Claes, 2004)

1 Create a regular discredited plane in post-perspective space that is orthogonal towards the viewer.
2 Project this plane onto a plane in the world-space.
3 Displace the world-space vertices according to the height field.
4 Render the projected plane.

They took this approach because it gives them an efficient polygon distribution over the screen (Kryachko, 2016). However, this method has several drawbacks: it causes flickering, instability in motion, is not very accurate and does not handle water deformations very well. By raising the triangle density up to three to four times and redistributing them in the vertical direction, they solved the low accuracy problem. To fix the flickering and instability in motion issue, the developers use a modified per pixel ray casting algorithm. It computes an improved version of Newton's method to solve the ray cast equation. Finally they manage to create realistic water deformations with the help of six times anti-aliasing (6xAA) (Kryachko, 2016).

Figure 9: Rendered ocean in *World of Warships*. Source: (Kryachko, 2016)

2.6 Candidate Applications for Embedding

We now present two candidate applications which need a real-time water model: *SLProject* and *0 A.D.* Although both have already one, in the former application the model does not work and in the latter it is outdated.

2.6.1 SLProject

SLProject is a Scene Library developed and maintained by the cpvrLab at the Bern University of Applied Sciences (BFH). It is free and open source[6] and runs on Windows, MacOS, Linux, Android and iOS. The project is a showcase for 3D computer graphics and

[6] Licensed under the GNU-GPL license.

image processing topics such as real-time rendering, raytracing and feature tracking. It is coded in C++ and OpenGL ES to ensure complete platform independence (Hudritsch, 2017; *SLProject Doxygen Documentation* 2017).

The project has a real-time water rendering demo but it is unfortunately broken (see open ticket, #36 "Fix water shader"[7]). The water plane is modeled by a simple multiplication of the sine and cosine functions and is illuminated 50% by a point light and 50% by a cube texture.

2.6.2 0 A.D.

0 A.D. is a real-time strategy game representing the 500 B.C to 500 A.D era. It is developed by Wildfire Games, an independent game development studio. The game is free, open source[8] and runs on Windows, MacOS and Linux.

The development of the game began in 2000 when three modding groups whished to create a free game engine. Up until 2009 the source code was accessible only to members of Wildfire Games. Anyone interested in participating could make an application and pass an interview. However, due to the fading interest of a mod[9] which used 0 A.D.'s game engine and the lack of programmers, they opened up the code in July 2009. From then on many contributions have been made, notably one redesigning entirely the code base, making new contributions easier. As of today the project is still actively maintained, with the latest alpha-release (number 22 code named "Venustas") dating from July 26, 2017 (*Project Overview* 2017; *The Story of 0 A.D.* 2017).

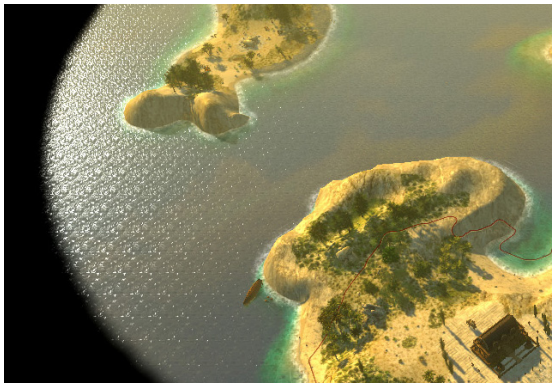
The game engine, Pyrogenesis, is heavily oriented towards modding. The core engine is coded in C++ and Javascript is used for the game's logic, like the artificial intelligence or the random generation of maps. This means that when *0 A.D.* is run, the game engine is started with a given mod. A mod contains all the graphical elements, 3D models, sound, shaders, scenarios, map generators and artificial intelligence.

