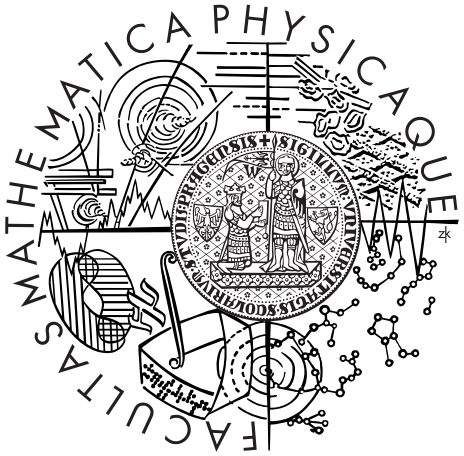
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Filip Krijt

Adaptive Simulation of Large-Scale Ocean Surface

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Petr Kadleček
Study programme: Computer Science
Specialization: Software Systems

Prague 2014

I would like to thank my supervisor Mgr. Petr Kadleček for his insights, his support and his willingness to dedicate a lot of time to the discussions about this thesis despite having a multitude of other responsibilities. I would also like to thank my family and my friends for being understanding about my preoccupation with the thesis and for supporting me in every way possible.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Adaptivní simulace rozsáhlého povrchu oceánu

Autor: Filip Krijt

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: Mgr. Petr Kadleček, Kabinet software a výuky informatiky

Abstrakt: Metody založené na fyzikální simulaci proudění kapalin a metody generování povrchu oceánu pomocí frekvenčních technik jsou v počítačové grafice dlouhodobě zkoumány. Jsou nicméně typicky používány odděleně, nebo bez jakékoliv možnosti vzájemné interakce. Tato diplomová práce se zaměřuje na možnost zkombinování obou metod do adaptivní metody navržením sjednocené reprezentace povrchu oceánu, prokládání výsledků simulačních metod a jednosměrné interakce mezi použitými metodami. Práce také předkládá několik možných budoucích vylepšení navržené metody a level-of-detail schéma založené na použití hardwarové teselace, jež může být použito nezávisle na konkrétních simulačních metodách.

Klíčová slova: fyzikální simulace, syntéza povrchu oceánu, proudění kapalin, teselace

Title: Adaptive Simulation of Large-Scale Ocean Surface

Author: Filip Krijt

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Petr Kadleček, Department of Software and Computer Science Education

Abstract: Physically-driven methods of simulating fluid dynamics and frequency-based ocean surface synthesis methods are of long-standing interest for the field of computer graphics. However, they have been historically used separately or without any interaction between them. This thesis focuses on the possibility of combining the approaches into one adaptive solution by proposing methods for unified surface representation, method result blending and one-way interaction between the methods. The thesis also outlines several future developments of the combined method and proposes a level-of-detail approach taking advantage of hardware tessellation that can be used regardless of what method was used for the simulation.

Keywords: physical simulation, ocean surface synthesis, fluid dynamics, tessellation

Contents

1. Introduction

1.1 Motivation

Despite the fact that the simulation of ocean water is a problem that has been well explored in the past decades and multiple models have been proposed by the scientific community, most of these models are suitable only for a specific part of the problem as a whole. On one hand, simulation of deep ocean waves was proved to be a problem that can be simulated with solid results using approaches based on gathering empirical data and transforming the ocean surface to comply with the data. The most famous method of simulating deep ocean waves is called the Tessendorf method, and its core is in taking a spectral model based on measurements done by weather buoys and transforming the spectral distribution into a fixed size height-field using inverse fast Fourier transform (FFT). The method is usable in real-time for very large water surfaces, but falls short when confronted with more complex water behaviours, such as interaction with floating objects, stormy conditions or shallow water wave simulation.

On the other hand, physically correct methods based on Navier-Stokes equations of fluid motion (NSE) have been shown to be able to produce complex and visually pleasing results, but the downside of the method is its high performance cost. This performance cost coupled with the huge volumes of water required for simulating an ocean convincingly make the method impractical to use for ocean simulation, especially in the context of real-time applications. However, the method is capable of handling a much more diverse set of cases – it can be used to simulate the shore and ocean floor influence on the approaching waves and can also be relatively easily modified to allow flooding of a previously dry area or interaction with buoyant objects.

These two methods have historically been developed and used mostly separately – the simulation engine was either based on some variant of the Tessendorf method, or on approximating the solution to the Navier-Stokes equations of fluid motion (or their derivatives, e.g. Shallow Water Equations, also known as Saint Venant Equations in their 1D variant). It is also apparent that the two methods have many complementary properties – the Tessendorf method is fast, tile-able and known to produce good results for open water scenes, but lacks in physically correct behaviour, especially near the shore, while the physically correct methods based on NSE are very accurate, but cannot be easily used for large scenes and carry an inherent performance overhead.

The purpose of this thesis is therefore in attempting to combine these two methods to create an adaptive scheme that allows for simulating large ocean scenes containing islands, using the Tessendorf method where applicable and transitioning to physical methods near to the shoreline and in other cases where the Tessendorf method would be unable to reproduce the water dynamic realistically. The transition between the methods must be as close to seamless as possible. No combined system was covered in [1], and the authors actually expressed a belief that such a system will emerge in the following years; however, to the best of our knowledge, no such system was described in any paper yet.

1.2 Thesis Goals

As stated above, the primary goal of the thesis is to combine deep and shallow water simulation methods into a system offering the advantages of both. To this end, we will research the most popular methods used for deep and shallow ocean simulation described in scientific literature and present a review of these methods, including their advantages and disadvantages from various viewpoints. We will then define criteria that must be taken into account when choosing the subset of these methods to combine, and then perform the selection itself based on these criteria. We will further propose a combination scheme for these methods and implement a pilot application including the selected methods and their combination.

However, the thesis also has several secondary goals. First, both the selection of the methods and their implementation should be done with eye towards game industry usage – the pilot application should ideally tend towards real-time framerates, at least on cutting-edge hardware. Second, the application should also isolate the used models from rendering as much as possible, to allow easy modification of used models or selection of another model altogether. To achieve this, we will propose and use a common way of rendering the resulting geometry regardless of the method being used. The application also needs to include basic support for data storage (such as reading terrain data from hard drive), basic rendering, and also a level of detail (LOD) system.

1.3 Technology

To provide reasonable results in terms of real-time usability of the method, the pilot implementation will be implemented in C++ using Microsoft Visual Studio 2012 and its compiler, which is the language and compiler used for most AAA games being developed in the game industry. We will also try to take advantage of modern technologies such as the DirectX 11 API, programmable HLSL shaders and general purpose graphics processing unit computing (GPGPU computing) via Nvidia CUDA in order to maximize the performance of the application.

1.4 Organization

This thesis is organized into several sections. First, we will summarize the previous research in the field of ocean simulation using information from selected papers. We will then move on to describing the available simulation methods in more detail, and then propose the scheme for combining the results gained from the chosen methods, defining the criteria we place on this combined system and justifying the method selection in the process. We will then move on to other important aspects of our combined solution, such as space organization and a level of detail scheme using modern GPU-side hardware tessellation. We will also include the technical documentation for our pilot application. Finally, we will review the performance results of the application and conclude with summarizing the results of our combination method, and propose future modifications.

2. Previous Work

In this chapter we present the research of available ocean simulation techniques. We start by a general review of all methods based on a survey research paper by E. Darles et al., following its structure and adopting its method categorization, then move on to detailing the work of Jerry Tessendorf which serves as a standard publication on the inverse FFT based deep water simulation method. We will then illustrate the potential of combining various methods and mention the performance problems of using the computational fluid dynamics (CFD) approach by reviewing the implementation and real-time focused system paper by Jensen and Goliáš before focusing solely on CFD method in the form of SWE (Shallow water equations), in particular the variant used by Chentanez and Müller.

2.1 Reviewing Known Methods

The principal paper on ocean simulation is a review of all known methods to date, compiled in 2010 by E. Darles, B. Crespin, D. Ghazanfarpour and J.C. Gonzato [1]. The paper contains methods used for the simulation itself, rendering, and various additional effects such as foam and spray. The authors divide the wave simulation methods into two categories based on the area of occurrence – deep ocean waves and shallow water waves – and describe the relevant methods separately. In order to ease the understanding of this thesis, we have adopted a similar classification of the methods and use it as a frame of reference in the next chapter. We have however renamed the categories from "Deep Ocean Waves" and "Shallow Ocean Waves" to "Empirical" and "Physical", respectively. This is because we believe that the methods by themselves are not defined by their usage but rather by the nature of the operations composing them, as we can theoretically use a physical method for deep ocean simulation on current hardware, even though the performance costs make such an approach unsuitable.

Ocean Simulation

The first part of the paper focuses on deep ocean waves, and describes two main method categories for solving this problem. The first approach is based on directly calculating the position of each point of surface approximation using parametric equations, the second uses spectral model to describe the ocean surface and inverse Fast Fourier Transform (FFT) to generate the heightfield based on these data. The paper also discusses suitability of the methods for GPGPU implementation and various modifications to offset the disadvantages of the methods, such as artificially skewing the shape of the waves to make them appear more choppy, and generating water sprays where applicable.

In the next part of the paper, the authors discuss the methods for simulating waves in shallow water. The approaches outlined in the previous chapter are reported as being unsuitable for shallow water simulation due to their inability to respond convincingly to various more complex situations. The primary method for simulating shallow water waves is based on the Navier-Stokes equations of fluid motion, which represent a physically correct description of viscous fluid

flow. To find an approximate solution of the equations in a given environment, the equations are discretized using one of two discretization schemes. The first scheme, called Eulerian in the literature, divides the simulation space in a 2D or 3D grid of cells, with each cell holding the properties of the fluid and calculations being performed on a per-cell basis, representing the fluid moving from cell to cell. The second approach, called Lagrangian, uses particle systems to represent the fluid, with each particle carrying a small volume of fluid and holding the fluid properties. While both approaches can be used for simulating the ocean volume, a common solution is using the Eulerian method for the surface generation itself, with the addition of Lagrangian particles representing the small scale details such as water spray. As with the deep water methods, the authors discuss various modifications to the general scheme, which are too detailed for the purpose of this thesis.

Another important scheme discussed here is a restriction of the NSE to water columns called the Shallow Water Equations, also known as the Saint-Venant Equations in its 1-dimensional variant. The restriction assumes that the horizontal scale of the fluid volume is much greater than the vertical scale and that fluid properties do not vary with depth for a specific water column. This allows a simplification to a 2D grid instead of a full-fledged 3D grid simulation, bringing a solid performance boost. The simplification of course brings some realism problems as well, but is generally favored as one of the more real-time capable NSE-based methods. All simulation methods mentioned up to this point will be described in greater detail in the next chapter, including some modifications and a review of advantages and disadvantages.

Rendering

The second part of the paper deals with realistic ocean rendering and lighting, as well as a summary of techniques used for generating additional visual effect, such as foam. Usually, such effects are based either on empirical approximations, which is the case of a method proposed by Jensen and Goliáš [2], or on Lagrangian particle systems mentioned above. Generally, such effects are applied in a post-processing step, based on ocean surface data generated using one of the methods already described in the preceding paragraphs. The remaining text in the chapter focuses on lighting itself. As the ocean is an irregular semi-transparent surface, to simulate the interaction of light with such a surface in a physically correct manner would require a ray-tracing engine. As such an engine is usually considered too demanding for real-time usage, implementations usually resort to approximations of light rays' reflection and refraction. The paper focuses on these methods, ranging from simple environment mapping to multiple-order approximation schemes, some of which are even taking chemical and biological properties of the ocean water into account. Overall the paper offers a broad overview of rendering techniques; however, the paper by Bruneton, Neyret and Holzschuch [3] offers a complete state-of-the-art ocean rendering system including lighting LOD methods and seamless transition from geometry to BRDF, making it the principal paper on ocean rendering for us.

Finally, it is worth mentioning that the review actually notes the separation of deep water and shallow water simulation methods and predicts that schemes

attempting to combine the two areas will emerge in the future.

2.2 Simulating Ocean Water Using the Tessendorf Method

In spite of not really being a research paper but rather a set of course notes, the work of Jerry Tessendorf [4] has essentially become the defacto standard for understanding the FFT-based spectral approach, being quoted in most papers based on spectral methods. The method is also widely referred to as the "Tessendorf method" in the community. The notes are more in-depth than most papers are and offer a solid basis for implementing an FFT ocean simulation.

After a short introduction and definition of the scope of the paper, as well as some notes about the limitations of the chosen methods, Tessendorf first focuses on the rendering of the water surface, in particular on the reflection and refraction of the incident light rays. The algorithm used here is one of the more simple ones, being a first-order approximation disregarding more than one reflection or refraction from the surface point. Apart from a helpful discussion of various light interaction cases, the section offers little information relevant to this thesis.

The second section of the notes is the most important one. Tessendorf explores "practical" ocean wave algorithms, which we surmise means "usable in real-time" in this context. He first describes the wave simulation based on the Gerstner theory in detail, illustrating the method first on one wave and then on multiple waves combined. He then moves on to illustrate the dispersion relation between the wave frequency and wave vectors, and how to take advantage of this relation to create a continuous loop of ocean surfaces for animation. He notes that the same method can be applied to FFT generated surfaces as well before moving on to explaining the spectrum-based FFT method in great detail. After first explaining the general idea and the role of FFT in converting the spectral distribution of the ocean into a spatial domain heightfield, Tessendorf provides details on the slope calculation and describes a simple oceanographic spectrum called Phillips spectrum. He also notes the artificial periodicity introduced by tiling the resulting heightfield, and conditions under which this periodicity is apparent to the viewer. The author included experience-based suggestions for the value of specific parameters (such as the size of the FFT grid), and discusses how these affect the visual quality of the outcome. He further provides a set of experimental results gained by applying the method with various parameters and shaders. Again, the method itself will be described in detail in the next chapter of this thesis.

Tessendorf further moves on to detailing a modification of his algorithm – the method he described up to this point produces waves with rounded tops, which are suitable for mild weather conditions. This method of creating more "choppy" waves is based on applying a displacement field on the resulting surface mesh, with the calculation of this field dependent on the heightfield result from the standard Tessendorf approach – the underlying spectrum is not modified, which means the method can be integrated with existing Tessendorf solutions without requiring many changes in the original code.

The rest of Tessendorf's work deals with rendering the resulting surface, sug-

gesting methods for approximating the reflection and refraction of light on the water surface, as well as volume effects including caustics and godrays. As before, while the section offers a comprehensive solution, we would refer to [3] for a fully detailed lighting and rendering model.

2.3 Deep Water Animation and Rendering

The work of Jensen and Goliáš describes a very comprehensive ocean simulation and rendering system capable of real-time performance. The model is a mix of various oceanographic methods, including the Tessendorf method, NSE, SWE and surface wave models; however, the model is not a combination of these methods in terms of using them for simulating different spatial areas of the simulation domain, as is the case with our model. Instead, the methods are used for simulating surface details of deep water waves on different scales vertically, i.e. the shapes of the waves are created using the Tessendorf method, NSE are solved in 2D to create a bumpmap that is applied over the surface to simulate fine surface details and surface waves are used for creating object-generated waves by superimposing them over the Tessendorf waves geometry. The work also includes a section on first-order approximation rendering with the addition of caustics, as well as a section focused on additional details, particularly foam and spray. The survey performed by E. Darles et al. mentions this paper mainly in context of the foam simulation part.

Models and Mixing

The first part of the work deals with surface generation and animation. The bulk of the simulation is based on the Tessendorf method, using the same approaches as outlined in [4], including the choppy waves modification. The authors describe all models comprising the simulation in great detail, including formulas and their derivations, as well as properties, use cases and limitations of the methods.

The only method included that was not mentioned up to this point is the "Surface waves" model. The model is even more restrictive than SWE, assuming fixed depth across the whole simulation domain in addition to the restrictions already imposed by SWE. The result is a method that is too simple to accurately simulate deep and shallow water waves, but can be used for small scale details and interaction with floating objects. The method is reportedly very fast.

The work then moves on to the actual mixing of the models. As we have mentioned above, the combination is performed in terms of levels of detail, not geographic circumstances; the method therefore remains a strictly deep-water simulation method. In addition to using the Tessendorf method for generating the geometry, the method uses 2D NSE calculation to generate surface details indicative of turbulent surface tension, applying them as a bump map. The authors considered whether to use SWE or Surface waves for waves generated by interaction with floating objects and eventually decided to go with the Surface waves model, as they felt it produced better looking results in this specific scenario. The actual combination of the Tessendorf method and the Surface waves consists of a superimposing the results of Surface waves simulation over the ocean geometry. While not physically correct in any sense, this works presumably because of the

differing scales of the waves, with the Surface waves being much smaller than the Tessendorf geometry.

As the generation of waves by the buoyant object is only half of the ocean – object interaction, the authors next focus on simulating the buoyancy of an object and the effect the surface waves have on its movement. As with everything else in this work, the focus is on real-time usability, with the core idea being the approximation of the floating objects by multiple patches, and the calculation of buoyancy forces for each patch. The calculated force is then applied to the object using the traditional methods of rigid body dynamics. The generation of surface waves is done by applying damping and displacement directly to the relevant cells. The authors note that with correct parameter selection, both splash and trail waves can be achieved.

Rendering, Foam and Spray

The rest of the work deals with rendering and graphical details. The rendering uses the standard first-order lighting approximation, but contains interesting additional effects such as caustic surface details and godray simulation. As this thesis is not primarily focused on rendering, we will not go into the details here. We will note however, that while the rendering methods proposed here are only first approximation of the lighting, the results look convincing and the additional details presented here add to the visual impact of the ocean surface.

The final section of the paper deals with foam and spray. The spray is implemented using standard particle systems, for the foam however, authors note that while using particle system as well would be an option, the performance cost might be noticeable. To maintain the real-time focus of their approach, the authors propose a method based on generating a foam texture in real-time and then overlaying this texture over the ocean texture using additive alpha blending. The amount of foam is defined per vertex and interpolated in pixel shader, and is used as a transparency factor for blending with the ocean texture. The spawning and dissipation of foam is based on the slope difference of the vertex and its neighbours – in case the difference is less than a chosen negative limit, the foam factor is increased, generating more foam, otherwise the foam amount is decreased by a constant, resulting in gradual linear dissipation of the foam.

2.4 Shallow Water Simulation

The paper Real-time Simulation of Large Bodies of Water with Small Scale Details [5] by Chentanez and Müller from Nvidia focuses solely on SWE-based methods. The paper is motivated primarily by the inability of classical SWE implementations to simulate water effects that can not be modelled using a heightfield. The authors therefore choose to combine a standard SWE solver and particle systems to overcome this limitation. This system can therefore be seen as combination of Eulerian and Lagrangian approaches to NSE space discretization. In addition, the authors propose several modifications to the SWE solver that allow for higher simulation stability and reacting to boundary conditions, which is the part of the work that is of particular interest to us in context of this thesis.

Solving SWE

The first part of the paper described the SWE method and the solver used in the simulation. The authors note that while the use of implicit integration in SWE has been shown to produce unconditionally stable results, such methods are very demanding performance-wise. They have therefore chosen to use explicit integration methods and take extra steps to increase the stability of the simulation.

The authors opt for using the standard staggered grid configuration (as illustrated in Figure 2.1), where the properties of the fluid are stored at the centers of the cells, while the velocity components are stored at the faces separating the two neighbouring cells. The self-advection of the velocity field is solved using the MacCormack integration method with a fallback to the common Semi-Lagrangian method, for more information on which the authors refer to [6]. The explicit integration scheme the authors used for integrating changes in height and velocity is described directly and in enough detail to allow reproduction of the system.

