Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Peixoto Antunes de Castro

Shallow Waters Simulation

June 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Peixoto Antunes de Castro

Shallow Waters Simulation

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António José Borba Ramires Fernandes

June 2022

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor António Ramires Fernandes for his continuous help and guidance throughout this work. I would also like to thank my parents for their support and encouragement during the whole process. Lastly, I extend these words of gratitude to my friends, with whom I shared the many successes and struggles.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Realistic simulation and rendering of water in real-time is a challenge within the field of computer graphics, as it is very computationally demanding. A common simulation approach is to reduce the problem from 3D to 2D by treating the water surface as a 2D heightfield. When simulating 2D fluids, the Shallow Water Equations (SWE) are often employed, which work under the assumption that the water's horizontal scale is much greater than it's vertical scale.

There are several methods that have been developed or adapted to model the SWE, each with its own advantages and disadvantages. A common solution is to use grid-based methods where there is the classic approach of solving the equations in a grid, but also the Lattice-Boltzmann Method (LBM) which originated from the field of statistical physics. Particle based methods have also been used for modeling the SWE, namely as a variation of the popular Smoothed-Particle Hydrodynamics (SPH) method.

This thesis presents an implementation for real-time simulation and rendering of a heightfield surface water volume. The water's behavior is modeled by a grid-based SWE scheme with an efficient single kernel compute shader implementation.

When it comes to visualizing the water volume created by the simulation, there are a variety of effects that can contribute to its realism and provide visual cues for its motion. In particular, When considering shallow water, there are certain features that can be highlighted, such as the refraction of the ground below and corresponding light attenuation, and the caustics patterns projected on it.

Using the state produced by the simulation, a water surface mesh is rendered, where set of visual effects are explored. First, the water's color is defined as a combination of reflected and transmitted light, while using a Cook-Torrance Bidirectional Reflectance Distribution Function (BRDF) to describe the Sun's reflection. These results are then enhanced by data from a separate pass which provides caustics patterns and improved attenuation computations. Lastly, small-scale details are added to the surface by applying a normal map generated using noise.

As part of the work, a thorough evaluation of the developed application is performed, providing a showcase of the results, insight into some of the parameters and options, and performance benchmarks.

KEYWORDS Shallow Water Equations, Fluid Simulation, Caustics, Real time, Heightfield

RESUMO

Simulação e renderização realista de água em tempo real é um desafio dentro do campo de computação gráfica, visto que é muito computacionalmente exigente. Uma abordagem comum de simulação é de reduzir o problema de 3D para 2D ao tratar a superfície da água como um campo de alturas 2D. Ao simular fluidos em 2D, é frequente usar as equações de águas rasas, que funcionam sobre o pressuposto de que a escala horizontal da água é muito maior que a sua escala vertical.

Há vários métodos que foram desenvolvidos ou adaptados para modelar as equações de águas rasas, cada uma com as suas vantagens e desvantagens. Uma solução comum é utilizar métodos baseados em grelhas onde existe a abordagem clássica de resolver as equações numa grelha, mas também existe o método de Lattice Boltzmann que originou do campo de física estatística. Métodos baseados em partículas também já foram usados para modelar as equações de águas rasas, nomeadamente como uma variação do popular método de SPH.

Esta tese apresenta uma implementação para simulação e renderização em tempo real de um volume de água com uma superfície de campo de alturas. O comportamento da água é modelado por um esquema de equações de águas rasas baseado na grelha com uma implementação eficiente de um único kernel de compute shader.

No que toca a visualizar o volume de água criado pela simulação, existe uma variedade de efeitos que podem contribuir para o seu realismo e fornecer dicas visuais sobre o seu movimento. Ao considerar águas rasas, existem certas características que podem ser destacadas, como a refração do terreno por baixo e correspondente atenuação da luz, e padrões de cáusticas projetados nele.

Usando o estado produzido pela simulação, uma malha da superfície da água é renderizada, onde um conjunto de efeitos visuais são explorados. Em primeiro lugar, a cor da água é definida como uma combinação de luz refletida e transmitida, sendo que uma BRDF de Cook-Torrance é usada para descrever a reflexão do Sol. Estes resultados são depois complementados com dados gerados num passo separado que fornece padrões de cáusticas e melhora as computações de atenuação. Por fim, detalhes de pequena escala são adicionados à superfície ao aplicar um mapa de normais gerado com ruído.

Como parte do trabalho desenvolvido, é feita uma avaliação detalhada da aplicação desenvolvida, onde é apresentada uma demonstração dos resultados, comentários sobre alguns dos parâmetros e opções, e referências de desempenho.

PALAVRAS-CHAVE Equações de Águas Pouco Profundas, Simulação de Fluídos, Cáusticas, Tempo Real, Campo de Alturas

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

BRDF Bidirectional Reflectance Distribution Function. iii, iv, viii, 19, 20, 44, 51, 52, 93

CFD computation fluid dynamics. 15, 16, 17
CFL Courant-Friedrichs-Lewy. 12

ENO Essentially Non-Oscillatory. 107

fBm fractal Brownian motion. 68
FDM finite difference method. 14
FFT Fast Fourier Transform. 24
FVM finite volume method. 14

GSM Group Shared Memory. 31, 32, 38, 39, 60, 61, 62, 63

LBM Lattice-Boltzmann Method. iii, 15, 16, 17, 94

NDF Normal Distribution Function. 20
NSE Navier-Stokes Equations. 1, 2, 4, 5, 7, 15

PIF Picard integral formulation. 27, 28

SPH Smoothed-Particle Hydrodynamics. iii, iv, 16, 17, 27, 94
SWE Shallow Water Equations. iii, viii, xii, xiii, 1, 2, 7, 8, 9, 13, 14, 15, 16, 17, 27, 28, 30, 78, 82, 83, 86, 93, 106

WENO Weighted Essentially Non-Oscillatory. ix, 27, 28, 30, 33, 37, 39, 107, 108, 109

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

Simulating water behavior in real-time can be very expensive, and thus, a 2D heightfield approximation is often used. This approximation can be achieved by ignoring certain components of the simulation under the assumption that the water is shallow. This chapter provides some context into this field of study (Section 1.1), as well as the goal of this thesis within it (Section 1.2). The final section provides a brief description of the chapters that compose this document (Section 1.3).

1.1 CONTEXT

Water is common element in nature, and as such, is often encountered in virtual environments. Recently, physically based effects have become more and more popular in these environments, which includes both the simulation and rendering of water.

The simulation of the water's behavior is often achieved by numerically solving the Navier-Stokes Equations (NSE) in offline applications, such as for movie special effects. In contrast, interactive systems require the simulation to be performed in real-time while achieving a frame rate of at least 60 Hz. Additionally, most applications require most of the computing budget to be used for other core features, leaving only a few milliseconds for physical simulation. Simply reducing the resolution of the methods used in offline simulations often yields blobby results and removes all the interesting details. Alternatively, many approaches have been developed in order to efficiently approximate the results obtained by those more complex methods.

A common simulation approach is to reduce the problem from 3D to 2D by treating the water surface as a 2D heightfield. Some simple and quite popular methods to simulate water surfaces are based on the wave equation or the pipe model (e.g., Dagenais et al. (2018), Kellomäki (2017)). However, these methods have the drawback of not being able to correctly simulate effects that are based on horizontal motion, such as whirlpools or the flow of a river. This effects can, however, be captured by the SWE, with this approach being used in both interactive graphics (e.g. Chentanez and Mueller (2010), Parna (2020)), as well as in engineering applications (e.g., Horváth et al. (2020)).