The documentation for developers provides good build instructions, coding conventions and codebase descriptions. Unfortunately we found that it is difficult for a newcomer to understand the game engine architecture. It is less documented and scattered accross multiple pages which are sometimes outadated. The Doxygen documentation is minimalistic and unlike the one from *SLProject*, does not help to understand the architecture.

The current water implementation has been created in 2006 by a student form the University of Waterloo, Canada, as a course project (*Matei Zaharia's CS 488 Project* 2006). Over the years it has been slightly modified and optimized but appears visually outdated (see Figure 10 on page 14) compared to the one used in commercial RTS games like Anno 1800 (see Figure 1) or Age of Empires 3. There is an open ticket, #48 "Advanced Water Rendering"[10], which was opened twelve years ago and modified three years ago but is still not closed. We have contacted the developers and they would gladly appreciate a contribution that would close the issue.

[7] `https://github.com/cpvrlab/SLProject/issues/36`
[8] Licensed under the GNU-GPL and CC BY-SA license.
[9] Shorthand for the term *modification*
[10] `https://trac.wildfiregames.com/ticket/48`

(a) Close view of the water. Source: `play0ad.com`



(b) Zoomed out view. Source: `play0ad.com`

Figure 10: *0 A.D.*'s water rendering. Notice on the lower left part of Figure 10a the repetition of the waves. In Figure 10b it is even more apparent.

3 The Simple Gerstner Wave Approach

We opted to implement the Gerstner Wave approach from section 2.2.1 because it was the easiest one to understand and well described in (Fernando, 2004). Furthermore, it does no represent a daunting task for someone new to the graphics pipeline programming.

We integrated the water model into a demonstration application coded in C++ using OpenGL and the *GLEW*, *GLFW* and *stb* libraries. C++ was used because both applications in subsection 2.6 are programmed with it. We choose *GLEW* because it is a cross-platform library to load the OpenGL extensions. *GLFW* is used for the window creation and keyboard interactions. Finally we utilize the simple but powerful capabilities of *stb* to read and load images into memory.

We programmed the following classes: `Camera`, `Plane`, `Skybox`, `FPSCounter`. The

`Camera` class represents the position of the camera in spherical coordinates[11]. This makes it easier to compute the zoom and rotation in the φ and θ directions. We use the `Plane` class to build a grid of vertices with the corresponding indices. The grid always points to the y direction[12], and can have a arbitrary resolution (amount of vertices) as well as size. This class is used to represent the water. The `Skybox` a simply cube whose indices are specified counterclockwise such that the normals point inwards. The particularity of it, is that the depth buffer is disabled before the OpenGL draw call is issued and re-enabled afterwards. Every object rendered after it, will be displayed in front of it. Finally the `FPSCounter` class tracks the time needed to compute each frame.

In the vertex shader of the water plane we compute the position, normal and tangent vectors as in eqs. (2.2.2), (2.2.3) and (2.2.5) with the only subtlety that the y and z coordinates are flipped. We use four waves ($i = 4$) with the constants $g = 9.81$, $\kappa = 0.2$, $\pi = 3.14159265358979$ and values for ω, φ, A, Q and $\mathbf{D}$ as follows:

$$\begin{aligned}
\omega &= \left\{\sqrt{g2\pi}, \sqrt{g4\pi}, \sqrt{g10\pi}, \sqrt{g\pi}\right\},\\
\varphi &= \{0.4\omega_1, 0.3\omega_2, 0.3\omega_3, 0.1\omega_4\},\\
A &= \{0.01, 0.008, 0.017, 0.003\},\\
Q &= \left\{\frac{\kappa}{\omega_1 A_1 i}, \frac{\kappa}{\omega_2 A_2 i}, \frac{\kappa}{\omega_3 A_3 i}, \frac{\kappa}{\omega_4 A_4 i}\right\},\\
\mathbf{D} &= \left\{\begin{bmatrix}0.0\\0.5\end{bmatrix}, \begin{bmatrix}0.8\\0.7\end{bmatrix}, \begin{bmatrix}0.6\\0.6\end{bmatrix}, \begin{bmatrix}0.7\\-0.3\end{bmatrix}\right\}
\end{aligned} \tag{3.0.1}$$

The implementation can be found in Listing 1 on page 16.