The next section of the paper describes the modifications that were applied to the standard SWE solver. The first of these is the treatment of boundary conditions. The method allows for marking arbitrary cell face as reflective – such a face will have velocity set to 0 and will not be updated during the velocity integration step. The authors also dynamically mark the cells satisfying certain conditions as reflective, simulating the situation where the terrain height of neighbouring cell lies higher than the water surface of the current cell. In addition to reflective faces, the authors also introduce absorbing faces. The core idea is in applying damping to waves approaching the absorbing face so the waves reaching the faces have negligible amplitudes. We take note of this approximation of absorption and use it for the SWE-Tessendorf transition border further in the thesis in Chapter 5.3.1.

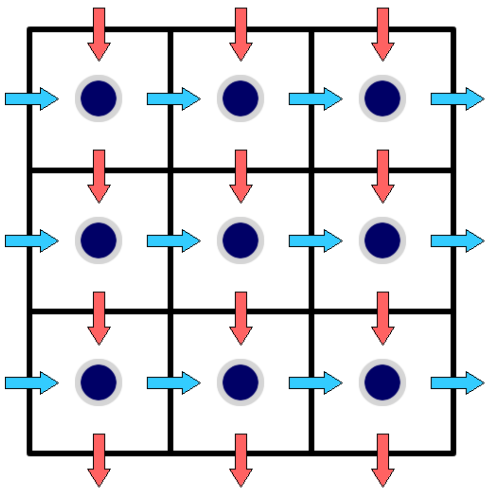


Figure 2.1: Illustration of the Marker and Cell (MAC) configuration of the Eulerian grid, with fluid properties being represented by blue circles and blue and red arrows representing the horizontal and vertical velocity components, respectively.

Stability Improvements

The authors then move on to describing the stability enhancements they have introduced into the SWE solver, defining various clamping ranges for different simulation parameters. The first stability improvement is clamping the height of the fluid in a cell to non-negative values, which can sometimes appear due to floating point inaccuracies. The authors also limit the velocities to a certain magnitude based on the cell spacing and timestep, as well as placing artificial limitations on wavelengths and amplitudes in deep water scenarios. While noting that these modifications reduce the quality of the results in some scenarios, the authors apparently consider the tradeoff worthwhile.

Eulerian to Lagrangian Simulation

The rest of the paper deals with the combination of the SWE solver with a particle system approach. The core idea is that the system automatically detects unstable heightfield shapes and turns them into Lagrangian particles carrying the fluid properties. Upon contact with either the terrain or water surface, the particles are again absorbed into the SWE simulation domain. While the proposed method is very well described and seems to offer visually appealing results, especially in the scenario of simulating vertically moving bodies of water, such as waterfalls, it is out of the scope of this thesis.

Open Water Simulation

The work also includes a short section on surrounding the SWE simulation with Tessendorf method surface; however, the scenario in which this combination takes place offers several liberties. First and foremost, there is no interaction at all between the two domains – the Tessendorf result does not impact the SWE in any way, and the SWE uses damping for the waves approaching the border. We also assume that linear interpolation is used between the method results from a rendering perspective – this is hard to determine however, as the authors do not go into much detail, and only include a small screenshot of the scenario, as it is only a marginal case for a work primarily focused on rivers.

3. Simulation Methods

After presenting the review of available methods and some state-of-the-art modifications in the previous chapter, we move on to the simulation methods themselves. As we have mentioned before, we use a similar categorization to that in the work of Darles et al. [1], only changing the names of the categories to better reflect our point of view. We will describe each method in greater detail than we have done in the previous chapters and include a paragraph detailing its known advantages and disadvantages for each method.

3.1 Empirical Methods

This category contains methods based either on empirical formulas or on historical oceanographic measurements. The survey by Darles et al. [1] uses the name "Deep water waves" for this category due to the typical usage of its representatives in simulating large deep water surfaces and their inability to simulate shallow water phenomena.

3.1.1 Parametric Equations

First approach used for simulating ocean surface is based on the idea that the surface can be approximated by a set of particles representing a height-field, and that the motion of each particle of this surface can be directly described by equations dependent on time and horizontal position variables. The method is first used by Max in [7], where the equation used for representing a single particle motion is based on evaluating a series of sinusoids of various amplitudes. The height (y coordinate) at time t of a particle with horizontal coordinates x and y is computed as given:

$$h(x, z, t) = -y_0 + \sum_{i=1}^{N_w} A_i cos(\boldsymbol{k}_{i_x} x + \boldsymbol{k}_{i_Z} z - w_i t) \tag{3.1}$$

where x and z are coordinates of a surface point, t is the simulation time and y_0 the height of the surface at rest. An individual wave is described by its amplitude A_i, its wave vector $\boldsymbol{k}_i$ and its pulsation w_i. The shape of a wave is in this case determined by the product of wave amplitude A_i and the norm of the wave vector K_i. For realistic motion, the product $p = A_i K_i$ must be either less than 0.5 (resulting in a trochoid movement) or equal to 0.5 (resulting in a cyclical movement). The exact process of calculating the wave vector has seen some research in the literature, with the original paper using an assumption of infinite depth and calculating the magnitude of the wave vector as follows:

$$K_i = 2\pi / \sqrt{\frac{gL_i}{2\pi}} \tag{3.2}$$

where g is the gravitational constant and L_i the wavelength of i-th wave.

An alternative formula for calculating the wave vector has been proposed by Peachey [8]. The author builds upon Airy Wave Theory and introduces a depth

parameter, thus removing the limitation of the original equation. The modified equation is as follows:

$$K_i = 2\pi / \sqrt{\frac{gL_i}{2\pi} tanh \frac{2\pi d}{L_i}} \tag{3.3}$$

where d is the depth of point related to the bottom of the sea.

Another parametric approach described in the literature is based on the Gerstner wave theory. This theory is a ocean wave approximation first derived by František Josef Gerstner in 1804, its first recorded usage in computer graphics according to [4] being in the work of Fournier and Reeves [9]. The Gerstner theory describes the motion of a surface particle due to a single wave using the following set of equations:

$$\boldsymbol{x} = \boldsymbol{x_0} - (\boldsymbol{k}/|\boldsymbol{k}|) A sin(\boldsymbol{k} \cdot \boldsymbol{x}_0 - \omega(\boldsymbol{k}t), \tag{3.4}$$

$$y = A cos(\boldsymbol{k} \cdot \boldsymbol{x}_0 - \omega(\boldsymbol{k})t). \tag{3.5}$$

$$|\boldsymbol{k}| = 2\pi/\lambda \tag{3.6}$$

$$\omega(\boldsymbol{k}) = \sqrt{g \cdot |\boldsymbol{k}|} \tag{3.7}$$

where $\boldsymbol{k}$ is the wave vector, t the current simulation time, A the amplitude, $\boldsymbol{x}$ the horizontal position of a surface point, $\boldsymbol{x}_0$ its horizontal position at rest, ω the frequency of the wave and λ the wavelength. These equations can be easily modified to allow simultaneous effects of multiple waves, as shown in [4].

According to the survey by Darles et al. [1], Fournier and Reeves modify this model by replacing the circular motion of the particle used in Gerstner theory by an elliptic motion and also add dependency on the topological changes in the seabed. Additional papers have researched this model, adding modifications allowing for greater realism, such as Gonzato and Le Saëc [10].

Overall however, the method of using parametric equations doesn't hold many advantages. Tessendorf reports that the oceanographic literature tends to downplay Gerstner wave theory (a prominent representant of the parametric approach) due to its low realism, and favors the spectral approaches described in the following chapter [4]. This makes sense, as the parametric methods are by nature a rough approximation, as opposed to spectral methods, which are generally based on empirical measurements gained from real-world ocean environments, and methods such as NSE, which offer physically correct surface calculation. However, the parametric approach reportedly allows for far greater direct control of the resulting surface shape than the spectral methods [11].

3.1.2 Spectral Approach

The Tessendorf Method

In contrast to the parametric approaches outlined above, the spectral model does not concern itself directly with the position of each point. Instead, the model describes the distribution of wavelengths and amplitudes appearing under specific conditions (describing the weather, type of ocean, etc.). By itself, such a model can not be used to generate the shape of the ocean surface. However, when

combined with an inverse Fast Fourier Transform capable of transforming the data from the spectral domain to the spatial domain representing essentially the uniformly spaced signal samples, this model is capable of generating fixed-size surface tiles.

Currently, the most popular spectral methods are all originally based on the work of Jerry Tessendorf, in particular on his course notes "Simulating ocean water" [4] introduced in the previous section. The method has been widely used both in the movie industry, where it was famously used to animate the ocean in the movie Titanic, and in computer graphics simulations. Even in its original form, the Tessendorf method, as it is widely called, was capable of reaching real-time framerates, albeit with relatively small grid size of 64 (Titanic used a grid of 2048). Since the introduction of this method, not only has the hardware improved, but several methods for increasing the performance of the Tessendorf method have been proposed, either using GPGPU computing for the FFT calculation, resulting in a respectable speedup, or implementing LOD methods to only calculate what is necessary. Currently, some implementations are capable of using the grid size of 1024 on a consumer-range hardware. Further increases in the grid size are complicated however, as the floating point inaccuracies reportedly influence the quality of the results starting from the size of 2048, as reported in the Tessendorf implementation presentation by Nvidia [12].

Properties of the Tessendorf Method

One of the advantages of the Tessendorf method is its inherent tile-ability. Because of the properties of the Fourier Transform, the opposite edges of the heightfield will always be symmetric. This allows us to place two heightfields adjacent to each other without producing a visible seam, or crack in the geometry. This ability to tile the ocean surface indefinitely (at least theoretically) plays an important role in the usability of the method, as it allows simulating large ocean surfaces without the need for additional computations. There is a tradeoff however, as the artificial periodicity introduced by the tiling becomes visible when viewed from further away. There are ways to offset this effect, such as using LOD methods to adapt the size of the grid to the observer distance, or introducing a noise function to hide the periodicity. Some implementations, such as [12], have successfully used Perlin noise to remove the regular appearance of the surface.

As it was described until now, the Tessendorf method is capable of synthesizing a fixed-size heightfield grid of convincing ocean surface at time $t0$. However, as the method is primarily intended for use in audio-visual media such as movies and videogames, the surface synthesis on its own is not enough – we also need a way to convincingly animate the said surface. In his original text, Tessendorf proposes an animation method that produces solid results and has been used in more or less unchanged form ever since. The animation is based on phase-shifting the waves in the original spectrum based on the difference of current time and $t0$, thus creating the illusion of movement for a particular wave. Since we are phase-shifting a wave with a fixed wavelength, each wave will also eventually return to its original configuration, creating a loop. The entire surface is therefore also looped after a fixed amount of time.

Oceanographic Spectrum

While the conversion from the spectral domain to spatial domain remains virtually the same across all variants of the method (disregarding implementation details such as the usage of GPU for FFT calculations), multiple oceanographic models have been experimented with in the literature. In his original notes, Tessendorf used the Phillips spectrum, notable for its easy description. While the same spectrum remains in use in many implementations, some authors have suggested the use of other, more sophisticated spectra. Apart from the traditional Pierson-Moskowitz spectrum, literature fairly often references the JONSWAP (JOint North Sea WAve Project) spectrum. As the JONSWAP was developed based on empirical measurements of fetch-limited seas and was built to simulate the same seas, it includes the fetch length [1] as a parameter. In the work of Lee at al. [11], another spectrum has been suggested for use in the field of ocean simulation. The TMA (Texel, Marson, Arsole) spectrum is an extension of the JONSWAP spectrum, introducing another parameter for the depth of the sea. While the authors of the paper note that the TMA is superior to more simple spectra such as Phillips and Pierson-Moskowitz due to having more user-controlled parameters, the actual benefit for simulating ocean waters is debatable, as the spectral methods are unsuitable for simulation of shallow waters regardless of the spectrum used, due to their inability to accurately react to the seabed and the shore. However, using such a spectrum might be useful for some LOD approaches, as well as for situations outside the ocean simulation field, such as modelling lakes or ponds.

Summary

To summarize the advantages of the Tessendorf method, the method by itself is very fast in comparison with both the parametric approaches and the physically-based methods while producing fairly convincing results for deep water scenarios. The properties of the Fast Fourier Transform also mean that the resulting surface is tile-able, making large ocean scenes possible, while at the same time allowing for visually pleasing animation loop using phase shifts. The disadvantages of the method are also quite apparent. The way the heightfield is generated means that the user can control the appearance of the surface only via the spectral parameters, if any are present for the chosen spectrum, not directly as is the case with parametric methods. As the surface shape is dependent only on the underlying spectra it is also very hard to modify the method to allow any additional influence over the ocean surface, such as reacting to floating objects or the the terrain itself. While some papers have tried to overcome these limitations (e.g. Jensen and Goliáš have included an approximation of two-way object-ocean interaction), these modifications are almost always very limited.

[1] The length of the surface over which the wind has blown

3.2 Physical Methods

Unlike the empirical approaches, physically-based methods are not specific to simulation of ocean waters, but instead follow common physical principles. In general, it can be observed that with the ongoing evolution of consumer-range hardware, interactive applications are slowly moving away from the specialized empirical methods to universal physical engines that drive most of the interactions in the simulated world. An example of this trend can be seen in the field of rigid body dynamics, where the general physical solvers are now standard for most of AAA videogame and movie titles. In the case of ocean surface simulation however, the performance cost of simulating the entire body of water is still too high to completely replace the empirical methods. In this chapter, we present some general methods from the field of Computational fluid dynamics, as well as their restrictions tailored specifically for ocean surface waves.

3.2.1 Navier-Stokes Equations

The primary method from the field of CFD is based on the set of equations known as the Navier-Stokes equations of fluid motion. The equations are derived by applying Newton's second law to continuous fluid motion and are known to describe viscous fluid flow. In the case of in-compressible fluid, which is the case usually assumed for large continuous bodies of fluid [1], the equations are simplified to the following form:

$$\rho\left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\bigtriangledown\boldsymbol{v}\right) = -\bigtriangledown p + \mu\bigtriangledown^2\boldsymbol{v} + \boldsymbol{f} \tag{3.8}$$

$$\bigtriangledown\boldsymbol{v} = 0 \tag{3.9}$$

Where ρ is the density of the fluid, $\boldsymbol{v}$ its velocity, t time, p its pressure, μ its viscosity and $\boldsymbol{f}$ other body forces. The first equation represents the application of Newton's second law and guarantees the conservation of momentum, while the second guarantees the conservation of mass.

While the NSE together capture the dynamic behaviour of viscous fluid flow from a theoretical standpoint, the actual solving of the equations is a very complicated matter, as these are a set of high-order partial differential equations. The literature therefore commonly resorts to discretizing the equations both in time and space to find an approximate solution [2]. As a detailed review of all methods used for approximating the NSE would require at least the scope of a scientific paper, we will resort to describing only the most common methods and those relevant to this thesis. Specifically, we will make no further mention of the particle-based Lagrangian discretization mentioned in the review of the survey by E. Darles et al., and we will focus solely on the grid-based Eulerian approach which is the common choice for ocean simulation in the literature.

The Eulerian approach of space discretization is based on a cell model – the simulation space is divided into uniformly-sized cells possibly containing some small volume of the fluid, and the calculations are performed on a per-cell basis. Two configurations are common in the literature – the collocated grid configuration, where both the fluid properties and the velocity vectors are stored at the

center of the cell, and the staggered grid model, often referred to as the MAC (Marker and Cell) configuration. Of the two configurations, MAC has seen more widespread use recently, as the collocated grid configuration allegedly suffers more from stability problems. In the staggered grid model, only the fluid properties are stored at the cell center, and the velocities are stored at the boundary faces dividing the neighbour cells. For visual comparison of the two grid configurations, see Figure 3.1

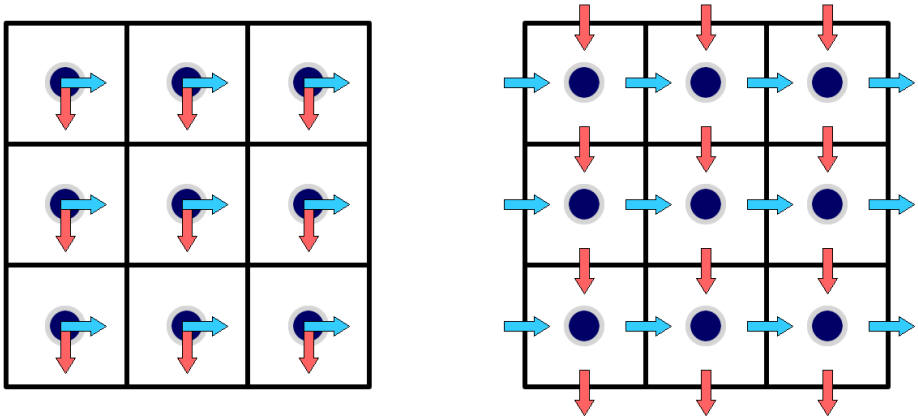


Figure 3.1: Comparison of the collocated (left) and staggered (right) Eulerian grid configuration, with fluid properties being represented by blue circles and blue and red arrows representing the horizontal and vertical velocity components, respectively.

As for time discretization, the most important factors influencing the quality of the result are the selection of integration scheme and the length of the time step. Both of these factors are in essence a tradeoff between speed and accuracy, and must therefore be chosen and implemented carefully in a real-time media such as a video game. Several integration schemes have been used for discretizing the NSE, the most common of these being the Semi-Lagrangian integration scheme. Essentially being an application of the backwards-Euler integration to particle-based simulation, the scheme has very good performance and is thus often used in situations calling for real-time framerates. The system proposed by Jos Stam [13] is an oft-quoted example of Semi-Lagrangian integration. Another widely used integration scheme potentially providing more accurate results is the MacCormack scheme, which is described in [6].

The properties of NSE are highly dependent on the exact scenario and the chosen discretization. In general however, it can be said that the NSE implementations fully capture the complex behaviour of viscous fluid flow [1], directly allowing for many effects that are unattainable with the empirical methods, such as separation of the fluid from the main body, flooding of previously dry areas, interaction of multiple fluids with one another, easy integration of two-way interaction with floating objects, etc. Unfortunately, providing such a complex fluid simulation also means that solving the equations is a notoriously hard problem. Jensen and Goliáš report the existence of solvers capable of $o(n^4)$ time complexity, but state that even this is too demanding for real-time purposes [2]. When

considering the huge volumes of water that would need to be simulated in the case of the ocean (including the entirety of its depth), it becomes obvious that the NSE can not be reasonably expected to run in interactive framerates in the case of the ocean.

3.2.2 Shallow Water Equations (Saint-Venant)

The Shallow Water Equations, also known as the Saint-Venant equations are derived from the full Navier-Stokes Equations of fluid motion. First used by Kass and Miller [14], the SWE take a simplified approach to simulation the ocean surface. As opposed to full NSE, which use uniform grid of cells across the whole 3D simulation domain, the SWE consider only a 2D range of cells, with each cell holding as a property the height of the fluid column instead of the fluid density. A visual comparison of the way the SWE divide and interpret the simulation space compared to the NSE can be seen in Figure 3.2. The reduction at the core of the SWE depends on several assumptions. First and foremost, it is assumed that the fluid shape can be modelled using heightfield – thus eliminating any possibility of vertically separated fluid bodies and indeed most fluid effects based on vertical velocity. Second, fluid properties usually carried in the NSE cells, such as the fluid density, pressure and velocity, are assumed to be constant across the whole fluid column. It is worth noting that while the method only models the horizontal velocity of the fluid, the vertical velocity is also simulated, albeit indirectly, by the height property changes Δh of each cell. Finally, the horizontal scale is assumed to be larger than the vertical scale, making the method less accurate and stable for deeper regions, hence the name. A form of SWE as used in [5] is as follows:

$$\frac{Dh}{Dt} = -h \bigtriangledown \cdot \boldsymbol{v} \tag{3.10}$$

$$\frac{D\boldsymbol{v}}{Dt} = -g \bigtriangledown \boldsymbol{\eta} + \boldsymbol{a}^{ext} \tag{3.11}$$

where h is the height of the fluid, D is the material derivative operator, g is the gravitational constant, $\eta = h + h_{terrain}$ is the absolute surface height, $\boldsymbol{v}$ is fluid velocity and $\boldsymbol{a}^{ext}$ represents external acceleration due to other forces.