The rendering of the water surface can be divided into various parts. An essential component is the reflection of the environment, including the distinct specular glitter caused by the Sun. Considering the case of shallow water, another big component is the refraction of the bottom terrain and the caustics caused in it by the water surface (e.g., Yang and Ouyang (2021)). There are also a variety of effects that focus on addressing limitations of

the heightfield representation, such as adding breaking waves, splashing particles, or small-scale perturbations below the simulation scale (e.g., Ojeda and Susín (2014), Fujisawa et al. (2017)).

1.2 OBJECTIVES

The aim of this dissertation is to develop an application for simulating water behavior by solving the SWE, and rendering the grid it outputs as a water volume. Furthermore, the application must allow both the simulation and rendering to run in real-time on current graphics hardware. To limit the scope of this work, rendering will be limited to the water surface's color (which includes reflections, refractions and shading), caustics, and small-scale details.

This can be split into the following sub-objectives:

- Study and understand the background on this field of study.
- Survey the state-of-the-art for both simulation and rendering techniques.
- Implement a solver for the SWE.
- Implement a visualization algorithm for the simulation.
- Evaluate the results and performance of the full application.

1.3 DOCUMENT STRUCTURE

This dissertation is split into various chapters that roughly match the previously presented objectives, and is organized as follows:

chapter 2 provides the needed background for the dissertation, starting from the NSE and introducing some simplifying assumptions used to achieve a form more feasible for real-time simulation, the SWE. This is followed by an overview of the current state-of-the-art, and a review of the more relevant fluid simulation methods. Then, a review of the state-of-art of rendering methods suitable for the simulation is presented, focusing on the previously mentioned set of effects.

chapter 3 presents the implemented simulation of the water's behavior. The chapter starts with a description of the numerical scheme, followed by the details of the GPU implementation. The next chapter (chapter 4) explores the rendering effects used to visualize the grid output by the water simulation, providing the implementation details. Both chapters end with a brief showcase of its results.

A thorough review of the results is presented in the following chapter (chapter 5). Here the SWE solver implementation is analyzed, reviewing different options, and presenting performance benchmarks. Then, a similar discussion is done for the rendering, going through each of the components individually. The chapter ends with a a showcase of varied example scenes, accompanied by a performance breakdown of the full application.

The final chapter (chapter 6) briefly discusses the main conclusions from the developed work and provides possible avenues of future work .

In addition to the work presented here, the full source code of the developed application is provided online at https://github.com/carlosc20/shallow-waters-nau.

2

STATE OF THE ART

This chapter provides an overview of the state-of-the-art in real-time simulation of shallow water. It starts by providing some background into the common ground of all fluid simulations, the NSE (Section 2.1). Then, it approaches the need of simplified solutions for real-time applications, focusing on the approximation provided by the shallow water equations (Section 2.2). This is followed by an overview of the main concepts of numerical fluid simulations (Section 2.3). Subsequently, the current methods used for developing shallow water solvers are presented and compared (Section 2.4). Finally, rendering methods used for realistic water visualization are reviewed, focusing on those adequate for a shallow water heightfield input (Section 2.5).

2.1 THE NAVIER-STOKES EQUATIONS

The NSE, a reformulation of Newton's Second Law, are often used to describe fluid flow. For incompressible and Newtonian fluids they consist in a set of partial differential equations, the momentum equation (Equation 1) and the incompressibility condition (Equation 2), as depicted in Bridson (2015) where a more in depth analysis can be found.

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \frac{1}{\rho} \nabla p = \boldsymbol{a} + \nu \nabla \cdot \nabla \boldsymbol{u} \tag{1}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2}$$

In Equations 1 and 2, $\boldsymbol{u}$ is the fluid velocity vector, p is the fluid pressure, ρ is the fluid density and $\nabla \cdot \nabla$ is the Laplacian operator. ν is the kinematic viscosity and is equal to η / ρ where η is the dynamic viscosity coefficient. Acceleration due to external forces is commonly represented by $\boldsymbol{g}$ when only gravity is considered. However, in interactive applications other forces often need to be considered, such as those created by user input. Considering that, a more generic $\boldsymbol{a}$ term is used to represent the acceleration due to all external body forces.

2.1.1 *The Momentum Equation*

The momentum equation (Equation 1) describes how the fluid velocity $\boldsymbol{u}$ evolves over time. Most of the components of the momentum equation represent different accelerations caused by forces affecting the water particles: external forces $\boldsymbol{a}$ (commonly gravity, wind or user interaction), pressure gradient $-\frac{1}{\rho}\nabla p$, and viscosity $\nu\nabla\cdot\nabla\boldsymbol{u}$.

Since water has low viscosity, which has a minor impact in the animation of large bodies of water, the term is typically dropped. This is further supported by the fact that numerical methods used to solve the NSE often introduce errors that can be physically reinterpreted as artificial viscosity (Bridson, 2015).

The process of a physical quantity moving with the velocity field of the fluid is called advection. This also affects the velocity itself, in what can be called self-advection, and is modeled by the convective acceleration term $\boldsymbol{u}\cdot\nabla\boldsymbol{u}$.

Advection can also be represented using the material derivative operator, this notation is exemplified in Equation 3.

$$\frac{\partial x}{\partial t}+\boldsymbol{u}\cdot\nabla x\equiv\frac{Dx}{Dt} \tag{3}$$

2.1.2 *The Incompressibility Condition*

The incompressibility condition (Equation 2), sometimes also called the continuity equation, ensures the conservation of mass, that is, the incompressibility of the fluid. In reality, fluids are not actually incompressible, but they are close enough to be considered as such in computer graphics. The equation states that density along the line of flow remains constant over time, therefore divergence of velocity is null all the time, i.e., water is flowing out as fast as it is flowing in from other directions.

2.2 SHALLOW WATER EQUATIONS

This section will explore a special case of water simulation that allows much faster and simpler algorithms, by making certain assumptions, namely that the water surface can be represented as an heightfield and that the horizontal scale of the simulation is much larger than the vertical scale.

2.2.1 *Heightfield approximations*

Simulation models based on the NSE are highly realistic, since they capture the three-dimensional dynamics of fluids. The downside of such models is that, for full-scale water simulations, full 3D simulations are often too slow to achieve real-time rates.

A common solution is to assume the water surface can be represented as a heightfield, in other words, the surface z-coordinate is a function $\eta(x, y)$, bounded from below by the bottom topography and from above by another fluid of negligible density (commonly air). This allows the simulation to take place in a 2D grid, which means that if n cells are required per dimension, the time complexity is typically lowered from $O(n^3)$ to $O(n^2)$.

Another advantage is that the rendering of the simulation results can be done using common, fast rendering techniques, similar to those used for e.g. terrain rendering. Full 3D simulations usually require surface tracking (e.g. level sets (Bridson, 2015)) and expensive volume rendering techniques (e.g. ray marching (Crane et al., 2007)) or mesh generating algorithms (e.g. marching cubes (Thürey, 2007)).

A regular 2D grid can be imagined as a group of adjacent water columns, where the water is defined by a nxm 2D array of depth values. A nxm heightfield is often visualized by creating $(n-1)x(m-1)$ squares that are rendered as two triangles each (Figure 1).