We pass to the fragment shader the local position, normal and tangent vectors as well as the local light position and the roughness of the surface.

In the fragment shader we update the normal with the values read from the normal map. Then we transform the light position, vertex position and normal into the eye space. We compute the reflection and refraction vectors as described in subsection 2.4 with the help of glsl's `reflect(...)` and `refract(...)` functions. The two corresponding colors are fetched from the cube map texture of the skybox and mixed together based on Schlick's approximation of the Fresnel term. This gives our `ambient` term, whose computation is shown in Listing 2 on 16. `F_0` results form Equation 2.4.4 with $n_{water} = 1.333$ and equal to 0.020373. `RATIO` is the ratio between the water and air refractive indices and equal to 0.75.

For the `specular` term we use the Cook-Torrance microfacet specular BRDF[13] as described in Equation 3.0.2.

$$f(\mathbf{l},\mathbf{v}) = \frac{F(\mathbf{l},\mathbf{h})G(\mathbf{l},\mathbf{v},\mathbf{h})D(\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})} \tag{3.0.2}$$

$F(\mathbf{l},\mathbf{h})$ is Schlick's approximation of the Fersnel term as seen in Equation 2.4.5. For $D(\mathbf{h})$,

[11] Our implementation is based on the recommendation of the Graphics Stackexchange post: `https://computergraphics.stackexchange.com/questions/151/`

[12] In OpenGL the z coordinate points out of the screen.

[13] Bidirectional reflectance distribution function

```
vec4 position = in_Position;
vec3 normal = vec3(0.0, 1.0, 0.0);
vec3 tangent = vec3(0.0, 0.0, 1.0);

for(int i = 0; i < NUMWAVES; i++) {
    float alpha = w[i] * dot(D[i], position.xz) + phi[i] * t;
    float WA = w[i] * A[i];
    float sinAlpha = sin(alpha);
    float cosAlpha = cos(alpha);
    float DxCos = D[i].x * cosAlpha;
    float DyCos = D[i].y * cosAlpha;
    float DxDy = D[i].x * D[i].y;

    position.x += Qs[i] * A[i] * DxCos;
    position.y += A[i] * sinAlpha;
    position.z += Qs[i] * A[i] * DyCos;

    normal.x -= D[i].x * WA * cosAlpha;
    normal.y -= Qs[i] * WA * sinAlpha;
    normal.z -= D[i].y * WA * cosAlpha;

    tangent.x -= Qs[i] * DxDy * WA * sinAlpha;
    tangent.y += D[i].y * WA * cosAlpha;
    tangent.z -= Qs[i] * D[i].y * D[i].y * WA * sinAlpha;
}
```

Listing 1: Implementation of eqs. (2.2.2), (2.2.3) and (2.2.5)

```
vec3 reflected = reflect(incident_eye, n);
reflected = vec3(inverse_V * vec4(reflected, 0.0));

vec3 refracted = refract(incident_eye, n, RATIO);
refracted = vec3(inverse_V * vec4(refracted, 0.0));

vec4 reflectColor = texture(cube_texture, reflected);
vec4 refractColor = texture(cube_texture, refracted);

vec4 ambient = mix(refractColor, reflectColor, F(F0, v, n));
```

Listing 2: Reflection and refraction color computation

the normal distribution function, we use the GGX/Trowbridge-Reitz function:

$$D(\mathbf{h}) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2-1)+1)^2} \tag{3.0.3}$$