Due to being in essence their restriction, the SWE share many traits with the NSE implementation-wise. Same approaches are applied to discretizing the equations in space and time. In particular, the SWE are almost always solved using the Eulerian grid approach, with the same configuration possibilities as in the NSE case. The typical implementation as being described in [5] consist of three steps, first self-advecting the velocity field itself, and then integrating the height and velocity forward in time. As with the NSE, the prevalent integration schemes for real-time application seem to be the Semi-Lagrangian approach used by Stam, and the MacCormack second-order scheme.

Unlike the NSE, the assumptions made in the SWE derivation make the method much more suitable for generating the ocean surface shape, as it is possible to simply use the fluid height values to generate the heightfield, whereas the literature has seen a lot of research regarding a visually convincing way of generating the surface based on the result of NSE calculations [1].

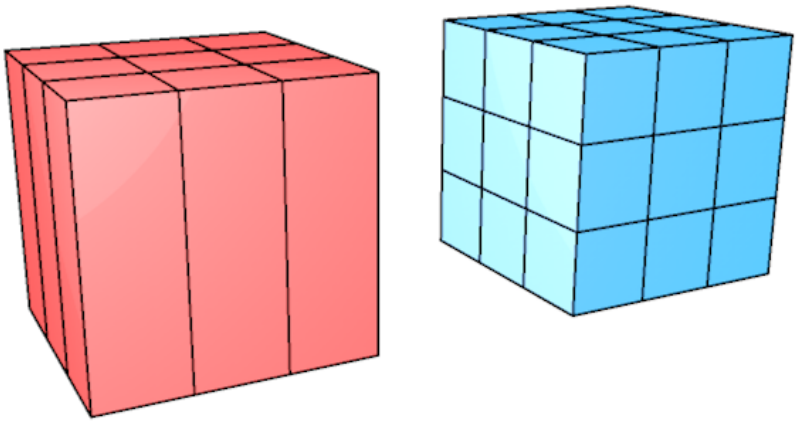


Figure 3.2: Visualization of the way the SWE (red) divide the simulation domain space compared to the NSE (blue).

Overall, the SWE seem like a good compromise between accuracy and performance. Despite losing the ability to simulate all fluid phenomena, especially under-surface flows and tensions as well as breaking waves, the method still exhibits the most important properties of the NSE, such as ability to react to terrain and floating objects. The 2D restriction allows us to dramatically reduce the computational complexity of the problem as well as its size, resulting in much better performance. However, as the method is devised for shallow water simulations, it can not be used as the only method in a large scale ocean surface simulation, the performance considerations of such an approach notwithstanding.

3.2.3 Surface Waves

For the sake of a complete review, we mention another method derived from the NSE. The Surface waves are a model that takes the simplification of SWE even further, assuming fixed depth across the whole simulation domain in addition to the restrictions introduced by the SWE. The method is described in the work by Gomez [15] and mentioned in Jensen and Goliáš [2], who use it as a supplementary model for generating small object-induced surface waves in addition to the large-scale ocean surface generated by the Tessendorf method. The form of the equations as used by Jensen and Goliáš is this:

$$\frac{\partial^2 h}{\partial t^2} = |V|^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) \tag{3.12}$$

where $|V|$ is the velocity of the wave across the surface, h is the height of the water surface, t is the time and x and y the horizontal coordinates.

While the model is too simplified to allow any meaningful interaction with terrain, the relatively lightweight character of the method performance-wise and the ease of implementation make it suitable for small-scale details and surface interaction in the manner explored by Jensen and Goliáš. Apart from that use case however, the methods mentioned up to this point appear superior in most regards.

4. Method Criteria

Having described the most popular methods in the field of ocean simulation, we now present the criteria we have formulated for the selection of two methods to be used in this thesis for our combined model, the exact form of the selected methods along with the reasons for choosing them.

4.1 Defining the Criteria

1. Large-scale Capability As the primary goal of the thesis is to simulate large ocean waters, the selection of methods must be appropriate for such usage. This limitation means that we can not, for example, base the entire model on NSE, as the volumes of water required to simulate deep ocean surface would be huge, not to mention the inefficiency of such approach.

2. Ability to React to Terrain in a Physically Convincing Manner The main problem with most existing deep ocean simulation implementations is their inability to correctly model the interaction with terrain features such as the seabed and the shoreline. The combined system we propose needs to be capable of simulating the complex effects of such an interaction in a manner that appears realistic to the observer.

3. Ability to Be Easily Integrated with Existing Approaches and Systems As we are basing the combined model on existing methods with a lot of research behind them, it is desirable that we keep the interoperability with the modifications and improvements of these methods. For example, the combined model should not be designed in such a way as to break the commonly used LOD approaches and other modifications in the relevant fields of study.

4. Real-time Capability The system must also support real-time usage, at least in theory, as the selection of methods for the combination is done with eye towards game industry usage. Choosing methods that we can not reasonably expect to allow modifications enabling real-time usage on current high-end machines is therefore not advisable.

4.2 Choosing the Methods

Based on the criteria presented in the preceding chapter, we have decided to base our combined model on two methods. We have chosen the Tessendorf method for simulating the general surface of ocean due to its popularity in the ocean simulation community, proven real-time capability and well-explored possibilities of simulating virtually infinite ocean surfaces. By itself, the Tessendorf method therefore satisfies all criteria except *2. Ability to React to Terrain in a Physically Convincing Manner*. Judging by the information provided in the survey [1], the Tessendorf method can be used to accurately and convincingly model deep ocean waves. Its limitations lie in the inherent inability to react to the terrain, as well as

any other objects interacting with the water surface. To offset these limitations and achieve the criteria, we need a physically correct method, along with a way of combining the results into one surface.

The choice of the exact physical method is more complicated. We need a method that is capable of capturing the complex behavior of ocean near the shore, but we also need the method to be as fast as possible to facilitate real-time usage. While it is theoretically possible to simulate ocean interaction with Navier-Stokes Equations, and such an approach is proven to provide dynamic results [1], the performance cost makes it unsuitable for real-time application, as noted in [2]. We must therefore turn our attention to the simplified versions of the NSE. The most prominent of these are the Shallow Water Equations. As we have described in the chapter dealing with the methods themselves, SWE reduce the complexity of the problem by assuming much larger horizontal scale than vertical scale, and instead of having uniformly-sized 3D cells, SWE deal with whole columns of water, thus restricting the size of the problem to two dimensions. While having some limitations, especially in the case of simulating more violent weather conditions where vertical movement of water and breaking waves become much more important, SWE have been shown to accurately model shallow water waves for non-violent ocean surfaces. SWE also operate on a uniform 2D grid of cells, which makes synthesizing the resulting heightfield to a similar format to that provided by the Tessendorf method easier than in the case of full NSE simulation. On the other hand, while SWE have been shown to be applicable in real-time, usually this was done on a spatially limited area. We are therefore unsure if SWE will not still be too computationally expensive in a combined model.

As for the exact form of the methods, we have decided not to include any additional modifications of the Tessendorf method, such as the choppy waves version often used for rough weather conditions. It should however be possible to include this and other modifications of the methods in the future, as the core approach is still the same. The SWE method is based on the work of Chentanez and Müller [5]with some additional modifications introduced to allow external influence from the Tessendorf method. The next chapter discusses the implementations and the modifications required in greater detail.

5. Combined Model

Having selected the methods to be used in our combination scheme, we can now move on to detailing our combination scheme itself. We will first describe the implementation of the selected methods, basing this implementation on the relevant papers and resources. We will then move on to our combination model itself, describing the transitioning between the two methods as well as the modifications we have had to incorporate to allow the transitioning without breaking the simulation.

5.1 Tessendorf Implementation

The implementation of the Tessendorf method follows standard steps outlined by Tessendorf in his original publication [4] and is also based on the implementation from [16]. The Phillips spectrum is used as the model for generating the spectral distribution of waves due to the popularity it has in various papers and systems using the Tessendorf method. However, the spectral model can easily be swapped with another, as it is only used during the initial computation of the heightfield in Fourier domain at time 0. We can therefore also consider using other spectral models, such as JONSWAP or perhaps TMA in the future, as suggested in [11]. The Phillips spectrum is defined as follows:

$$P_h(\boldsymbol{k}) = A\frac{e^{(-1/(|\boldsymbol{k}|L)^2)}}{|\boldsymbol{k}|^4}|\boldsymbol{k}\cdot\boldsymbol{w}|^2 \tag{5.1}$$

where A is the wave shape parameter, $L = W^2/g$ defines the limit of wave size based on wind speed W and gravitational constant g. The vector $\boldsymbol{k}$ is the wave vector and $\boldsymbol{w}$ represents the wind direction.

The primary application-controlled parameter of the Phillips spectrum is the wind direction $\boldsymbol{w}$ and wind speed represented by W. Using the Phillips spectrum, we can define the values of the heightfield in the spectral domain at time 0:

$$\tilde{h}_0(\boldsymbol{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{S(\boldsymbol{k})} \tag{5.2}$$

where ξ is a random Gaussian number, with ξ_r and ξ_i being its real and imaginary components, respectively. $S(\boldsymbol{k})$ is the underlying oceanographic spectrum, i.e. in our case $P_h(\boldsymbol{k})$. To animate this static heightfield, we use the method Tessendorf proposed in his original work. This method is based on the dispersion relationship between wave vectors and the frequency of the wave. For deep water waves (assuming infinite ocean depth), the relationship is known to take the following form:

$$\omega(\boldsymbol{k}) = \sqrt{g|\boldsymbol{k}|} \tag{5.3}$$

where $\omega(\boldsymbol{k})$ represents the wave frequency of a given wave vector. Using the given dispersion relationship, we use the equation from [4] to gain the spectral heightfield representation at time t as follows:

$$\tilde{h}(\boldsymbol{k}, t) = \tilde{h}_0(\boldsymbol{k})e^{i\omega(\boldsymbol{k})t} + \tilde{h}_0^*(-\boldsymbol{k})e^{-i\omega(\boldsymbol{k})t} \tag{5.4}$$

where $*$ represents the conjugation operation. This modified spectrum at time t can then be transformed using the inverse FFT algorithm, resulting in an array of complex values whose real components represent the desired heights of the surface described by the transformed spectrum.

From a technical viewpoint, we have opted for using the Nvidia CUDA technology to calculate the FFT operations as fast as possible. CUDA contains a library called cufft which provides high-level functions for performing the transform, we only need to provide the data. We start by creating the initial Fourier domain heightfield at time 0 on the CPU, using the relationships described above. We then allocate a suitably sized memory block on the GPU and copy the initial heightfield to this memory block, storing the CUDA pointer. We also allocate memory for the spectrum at time t and create a cufft plan handle to allow FFT transforms using the supplied GPU memory pointers in the future. During each frame, we then create a modified heightfield at time t using the original heightfield from time 0 and the wave dispersion relationship, as proposed by Tessendorf. The creation of this new spectrum is performed entirely on the GPU using custom CUDA kernel. We then feed the modified spectrum to the cufft library function providing complex to complex inverse FFT transformation based on our defined cufft plan. The resulting heightfield in spatial domain is then gained by taking only real components of the inverse FFT operation.

Note that unlike the original Tessendorf's implementation, we are not using the calculated heightfield to assign heights to a set of vertices. Instead, we are storing the resulting heightfield in a texture further referred to as the Tessendorf heightmap. This modification allows us to later use the Tessendorf results in the domain shader to displace the newly created dynamic vertices. In addition to the heightmap, our method also requires the presence of a normalmap specifying the normal vector distribution of the surface. We use another custom CUDA kernel and provide it with the spatial representation of the heightfield at time t using the heightmap and the spacing of the pixels when converted to world space. The actual generation of the normals uses a standard Finite difference method [17], basing the approximation of the surface slope on the heights of surrounding elements (represented by the neighbouring pixels) and the spacing.

It is also important to mention that the textures cannot be used simultaneously by the CUDA driver and Direct3D 11. Before accessing the textures from CUDA, the textures must be registered as CUDA resources and mapped for use. After all CUDA work on a texture is complete, the texture must be again unmapped to allow Direct3D 11 to bind the texture to the graphics pipeline. The basic results of the Tessendorf implementation can be seen in Figure 5.1. As we have not taken any steps to counter it, the implementation contains tiling periodicity common to the unmodified Tessendorf method. The periodicity is visible from certain points of view, but is negligible when the camera is near the surface.

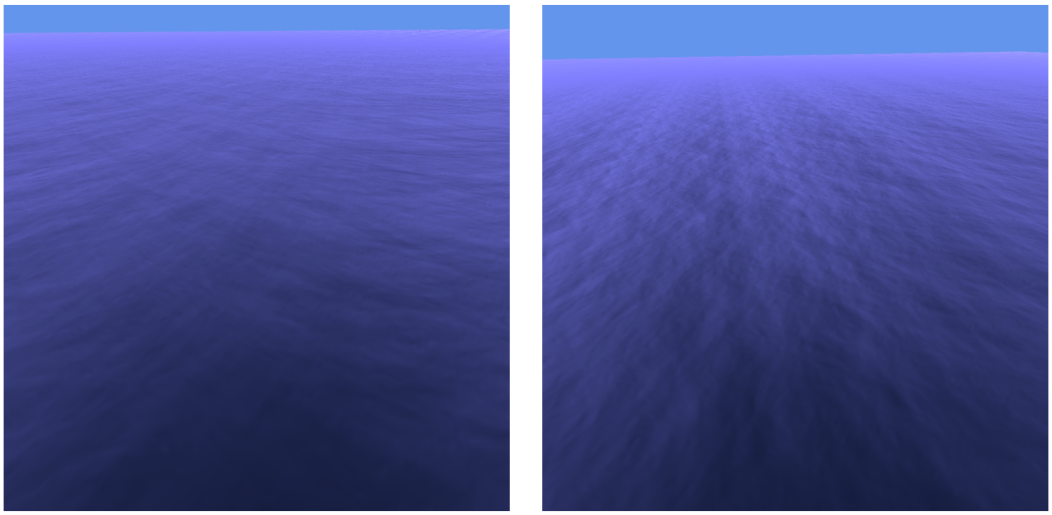


Figure 5.1: Screenshots of the Tessendorf method from the pilot application. Periodicity is visible in the screenshot on the right.

5.2 SWE Implementation

We have chosen to base the SWE implementation on the paper by Chentanez and Müller from Nvidia. Not only do the authors describe the system in high detail, they also include modifications meant to improve the stability of the SWE solver, and propose a way of dealing with open-water boundary conditions. The authors opt for using the MAC grid configuration and explicit integration, as they state that the implicit integration methods are too demanding for real-time usage. They base their system on the following form of the SWE:

$$\frac{Dh}{Dt} = -h \bigtriangledown \cdot \boldsymbol{v} \tag{5.5}$$

$$\frac{D\boldsymbol{v}}{Dt} = -g \bigtriangledown \eta + \boldsymbol{a}^{ext} \tag{5.6}$$

where h is the height of the fluid, D is the material derivative operator, g is the gravitational constant, $\eta = h + h_{terrain}$ is the absolute surface height, $\boldsymbol{v}$ is fluid velocity and $\boldsymbol{a}^{ext}$ represents external acceleration due to other forces.

The authors refer to another full paper for the implementation of the velocity self-advection, suggesting the use of the MacCormack second-order integration scheme. For the sake of implementation simplicity, we have instead opted for the standard Semi-Lagrangian advection, as implemented, for example, by Stam [13]. This velocity advection scheme works in the fashion of backwards Euler integration – instead of taking the current velocity of a particle and integrating it forward in time, the velocity is inverted, and an approximate point of origin in the previous frame is calculated. As the point of origin will usually lie between four sample points contained in the grid, we calculate the original velocity value using bilinear interpolation based on the four values in the adjacent grid points, as illustrated on Figure 5.2. The resulting value is then used as the new velocity value of the particle. It is worth noting that the velocity advection scheme is

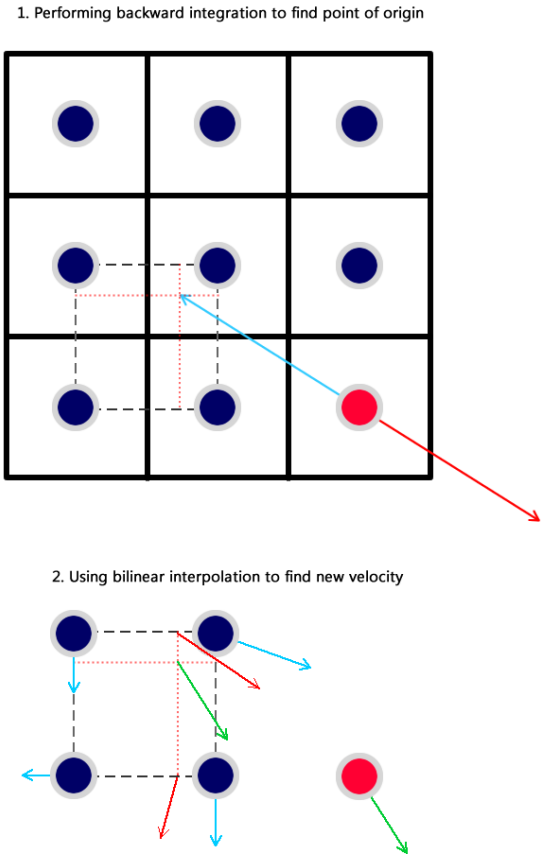


Figure 5.2: Illustration of Stam's advection scheme. In the first step, the current velocity of the particle (red) is inverted (blue) and used to find the approximate point of origin. In the second step, approximation of the point of origin velocity is found using bilinear interpolation based on the velocities of the surrounding grid elements (blue). The final interpolated velocity (green) is then assigned as the new velocity for the particle.

separated from the rest of the calculations, and could be easily replaced by a more accurate one, should the need arise.