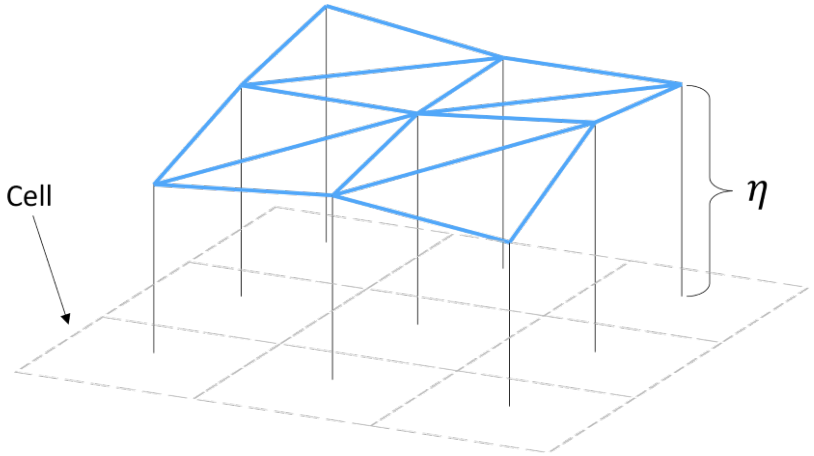

Figure 1: A $3x3$ heightfield with one of the nine simulation cells drawn. The value η of each cell represents the vertical displacement in its center. The cells' centers are connected to form triangles for visualization.

This approach results in a clear limitation, behavior that breaks the water surface can not be represented, such as breaking waves, waterfalls and splashes. However, some of these effects only occur in limited areas in the simulation and can be added using, e.g., a local small-scale particle simulation, while the main body of water is simulated using 2D methods (Chentanez and Mueller, 2010).

In many virtual environments it is common that the terrain is also represented as a heightfield. In this cases it is therefore interesting to simulate the interaction of the water with the terrain below. Assuming that the water flows on top of an arbitrary heightfield terrain $b(x, y)$, and can freely spread to dry areas, it can be represented by the depth $h(x, y) \geq 0$, with surface height $\eta = b + h$ (as shown in Figure 2). Most simulations for this setup use the depth instead of the surface height as the main simulation variable.

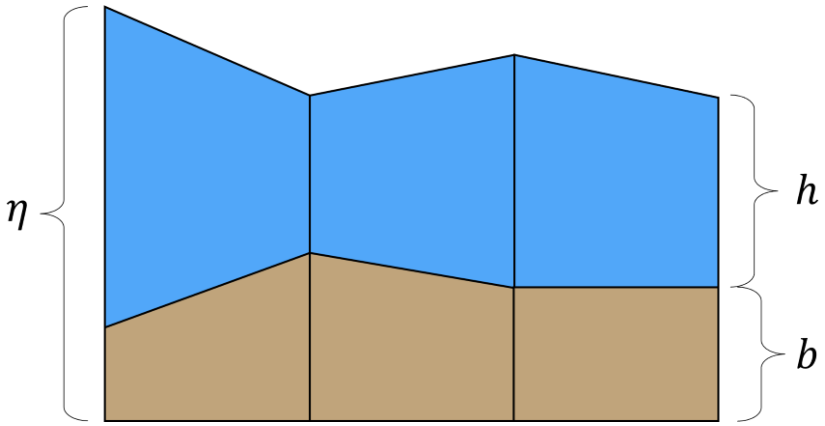

Figure 2: A fluid volume above terrain where both are represented as a heightfield elevation, where η denotes the height of the fluid above zero level, b denotes the terrain height and h denotes the fluid depth, or height above terrain.

2.2.2 *The shallow water equations*

A very common approximation used to create a heightfield simulation on top of a heightfield terrain is to ignore vertical variations in the velocity field, resulting in the SWE. These are a simpler set of equations derived from the NSE, and were first introduced in computer graphics in Layton and Panne (2002).

Another important assumption of the SWE is that the vertical movement of the water is dominated by gravity and pressure, which is a suitable approximation only if other accelerations in the fluid are much smaller than the acceleration due to gravity. By taking the vertical component of the momentum equation (Equation 1), dropping all other terms except the pressure and gravity, and integrating from some coordinate z to the surface η, the hydrostatic pressure equation (Equation 4) is obtained.

$$p(z) = \rho g(\eta - z). \tag{4}$$

As in this model the pressure is fully determined by the depth, there is no longer a need to store it. With these assumptions, the non-conservative form of the SWE (Equation 5 and Equation 6) is derived from the NSE, where g denotes the vertical acceleration of the fluid due to gravity, and h denotes the depth. The horizontal velocities, represented by $\boldsymbol{u}$, are tracked as an average in the whole column of water. The proof is left out for brevity but can be found in Bridson (2015).

$$\frac{\partial h}{\partial t} + \boldsymbol{u} \cdot \nabla h = -h(\nabla \cdot \boldsymbol{u}) \tag{5}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -g\nabla\eta \tag{6}$$

Equation 5 describes the rate of change of the fluid depth. The depth is advected by the term $\boldsymbol{u} \cdot \nabla h$. On the right side, the depth, representing the vertical direction, that has been integrated out, is multiplied by the negative divergence of the velocity, which represents the volume density of the inwards flux to the point.

Equation 6 describes how the velocity $\boldsymbol{u}$ changes. It has two components: the velocity self-advection term on the left ($\boldsymbol{u} \cdot \nabla \boldsymbol{u}$), and on the right the term describing acceleration caused by the pressure difference ($-g\nabla\eta$). As the vertical pressure gradients are assumed to be nearly hydrostatic, this acceleration is simply the acceleration by gravity times the surface gradient.

An alternative conservation law form of the SWE is often used by the scientific community, with some appearances in computer graphics literature (e.g. Brodtkorb et al. (2012), Parna et al. (2018)). This form of the SWE is given by Equation 7 and Equation 8 (as in Parna et al. (2018), where a full derivation can also be found). It should be noted that the velocity vector $\boldsymbol{u}$ is now represented by its two separate components, as in $\boldsymbol{u} = (u, v)$.

$$\frac{\partial \boldsymbol{U}}{\partial t} + \frac{\partial \boldsymbol{F}}{\partial x} + \frac{\partial \boldsymbol{G}}{\partial y} = \boldsymbol{S} \tag{7}$$

$$\boldsymbol{U} = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix} \boldsymbol{F} = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix} \qquad \boldsymbol{G} = \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix} \boldsymbol{S} = \begin{pmatrix} 0 \\ -gh\frac{\partial b}{\partial x} \\ -gh\frac{\partial b}{\partial y} \end{pmatrix} \tag{8}$$

$\boldsymbol{U}$ is the vector of conserved variables (fluid height h and horizontal momenta hu and hv, where u, v are the horizontal fluid velocities); $\boldsymbol{F}$ and $\boldsymbol{G}$ are respectively the x and y directional fluxes; $\boldsymbol{S}$ is the source term due to the underlying bathymetry/topography function b; g is the gravitational constant. The difference between the two forms, and which makes it more common in the scientific community, is that the conservation form produces a correct solution with shock conditions, whereas the non-conservation form might not (Toro, 2009). A shock or hydraulic jump is caused when a fluid at high velocity discharges into a zone of lower velocity, causing a sudden rise in the fluid surface.