α is the squared roughness. And for the specular geometric attenuation term $G(\mathbf{l},\mathbf{v},\mathbf{h})$

we take the height correlated Smith function:

$$k = \frac{\alpha}{2} \tag{3.0.4}$$

$$G_1(\mathbf{x}) = \frac{\mathbf{n} \cdot \mathbf{x}}{(\mathbf{n} \cdot \mathbf{x})(1-k)+k} \tag{3.0.5}$$

$$G(\mathbf{l}, \mathbf{v}, \mathbf{h}) = G_1(\mathbf{l}) G_1(\mathbf{v}) \tag{3.0.6}$$

We choose to use the same equations as in (Karis and Games, 2013) for $F(\mathbf{l}, \mathbf{h})$, $G(\mathbf{l}, \mathbf{v}, \mathbf{h})$ and $D(\mathbf{h})$. The implementation of those functions can be found in Listing 3 on page 17. To reduce the computations we avoid using the `dot(...)` function inside of them. Instead we pass the value of the computed dot product.

```
// Shlick approximation
float F(float F_0, vec3 v, vec3 h) {

    return F_0 + (1 - F_0) * pow((1 - max(dot(v, h), 0)), 5);
}

float G_smith(float nx, float k) {

    return nx / (nx * (1 - k) + k);
}

// Height correlated Smith
float G(float nl, float nv, float k) {

    return G_smith(nl, k) * G_smith(nv, k);
}

// Trowbridge-Reitz
float D(float nh, float alpha2) {

    float denom = nh * nh * (alpha2 - 1) + 1;
    return alpha2 / (PI * denom * denom);
}
```

Listing 3: Cook-Torrance microfacet specular BRDF helper functions

Finally we output the final fragment color as the sum of the ambient and specular values: `out_Color = specular + ambient`.

4 Performance and Visual Results

We tested our application on a MacBook Pro Retina (Late 2013) with an Intel i7 2.8 GHz processor and an integrated Intel Iris 1.5 GB graphic card. The application ran on an external monitor having the resolution of 1920×1080 pixels. The application itself ran in 800×600 window. The number of vertices used to represent the water was 900. The average computation time of a frame was 58.8978ms.

The results of our implementation can be seen in Figure 11 on page 19. Figure 12 on page 24 shows our reference images.

5 Discussion

Having seen Figure 3 and other figures from (Fernando, 2004) we knew from the start that the result would not be a realistic ocean water. After implementing the Gerstner water model ourselves, we now have the confirmation that this model alone is indeed

Figure 11: Result of our implementation

not enough to represent a realistic water surface. We will discuss two important issues further below: visual accuracy and performance.

5.1 Visual Accuracy

Four waves is in our view insufficient to approximate realistically the shape of the ocean. We had trouble to find constants which produced acceptable results: in all the resources explaining the Gerstner model, finding them where “left to the artist” (Fernando, 2004,

Chapter 1). We came across one resource (Williams, 2017) in which the author implemented the model with the help of the Unreal Engine but unfortunately her constants did not produce acceptable results in our application. The rounded shape is only suitable in cases where it is artistically needed or in those displayed in Figure 12.

Our implementation does not take into account the interaction with other objects on its surface. Hence they are not reflected appropriately. We propose to compute the reflection with the help of another camera placed below the main one and looking up to the sky. We then have to make only a texture lookup to get the reflection fragment color. To render the interactions with the object we could store them like in (*Meersimulation In Anno 1404* 2008) into a displacement map and combine it with another one describing the ocean's shape. Another possibility is to combine the FFT approach from section 2.2.2 with the *Wave Particles* from section 2.3.1. Foam could be blended in based on the height of the wave, its surrounding slope and on the depth of the water as described in section 2.4.3. Caustics could also be faded in based on the depth of the water. Finally there should be a change in color of the water based again on its height.

5.2 Performance

58ms to render the plane is much too costly for an application having other objects moving on the screen. This result is due to the expensive computations done in the vertex and fragment shader, to the hardware on which we tested our application and the fact that we did not use any kind of level of detail. The plane has always the same amount of vertices describing it, no matter if viewed from far away or up close. Hence it is imperative to implement a level of detail solution, like the projected grid from algorithm 2, or the radial grid from (Pharr and Fernando, 2005, Chapter 18).