The implementation employs a time-splitting technique, first advecting the velocity field of the fluid, and then integrating the height and velocity forward in time. For solving the height integration, the authors rewrite the Equation 5.5 and discretize it into grid cells:

$$\frac{\partial h}{\partial t} = - \bigtriangledown \cdot (h\boldsymbol{v}) \tag{5.7}$$

$$\frac{\partial h_{i,j}}{\partial t} = - \left(\frac{(\bar{h}u)_{i+\frac{1}{2},j} - (\bar{h}u)_{i-\frac{1}{2},j}}{\Delta x} + \frac{(\bar{h}w)_{i,j+\frac{1}{2}} - (\bar{h}w)_{i,j-\frac{1}{2}}}{\Delta x} \right) \tag{5.8}$$

where u and w are the horizontal, resp. vertical components of the velocity field, Δx is the spacing of grid cells and $\bar{h}$ represents an estimate height value at a point corresponding to the placement of a velocity component. The authors

suggest using the upwind cell height value for calculating $\bar{h}$ instead of using the average of the two cells, claiming that this choice results in more stable simulation and better results. We have experimented with both options while developing our implementation, and can support this claim in general – using the average indeed often introduces unwanted oscillations. Formally, the values of $\bar{h}$ are defined as follows:

$$\bar{h}_{i+\frac{1}{2},j} = \begin{cases} h_{i+1,j} & \text{if } u_{i+\frac{1}{2},j} \leq 0, \\ h_{i,j} & \text{otherwise} \end{cases} \tag{5.9}$$

$$\bar{h}_{i,j+\frac{1}{2}} = \begin{cases} h_{i,j+1} & \text{if } w_{i,j+\frac{1}{2}} \leq 0, \\ h_{i,j} & \text{otherwise} \end{cases} \tag{5.10}$$

Finally, the height is integrated explicitly based on the calculated values using basic forward Euler integration:

$$h_{i,j} = \frac{\partial h_{i,j}}{\partial t} \Delta t \tag{5.11}$$

The velocity integration is based on taking the fluid gradient into account as follows:

$$u_{i+\frac{1}{2},j} += \left(\frac{-g}{\Delta x} \left(\eta_{i+1,j} - \eta_{i,j} \right) + \boldsymbol{a}_x^{ext} \right) \Delta t \tag{5.12}$$

$$w_{i,j+\frac{1}{2}} += \left(\frac{-g}{\Delta x} \left(\eta_{i,j+1} - \eta_{i,j} \right) + \boldsymbol{a}_z^{ext} \right) \Delta t \tag{5.13}$$

This step can be intuitively interpreted as the implementation of the communicating vessels principle – the velocities are updated in order to balance the water surface of neighbouring cells at the same total height, regardless of the underlying terrain.

In addition to the basic implementation outlined above, we have also included several of the stability improvements suggested by the authors, clamping the height of the fluid to positive values in each simulation step, as well as limiting the magnitude of the velocity components to a value calculated as follows using a user-defined parameter α:

$$max_u, max_w = \alpha \frac{\Delta x}{\Delta t} \tag{5.14}$$

Another stability improvement proposed by the authors is artificially limiting the fluid height taken into account during the height integration step. The original value of $\bar{h}$ is modified by subtracting an adjustment value h_{adj}:

$$h_{adj} = max \left(0, \frac{\bar{h}_{i+1,j} + \bar{h}_{i-1,j} + \bar{h}_{i,j+1} + \bar{h}_{i,j-1}}{4} - \bar{h}_{max} \right) \tag{5.15}$$

where $\bar{h}_{max}$ is defined as follows using a user-defined parameter β:

$$\bar{h}_{max} = \beta \frac{\Delta x}{g \Delta t} \tag{5.16}$$

Despite using these modifications, we have also encountered an artifact causing cells near the shore to develop unstable oscillations. This same artifact caused a sort of repelling effect in cases where the velocities would be oriented away from the shore. In the end, we have found that this artifact was caused by the velocity integration step, where the change in velocity is calculated based on the total height difference between the two cells, regardless of whether any fluid was actually present in the source cell. In combination with floating point precision errors, this velocity was then advected during the Semi-Lagrangian integration step, effectively appearing as a repelling force. The authors probably did not encounter this problem due to using the MacCormack integration scheme. We have fixed the problem by specifically setting velocities to 0 in the case the velocity would draw fluid from an empty cell.

5.3 Combining the Methods

The first important step in combining the two methods is in determining where to use which method. The terrain layout of our system is represented by terrain heightmaps loaded from the hard drive, with each heightmap corresponding to one tile in the world space (more on tiles in Chapter 6.1). During the tile loading, this heightmap (or the absence thereof) determines whether the tile will contain SWE cells or will be a pure Tessendorf tile. The SWE cells for the tile being loaded are generated with the consideration of the terrain heightmap data – cell terrain heights are set to the corresponding values read from the heightmap.

To allow for visually convincing combination of the SWE and the Tessendorf method, it is necessary to solve several problems. First we must find a way to simulate open water boundary conditions in the SWE in such a way that waves do not reflect off the boundary, which is the standard behavior of the SWE solver we have implemented. Since the Tessendorf method is by its very nature dependent only on the spectral model underneath, it is virtually impossible to project any SWE results back into the Tessendorf surface. However, converting the method results from the Tessendorf to the SWE representation is possible, albeit hard to implement in a physically correct manner. We must also first determine where to apply which type of transition. Currently, this is implemented in per-tile way; while this makes the method transition more visible, it is also more convenient for standard representations of heightmap data, as these are most often based on encoding only the terrain above the sea level. Generating the cells only for certain shallow depth levels, as was originally intended, thus becomes impractical. However, we have included a section on implementing these changes in Chapter 9.

For a given SWE tile, we must therefore first assign the transition methods to each side of the tile. For the sake of simplicity, we assume that this hypothetical SWE tile borders with pure Tessendorf tiles on all sides, as in the case of sharing a border with another SWE tile, no method transition is required; we must merely ensure correct communication between the tiles during the physical updates. For a surrounded SWE tile, we first use basic goniometric functions to calculate the wind vector. Assuming the vector is situated at the center of the tile, we then assign SWE-Tessendorf transition to the tile sides towards which the wind vector is points, and Tessendorf-SWE transition to the sides the wind vector is

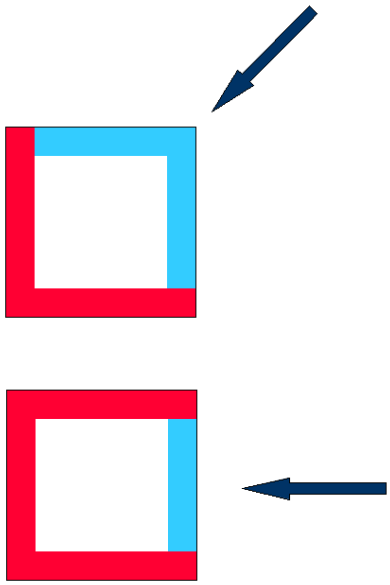


Figure 5.3: Visualization of how the wind direction impacts the creation of transitioning borders and damping zones of a tile. Blue borders represent Tessendorf-SWE transition, while red borders represent SWE-Tessendorf transition in the form of damping zones.

originating from. Generally, when the wind vector is not aligned with any of the axes, the transition distribution will be two by two, in the case of axis alignment, three sides will be using SWE-Tessendorf transition, and only one Tessendorf-SWE transition, as illustrated on Figure 5.3.

5.3.1 SWE-Tessendorf Transition

Unlike the other transition direction, the transition from the SWE to FFT is occasionally mentioned in the literature. However, as we have mentioned earlier, it is virtually impossible to influence the results of the Tessendorf method in any other way than by modifying the spectrum, or by directly displacing the resulting heightfield, both of which is impractical, mainly from the realism standpoint. Modifying the spectrum for a specific tile would also deprive us of the advantage of only having a constant number of FFT calculations per frame, as well as break the tile-ability of the entire approach. The problem is therefore usually reduced to making sure the SWE part of the system is equipped for dealing with the boundaries in a manner that is in accordance with the laws of physics, i.e. that the borders do not reflect the outgoing waves back into the simulation domain.

Our implementation uses the same concept as outlined in the work of Chentanez and Müller [5] – the SWE cells nearest to the transition borders are outfitted with a damping field, gradually smoothing out waves approaching the border. It is important to note that care must be taken in choosing the damping factors, as large values will make the damped zone act like a wall, essentially replicating the reflection effect we are trying to smooth out. On the other hand, if the damping factor is too small, the damping zone will be incapable of sufficiently suppressing the waves approaching the border. In our pilot application, the damping factors

are set as follows:

$$damp(i) = (|i - size/2| - size/2 - 10))/200 \tag{5.17}$$

where i is the distance of the cell from border in terms cell units and $size$ is the size of the SWE tile side in cell units. This selection behaves well during the simulation itself, but introduces small wave artifacts after the initialization. Only cells with within 10 cell units from the border are damped.

5.3.2 Tessendorf-SWE Transition

In contrast to the SWE-Tessendorf transition, we can actually directly influence the SWE surface, even if such an action is questionable from a purely physical standpoint. However, we are faced with several difficulties. First and foremost, the input data we are obtaining from the Tessendorf method border are limited only to a heightfield representing the surface. While it would be theoretically possible to decompose the FFT-generated heightfield back into its separate wave components, not only would it be contrary to the whole purpose of using the Tessendorf method in the first place, it would probably not make the problem any easier, as there is still no direct mapping between the wave components and the SWE properties. Thus we will base the transition only on the heightfield values at the border as well as the physical parameters of the scene, such as wind direction.

We have tried out several approaches of converting the heightfield to SWE cell values. The core problem of the Tessendorf-SWE transition is that we are modifying a physically correct system on the basis of purely empirical data, which can easily break the stability of the physical system. To allow the transition between the methods, we have introduced a new concept into the SWE solver implementation, further referred to as the "fixed-height cells". The fixed-height cells are excluded from the height-integration step, making them entities not conforming to the underlying physical system. Fixed height cells do however take part in the velocity integration step, which enables them to influence the surrounding cells without changing themselves. This also induces a hidden problem that is described and solved in Chapter 5.3.3. On the whole however, the fixed-height cells provide us with a way of influencing the SWE system without breaking it too much, even though extra steps must be taken to compensate for the non-physical volume gain. Having defined the fixed-height cells, we can now describe our experimental methods of transitioning from the Tessendorf method to SWE. The methods we used perform a variety of operations, including modifying heights of the cells, velocities of the cells and even using full-scale interpolation with the Tessendorf method.

Height Copy

In our first method, we use the wind direction to find which borders require the Tessendorf-SWE direction using the method described above, and then mark the first row of cells along each border as fixed-height cells. In each frame of the simulation we then access the texture describing the Tessendorf heightfield

and sample the heights corresponding to the fixed-height cells on the Tessendorf-SWE border and assign these as new heights of the relevant cells considering their underlying terrain height. The fixed-height cells along the other borders are permanently set to the default water level – zero.

Wind Velocity Fixing

Another approach we have tried is fixing also another parameter of the borderline cells – the velocities. Our expectation was that such a modification might result in a flow forming between the transition border, effectively simulating the water volumes moving to and from the system. This flow did form, but the results were not visually pleasing – as the velocities on the borders were constant, they quickly overtook the velocities forming as a result of the fixed-height cells, making the water move in much larger masses. In addition, the larger volumes of water lost and gained made it harder for the mass conservation step to counter, resulting in a situation where the water level was slightly skewed in the direction of the wind.

Full Tile Interpolation

We have also attempted to use a full-tile Tessendorf interpolation in addition to the standard border heightmap copy. In this case, all cells in the tile would receive height changes without being fixed-height. The copying from the Tessendorf data needs to be done using linear interpolation, otherwise the SWE tile ends up being essentially a low-resolution copy of the Tessendorf tile. The copy step can therefore be interpreted as slowly forcing the cell configuration into a rest state defined by the Tessendorf heightmap. However, even for small factors the impact on the cell structure is significant, but in a negative way – the constant stabilization tends to blur the SWE effects without providing much additional detail. We have therefore decided that such an approach is counterproductive, even though it makes the difference between SWE and Tessendorf tiles slightly harder to spot. To achieve a similar effect without compromising the SWE simulations, we would suggest overlaying the Tessendorf data over the SWE results in a purely visual way, such as in the domain shader. The comparison of the height-copy method and tile-wide interpolation can be seen in Figure 5.4

Multi-cell Height Copy

The last approach we have taken was to use a larger array of fixed-height cells to transition from the Tessendorf surface, hoping for a better reproduction of the surface details in the SWE systems. This modification did very little to change the result – in fact, as only the neighbouring cells are taken into account during the height and velocity integration step, it is impossible for these cells to influence the result in any other way than by velocity advection. In order to be advected however, the velocities inside the non-fixed cells and the simulation timestep would need to be large enough to skip the neighbouring cells during the advection step. However, the timestep is usually as small as possible in a SWE solver to allow for accurate integration and to avoid stability issues that commonly arise. Also, we are of the opinion that even if the advection did indeed pull the fixed-height cell array velocities into the simulation, the velocities would

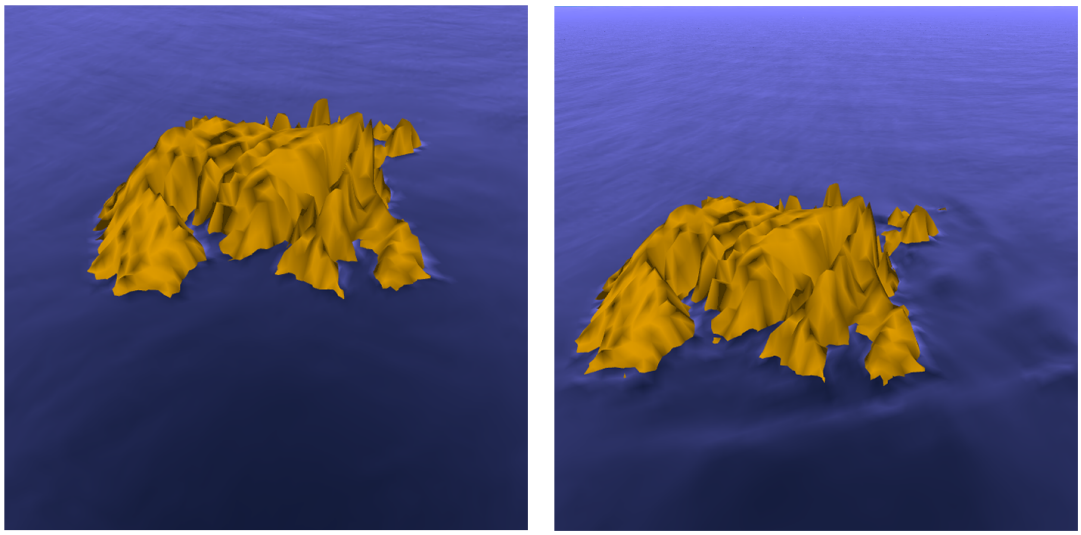


Figure 5.4: Screenshot comparison of the Full tile interpolation (left) and Height-copy (right) methods from the pilot application.

be off and would actually introduce more problems into the system – the cells would have fixed height but not fixed velocities, meaning that the velocities would be updated without affecting the resulting heights. As we describe in Chapter 5.3.3, this introduces problems even for one row of fixed-height cells bordering with a standard row of cells, let alone for a whole array of fixed-height cells.

Interpreting the Results

In the end, it turns out that the approach with most promise is the first one, using only one row of cells to transfer the Tessendorf heightfield for the border each frame. While this may seem counter-intuitive, it is actually quite reasonable when we consider what we have mentioned before – any modifications to the SWE system to enable the transition are in some way violating the underlying physical properties of the system, essentially introducing inaccuracies and unrealistic behavior. The less changes we make each frame the better, as this makes it easier for the system to deal with the artifacts we have introduced. However, we are still interested in performing more experiments with velocity modifications in the future, as we believe the effects could be more dynamic.

Conditional Damping

In addition to approaches outlined above, we must also take steps to ensure the boundary does not act as a reflective wall, similar to the problem in the SWE-Tessendorf transition. However, we cannot use the same damping field, as not only the waves approaching the border would be smoothed out, so would the incoming waves we are trying to create. To this end, we have modified the damping fields to allow for a special case we have called "conditional damping". This modification represents a damping field that only smooths out the affected cells if the relevant velocities satisfy a predefined condition. The condition for smoothing out directly depends on the wind direction, and thus gradually smooths

out the waves moving against the wind and approaching the source boundary from inside the SWE domain.

5.3.3 Mass Conservation

The SWE solvers as implemented in the literature are usually closed systems – no fluid leaves the simulation domain, and none enters. However, in the design of the transition methods described above, we have taken steps that are directly modifying the configuration of the SWE system both by damping out the waves using the damping field, and generating additional water volumes on the Tessendorf-SWE transition borders. These steps reflect the open-water situation of the simulation scenario to some degree, in particular the water leaving and entering the system, respectively. However, we cannot guarantee that the modifications represent a physically correct situation, as the outside system is not governed by the rules of physics and the damping fields function in a purely empirical way. In particular, by directly modifying the heights of the cells in a non-physical matter, we may have violated the conservation of mass, as the sum of fluid leaving and entering the systems is not guaranteed to be zero. We must therefore take extra steps to ensure that conservation of mass is enforced.

To this end, we have modified the height integration algorithm as described in [5] and the SWE solver itself by including additional steps. Before the calculations themselves, during the initial population of the *CellManager* data (see 7.3.6), we sum the height volumes of all non-empty cells that do not have a fixed height (as is the case with cells bordering with the Tessendorf-SWE transition) and store this number as the "correct" fluid volume in the system. While integrating the height across the cells, we then sum all current fluid heights of the cells, calculating the current fluid volume. If the current fluid volume differs from the correct fluid volume by a value larger than a certain constant, we modify each integrated height value by multiplying it by the ratio between the volumes. This ensures that the sum of new cell heights across the SWE simulation is equal to the correct volume, stabilizing the system. Note that the integrated values are modified before they replace the current height values, limiting the impact the inaccuracies have on the system as a whole. The calculation of the height factor and its effect on cell heights can be formulated using the following formulas:

$$ActiveCells = \{c | c \in Cells, h(c) > 0, active(c), '(fixed(c))\}, \tag{5.18}$$

$$h_f = \frac{H_{canon}}{\sum_{c \in ActiveCells}}, \tag{5.19}$$

$$h_{new}(c) = max(0, h(c)h_f) \tag{5.20}$$

where h_f is the height factor, $h(c)$ is the height of cell c and $h_{new}(c)$ is the replacement height of the cell. Functions $active(c)$ and $fixed(c)$ represent the active and fixed-height cell properties, respectively.

It is also important to ensure that the velocities lying on the boundaries between standard cells and fixed-height cells behave in a reasonable manner. In particular, it is desirable that no velocity ever indicates fluid movement into a fixed-height cell. While the mass conservation mechanism is in theory capable of

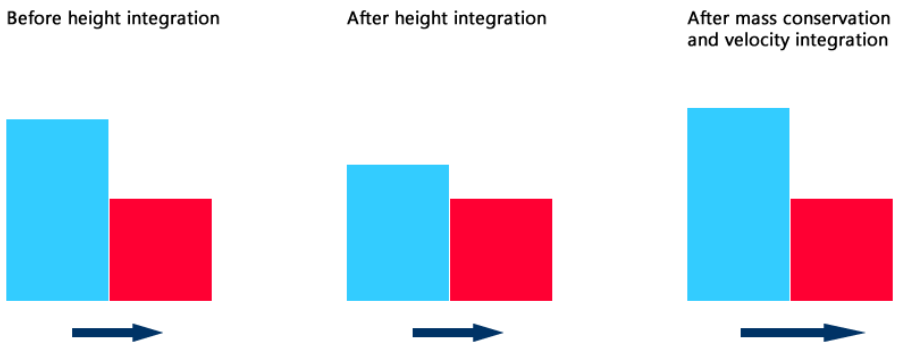


Figure 5.5: Visualization of the problem arising when velocities between fixed-height cells (red) on the borders and standard cells (blue) are not corrected. The algorithm first transfers height property from the standard cell without raising the height of the fixed-height cell in the height integration step. The mass conservation step then raises the heights of all standard cells (including the one pictured) to account for the fluid loss. This raises the height of the cell (possibly even to higher point than before due fluid loss in other parts of the system), which is then used to further increase the velocity of the fluid escaping the simulation (dark blue arrow).

compensating for the fluid that would be leaving the system in this way, this very action might create an explosive chain reaction that would eventually result in an utterly non-physical surface. The exact nature of this problem is illustrated on Figure 5.5 – with the height of the cells neighbouring with the fixed-height cells being higher, the velocity integration detects this slope and sets velocities for a "downhill" flow. As the source cell loses fluid but the fixed-height cell never gains it, the overall system loses a potentially significant amount of fluid. The mass conservation system responds to this loss by raising the heights of all nonempty cells by a factor required to achieve the original volume, as described above. However, this includes the cell neighbouring with the fixed-height cell, again increasing the height difference and thus the velocity integrated for the flow, resulting in even more fluid leaving the system next frame. Ultimately this escalates in a situation where most of the fluid is concentrated in several "blobs" and the system is oscillating between fluid loss and compensation. To counter this, we have modified the velocity integration step to test for such scenarios and set the velocity component compromising the system in this way to zero.