2.2.3 *Other heightfield methods*

A simple alternative technique to simulate water surfaces is to further simplify the SWE by dropping the advection terms completely, as was first done in computer graphics in Kass and Miller (1990). Equations 5 and 6 become Equations 9 and 10.

$$\frac{\partial h}{\partial t} = -h(\nabla \cdot \boldsymbol{u}) \tag{9}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} = -g\nabla\eta \tag{10}$$

Deriving Equation 9 assumes that the depth varies only slowly (Kass and Miller, 1990), which means it can work well for calm scenes with local phenomena like puddles or waves around a boat.

As shown in, e.g., Kass and Miller (1990), these linearized shallow water equations are actually equivalent to the 2D wave equation. Bridson (2015) points out that the wave equation has solutions corresponding to waves moving at speed $\sqrt{gh}$. Yuksel et al. (2007) directly made a very fast wave solver from this observation, using "wave particles" traveling at this speed which locally change the height of the water. This idea, based on displacing a 2D domain, was later extended and stands and as a middle ground between Fourier-based methods and SWE solvers (Jeschke and Wojtan (2017), Jeschke et al. (2018)). It allows for a highly detailed surface and interaction with objects/terrain, but also requires a flat domain, so the water can not flow over dry or different height areas.

A different approach that leads to a method similar to the wave equation, is to think of the heightfield water as a group of adjacent water columns, where the flow is modeled by virtual pipes between columns of water on the grid. Hydrostatic pressure is used to calculate the flow in each pipe. This method that was originally formulated by O'Brien and Hodgins (1995) and is often called the pipe method (Kellomäki (2015), Dagenais et al. (2018)).

Figure 3: An example of the difference between linearized (left) and full SWE (right). The whirlpool is not created in the linearized version (Kellomäki and Saari, 2014).

The main drawback of the mathematical model used in methods based on the wave equation or the pipe model is that it only works with a vertical velocity field. Therefore, effects that are based on horizontal motion such as whirlpools or the flow of a river cannot be treated correctly (Figure 3). The SWE can capture these effects because in addition to the heightfield, they describe the evolution of a 2D velocity field normal to the water columns. Because of this, and its similar performance (Kellomäki, 2015) to full shallow water solvers, only the later will be considered on the following sections.

2.3 NUMERICAL SIMULATION

This section will approach the main concepts to consider when numerically simulating the behavior of a fluid.

2.3.1 *Lagrangian and Eulerian Viewpoints*

For a numerical solution, the domain needs to be discrete. When it comes to a continuum (like a fluid) moving, there are two approaches to tracking its motion, the Lagrangian viewpoint and the Eulerian viewpoint.

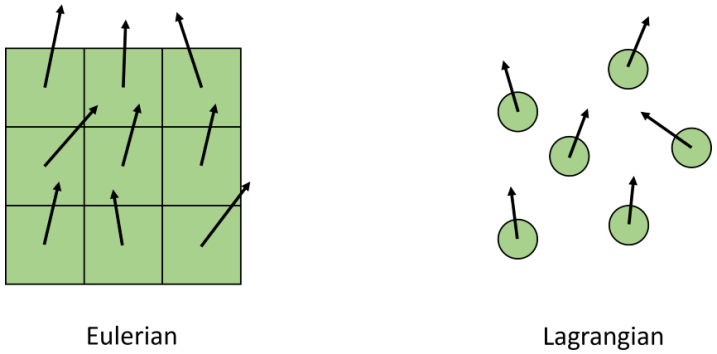

Figure 4: Velocity property represented as a vector on the Eulerian (points in a fixed grid) and Lagrangian (particles) viewpoints

Lagrangian / Particle-based Viewpoint

The Lagrangian approach treats the fluid like a particle system. Each particle represents a discrete blob of fluid and has a set of properties such as mass and velocity (Figure 4, right). In a simulation, a certain number of small particles is followed. The forces affecting each particle are evaluated to get the acceleration of each particle. The particles' properties are automatically moved or advected by them. These properties can then be evaluated in any point of the continuum, e.g., as a weighted-average of the corresponding properties of the particles.

Eulerian / Grid-based Viewpoint

The Eulerian approach looks at sample points fixed in space, that represent small fluid volumes, and sees how the fluid properties measured at those points change in time (Figure 4, left). These points are commonly aligned to a grid, which is why the Eulerian approach is also known as the grid-based approach. In this framework, the fluid properties change in time due to external forces, and also because of advection, i.e., the fact that the fluid itself is flowing past these points.

Comparison

When it comes to advection, for the Eulerian methods it is an additional burden compared to the Lagrangian methods, as in the latter it is automatically kept track of, because the observation point (the particle) is also advected along the velocity field. However, it is easier to analytically work with the spatial derivatives in the Eulerian viewpoint, and to numerically approximate those spatial derivatives on a fixed Eulerian grid than on a

cloud of arbitrarily moving particles (Bridson, 2015). Determining particle neighbors at each simulation step is generally accelerated by the use of specific data structures Koschier et al. (2019).

For the Lagrangian approach, conservation of mass comes easily since the mass property is only associated with each particle and, therefore, is not altered as long as particles are not created or destroyed. A major problem with this approach is that the particles can get arbitrarily close to each other, which causes a large pressure force pushing them apart. Similarly, particles can spread out, allowing for regions with low density, where the calculations will become less accurate as there is not enough information stored in the nearby particles. For a discretization based on a regular grid this obviously cannot happen, because the forces are always calculated from quantities evaluated at the same distance from each other. For the same reason, the Eulerian approach does not suffer from the same problems with low-density regions as particle-based representations do. Overall, this leads to the Lagrangian methods in general requiring smaller timesteps than the Eulerian approach (Kellomäki, 2017).

Discretizing a heightfield with particles has the benefit over grid-based approaches when it comes to the handling of complex and sparsely filled domains. These limitations are overcome since particles can move to arbitrary locations and can separate from the main body of fluid, which allows the interaction of a user with the flow and environment in a more flexible way.

2.3.2 *Discretizing in time*

To solve the value of $q(t)$ in a differential equation of the form $\partial q/\partial t = f(q)$, the equation needs to be integrated over time. In most numerical methods this is approximated by finding the value of $q(t)$ at a finite number of time instants with timestep Δt between each instant (denoted by the superscripts): $q^1, q^2, \ldots, q^n$. There are multiple methods for achieving this with varying degrees of accuracy. One simple and commonly used method is the forward Euler, which simply evaluates the next value as a function of the previous value (Equation 11).

$$q^{i+1} = q^i + \Delta t f(q^i) \tag{11}$$

This is only accurate if $f(q)$ stays constant for the duration of the timestep. There are many time integration methods that are more sophisticated than forward Euler, while also being more costly to evaluate, such as the Runge-Kutta family (Butcher, 1987). Still, it is possible for one of these methods to allow using a longer timestep, which might compensate for the slower evaluation.

An important feature of the simplistic forward Euler method is that it is an explicit method, meaning that q^{i+1} is only a function of the state at the previous time instant q^i. In contrast, implicit methods need to solve an equation that includes the future state, which leads to them being usually more stable, but also more expensive to calculate.