6 Conclusion

Although the result of our implementation did not produce convincing realistic water, it confirmed our intuitions. The Gerstner water model has to be combined with another one or completely discarded. Water rendering is a complex process involving many physical properties, each having to be approximated as discussed in section 2. Every one of them has to be carefully selected in order to avoid a major slowdown during rendering. Building an application from scratch is not recommended as too many concepts like the scene graph, need to be programmed.

As eighteen weeks ago we knew very little about the rendering pipeline and practically nothing about its programming, we are confident to obtain good results for our further work. We will try to implement a hybrid approach which handles shallow and deep water into *0 A.D.* using an enhanced projected grid like the one from (Kryachko, 2016). For the shallow water we think to use one precomputed Gerstner wave adapted to the curvatures of the beaches and stored as a displacement map. The fast Fourier transform from (Tessendorf et al., 2001) is rather appealing to use for deep water. This will be the main focus of our further work. If there is time left, we will add the other elements discussed below.

We think that blending the foam in an out based on the water surface height is amply sufficient. Interactions between objects and the water surface will have to be placed also

in a displacement texture and computed with an adapted form of the *Wave Particles* model.

References

Akenine-Möller, Tomas, Eric Haines, and Naty Hoffman (2008). *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., p. 1045. ISBN: 987-1-56881-424-7 (cit. on pp. 2, 9, 10).

Arvo, James (1986). "Backward ray tracing". In: *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*, pp. 259–263 (cit. on p. 9).

Claes, Johanson (2004). "Real-time water rendering-introducing the projected grid concept". In: *Master's thesis* (cit. on pp. 11, 12).

Darles, Emmanuelle et al. (2011). "A Survey of Ocean Simulation and Rendering Techniques in Computer Graphics". In: *CoRR* abs/1109.6494. arXiv: 1109.6494. URL: http://arxiv.org/abs/1109.6494 (cit. on pp. 2–4, 6, 7, 9, 10).

Fernando, Randima (2004). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education (cit. on pp. 4, 9, 10, 14, 18, 19).

Gallagher, Isabelle (Jan. 2010). "Autour des équations de Navier-Stokes". In: *Images des Mathématiques*. URL: http://images.math.cnrs.fr/Autour-des-equations-de-Navier-Stokes?lang=fr (visited on 01/02/2018) (cit. on p. 3).

Gerstner, F (1802). *Theorie der wellen: Abhandlungen der Koniglichen Bohmischen Gesellschaft der Wissenschaften* (cit. on p. 3).

Gonzalez-Ochoa, Carlos (2016). "Rendering Rapids in Uncharted 4". In: Naughty Dog (cit. on pp. 6, 11).

Gonzalez-Ochoa, Carlos and Doug Holder (2012). "Water Technology of Uncharted". In: URL: http://www.gdcvault.com/play/1015309/Water-Technology-of (cit. on p. 11).

Gonzato, Jean-Christophe and Bertrand Le Saëc (2000). "On modelling and rendering ocean scenes". In: *The Journal of Visualization and Computer Animation* 11.1, pp. 27–37 (cit. on p. 10).

Heckbert, Paul S and Pat Hanrahan (1984). "Beam tracing polygonal objects". In: *ACM SIGGRAPH Computer Graphics* 18.3, pp. 119–127 (cit. on p. 9).

Hudritsch, Marcus (Sept. 2017). *SLProject Features*. URL: https://github.com/cpvrlab/SLProject/wiki/SLProject-Features (visited on 12/31/2017) (cit. on p. 13).

Ivanoff, Alexandre (1972). *Introduction à l'océanographie: propriétés physiques et chimiques des eaux de mer*. Vol. 2. Vuibert (cit. on p. 10).