5.3.4 Blending

After the simulations of all involved methods are finished and the results are stored in a texture, the result blending takes part. As will be mentioned further on in Chapter 6.4, all remaining combination operations are applied on the GPU in the form of shader programs. Theoretically, the result combination is a special texture blending operation. After first generating the surface sampling points based on the observer position and other level of detail parameters, the shader

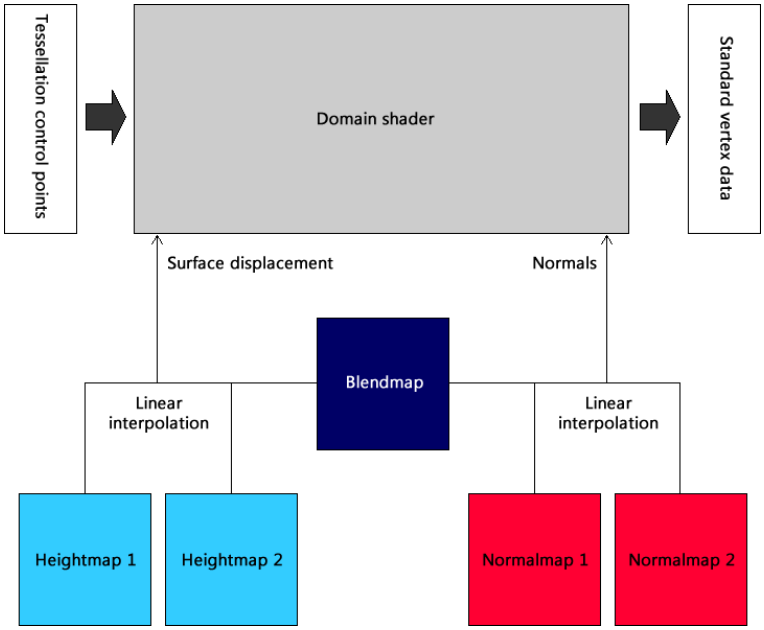


Figure 5.6: Visualization of the blending operations. For each vertex being generated by the domain shader, each texture is sampled using the same surface coordinates. The results from the heightmaps and normalmaps are then interpolated based on the factor sampled from the blendmap.

program calculates the height of each of these sampling points. The shader is provided with the two textures representing the results of the methods, and a third "blendmap" texture. The blendmap texture is either loaded from the drive alongside the terrain heightmap or generated in the tile loading step and defines the ratio of influence the simulation methods have on the height for the entire surface. Currently, as it is necessary to encode only two methods, the ratio can be encoded into a single floating point value, representing a linear interpolation factor, but the exact way of creating and interpreting the blendmap is easy to modify in case of changes.

As the heightmap by itself is insufficient to describe a smooth surface reasonably, the blending shader must also be provided with two normalmaps representing the normal vector distribution on the surface in the standard encoded way, i.e. representing a normalized vector xyz by a pixel with values of $red = (x + 1)/2, green = (y + 1)/2, blue = (z + 1)/2$ and $alpha = 1$. These normalmaps must be created by the methods themselves and updated whenever there are changes to the heightfield. The blending step then also interpolates between the normals in the same way as between heights, using the blending factor loaded from the blendmap as a parameter for linear interpolation. The entire blending sequence is described in Figure 5.6.

5.3.5 Limitations

The method as proposed above suffers from several limitations. First and foremost, the square shape of the SWE cells makes it hard both to generate waves for wind directions that are misaligned with the coordinate axes as well as to hide the tiling visually. The second problem can be alleviated to a certain degree by

using a suitable blendmap, but the first is harder to counter. In Chapter 9.2 we outline a modification that would allow for arbitrary border shapes, ideally based on another future modification, depthmaps (see 9.1).

The result also suffers from the lack of detail in the SWE domain. This is caused by the fact that the SWE tiles have lower resolution than the Tessendorf tiles, being sized only 128 by 128 cells. This means 16 384 cells per tile which is essentially a limit of what the current implementation is capable of simulating with reasonable framerates. Note however that no special optimizations were implemented, and it should be possible to reach much larger cell counts in future versions. Another improvement might lie in moving the SWE implementation to the GPU, we would however expect some complications in this effort. The possible move to the GPU is also discussed in Chapter 9.4.

5.3.6 SWE Initialization

There are situations where we must initialize the SWE system to some default configuration, one of these being the startup itself, and the other a situation where the tile converts from the default pure Tessendorf representation used for tiles that contain terrain, but are too far from the observer, to the SWE representation used near the observer. In such a scenario, using a flat surface as default would be jarring to the observer, and it would also take a lot of time before the system is filled with at least some waves, because they need to travel from the border. As such, we have decided to initialize the SWE system based on the Tessendorf data – despite quickly degrading to a wavy surface unrelated to the original Tessendorf source, this provides some texture to the tile, making the conversion less abrupt, as well as giving the incoming waves time to form.

6. Level of Detail (LOD)

As we are trying to focus our experimental methods on usability in real-time, it was necessary to include some form of level of detail methods in the pilot application, preferably in such a way as to be independent of the underlying methods. While the methods used are not the most advanced the literature has to offer, this is due to the LOD methods being a separate and extensive field of study on their own. Implementing the state-of-the-art LOD algorithms is therefore out of the scope of this thesis. We have however tried to take advantage of modern techniques that have emerged only recently due to technological advances, as well as the properties of methods themselves, to propose a level of detail scheme capable of reasonable application speedup.

6.1 Tiles and Range Culling

The first element of the level of detail scheme comes from the properties of the Tessendorf method, as well as the hardware limitations of the graphic cards. As is described above, the Tessendorf algorithm produces a fixed-size tileable heightmap describing the surface of a given square-shaped part of the ocean. Additionally, we need to represent and store the description of the terrain data, both above the water level and below, ideally with fast lookup times of terrain height value for a specific set of coordinates.

To solve this, we have elected to artificially partition the world space into a uniform grid structure of elements, which we refer to as tiles. This allows us to perform many operations that would otherwise be costly performance-wise or much harder to implement. First, the terrain data can now be represented by a texture representing the height of the terrain in the same way as the Tessendorf heightmap does for the water. Not only does this allow us to treat terrain geometry similarly to the water geometry, it also provides us with an easy way of persisting and loading the terrain data, as we can simple store DDS (Direct draw surface) textures on the disk as the simulation content. As the representation of terrain using the heightmaps is very widespread, it also makes the application easy to integrate with existing solutions and data.

In addition to these conceptual advantages, the division of world space into tiles also provides us with the first basic LOD method. As we can detect the tile that is nearest to the camera position, we can limit the range of the tiles that the *TerrainManager* (see Chapter 7.3.5 for more detail) request to be drawn to a certain neighbourhood of the "current" tile. The size and shape of the neighbourhood can be easily modified according to FPS (frames per second) results and limitations of the camera (e.g. we don't need to render many tiles when we limit the maximum height the camera can be placed at).

While the tile approach is used for the water and terrain geometry, note that simulation objects are not required to adopt it as well. The world is capable of holding any number of simulation objects, including updating and drawing them with the rest of the world. However, the pilot application does not contain any LOD methods for non-tile based objects, which means that they will all be drawn regardless of camera position and any other circumstances, unless they implement

some kind of LOD scheme themselves (e.g. at least aborting the draw call in case of large distance from the camera). Unless a complex LOD scheme such as a quadtree or octree spatial division algorithm is applied, it is therefore advisable to include a very limited amount of simple non-tile objects.

6.2 Near / Far Model

As the calculations performed by the SWE tiles are demanding, we have included two versions of simulation objects per tile – a "near" and "far" model . For pure Tessendorf tiles, both of these are the same tessellated patches using the shared Tessendorf heightmap. For SWE-based tiles, the near model contains a simulation object capable of processing the SWE calculations in the near mode, and a standard shared Tessendorf patch in the far mode. The actual rendering of the near or far model is based directly on the distance to the observer. The SWE patch itself keeps track of when it was last drawn, and if the difference between current time and last draw time is larger than a given constant, all updating of the patch is postponed, and the patch is marked as requiring a reset. The delay is introduced to prevent cases where quick camera movements would rapidly activate and deactivate the patch, resetting the simulation each time. Note that the deactivation based on last draw call also means that a patch is deactivated after not being in the observer field of view for a while due to the view-frustum culling modification described in the next section. The actual reset procedure was described above – the SWE cells in the patch are initialized from the Tessendorf heightmap.

6.3 View-Dependent LOD

6.3.1 View-frustum Culling

While the range-based tile culling reduces the amount of actual tiles being drawn, it does so depending only on the observer position, not the matrices describing what can actually be seen by the observer – even tiles behind the camera are still sent to the graphics card and get processed in the entire device pipeline up to the pixel shader stage, where they are discarded due to not being in the visible space. This is usually very wasteful, as processing each tile includes a lot of operations, including tessellation and heightmap sampling. To address this issue, we perform the same test that eventually discards the pixels of objects outside the visible space before the pixel shader stage, but we perform it on a per-object basis on the host before actually requesting any additional draw calls. The method itself is called View-frustum culling and in essence combines six half-plane tests per point to find out if a point lies inside or outside the view-frustum, as illustrated in Figure 6.1. The frustum itself is constructed from the view and projection matrices, and the testing itself is done in the world space. As our tiles are objects that are naturally square-shaped heightfields, we can use axis aligned bounding box (AABB) to easily represent its world space position. While drawing the range-culled tiles, we construct a view-frustum for the current camera position and orientation and then test the bounding box containing each

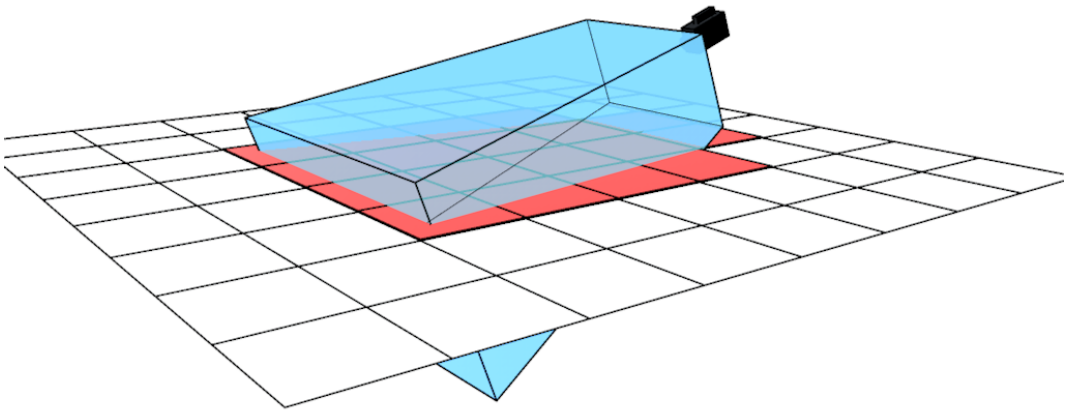


Figure 6.1: Visualization of the view-frustum (blue) of the camera and the effect it has on rendering the tiles. Tiles intersecting the view-frustum (red) will be rendered, while the other tiles (white) will be discarded.

tile for intersection with the view-frustum. If the bounding box intersects the view-frustum, the corresponding tile is drawn, otherwise it is discarded. The bounding boxes themselves are created during the tile creation and stored among other per-tile data.

Despite limiting the amount of onscreen triangles representing the water surface to a fixed number using the range based tile culling and then further reducing this amount via view-frustum culling, both of these methods represent a crude LOD scheme, as they operate only on boolean values indicating whether a tile should or should not be drawn. Tiles being drawn far away from the observer will have the same level of detail as those directly in front of the camera. To address this issue, we have included another LOD pass in the rendering scheme using progressive mesh tessellation.

6.3.2 Tessellation

The core idea is to store and display the tile geometry in the lowest level of detail by default, and only create the higher levels dynamically where needed, as indicated by the observer position in world space. This process is essentially inverse to traditional LOD methods which store multiple levels of detail generated either programmatically from the highest detailed geometry, or created directly by the artist to accurately represent the object. Since we are rendering the surface of an ocean, we are in essence approximating a smooth surface using a number

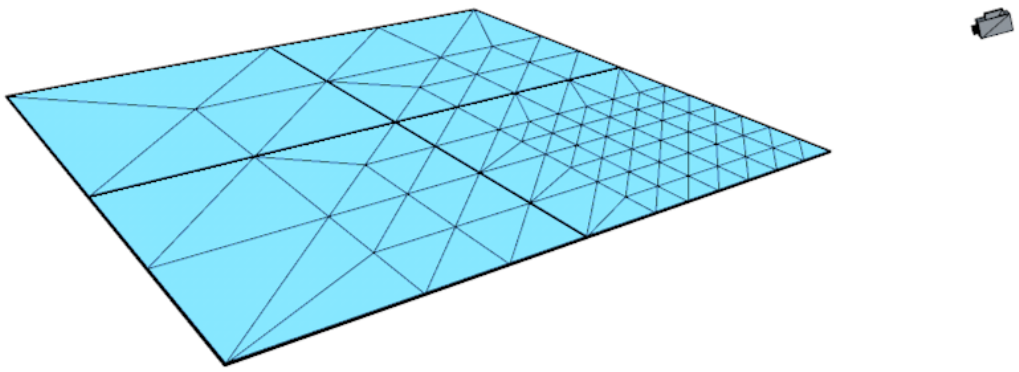


Figure 6.2: Illustration of the effect the view-dependent tessellation has on the tile vertex structure.

of sampling points defined by a heightmap. While the Tessendorf method mostly uses uniform fixed-size grid to sample the surface, this has arisen from the natural mapping of vertices to the FFT-generated heightmap, and is not the only way to sample the surface.

The tessellation itself is performed on a per-tile basis and the control of the tile tessellation is not influenced by any surrounding objects, only observer position. The lowest level of detail for a tile may consist of only four vertices, or of multiple four-point patches, depending on how we want to model the furthest tiles, but also on how much detail we require in the nearest tiles [1]. The exact algorithm for calculating the tessellation factors will be presented in the next section; it is worth noting however, that it can be changed very easily to support various other cases. An example visualization of four tiles being tessellated based on the camera position can be found in Figure 6.2.

6.3.3 Cracking

One problem that is often associated with using tessellation as a LOD scheme is the creation of cracks in the surface where adjacent meshes meet due to T-vertices. However, in the case of ocean water, we can take steps to ensure that the resulting surface is waterproof. First, we need to avoid having different heightmap data for the two adjacent edges. For Tessendorf tiles, this is automatically satisfied,

[1] HW tessellation factors have built-in limits, so the subdivision granularity depends on the original geometry. For integer mode, the factor ranges are 1-64. For more information see [18]

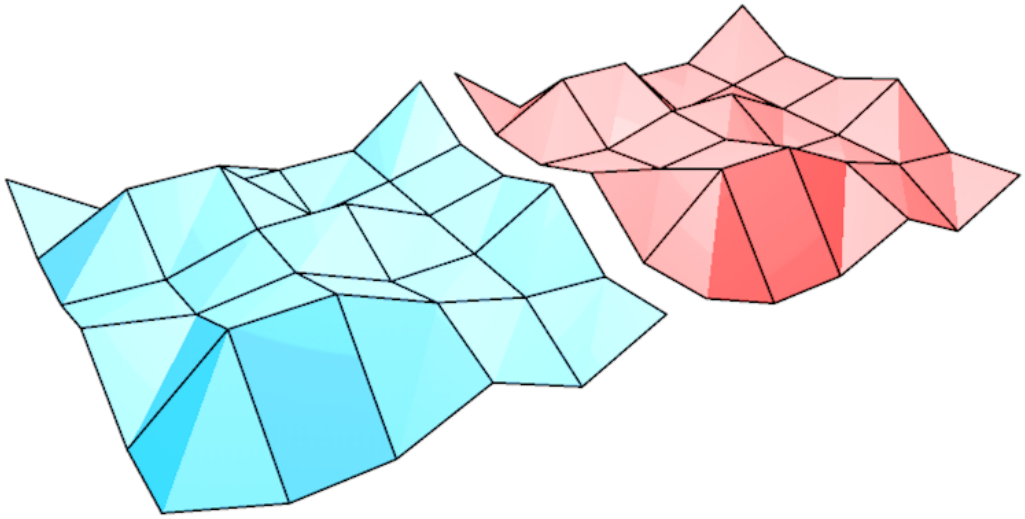


Figure 6.3: Illustration of the cracking problem in tessellation. Due to sampling the same heightmaps and using same tessellation factors for the adjacent edges, our tiles don't have this problem.

as the periodicity property guaranteed by Tessendorf method ensures the same height for the borders. For tile edges representing the boundary between the two methods, the property must be satisfied artificially by copying the height values for border SWE cells from Tessendorf and using linear interpolation. Next, we need to avoid the creation of T-vertices on the tile edges. We can satisfy this requirement by basing the edge tessellation factor calculation directly on the position of the vertices comprising the edge. As the vertex positions are the same for both edges, the tessellation factors will be the same, and the newly created vertices will be overlapping as well. Since the vertices will also sample the same height from the heightmap, the surface will appear to be smooth. An illustration of adjacent tile tessellation can be found in Figure 6.3.

6.4 Hardware Tessellation

6.4.1 Overview

Both in the design and implementation of the LOD scheme outlined above, we have tried to take advantage of the capabilities of modern graphics cards. Since we have opted for using Direct3D 11 in our pilot implementation, we have access to hardware-accelerated tessellation integrated in most modern GPU units via the hull and domain shaders. The host can therefore treat each water tile as a

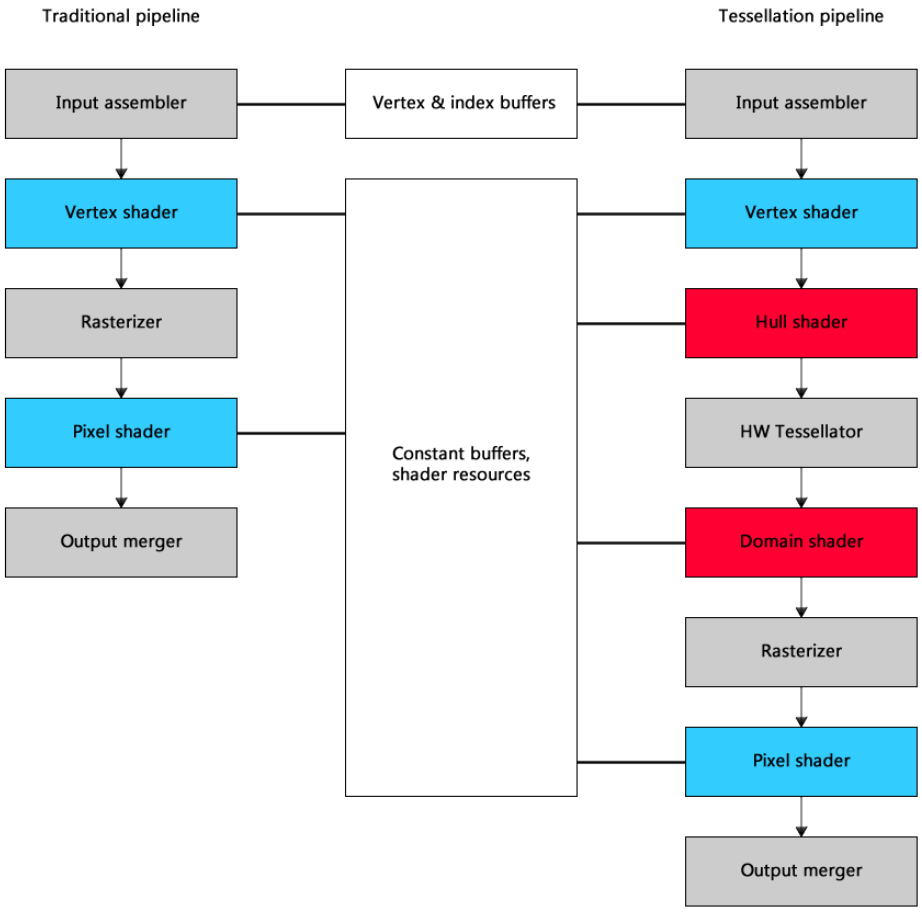


Figure 6.4: Comparison of the traditional graphics pipeline (left) and the tessellation-enabled pipeline (right). The arrows illustrate how the graphical data flow through the pipeline, while the lines represent the access to read-only data stored in buffers set by host code.

square patch of control vertices (or multiple patches) and leave the process of using tessellation to provide sufficient view-dependent detail to the GPU.