Stability

Numerical integration methods provide only an approximation of the true solution for the corresponding partial differential equations. When $\Delta t \longrightarrow 0$ the numerical solution usually converges to the actual solution, but as larger timesteps are used, many methods can become unstable, i.e., start to deviate exponentially from the true solution (a simple example is given in Figure 5). In off-line simulations adaptive time-stepping can be used in situations where stability problems arise. However, in real-time applications that should run at a fixed frame-rate, the simulation method must be stable for the given timestep size no matter the situation. Since the goal is to simulate as fast as possible without losing numerical accuracy, ideally the timestep is as large as possible, but not enough to destabilize the simulation.

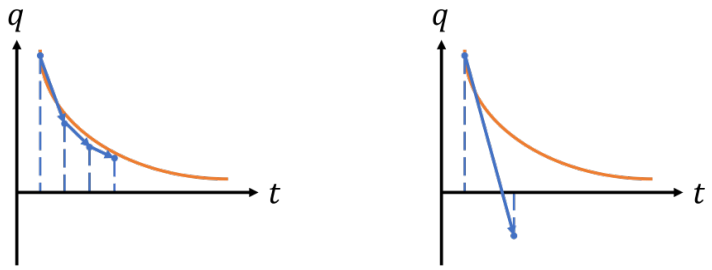

Figure 5: Consider a simple simulation that aims at reproducing a function $q(t)$ (orange line). The state of the simulation is advanced iteratively, where the estimates are calculated using the last estimate and slope of the function. With small timesteps (left graph), the estimates (blue points) approximate the function, but if the timestep is too large (right graph) it will cause the estimate to completely overshoot the function, potentially resulting in instability.

The most common tool for analyzing the stability of an explicit numerical fluid solver is the Courant-Friedrichs-Lewy (CFL) condition (Courant et al. (1928), Bridson (2015)). Considering a grid-based simulation, the CFL condition says to choose a value of Δt small enough so that when any quantity is moved from the center of some cell through the velocity field, it will only move Δx distance, Δx being the size of the cell's sides. This makes sense intuitively, since if the quantities were allowed to move in any larger amounts than this, some parts of the velocity field would be ignored. Using this idea, a necessary (but not sufficient) condition for the convergence of most explicit time integration methods based on finite differences can be formed (Equation 12, Bridson (2015)). Here, α is the CFL number which is a parameter that changes with the method used to solve the discretized equation.

$$\Delta t \leq \alpha \frac{\Delta x}{|\boldsymbol{u}|} \tag{12}$$

As stated in Equation 12, given a maximum fluid velocity $|\boldsymbol{u}|$, the stability condition sets an upper bound for the timestep, given Δx and maximum fluid velocity $|\boldsymbol{u}|$. This is set as a condition on the timestep since the Δx is usually a fixed value as to achieve a certain level of detail in the simulation. A similar condition can be obtained for particle-based simulations that instead takes the particle size into consideration (Koschier et al., 2019).

2.3.3 *Boundary Conditions*

SWE need boundary conditions where the water ends (in the x–y horizontal plane) or the simulation domain ends. As in heightfield simulations the free surface is automatically handled, only solid walls or open boundaries need to be addressed.

A solid wall (or reflective) boundary is where the fluid is in contact with a solid (Figure 6, top row). The fluid should not be flowing into or out of the solid, therefore the normal component of velocity should be zero, as defined by Equation 13a, where $\hat{n}$ is the normal to the solid boundary. In case the solid is moving, the normal component of the fluid velocity needs to match the normal component of the solid's velocity, as in Equation 13b. This kind of condition is also referred to as "no-stick" condition, since it only restricts the normal component of velocity, allowing the fluid to freely slip past in the tangential direction (Bridson, 2015).

$$\boldsymbol{u} \cdot \hat{\boldsymbol{n}} = 0 \tag{13a}$$

$$\boldsymbol{u} \cdot \hat{\boldsymbol{n}} = \boldsymbol{u}_{solid} \cdot \hat{\boldsymbol{n}} \tag{13b}$$

This is correct only for a inviscid fluid. For fluids that do have viscosity, the stickiness of the solid generally influences the tangential component of the fluid's velocity, forcing it to match. This is called the "no-slip" boundary condition and can be defined by Equation 14a, or if the solid is moving Equation 14b. For fluids with very low but nonzero viscosity this is more accurate than the "no-stick" condition only in microscopic detail, so a "no-stick" condition is often used instead (Bridson, 2015).

$$\boldsymbol{u} = 0 \tag{14a}$$

$$\boldsymbol{u} = \boldsymbol{u}_{solid} \tag{14b}$$

It is much more difficult dealing with the edge of the simulation domain if it is assumed that the water continues on past the edge. If waves should be entering along one edge, perhaps from a simple sinusoid model, the normal velocity and height can be further specified. However, if the waves are expected to leave through the edge, giving the effect of an open water surface, things are much trickier. This is called a absorbing (or non-reflecting) boundary condition (Figure 6, bottom row), and is a long known problem that continues as a subject of research in numerical methods Düz et al. (2017). The usual approach taken is to gradually blend away the simulated velocities and heights with a background field (such as a basic sinusoid wave, or flat water at rest), over the course of many grid cells (or particles): if the blend is smooth and gradual enough, reflections should be minimal. A commonly used method to achieve this, is the perfectly matched layer introduced by Berenger (1994), a method originally used for the absorption of electromagnetic waves and adapted for use in fluid simulation (e.g. Söderström and Museth (2009)).

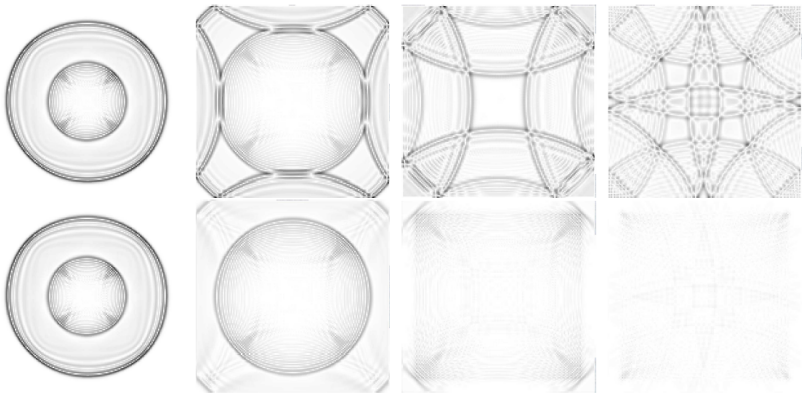

Figure 6: A comparison between reflecting boundary conditions (top row of pictures), and absorbing ones (bottom row) (Müller et al., 2008).

For boundaries where fluid should flow into or out of the domain, the two types above can be used. Inflow boundary conditions can be achieved by specifying reflecting ones, with an additional fixed normal velocity. For outflow boundary conditions, absorbing ones with free normal velocities can be used. These boundaries can then be enhanced by adding a source term to the height equation, directly adding (or subtracting) water in some regions. Such source terms are also ideal for modeling vertical sinks or sources of water (such as a drop falling from above, for instance in a particle system, or a drainage hole).

2.4 FLUID SOLVERS

This section provides an overview of the methods used to model water behavior, focusing on methods that solve SWE in real-time applications.

2.4.1 *Eulerian Solvers*

The traditional Eulerian fluid solvers are based directly on solving the underlying equations using a finite difference method (FDM) or finite volume method (FVM).