Iwasaki, Kei, Yoshinori Dobashi, and Tomoyuki Nishita (2001). "Efficient rendering of optical effects within water using graphics hardware". In: *Computer*

Graphics and Applications, 2001. Proceedings. Ninth Pacific Conference on. IEEE, pp. 374–383 (cit. on p. 9).

Jensen, Lasse Staff and Robert Golias (2001). "Deep-water animation and rendering". In: *Game Developer's Conference (Gamasutra)* (cit. on pp. 6, 10).

Jeschke, Stefan, Hermann Birkholz, and Heidrun Schmann (2003). "A procedural model for interactive animation of breaking ocean waves". In: (cit. on p. 10).

Karis, Brian and Epic Games (2013). "Real shading in unreal engine 4". In: *Proc. Physically Based Shading Theory Practice* (cit. on p. 17).

Kryachko, Yury A (2016). "Sea surface visualization in world of warships". In: *ACM SIGGRAPH 2016 Talks*. ACM, p. 22 (cit. on pp. 11, 12, 20).

Mastin, Gary A, Peter A Watterberg, and John F Mareda (1987). "Fourier synthesis of ocean scenes". In: *IEEE Computer graphics and Applications* 7.3, pp. 16–23 (cit. on pp. 5, 6).

Matei Zaharia's CS 488 Project (2006). URL: https://www.student.cs.uwaterloo.ca/~cs488/Contrib/w06/a5/mazahari.html (visited on 12/29/2017) (cit. on p. 13).

Max, Nelson L (1981). "Vectorized procedural models for natural terrain: Waves and islands in the sunset". In: *ACM SIGGRAPH Computer Graphics* 15.3, pp. 317–324 (cit. on pp. 4, 5).

Meersimulation In Anno 1404 (2008). URL: http://anno.de.ubi.com/devdiary.php?diary=devdiary_1404_2 (visited on 12/29/2017) (cit. on p. 20).

Munk, Walter H (1947). "A critical wind speed for air-sea boundary processes". In: *J. mar. Res.* 6, pp. 203–218 (cit. on p. 10).

Oceanic, National and Atmospheric Administration (Oct. 2017). *What is sea foam? Sea foam forms when dissolved organic matter in the ocean is churned up.* Ed. by National Ocean Service website. URL: https://oceanservice.noaa.gov/facts/seafoam.html (visited on 01/19/2018) (cit. on p. 10).

Perlman, Howard (Dec. 2016). *How much water is there on, in, and above the Earth?* URL: https://water.usgs.gov/edu/earthhowmuch.html (visited on 01/06/2018) (cit. on p. 2).

Pharr, Matt and Randima Fernando (2005). *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional (cit. on pp. 11, 20).

Premože, Simon and Michael Ashikhmin (2001). "Rendering natural waters". In: *Computer graphics forum*. Vol. 20. 4. Wiley Online Library, pp. 189–200 (cit. on pp. 10, 11).

Project Overview (2017). URL: https://play0ad.com/game-info/project-overview/ (visited on 12/29/2017) (cit. on p. 13).

SLProject Doxygen Documentation (Sept. 2017). URL: http://cpvrlab.github.io/SLProject_doc/html/index.html (visited on 12/31/2017) (cit. on p. 13).

Strugar, Filip (2009). "Continuous distance-dependent level of detail for rendering heightmaps". In: *Journal of graphics, GPU, and game tools* 14.4, pp. 57–74 (cit. on p. 11).

Tessendorf, Jerry et al. (2001). “Simulating ocean water”. In: *Simulating nature: realistic and interactive techniques. SIGGRAPH* 1.2, p. 5 (cit. on pp. 6, 7, 11, 20).

The Story of 0 A.D. (2017). URL: https://play0ad.com/about/the-story-of-0-a-d/ (visited on 12/29/2017) (cit. on p. 13).