After first binding the shaders and heightmap parameters via material system (for more information see Chapter 7.5 of technical documentation) and providing tessellation-capable vertex data, the process itself can start. In addition to standard vertex shading and pixel shading, the pipeline now contains two more stages. The hull stage is responsible for setting the parameters of the hardware tessellator and preparing the control point data for use with the domain shader. The control points are then fed to the hardware tessellator, which generates new vertices on the GPU and interprets them as a surface. The visual comparison of the two pipeline configurations is illustrated in Figure 6.4.

Before moving on to describing the two new shaders, let us first discuss the impact the tessellation has on the standard pipeline setup. Because we want to base our tessellation algorithm on the position of the viewer, the input vertices of the hull shader need to be still based in world space, not screen space. We must therefore modify the vertex shader, which precedes the hull shader in the

tessellation pipeline, to only apply world transformation on the geometry. The pixel shader now follows the domain shader in the pipeline, not the vertex shader, we must therefore take care that the data layout is consistent across these two stages.

6.4.2 Hull Shader and Domain Shader

As we base the tessellation on a square shape, the hull shader input will be four patch control points. To allow basing the tessellation on the viewer position, the shader is provided with the world space coordinates of camera. To work correctly, the hull shader has several responsibilities to maintain. First, it must create the output control point data, which is mostly directly based on the input control points received from the vertex shader. Second, it must set the tessellator parameters. This includes the output topology that is to be generated, as well the type of partitioning. The partitioning type directly controls how the subdivision behaves and Direct3D 11 offers four options: fractional_odd, fractional_even, integer and pow2. We have chosen the integer partitioning as it corresponds to how the heightfield is traditionally modelled in most of the Tessendorf method implementations. The final task of the hull shader is to calculate the tessellation factors for each triangle edge in the input primitive using the defined constant function. As we are using the quad domain, we need to set four factors for each outside edge, and two factors for the inner edge tessellation. Our method of calculating the tessellation factors is essentially a shifted linear falloff function of the point position in world space :

$$T_f(\boldsymbol{p}) = clamp(MAXDETAIL - \frac{|\boldsymbol{o} - \boldsymbol{p}| - 200}{10}, 1, MAXDETAIL) \qquad (6.1)$$

where *clamp* is a function clamping a value to a certain range, p is the position of the point in world space, o is the position of the observer and $MAXDETAIL$ is a constant representing maximum tessellation factor. A graph illustrating the progression of tessellation factor based on the distance from the observer can be found in Figure 6.5.

As the factor function is a function of only one world space position, we calculate the factors for outside quad edges as factor function of the point lying in the middle of the line segment connecting the points. For the inside edge factors, we first calculate the center of the patch and then transform it using the factor function.

After the hardware tessellator partitions the patch based on the calculations in the hull shader, the domain shader is executed once for each resulting vertex. The domain shader must set the vertex properties accordingly to the control point data and domain position (a two dimensional coordinate specifying the position on a surface defined by control points). In our case, we use bilinear interpolation to generate horizontal position and texture coordinates for the new vertex, and then use the texture coordinates to sample the heightmap, normalmap and additional textures that are provided to the domain shader. For example, in the blending variant of the shader, there would be two heightmaps and two normalmaps alongside a blendmap texture. The calculated surface height is then used to displace the vertex along the y axis. The final task of the domain shader

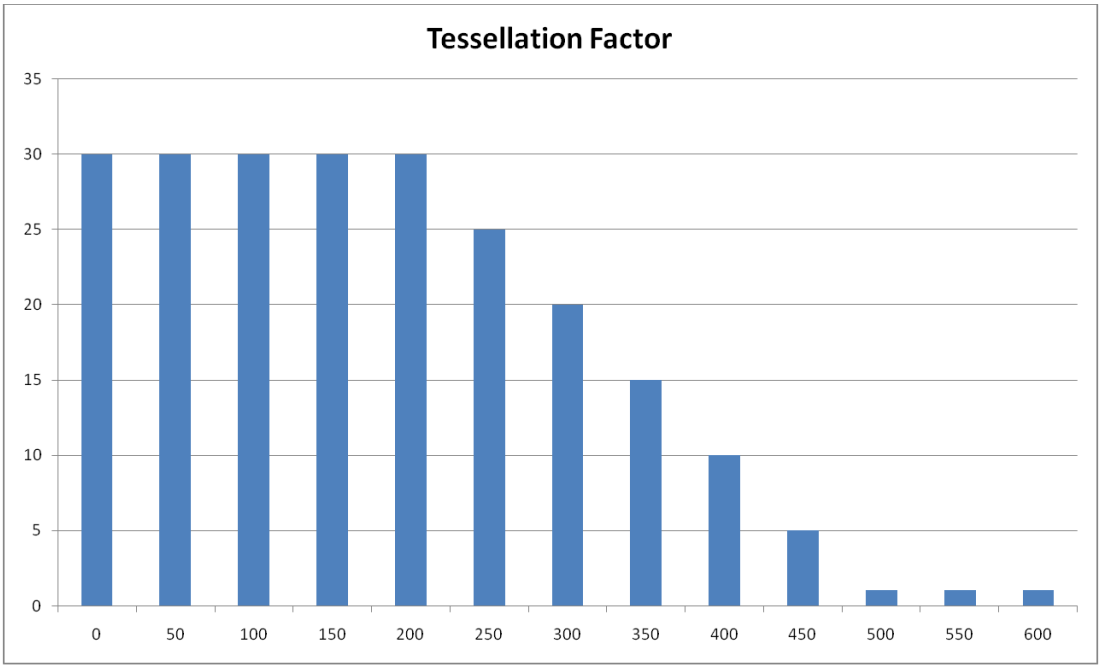


Figure 6.5: Graph describing the dependency of the Tessellation factor (vertical axis) on the distance from the observer (horizontal axis).

is to perform the world space to screen space transformation that usually falls to the vertex shader in a traditional pipeline configuration.

7. Technical Documentation

In this chapter of the thesis we briefly describe the project properties and architecture of the pilot application in such a detail as to allow sufficient and reasonably fast understanding of the principles behind it, with the goal of enabling future modifications and improvements of the application. We also detail the process of changing the simulation methods themselves and quick incorporation of new methods using the material systems, as we consider these an important aspect of the implementation.

7.1 Project Structure

The application is a standard Qt window application consisting of one main window named "ocean", the layout of which was created using the Qt Designer. As with all Qt windows, the main window consists of multiple files, some of which are generated by the Qt library during compilation. These files are contained in the "Generated Files" filter. Apart from the standard C++ filters containing the header files, source files and resource files, the application further contains additional file types, with corresponding filters gathering these files together. Specifically, the application contains a Qt form file mentioned above in the filter "Form Files", files compiled using Nvidia CUDA in the filter "CUDA Kernel Files" and HLSL shaders in the filter "Shader Files". For the header and source files, we have included additional filters grouping together the files sharing a common ancestor, such as cameras and materials.

7.2 Dependencies

The project depends on several libraries and frameworks which are required for successfully building the application. The following libraries were used in the application.

7.2.1 Qt GUI Framework

For basic presentation and controls the project requires a GUI library. The library chosen is the well known Visual C++ GUI library named Qt Project. The application was developed and built with the version 5.2.1 of the Qt library, which can be found on the following website: http://download.qt-project.org/archive/qt/5.2/5.2.1/

7.2.2 Direct3D 11 (via WindowsSDK 8.0)

The project uses Microsoft Direct3D 11 for rendering the scene. Direct3D is currently part of the Windows SDK, which is shipped with Microsoft Visual Studio 2012 and higher. For deployment purposes, Direct3D 11 should be available on all versions of Windows from Windows 7 onward. The application uses the built-in Visual Studio 2012 shader compilation tools, so no additional tools are required for this task.

7.2.3 Nvidia CUDA

The GPGPU computing library by Nvidia is used for calculating the FFT transformations, heightfield and normal textures, and additional computationally demanding tasks. The specific version of CUDA used in the project is 5.5, which is currently outdated, as it was replaced by version 6 while the project was being developed. The SDK for version 5.5 can still be found on https://developer.nvidia.com/cuda-toolkit-55-archive.

7.2.4 DirectX Toolkit (DirectXTK)

Additional library used in the project is the DirectX Toolkit created by Shawn Hargreaves and other members of Microsoft. The library provides several higher-level abstractions for working with DirectX 11, some of which are based on the discontinued XNA library. The toolkit also includes some basic classes supplementing the removed DXGI functionality and can be found on http://directxtex.codeplex.com/. The compiled libraries are also included in the project.

7.2.5 DirectX Texture Tool (DirectXTex)

The last library used in the project is a companion library to DirectXTK, providing methods for loading and processing textures via the WIC2 interface. The library is hosted on codeplex and includes basic documentation: http://directxtex.codeplex.com/. The compiled libraries are also included in the project.

7.3 Overall Architecture

This section includes brief documentation for the most important classes in the project. Some of the more complicated class hierarchies are discussed separately in the following chapters.

7.3.1 Ocean

The highermost architectural class in the application is the *Ocean* class, representing the actual application window. The class extends the Qt class *QMainWindow*, and is directly instantiated and activated in the application entry point. Apart from presenting the window and its controls to the user, and processing user input by delegating it down the chain of responsibility to the relevant parts of the application, the class also contains a *QTimer* object with 0 interval, which is bound to the update method of the *D3DWidget* class described below. This allows the *D3DWidget* instance to control the frequency of its updates and draws independently. The visual representation of the window is contained in the *ocean.ui* file and should only be edited using the Qt Designer.

7.3.2 D3DWidget

The *D3DWidget* class represents a Qt widget capable of rendering a Direct3D surface and updating the state of the application. The most important part of

the class is the update method, which is being called during each update of the window, as was mentioned above. The update method controls the timing of actual work by trying to reach a fixed amount of frames per second, and the same number of updates. Total elapsed time since last frame must therefore be greater than Δt, defined by this simple relationship:

$$\Delta t = 1000.0/TargetFPS \tag{7.1}$$

$TargetFPS$ is a numerical constant contained in *D3DWidget* currently set to 60. The end result is that while the application is running normally, there will be approximately $TargetFPS$ number of update and draw calls. If the computations done in the update section are too performance heavy, the application will run as fast as possible.

Most of the actual work that should be done during the update and render calls is delegated further to separate objects. The *D3DWidget* merely holds references to these objects and calls the relevant methods where necessary. In particular, the class contains a single owning reference (modelled using the *unique_ptr* template class) to instances of *TerrainManager*, *Renderer* and *Camera*. It also contains a vector of *ISimObject* references, which represents physical objects contained in the simulation world.

For processing the user input, *D3DWidget* uses the Qt slot system. The slot system allows different widgets in the window to react to each other by binding actions with reactions. The actual set of methods available for binding is marked by keyword "slots". For more information, see Qt documentation (http://qt-project.org/doc/qt-4.8/signalsandslots.html along with http://qt-project.org/wiki/New_Signal_Slot_Syntax) and the header of *D3DWidget*. The slots in *D3DWidget* are used mainly to allow changes to the camera placement and simulation mode.

7.3.3 ISimObject

The abstract class *ISimObject* provides an interface for updating and drawing simulation objects. Depending on abstraction in this case allows us to easily create and insert a new type of object into the world, provided we implement the required methods correctly. In particular, the draw method should make no assumptions about the state of the *Renderer* object, meaning it is mandatory to set the material states during each draw call. The pilot implementation contains several implementations (extensions) of the *ISimObject*, such as the *SWEPatch*, *TessPatch* and more.

7.3.4 Renderer

The *Renderer* class represents an object used for hiding some of the complexity of working with Direct3D device, providing instead a set of relatively simple methods for rendering the simulation objects. The initialization of the device itself is done in the constructor of the class via the method *createDevice*, immediately followed by the *initPipeline* method, which takes care of view-port creation, basic device state management, as well as the creation of the backbuffer. The depth-stencil buffer creation was separated into the *initDepthStencilBuffer* method.

Rendering itself is initiated via a call to the *render* method, which receives an instance of the *Camera* class as well as a vector of *ISimObjects*. The *Renderer* performs basic preparation of the device state, and then simply delegates to the *draw* methods on the provided *ISimObjects*, eventually presenting the render result.

The *Renderer* also serves as the manager of the available *IMaterial* objects, and allows the retrieval of a specific *Material* using the *getMaterial* method.

7.3.5 TerrainManager

The *TerrainManager* class is responsible for storing the geographic terrain and water data, as well as taking care of the generation of shared water surface textures generated by the Tessendorf method. The class takes care of texture creation and CUDA initialization in the constructor, using the *SharedTexture* structure to store data about the textures accessible by both Direct3D and CUDA kernels.

In the calls to *update*, *TerrainManager* uses CUDA kernels to create textures describing the heightfield and normal map of the resulting Tessendorf surface. In order to allow writing to D3D textures, the textures need to be mapped for CUDA usage before the kernel execution. The kernels themselves use linear memory to store the results, which is then copied to the CUDA array representing the mapped texture. The exact process of generating the texture has been outlined in the algorithmic section of this thesis. In addition to generating the shared Tessendorf texture, the *TerrainManager* also triggers *update* methods for tiles requiring the SWE calculations as well as copying the Tessendorf heightfield data to corresponding SWE cells where necessary before unmapping the graphical resources, thus again allowing Direct3D to access the textures.

As for the *draw* method, the *TerrainManager* draws only tiles which are within a fixed distance to the camera as well as within the view-frustum. Both the water and terrain geometry is rendered for each tile to be drawn, provided that the tile contains both. In many cases, the terrain part of the tile will be non-existent, which is represented by being set to *nullptr* value.

The actual access to tile data is always done via the *getTile* method, which provides the tile data based on the supplied coordinates. The tiles themselves are instances of the *TileData* class, and are stored using a hash-based cache. The cache contains a fixed amount of buckets, and solves the hash function collision by chaining the tiles together. If no *TileData* is present in the cache for the requested coordinates, the corresponding *TileData* will be created transparently and added to the cache.

The creation of *TileData* is in itself fairly complicated, as the *TerrainManager* needs to load a texture from the drive storage, and use this texture both as a heightmap for the terrain component and as a source of SWE cells for the water simulation. The loading of heightmap is done using the DirectXTK function *CreateDDSTextureFromFile*; however, we still need the normal map to allow for at least a basic form of rendering. The normal map is generated using the same CUDA kernel that is used for generating the Tessendorf normal map.

Next, the cells for SWE are created when necessary. Instead of the already mentioned function *CreateDDSTextureFromFile*, we use the function *LoadFromDDSFile* defined in DirectXTex. The reason for this is that this method

creates a CPU-side representation of the texture, which allows us to easily access the pixel values in the texture from the host. We then create a new instance of *CellManager* class, and fill it with cells using the method *addCell*, setting the cell properties such as height and terrain height based on the data in the texture.

Finally, the complete *TileData* object is gained by combining the water and terrain objects into one structure, which is then returned as the result of the *getTile* operation.

7.3.6 CellManager

The *CellManager* class is responsible for storing a set of *FlowCells* and allowing access to them. The primary purpose of the class is to hide the exact implementation of the cell storage, allowing for easy changes. Currently, the *FlowCells* are stored in a pre-resized vector, which results in very fast lookup times, but inefficient memory usage in case of sparse fields. However, the implementation hiding would theoretically allow us to, for example, use caching by cell coordinates when necessary.

7.3.7 FlowCell

The *FlowCell* class represents a single SWE cell, storing the fluid properties such as water height and terrain height in the cell center, and the velocities in the cell boundaries. Since the changes to fluid properties must be usually applied together across the whole cell grid, the cells allow for storing modified values for later usage in the *newHeight*, *newVelocityX* and *newVelocityZ* fields. The changes can be applied on demand using the *applyVelocityChanges* and *applyChanges* methods.

7.3.8 Cameras

The abstract class *Camera* provides methods and some fields necessary for description of a 3D space camera. The primary purpose is therefore in storing the two standard 3D transformation matrices, the view matrix and projection matrix. The projection matrix depends only on the projection type and view-port properties and can therefore usually be set only once during the creation of the camera. The view matrix depends on the position and orientation of the camera, and must be therefore updated as these values change. This modification is hidden from the camera user, who can access the matrices only as *const* values. Instead, the matrix changes are applied transparently in the implementation of the virtual methods in the classes extending the *Camera*.

The *BasicCamera* class extends the *Camera*, implementing the methods in a way that is common to most cameras, i.e. taking care of keeping the parameter ranges within specified values in the setter methods, as well as introducing the abstract *recalculatePosition* method that triggers the modification of the view matrix. This allows the classes extending the *BasicCamera* to focus only on how this modification is performed.

There are two classes extending the *BasicCamera*. The *CenteredCamera* is a camera observing the origin of the world space, i.e. the vector $(0, 0, 0)$. The camera therefore allows both horizontal and vertical rotation around the origin as

well as zooming. The *CenterdCamera* was used initially during the development and has been later replaced by *FreeCamera*, which allows arbitrary movement in the world space, along with rotation. To this end, the *FreeCamera* contains methods *moveForward* and *moveBackward* as well as *getDirection*, in addition to those methods inherited from the *BasicCamera* and *Camera*.

For a complete overview of the relationships between cameras, see Figure 7.1.

7.4 Swapping Models

As the primary purpose of the application to allow experimentation with different ocean simulation models, we have tried to decouple the simulation part of the process from the rendering as much as possible. On the other hand however, going too far in the separation could potentially result in a notable performance loss. To avoid this, the separation goes only as far as to allow easy modifications of the existing methods and their addition, but without any design decisions that could massively influence the performance.

The core idea for allowing easy modification is to unify the way in which the geometry is generated from the models, and to delay the actual surface creation for as long as possible. In our implementation, we have chosen to take advantage of the cutting-edge hardware tessellation on the GPU, which essentially means that no specific geometry is passed from the host to the graphics device. Instead, we pass a set of vertices (usually called *patch* in the tessellation context) and the hull and domain shaders generate the actual surface vertices and assign their properties, including position. The details are explained in the Chapter 6.4; however, the choice of hardware tessellation has impact on the swapping of the models. Essentially, the shaders need to read the height and normal data from a Direct3D texture. We have therefore elected to make the heightmap and normalmap stored in a texture the unified model for all simulation methods. This generalization is reasonable, as while generating a Direct3D texture might not be the natural choice for some methods and may require additional processing, all methods must generate the heightfield in some form sooner or later. This approach also allows us to use the same level of detail methods for all water geometry, irrespective of what method was actually used to generate the geometry in the first place.