Eulerian fluid simulations were first introduced to computer graphics in Foster and Metaxas (1996) which simulated a fluid surface and included a one-way water-object interaction by advecting the floating objects using the fluid velocity.

A breakthrough came when the stable fluids semi-Lagrangian method was introduced in Stam (1999). It achieved significant performance improvements, ensuring the solver was unconditionally stable at larger timesteps. It should be noted, however, that the interpolation which guarantees stability, also introduces excessive dissipation, leading to a loss of small details and a less lively look. Most Eulerian simulation methods proposed afterwards use some variation of this approach. More advanced methods were also developed, such as the MacCormack method and its variants, to combat the deficiencies of this simple approach (Selle et al., 2008).

For shallow water, water surfaces were first simulated in Kass and Miller (1990) by solving the linearized SWE. Later, this work was in O'Brien and Hodgins (1995), modeling the flow between fluid columns with virtual pipes, coupling objects as they impacted the surface of the fluid and adding particle based splashes. The full SWE equations were first introduced in Layton and Panne (2002) to computer graphics, using the non-conservative form of the SWE.

Since then the computer graphics community has continued using the non-conservative form where the governing equations are usually solved using splitting and explicit time-stepping on a staggered grid (Bridson (2015), Chentanez and Mueller (2010), Thurey et al. (2007)): first, the self advection of the velocity field is solved, then the heightfield and velocity field are integrated forward in time. The advection term ($\frac{\partial x}{\partial t} + \boldsymbol{u} \cdot \nabla x$) is commonly solved using the semi-Lagrangian method introduced in Stam (1999).

An alternative with a simpler solver was proposed in Lee and O'Sullivan (2007) by ignoring the divergence term and using a uniform collocated grid. Although sacrificing unconditional stability for speed, it was shown to be sufficiently stable. In Wang et al. (2007) surface tension forces were added to the SWE to simulate water flow on arbitrary surfaces. An implicit Newmark integration scheme was used in Angst et al. (2008) to reduce numerical dissipation in the velocity advection step. In Chentanez and Mueller (2010) a modified MacCormack method (Selle et al., 2008) is used for the advection and stability is further improved by evaluating water height values in the upwind direction.

SWE are also explored in the scientific community, namely in computation fluid dynamics (CFD). Here, the conservation law form of the SWE is preferred. The accuracy of the numerical solution is also prioritized over the computational efficiency, whereas in computer graphics the opposite is the norm. Methods that originated in CFD have also been used for realistic water animation. A finite volume scheme based on the formulation in Kurganov and Petrova (2007) was implemented on GPUs for real-time simulations with photorealistic rendering in Brodtkorb et al. (2012). Parna et al. (2018) demonstrated that with the power of modern era GPUs, higher order numerical techniques usually applied to CFD use cases, can be used in computer graphics for more visually compelling results. The technique presented avoids some commonly encountered issues (e.g. oscillatory solution profiles and mass conservation issues) and do not need to rely on post-simulation unphysical fixes (such as in Chentanez and Mueller (2010)).

2.4.2 *Lattice-Boltzmann Method*

LBM is a grid-based method that originated from the field of statistical physics. Although it comes from descriptions on the molecular level, the method can still be used to compute fluids on a larger scale, and is applicable both in 3D and 2D. For simulations with LBM, the simulation region is usually represented by a Cartesian equidistant grid, where each cell only interacts with its direct neighborhood.

While conventional solvers directly discretize the NSE, LBM is essentially a first order explicit discretization of the Boltzmann equation in a discrete phase-space. It has be shown that LBM approximates the NSE with good accuracy, a detailed overview of LBM and derivations of the NSE and SWE from the LBM can be found in Zhou

(2004). An in-depth review on the history and development of the LBM is presented in Arumuga Perumal and Dass (2015).

LBM has several advantages that make it appealing for real-time applications. The LBM algorithm uses only arithmetic operations and is totally local, so it maps very well to parallel architectures such as GPUs. The method is also considered simple to understand and implement.

Its shortcomings include the strict limit on the timestep to ensure stability, and large memory consumption (Ojeda Contreras, 2013). It also has the same drawback of conventional SWE simulations when it comes to coupling with external dynamic objects, as it needs ad hoc processes to handle those interactions.

The LBM has also proven to be an accurate and efficient alternative for solving the SWE. The first reports on this topic were conducted in Salmon (1999) and Zhou (2002). As the LBM is very suitable for parallel architectures, several publications can be found on solving the SWE with a GPU implementation (e.g. Kuznik et al. (2010), Tubbs and Tsai (2011)). Navarro-Hinojosa et al. (2018) provides a survey on GPU accelerated LBM applications.

The LBM has seen some use in computer graphics, both for both 3D (Thürey et al., 2005) and shallow water (2D) simulations García Bauza et al. (2010). A series of publications by Ojeda and Susín can be highlighted, where they explore 2D versions of the method for shallow water simulations with a focus on real-time execution and GPU implementation (Ojeda Contreras (2013), Ojeda and Susín (2013), Ojeda and Susín (2014)). They also present an approach that couples the LBM for shallow waters with a particle system that adds more details, and supports breaking waves. Its performance is similar to the Eulerian alternative developed in Chentanez and Mueller (2010) (Kellomäki, 2017).

Recently, although the LBM for solving the SWE still sees extensive use in CFD, it has received little attention in computer graphics. A novel simplified LBM presented in Zhou (2022) could possibly be of interest as it has a reduced computational and memory cost while having similar accuracy.

2.4.3 *Smoothed-Particle Hydrodynamics*

SPH is probably the most common of the particle-based methods. As it is a Lagrangian method, the advection term is automatically handled by the movement of the particles. All that remains is determining the various forces affecting the particles to update their velocity, and then to update the position based on the velocity. For this, the necessary fluid physical quantities and their derivatives need to be evaluated at the particle locations. SPH is a method used for interpolating these quantities as a distance-weighted sum from the values stored in the nearby particles (using kernel functions). External forces can be simply applied to each particle independently of the others.

It was introduced to the computer graphics community with fire and gas simulations in Stam and Fiume (1995), with interactive simulations being demonstrated in Müller et al. (2003) with a limited particle count. There is a vast literature of variants and extensions of SPH that include, e.g., GPU implementations, different methods for enforcing incompressibility and handling boundaries, combining particles with varying sizes or timesteps, and multiphase fluids. However, most of the methods are not designed for a real-time context. A review of the theory

and applications of SPH in CFD and computer graphics can be found in Weaver and Xiao (2016) and Koschier et al. (2019).

As was the case with grid-based methods, 3D particles methods were adapted to 2D with the goal of improving performance for interactive applications. The first case was in Lee and Han (2010), where the SPH model was applied to solve the SWE. This work was extended in Solenthaler et al. (2011) which introduces arbitrary domain boundaries and terrain slopes. They also add two-way coupling of the particle based fluid with rigid bodies, and improve the surface rendering of low density particle regions. In both cases the surface is defined as a height map, using the density to displace the vertices.

Shallow water methods are naturally much faster than a full 3D solution, as they require far fewer particles, while also retaining the good rigid body coupling of SPH. However, shallow water SPH methods still have significantly worse performance than similar grid-based methods due to requiring very short timesteps (Kellomäki, 2017).

Further work was done in Chladek and Durikovic (2015), which improved simulation accuracy and boundary handling. The shallow water simulation can also be coupled with a full 3D SPH simulation, although this has a considerable impact in performance.