Watt, Mark (1990). “Light-water interaction using backward beam tracing”. In: *ACM SIGGRAPH Computer Graphics* 24.4, pp. 377–385 (cit. on p. 9).

Weisstein, Eric W. (2018). *Navier-Stokes Equations. From MathWorld–A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/NavierStokesEquations.html (visited on 01/04/2018) (cit. on p. 3).

Williams, Hailey (May 2017). *Tutorial: Ocean Shader with Gerstner Waves*. URL: https://80.lv/articles/tutorial-ocean-shader-with-gerstner-waves/ (visited on 01/19/2018) (cit. on p. 20).

Yuksel, Cem, Donald H House, and John Keyser (2007). “Wave particles”. In: *ACM Transactions on Graphics (TOG)*. Vol. 26. 3. ACM, p. 99 (cit. on pp. 7, 8).

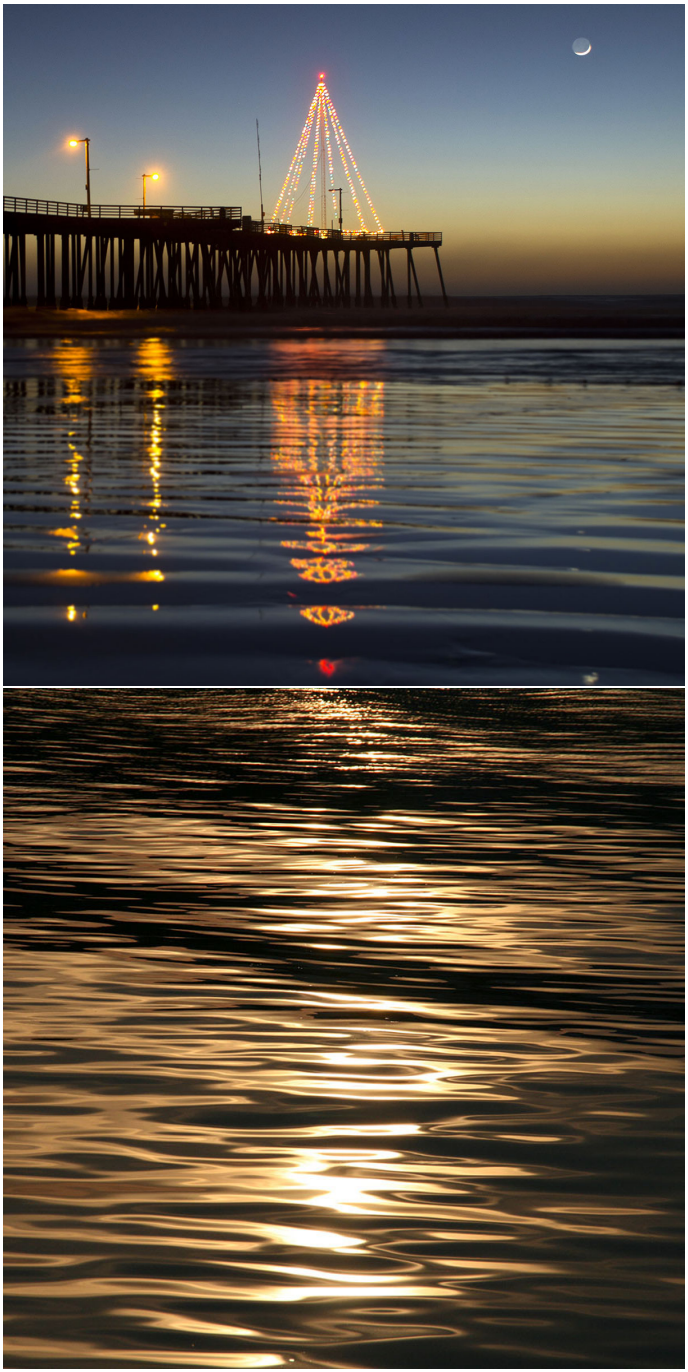


Figure 12: Reference images