The actual placement of the implementation of the new method depends on the method itself. In theory, if the data and calculations differ for each tile, the easiest way is to create a new implementation of *ISimObject* via extending it and overriding its methods, as is the case with, for example, the *SWETessPatch* class. Next, it is necessary to change the creation of *TileData* in the *TerrainManager::getTile* method to use the new implementation, at least in those cases that require it. As the *TileData* provides the water model to other parts of the system only in the form of an *ISimObject* pointer, no changes need to be done to the usage. Note however that the new implementation must correctly set the material for rendering and also provide the simulation result in the corresponding form in its implementation of the draw method.

The approach outlined above is well suited for methods that generate unique data per tile, such as the SWE and NSE physical approaches. It is however unsuitable for methods that perform the same calculations regardless of data

and share the resulting heightfield across many tiles, which is the case of the Tessendorf method. In this case, we would suggest taking the same steps we have taken with our Tessendorf implementation – the actual calculation is performed in the *TerrainManager* itself, and the texture is bound to the material shared between the tiles. This ensures that no unnecessary operations are performed during the update and draw passes.

7.5 Materials

To further aid the decoupling of rendering and simulation parts of the application, we have introduced a material system. Apart from hiding the low-level implementation dealing with setting the Direct3D states and working with shaders, the material system also has a secondary purpose in facilitating easy code reuse and allowing new simulation objects and methods to be quickly integrated without having to implement full rendering pipeline usage.

The most important abstraction in the material system is the *IMaterial* abstract class. The class contains a single abstract method *apply*, which requires instances of *Renderer* and *Camera* classes as parameters. As the material can access the Direct3D device and device context via the *Renderer* instance, the materials have full control over all aspects of rendering. To apply a material, simulation objects need to request a reference from the *Renderer* class managing the materials. The *getMaterial* method performs a lookup based on a string identifier and returns a pointer to the material masked as the abstract *IMaterial* class. The downside to this approach is that before the specific properties of the material can be accessed, the instance received from the *Renderer* by using the *getMaterial* method needs to be cast to the specific extension of *IMaterial*. As this cast is inherently unsafe, it can result in run-time errors and unpredictable behavior; fortunately however, this kind of class mismatch can only be created due to coding error, and will be immediately obvious during the first render of the new object.

The classes implementing *IMaterial* must therefore take care primarily of setting up the graphics pipeline – setting shader objects for the relevant stages of the pipeline, preparing the buffer objects for the shaders and binding them to the graphics pipeline. Note that the responsibility for correct pipeline setup lies with the author of the extending material class, who must also take care that the pipeline configuration is consistent with its usage in the simulation objects, e.g. rendering primitives of one of the tessellation patch types requires also the binding of hull and domain shaders in the corresponding stages, otherwise the rendering pass will result in Direct3D errors. The same goes for vertex types – the vertex data provided by the geometry must contain all values required by the shader, in the same order and using the correct semantics.

The pilot implementation currently contains several classes extending the *IMaterial* class. The base of all other material classes is the abstract *Material* class, which contains basic loading of pixel and vertex shaders, input layout description creation as well as the pixel and vertex buffer creation and assignments. Two classes inherit from the *Material* class. *ClassicMaterial* represents a traditional pipeline configuration, with only vertex and pixel shaders being active, requiring the provided primitives to be one of the traditional types, e.g. triangle

list or triangle strip. *TessMaterial* on the other hand represent a tessellation-enabled pipeline, indicating that hull and domain shaders should be used and that the primitives supplied to the material for rendering need to be of one of the types supporting tessellation. Another two classes extend the *TessMaterial*. The *TerrainMaterial* class is a simple extension using a different pixel shader, but is otherwise identical. The *BlendMaterial* introduces three additional textures (SWE heightmap and normalmap, along with a blendmap) for use with the blending domain shader, as described in the Chapter 7.7. For a better illustration of the relationships between the various materials, see the class diagram in Figure 7.2.

7.6 Input Data

While we have mentioned that the terrain data are loaded from a hard drive, we have not specified the exact form that is required for the data to be processes accordingly. All data must be in the folder "Content" located in the current working directory of the application. The height of each terrain tile is controlled by a texture named *"X-Y.dds"*, depending on the coordinates of the tile. This texture must be in the Direct Draw Surface (DDS) file format compatible with Direct3D 11. The pixel format must be 32 bits per channel ABGR floating point format. These same basic limitations apply to a pre-made blendmap for a tile, with the only difference being that the texture name is now *"bX-Y.dds"*, with the "b" prefix indicating that this is a blendmap, not a heightmap texture. Neither the heightmap nor the blendmap are necessary for displaying a tile – a missing heightmap results in a pure Tessendorf tile, and a missing blendmap defaults to linear interpolation along the tile borders.

7.7 Rendering

As the rendering is not the focus of this thesis, and is in fact a field of study of its own, we have included only a simple rendering and lighting implementation. Nevertheless, we feel obligated to at least mention the structure and steps taken for the rendering and lighting in this chapter dedicated to technical documentation, as it is information relevant to anyone wishing to modify the application.

The exact configuration of the graphics pipeline depends on the used material. As was mentioned before in Chapter 6.4 and Chapter 7.5, using hardware tessellation changes the role of the vertex shader and delegates some of its responsibilities to the domain shader. The pilot application therefore contains two versions of vertex shader – *OceanVertex* and *OceanVertexTess*. *OceanVertex* performs the transformation from world space to screen space, whereas the *OceanVertexTess* leaves this to the domain shader and merely passes the vertices in world space to the hull shader. The hull shader is contained in the *OceanHull* shader file and is only used when the tessellation is active, calculating the per-patch constants and control points, both of which has been described above in Chapter 6.4.

As we require domain shaders both for the case with and without blending, there are two versions of domain shaders as well. As was mentioned above, the primary role of the domain shader is to interpret the surface represented by the

vertices from the hull shader and transform the resulting vertices to screen space. The *OceanDomain* shader implements this by using bi-linear interpolation to find the texture coordinates of the vertex, and then sampling the heightmap and normalmap to find the properties of the resulting vertex, before transforming its position to screen space. When blending is activated, the *OceanDomainBlend* shader is used instead, extending the parameters of the shader by including additional heightmap, normalmap and blendmap. The process of blending was already described above – a floating point blending factor is read from the blendmap and linear interpolation is applied both to the displacement defined by the heightmaps and the normal vectors.

The final stage of the rendering pipeline is defined by the pixel shader. The application contains two pixel shaders – *OceanPixel* and *TerrainPixel*. The *TerrainPixel* shader is used for rendering matte geometry, such as terrain, and uses a variant of the Phong shading algorithm [19]. The *OceanPixel* instead implements an approximation of the Fresnel term and uses it to combine a reflective color with a refractive color, representing a very simple first-order approximation of the two principal ocean lighting components. As the application does not contain a skybox, the reflective color is a constant. In a more complex scenario, we would have to introduce an environment map and sample it to get reflective color, as suggested in [2].

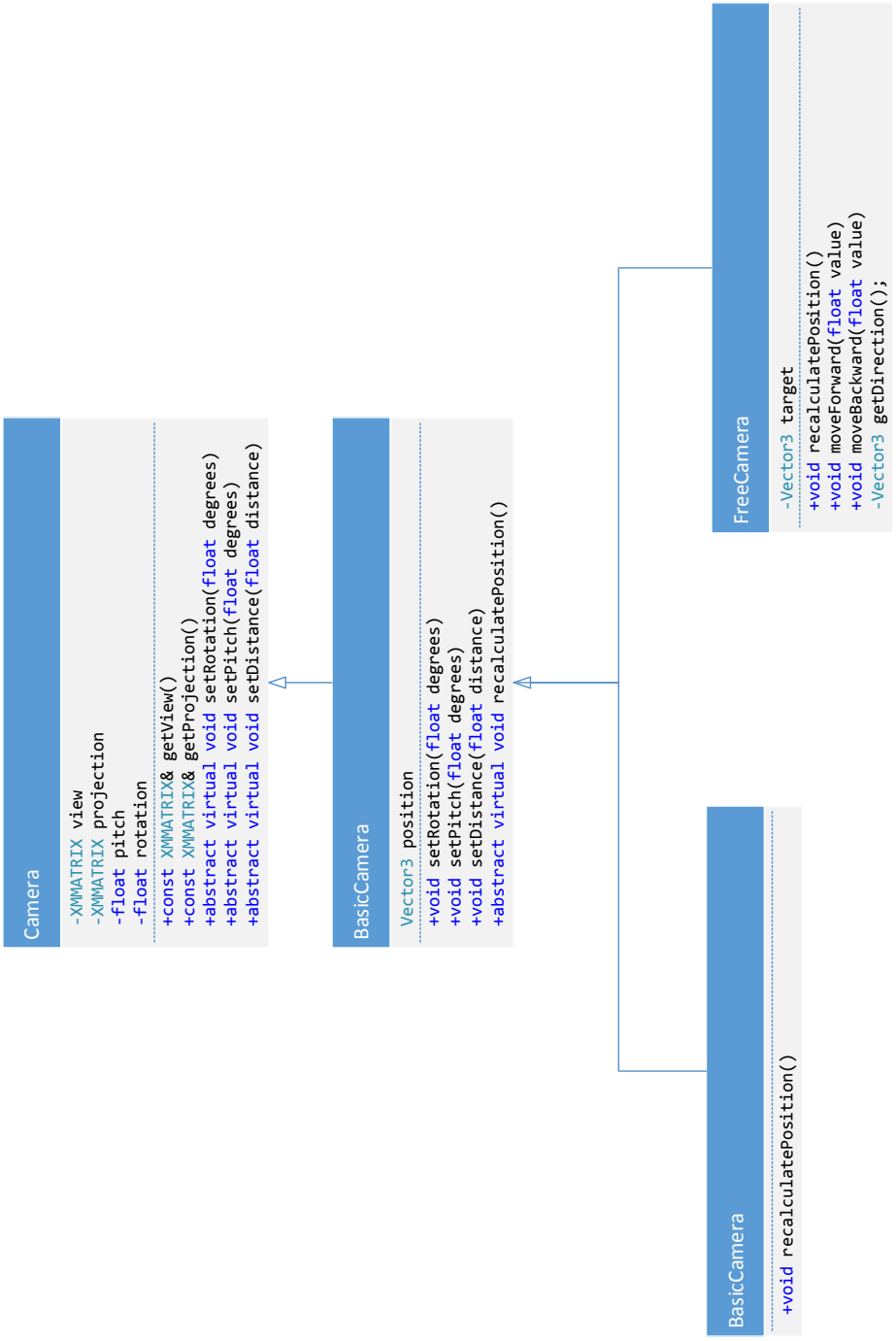


Figure 7.1: Class diagram describing the camera hierarchy. Note that only the most important fields and methods are included.

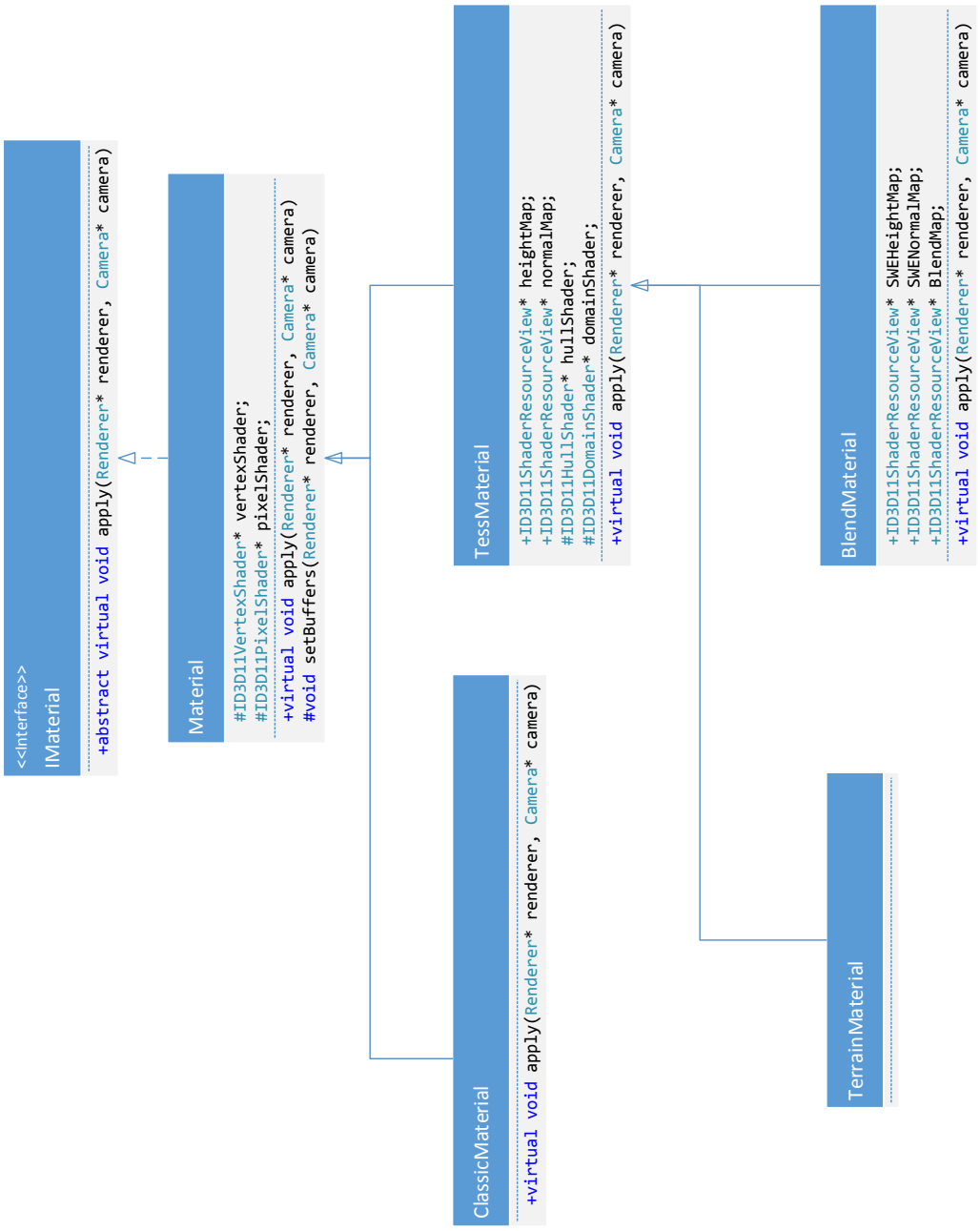


Figure 7.2: Class diagram describing the material hierarchy. Note that only the most important fields and methods are included.

8. Performance

In this section we briefly present the performance results of our pilot implementation, along with identification of several bottlenecks and suggestions for improving the performance in the future versions of the method implementation.

8.1 Environment

The measurements were taken on a machine consisting of Intel Core i5 2500K CPU, 16 GB RAM and an Nvidia GTX 760 class GPU. All of the measurements were taken in the Release configuration and cannot be interpreted in any other context.

8.2 Results

The pilot implementation is somewhere below the real-time framerates in terms of performance. The chief factor governing the overall performance is the size of the SWE grid used in the SWE tile. With a size of 64 cells per tile, the application reaches about 40 frames per second, which can be reasonably described as a real-time capable value. The 64 grid resolution is not very visually appealing however, so we instead use the size of 128 in the pilot application. While only reaching about 25 frames per second, the results are much better while the animation is still reasonably fluid. While the value of 25 frames is not real-time capable, we believe that with improvements in LOD schemes and implementation of various parts of the application, the method can be made into a truly real-time capable solution usable for example in the game industry.

8.3 Main Bottlenecks

To identify the main bottlenecks of the application, we have used performance analysis tools available in Visual Studio 2012. The approach taken was sampling, where the performance analyzer takes snapshots of the application at regular intervals, recording the call stack at the sampled point as well as the current function and line being executed. As expected, the bulk of the most expensive operations is performed in the SWE simulation. In particular, we have identified several key bottlenecks that slow down the simulation as a whole.

The first important bottleneck is the *CellManager::getCell* method with about 15% of exclusive samples. This observation suggests that while our decision to separate the implementation of the cell memory storage from the SWE might have been correct from the design perspective, as it, for example, allows for the usage of hashed access to cell values, it was an incorrect decision from the performance point of view. While the method is not demanding on its own, its widespread use in the SWE simulator means that it is called very often. We would therefore suggest that future implementations rather sacrifice the modifiability of the code and use an inline replacement of the *getCell* method merely converting

the coordinates of the cell to an index into a linear array memory. Extra effort would be required to enable interaction between two SWE tiles.

Another bottleneck in the application appears to be the integration itself contained in the method *SWEPatch::performIntegration*. This was expected, as the integration step has to perform multiple passes over the entire SWE domain, accessing surrounding cells and performing a lot of computations in the process. The method usually amounts to about 15% of exclusive samples. We expect that increasing the performance of the step without radically changing it would be complicated. Some possibilities worth exploring are attempting to eliminate branching, inlining calls to simple functions and combining the height and velocity integration steps together (even though this will also fairly reduce the readability of the code).

The standout bottleneck of the application appears to be in the implementation of the *SWEPatch::finalize* method, which is responsible for copying the temporary height and velocity values to the current height and velocity fields for all cells, as well as generating normal vectors for the surface geometry. Apart from doing a lot of work on its own with 10% of exclusive frames, the method also calls the virtual method *copyToGPU*, which is responsible for copying either the vertex or texture data to the GPU and takes additional 6% of exclusive samples. In terms of inclusive samples, which means mostly calls to CUDA and Direct3D functions, the overall inclusive cost of the finalize method is about 23%. The method could be improved in several ways, the most promising of which appears to be the move to the GPU.

8.4 Increasing Performance

Apart from the modifications outlined above, it is our belief that while complicated, moving the SWE implementation to the GPU might dramatically improve the performance of the application. Such a move would mean that the costly step of interpreting the vertex data as textures and copying these textures to the GPU could be avoided altogether, as the textures could be accessed directly on the graphics card. The massive GPU parallelization and the nature of the SWE cell-based data suggests that there could be a significant increase in performance of the computations as well. This modification is described in more detail in the next chapter of the thesis, including discussion of several potential complications.

9. Future Modifications

In this section we present possible future modifications of our combination method and the accompanying pilot application we would like to focus on in the future, and those that we believe might improve the method either in terms of the result quality or performance. For each modification we consider what changes would need to be incorporated into the current form of our combination method, what problems or limitations it might solve, and what complications we expect to arise.

9.1 Depthmaps

One approach we have considered originally was to use a texture describing the shape of the ocean floor (we will refer to such texture as a "depthmap" from now on) to initialize the cell heights and also as an influence in generating the blendmaps for tiles. However, we have found that such textures are hard to come by – most heightmaps representing islands and other ocean-related terrain only contain the terrain above the sea level. We have therefore abandoned the original idea in favor of using hand-made blendmaps and repositioning the available heightmaps vertically to be partially submerged. While this produces reasonable results, we still believe that using complete depthmaps will produce result of superior quality compared to the current result. One possible way of getting the depthmaps would be to use a random terrain generation algorithm or results from entire terrain generation suites, such as Terragen [20].

Modifying the pilot application to include depthmaps should actually be straightforward – we need to change the tile loading step in *TerrainManager* to read the height data in a different way to include negative terrain heights, setting the cells to the resulting terrain height. In addition, the blendmap creation would now not consist of loading a blendmap as a separate texture from the drive or using a simple blendmap based on tile borders, but instead would generate the blendmap based on the terrain data represented by the heightmap.