Fujisawa et al. (2017) extends the work in Solenthaler et al. (2011) by combining the 2D simulation with a 3D simulation to address missing behaviors such as splashes and breaking waves. The 3D particles are generated from 2D particles by categorizing these particles according to motion and position and deleted when they reenter the water's surface. The 3D particles behavior is simulated using the position based fluid method (Macklin and Müller, 2013) and both particle types are rendered using a screen-space rendering technique.

Since then, significant work has been done in SPH, which reduces the severe time step restrictions of standard techniques (Koschier et al., 2019), however, this has still not been applied to SWE 2D simulations.

2.4.4 *Summary*

The first and most researched solvers were of the Eulerian type. In particular, the splitting scheme and semi-Lagrangian advection combination became the most popular in computer graphics (Bridson, 2015). Alternatively, recent work (Parna et al., 2018) argues that higher order numerical methods, commonly found in the scientific community, can provide a higher quality solutions and avoid unphysical modifications, with an acceptable increase in computational cost.

An alternative is the LBM which has a simple and easily parallelizable algorithm. Its update step is fast but forces the use of small timesteps to ensure stability, which leads to a similar performance to other Eulerian methods (Kellomäki, 2015), while having increased memory requirements (Ojeda and Susín, 2014).

A third counterpart is the SPH model, which provides unique advantages due to being a particle based method, such as the inherent conservation of mass and coupling with objects. Although SPH has seen significant advances recently (Koschier et al., 2019), they are yet to be applied to the SWE, where previous solvers have significantly worse performance than their grid-based counterparts (Solenthaler et al. (2011), Kellomäki (2017)).

This is further accentuated by the requirement of extra heightfield surface construction procedures, and needing high particle counts to avoid blobby a surface (Chladek and Durikovic (2015), Fujisawa et al. (2017)).

2.5 RENDERING

The previous section focused solely on the dynamics of the fluid and its interaction with the terrain. The goal of this section is to explore real-time rendering effects that can achieve a realistic representation of the developed fluid simulation, including the water color, small surface details and caustics.

2.5.1 *Water Surface Color*

The water's surface color is composed of reflected and transmitted/refracted light, which can be handled as two separate components. The ratio between the two is given by the Fresnel reflectance F, which specifies how much light is reflected at an interface between two different media. Consequently, the transmitted fraction is given by $1 - F$, resulting in a final composition as in Equation 15. C_{refl} and C_{trans} are the light intensities that are reflected and transmitted, respectively, before applying the Fresnel factor.

$$C = F(\theta_i)C_{refl} + (1 - F(\theta_i))C_{trans} \tag{15}$$

F is often calculated using Schlick's approximation (Schlick, 1994), given in Equation 16, which takes as input the angle of incidence θ_i and the specular reflectance at normal incidence f_0 (for water, $f_0 = 0.02$, Akenine-Mller et al. (2018)).

$$F(\theta_i, f_0) = f_0 + (1 - f_0)(1 - \theta)^5 \tag{16}$$

Reflection

A surface's BRDF, denoted by $f(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, n)$, describes for a particular point, how much of the incoming light in a direction $\boldsymbol{\omega_i}$ is reflected in an outgoing direction $\boldsymbol{\omega_o}$, where n is the surface normal. Considering a surface lit from a single direction, such as by a directional light, the reflected light at the surface can be given by Equation 17, where C_{light} is the light's intensity and all vectors are considered to be normalized. This is illustrated in Figure 7 where the value θ_i represents the angle between $\boldsymbol{n}$ and $\boldsymbol{\omega_i}$.

$$C_{refl} = f(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, \boldsymbol{n})C_{light}(\boldsymbol{n} \cdot \boldsymbol{\omega_i}) \tag{17}$$

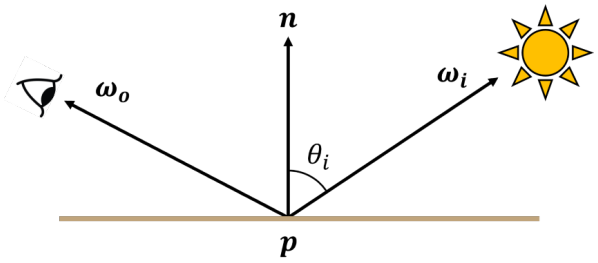

Figure 7: The BRDF defines the amount of light that is reflected at a surface point $\boldsymbol{p}$ with normal $\boldsymbol{n}$ from incident direction $\boldsymbol{\omega_i}$ in the outgoing direction $\boldsymbol{\omega_o}$.

The BRDF, is typically made of two terms: a diffuse component and a specular component (Pharr and Humphreys, 2018). However, when describing the surface of a transparent media such as water, only the specular component is considered, as the refracted/transmitted light is handled separately.

Physically based specular BRDF terms are typically based on microfacet theory (Pharr and Humphreys, 2018). The core idea is that any surface at a microscopic scale can be described by tiny perfectly reflective mirrors called microfacets. The light reflectance is then statistically modeled based on these surface variations that are present below the observation scale. One of the most commonly used models is the Cook-Torrance model (Cook and Torrance, 1982), given by Equation 18.

$$f(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, \boldsymbol{n}) = \frac{D(\boldsymbol{h})F(\boldsymbol{\omega_i}, \boldsymbol{h})G(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, \boldsymbol{h})}{4(\boldsymbol{n} \cdot \boldsymbol{\omega_i})(\boldsymbol{n} \cdot \boldsymbol{\omega_o})} \tag{18}$$

As the microfacets are considered to be perfect mirrors, for a microfacet to reflect light from $\boldsymbol{\omega_i}$ into $\boldsymbol{\omega_o}$ its surface normal has to be oriented exactly halfway between $\boldsymbol{\omega_i}$ and $\boldsymbol{\omega_o}$. The vector halfway between $\boldsymbol{\omega_i}$ and $\boldsymbol{\omega_o}$ is called the half-vector and denoted as $\boldsymbol{h}$. $D(\boldsymbol{h})$ is the microgeometry Normal Distribution Function (NDF) evaluated at the half-vector $\boldsymbol{h}$ and represents the probability that a given microfacet's normal $\boldsymbol{m}$ is oriented correctly, i.e. $\boldsymbol{m} = \boldsymbol{h}$.

$G(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, \boldsymbol{h})$ is the geometry function[1] and it represents the proportion of surface points with $\boldsymbol{m} = \boldsymbol{h}$ that are not shadowed (microfacets not being directly lit) or masked (microfacets being occluded by other microfacets from the viewing direction). Therefore, the product of $D(\boldsymbol{h})$ and $G(\boldsymbol{\omega_i}, \boldsymbol{\omega_o}, \boldsymbol{h})$ gives the concentration of active surface points, the surface points that actively participate in the reflectance by successfully reflecting light from $\boldsymbol{\omega_i}$ into $\boldsymbol{\omega_o}$.

$F(\boldsymbol{\omega_i}, \boldsymbol{h})$ is the Fresnel reflectance of the active surface points as a function of the light direction $\boldsymbol{\omega_i}$ and the active microgeometry normal $\boldsymbol{m} = \boldsymbol{h}$.

Finally, the denominator $4(\boldsymbol{n} \cdot \boldsymbol{\omega_i})(\boldsymbol{n} \cdot \boldsymbol{\omega_o})$ is a correction factor which accounts for the transformation of quantities between the local space of the microgeometry and that of the macro-surface.