Overall, we believe the depthmaps would improve the quality of the result due to the creation of more natural blendmaps as well as cell terrain height values. This would allow the SWE to respond to the shape of the seabed, which would produce more dynamic behavior of the water surface. One problem that would need to be addressed is finding the best values for parameters controlling the cell and blendmap creation, i.e. which depths are considered infinite and how the depths impact the creation of the blendmap.

9.2 Irregular Borders

With the addition of depthmaps it would make sense to be able to define any shape of the cell sets using irregular cell placement instead of a fixed grid, as is the case with the current implementation. Such irregular SWE cell sets have been occasionally used and described in the literature. The core idea is to use a hash function to control the actual placement of cells in the memory. As we have anticipated the possibility of implementing this hashmap scheme, the actual rep-

resentation of the cell memory is hidden and cells are uniformly accessed via the *CellManager* class. Implementing this modification would therefore theoretically consist only of modifying the way *CellManager* works with cells and modifying the cell generation in the tile loading step. However, note that the current implementation takes advantage of the grid placement to facilitate fluid transfer to and from neighbouring tiles – the hash table implementation would need to find another solution to this problem.

Another aspect of using irregular borders as opposed to the fixed grid is that it becomes harder to decide which part of the shape is "inside" (requiring the placement of the cells) and which is "outside" (pure Tessendorf). It is also harder to decide where to place the damping fields and the fixed-height cells for method transition based on the wind direction. We assume that both of these problems are solvable and we would probably opt for experimenting with some modification of the sweep line algorithm [21].

In addition to better reflecting the shape of the ocean floor based on the depthmaps, the irregular borders could be also used to alleviate another limitation of the current implementation. As the fixed-height cells are currently placed on tile borders, the transfer of water movement not aligned with the axes suffers from inaccuracies. We propose to define the transition border in such a way as to be perpendicular to the wind vector as to allow best transition quality in the direction of the water movement, as illustrated on Figure 9.1. However, this introduces some problems, notably that the tile grid would include cell elements overlapping neighbouring tiles. It would therefore be necessary to find a way to define the new cells in a reasonable way across the whole scene without any overlaps.

9.3 SWE LOD Scheme

In the current implementation, the SWE cells have fixed size after their creation, regardless of the current observer position. While we have included a near and far model of water and stop updating the SWE cells when applicable, this LOD approach works only on a boolean basis – the cells are either being updated, or they are frozen. To improve the performance and scalability of our method, it would be advisable to include some form of view-dependent LOD method for the cells as well. A good starting point would be the system described in [22], but more research would be required before selecting a LOD method best applicable to our situation, as well as requiring a considerable effort to incorporate into the combined method and pilot application.

9.4 SWE GPU Conversion

In the current implementation, the SWE solver runs entirely on the host CPU. In addition to lacking any parallelism, this placement of the solver also means that each SWE patch must copy several textures to and from the GPU each frame, introducing a significant performance overhead. Considering the fact that the updating of the SWE solver consists of applying several numerical steps across a set of cells, it stands to reason that the GPU might provide a good environ-

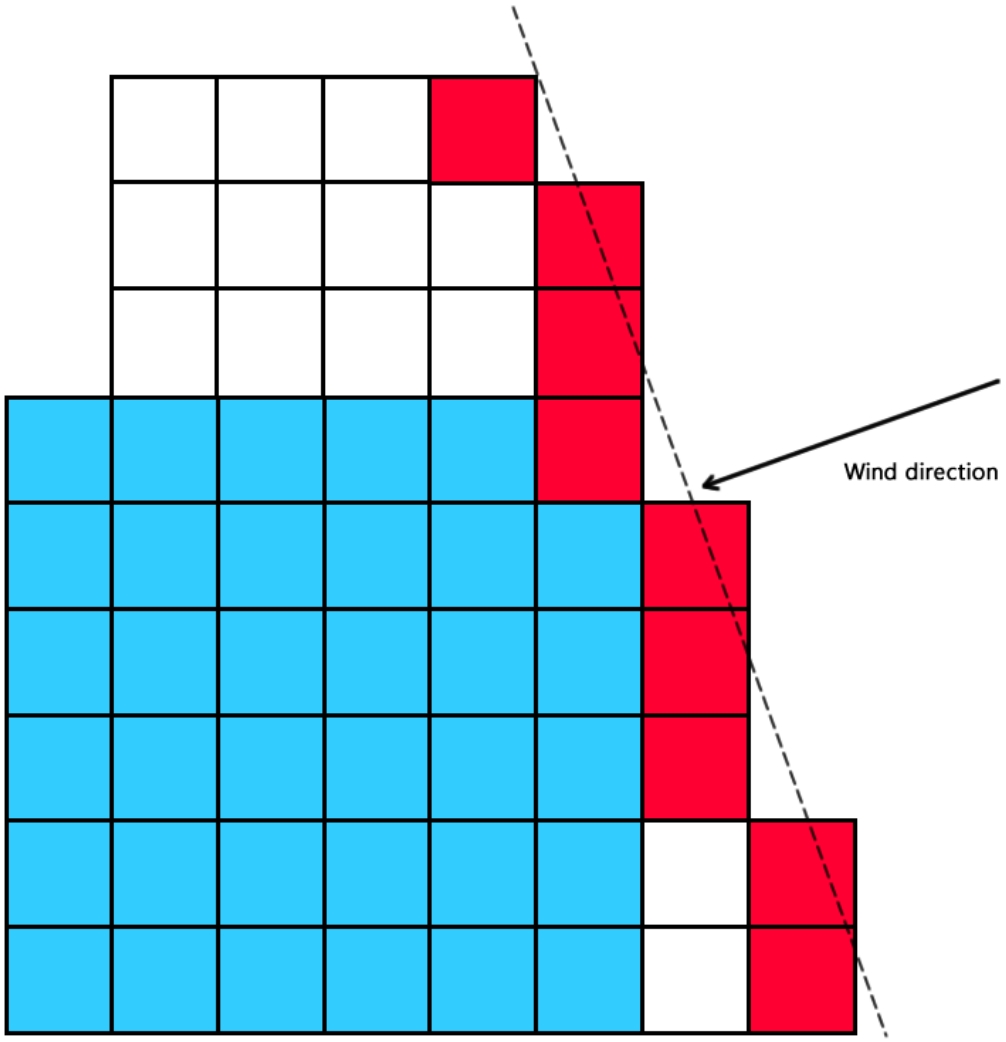


Figure 9.1: Example of irregular border shape used in place of the current square-shaped model corresponding to a tile. The original tile cells are blue, while the new cells are white and the fixed-height cells are red.

ment for such calculations and that moving the SWE to the GPU would result in notable performance improvements. However, implementing such a change would also present several problems. While the cells represent individual entities and could therefore naturally be mapped to threads on the GPU, the integration and advection steps require data from the neighbouring cells. As accessing global memory without any pattern is a notoriously costly operation in context of GPGPU computing, an important aspect of the SWE to GPU conversion would be designing a cell storage scheme that could take advantage of the coalesced loading [1] property of the GPU. We would suggest starting with converting the current pitched memory representation of the cell grid into 2D blocks corresponding to

[1]GPU device attempts to minimize global memory access by performing load and store instructions for entire thread blocks, where applicable. Most efficient way of grouping threads is therefore organizing the memory accessed by a thread block as a continuous segment. For more information, see [23].

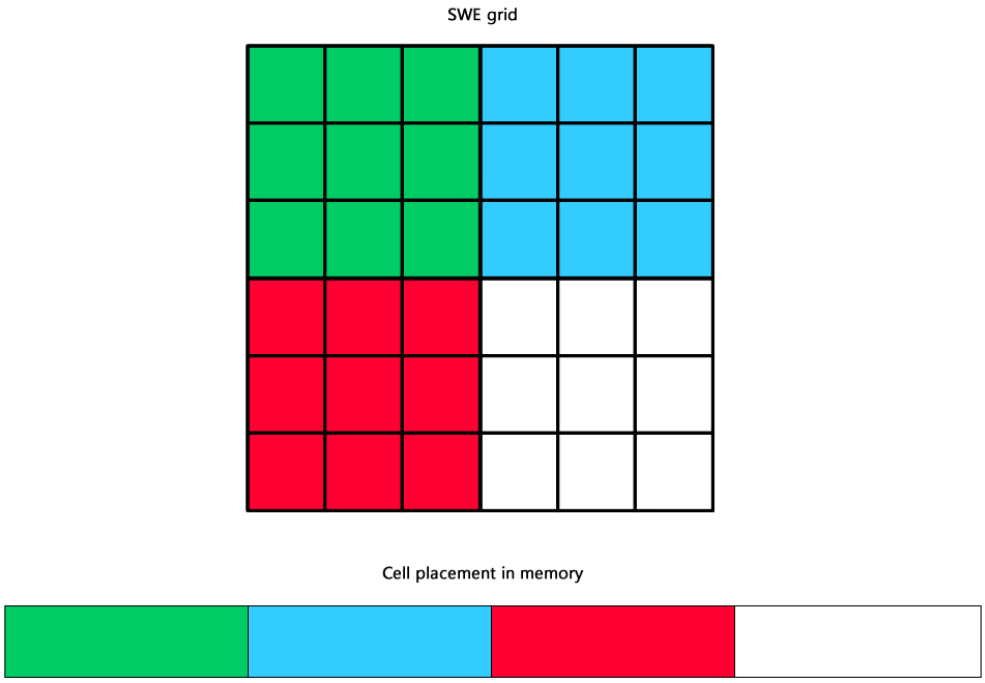


Figure 9.2: Proposed memory placement of cells in case of moving the SWE implementation to the GPU. Differently colored cell sets correspond to thread block sizes on the GPU, and are placed together in the global GPU memory array to better support coalesced loading.

the size of a thread block on the GPU, as illustrated on Figure 9.2.

Another problematic aspect of the conversion would be branching. Branching is another operation that is costly on the GPU, as the GPU thread blocks essentially have to perform all variants that are required by the threads in the block. This should be solved by replacing branching with arithmetical operations where applicable and by ensuring that the amount of branching variants in a block is minimal. Fortunately, in the current form of our method the properties that influence branching (such as damping and fixed-height properties) are distributed fairly regularly and only on the edges of the grid, which should mean that they slow down only a limited amount of thread blocks adjacent to the border, depending on the size of the block.

When used in conjunction with irregular borders, the conversion would become a harder problem due to several assumptions becoming invalid. As the cell placement in memory would now be subject to a hash function, it would be hard to ensure that the coalesced loading is used properly, and the fixed-height cells might also be contained in many blocks. In fact, it might be advantageous to abandon the hashmap entirely in favor of wasting memory but reducing the randomness of global memory access, effectively storing the entire grid with special "null cells" located at coordinates outside the boundary defined by fixed-height cells. Overall, we believe that while the conversion to GPU might prove problematic, the potential performance gain is worth at least attempting the modification in the future.

9.5 Better Rendering

While rendering is not a significant part of this thesis, we believe that including a realistic ocean lighting model would dramatically improve the overall look and feel of the system. The most interesting paper we have read on the subject is the recent work of Brunetton, Neyret and Holzschuch [3] as it describes a complete rendering and lighting system including LOD methods and offers some visually impressive results. The rendering system is built for the Tessendorf method which uses tiles, and should therefore be applicable to our combined solution with some modifications. Implementing such a complex lighting and LOD system including the method-specific modifications must be expected to take a lot of time though, even though we believe the results would be worth the effort.

In addition to lighting, it is possible to include additional effects such as foam and spray. Our combined method might actually be easily extensible to include these phenomena, as the cells in the SWE domain provide physical parameters that can be used to indicate the generation of these effects, e.g. cells with a lot of incoming fluid can be expected to produce sprays. For the Tessendorf domain, it is possible to use approaches suggested by Jensen and Goliáš in [2].

10. Conclusion

In this thesis we have attempted to outline an approach for combining a selection of methods used for simulating deep water and shallow water waves in order to end up with a system having the favorable properties of both empirical and physical approaches. To our knowledge, no system containing the conversion from the Tessendorf method to SWE domain is described in the literature, so we were initially unsure if the idea of combining the methods was feasible in the first place. Nevertheless, we wanted to explore the possibility, as we believe such a system could be capable of offering a visible quality improvement over approaches currently used for ocean simulation in the game industry.

To achieve this goal, we have performed a survey of currently existing popular ocean simulation methods based on multiple relevant papers and analyzed these methods from the viewpoint of the combination method we were trying to create. Then we defined the criteria we are focusing on and selected the two methods based on these criteria. We have implemented the two methods and the combined system, detailing the modifications that were required to enable the combination and the combination scheme itself. The resulting method is capable of simulating large ocean surfaces using the Tessendorf method while transitioning to SWE approach near terrain objects, showing a visible influence the Tessendorf method has on the SWE domain. The SWE simulation near the shore is a full-fledged physical simulation having all the favorable properties associated with such approaches, such as high interactivity and accuracy.

Of the methods we tried, we have found that the most appealing results were obtained by the method with least influence on the physical SWE solver. We have speculated that this is because any changes that are externally applied on the SWE domain are by their very nature introducing physically incorrect behavior and inaccuracies into the simulation. We have formulated a hypothesis that the less physical properties are changed in the transition between the methods, the more physically correct the result will be, based on this observation.

We have also designed a LOD scheme applicable regardless of what methods were used to generate the surface, defining and using a common data storage method in the form of textures representing the height and normal data of the resulting surface. We have taken an approach of deferring the combination of the methods until the last possible moment, allowing us to use hardware tessellation.

Technologically, we have attempted to take advantage of modern technologies such as Direct3D 11, Nvidia CUDA GPGPU computing toolkit and hardware tessellation using domain and hull shaders, taking the extra time to learn these technologies and methods in order to maximize the relevance of the result.

To conclude, we believe that our combination method has shown the potential of the idea of combining the methods that were historically used separately into one system. While the current version of our combination method and the pilot application has some limitations, notably the inaccuracies when dealing with diagonal wind directions and the low limit of grid size usable for real-time, we have shown a list of future modifications that we would like to implement next and that we believe should improve the system in these areas. In time, we see our combination method evolving into a complex real-time system offering both the

advantages of physical methods along with the Tessendorf large-scale capability.

Bibliography

[1] Emmanuelle Darles, Benoît Crespin, Djamchid Ghazanfarpour, and Jean-Christophe Gonzato. A survey of ocean simulation and rendering techniques in computer graphics. In *Computer Graphics Forum*, volume 30, pages 43–60. Wiley Online Library, 2011.

[2] Lasse Staff Jensen and Robert Golias. Deep-water animation and rendering. In *Game Developer's Conference (Gamasutra)*, 2001.

[3] Eric Bruneton, Fabrice Neyret, and Nicolas Holzschuch. Real-time realistic ocean lighting using seamless transitions from geometry to brdf. In *Computer Graphics Forum*, volume 29, pages 487–496. Wiley Online Library, 2010.

[4] Jerry Tessendorf. Simulating ocean water. *Simulating Nature: Realistic and Interactive Techniques. SIGGRAPH*, 1, 2001.

[5] Nuttapong Chentanez and Matthias Müller. Real-time simulation of large bodies of water with small scale details. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on computer animation*, pages 197–206. Eurographics Association, 2010.

[6] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3):350–371, 2008.

[7] Nelson L Max. Vectorized procedural models for natural terrain: Waves and islands in the sunset. *ACM SIGGRAPH Computer Graphics*, 15(3):317–324, 1981.

[8] Darwyn R Peachey. Modeling waves and surf. In *ACM Siggraph Computer Graphics*, volume 20, pages 65–74. ACM, 1986.

[9] Alain Fournier and William T Reeves. A simple model of ocean waves. In *ACM Siggraph Computer Graphics*, volume 20, pages 75–84. ACM, 1986.

[10] Jean-Christophe Gonzato and B Le Saëc. On modeling and rendering ocean scenes (diffraction, surface tracking and illumination). In *WSCG'99 (Seventh International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media)*, pages 93–101, 1999.

[11] Namkyung Lee, Nakhoon Baek, and Kwan Woo Ryu. A real-time method for ocean surface simulation using the tma model. *International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)*, 1:15–21, 2009.

[12] Nvidia Corporation. Ocean surface simulation. https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/OceanCS_Slides.pdf, 2011. Accessed 24-July-2014.

[13] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[14] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 49–57. ACM, 1990.

[15] Miguel Gomez. Interactive simulation of water surfaces. In Mark DeLoura, editor, *Game Programming Gems*, pages 187–194. Charles River Media, 2000.

[16] Nvidia Corporation. Nvidia cuda 5.5 sdk.

[17] Wikipedia. Finite difference — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Finite_difference&oldid=618451623, 2014. Accessed 27-July-2014.

[18] Microsoft Corporation. Tessellation overview. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476340%28v=vs.85%29.aspx. Accessed 26-July-2014.

[19] Wikipedia. Phong reflection model — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Phong_reflection_model&oldid=602498842, 2014. Accessed 27-July-2014.

[20] Terragen software suite homepage. http://planetside.co.uk/products/terragen3. Accessed: 26-July-2014.

[21] Wikipedia. Sweep line algorithm — wikipedia, the free encyclopedia. "http://en.wikipedia.org/w/index.php?title=Sweep_line_algorithm&oldid=601866462", 2014. Accessed 26-July-2014.

[22] Hilko Cords and Oliver Staadt. Real-time open water environments with interacting objects. In *Proceedings of the Fifth Eurographics conference on Natural Phenomena*, pages 35–42. Eurographics Association, 2009.

[23] Mark Harris. How to access global memory efficiently in cuda c/c++ kernels. http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/, 2013. Accessed 25-July-2014.

List of Definitions and Abbreviations

FFT	Fast Fourier Transform
NSE	Navier Stokes Equations of fluid motion
SWE	Shallow Water Equations
GPGPU computing	General purpose graphics processing unit computing
AAA game	Video-game with big budget and promotion, generally funded by a distributor
LOD	Level of Detail, used in context of various levels of simulation and rendering
AABB	Axis-aligned Bounding Box
DDS	Direct Draw Surface
CUDA	Compute Unified Device Architecture
MAC	Marker and Cell, grid configuration used in space discretization of the NSE
JONSWAP	Joint North Sea Wave Project, oceanographic spectrum
TMA	Texel, Marson and Arsole, oceanographic spectrum

Attachments

A. Structure of the accompanying CD

Following structure is used for organization of the CD included with this thesis:

- **Binaries** – Folder containing compiled binary files including content resources and DLLs for easy deployment.
- **Content** – Folder containing example content data. This folder must be copied to the application folder.
- **DLLs** – DLL files that must be present in the application root folder before running.
- **Source** – Folder containg complete sources of the project.
- Thesis.pdf – The thesis itself, including technical documentation and application requirements.
- readme.txt – A readme file

B. Application Requirements

The requirements for running the application are as follows:

- Intel i5 2500K or faster
- Nvidia GTX 760 or faster, including Direct3D 11 and CUDA support (NOTE: the manufacturer of the GPU must be Nvidia, to enable CUDA).
- 4 GB RAM
- Newest CUDA and Nvidia GPU drivers
- Installed and configured Qt 5.2.1 x86 (may be bypassed in some cases by using the enclosed DLLs)
- Windows 7 SP1 or newer

For compiling the application, additional requirements apply:

- Microsoft Visual Studio 2012
- Installed Windows SDK 8.0
- Installed CUDA SDK
- Installed Qt 5.2.1 x86 SDK

See the technical documentation section of the thesis for a guide on getting these SDKs.