A widely adopted choice for the D and G terms is the GGX distribution and its matching Smith geometric function (Burley (2012), Heitz (2018), Guy and Agopian (2019)). A review of the options for these terms and

1 Can also be found in the literature with various other names, such as: geometry factor, shadowing-masking function, shadowing function, geometry term, geometric attenuation, amongst other denominations.

additional information on the subject can be found in Pharr and Humphreys (2018) and Hoffman (2013). Specific models have been used in ocean rendering, however these rely on the statistical properties of ocean wind waves (Bruneton et al., 2010).

To compensate for the lack of indirect lighting, which is especially noticeable with highly specular surfaces, some kind of ambient or image-based lighting is typically added (Hoffman, 2013).

Transmission

Transmitted/refracted light is given by the reflection of the terrain below. However, this is further complicated by the fact that water acts as a participating media and therefore affects the distribution of radiance in it. Light that enters the water volume, after being refracted at its boundary, will be affected by two main processes (Pharr and Humphreys, 2018):

- Absorption, which causes reduction in radiance due to conversion from light to some other form of energy, such as heat.
- Scattering, which causes the light direction to change due to particle collisions.

As a ray of light passes through a media, it can collide with particles and be scattered in different directions, which results on the ray losing intensity. Consequently, radiance from other rays can also be scattered into the path of this ray increasing its intensity. These interactions are referred to as out-scattering and in-scattering respectively. Furthermore, single and multiple scattering can be distinguished when considering the number of times a light ray is scattered when traveling through the media (Figure 8).

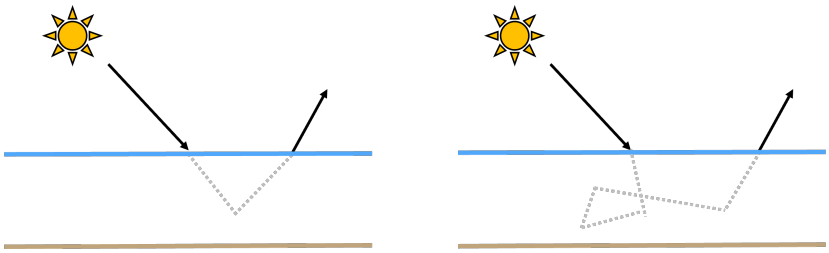

Figure 8: When entering a participating media, light can be scattered once (left) or multiple times (right) before exiting the media.

How much scattering happens is described by the scattering coefficient σ_s. Similarly, how much the media absorbs light is determined by the absorption coefficient σ_s. Combining both coefficients results in the extinction/attenuation coefficient $\sigma_t = \sigma_a + \sigma_s$ which describes how much light is attenuated along its path through the media. The fraction of light that is not absorbed or scattered, while traveling a distance d through the media, is defined as the transmittance (given by the Beer-Lambert Law) as in Equation 19 (Pharr and Humphreys, 2018).

$$T(d) = e^{-\sigma_t d} \tag{19}$$

The variance of the photon flux in a light ray traveling through the media can be visualized in Figure 9.

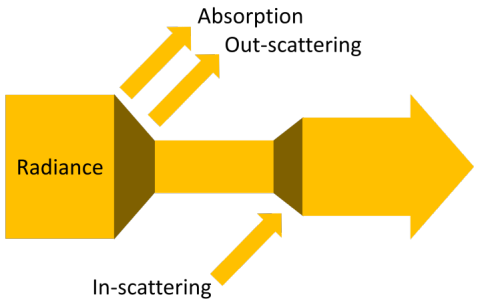

Figure 9: Illustration of how the photon flux of a light ray is affected when in a participating media.

2.5.2 *Caustics*

Figure 10: Visible caustics caused by perturbations in the water surface. Examples from Yuksel and Keyser (2009), Parna (2020) and Yang and Ouyang (2021), from left to right respectively.

Caustics are concentrations of light in diffuse surfaces (receivers) caused by refraction or reflection of light from a highly specular surface (generator). In water rendering, caustics are interesting visual elements that enhance the quality and realism of generated images, while also providing additional cues to perceive the shape of the water surface (examples presented in Figure 10). The efficient rendering of caustics in real-time is rather difficult, as a single point may depend on light redirected from several points.

In an early attempt at real-time caustics, a technique was presented in Stam (1996) which projected a pre-computed texture onto the scene geometry using additive blending, animated by texture coordinate perturbation. Although this technique is extremely fast, evidently the caustics produced are not correct in relation to the water surface and receiver geometry.

A method is proposed in Guardado and Sanchez-Crespo (2007) that computes caustics by tracing rays from the receiver surface to the water surface, where each receiving point is computed from a single point in the surface. The intensity is modulated based on the direction of these rays and samples a light map in the normal direction at the intersection point. The results produced by this method are not physically correct either, and

can only roughly approximate caustics for sun light coming from directly above the water surface. Since then, real-time techniques have significantly advanced towards physical accuracy.

Caustics mapping is introduced in Shah et al. (2007), a technique similar to shadow mapping that generates a caustics texture that can be projected into the receiver geometry. The algorithm starts by rendering the generator and receiver objects from the light's view onto textures, storing its position and normals. A vertex grid of equal resolution to the generator's texture is then used in place of the object, where each vertex maps to a texel. At each vertex, the incoming light ray is refracted and intersected with the receiver's positions (stored in the previously mentioned texture) and the light intensity is estimated at this point. The intersection positions are then used to splat points from the light's view onto the caustics map. The same idea was explored concurrently in Wyman and Davis (2006) and Hu and Qin (2007) which present similar methods.

Hierarchical and adaptive photon casting methods were proposed in Wyman (2008) and Wyman and Nichols (2009) to improve caustic mapping performance by reducing redundant processing.

A similar method in Papadopoulos and Papaioannou (2010) utilizes the image-space ray-scene intersection method introduced in Shah et al. (2007), but uses instead a uniform light screen space grid, to decouple the effect accuracy from the refractive geometry. This method additionally implements dynamically varying splatting size and improves image quality via additional filtering.

A method for simulating caustics in a flat receiver due to water refraction was proposed in Yuksel and Keyser (2009). It considers the refracted radiance towards a point on a receiver plane from a singular rectangular region of the heightfield surface. This approach is fast, but has the drawback of only working with flat receiver geometry.

Alternatively to splatting based methods, some work rendered continuous geometry instead of discrete photon hits. Real-time beam tracing is applied in Ernst et al. (2005), where a bounding volume is drawn for each caustic volume created by a triangle in the generator mesh. The caustics intensity is then accumulated by performing point-in-volume tests for every rendered pixel of the receiver. Liktor and Dachsbacher (2010) separates the generator mesh density from photon casting density by projecting a vertex grid onto a light space rendering of the generator geometry. This generator mesh is then extruded into the receiver surface using a geometry shader to create the caustics volume. Liktor and Dachsbacher (2011) further improves performance by implementing adaptive beam generation while also supporting multiple bounces.

Parna (2020) extends Shah et al. (2007) caustics mapping to work directly with heightfield water simulations by using its grids points as photons to be traced. The caustics intensities are computed efficiently by exploiting compute shader capabilities. Both splatting and geometry based reconstruction are implemented.

Recently, hardware accelerated ray-tracing has also been used for real-time caustics. A method is presented in Gruen (2019) that follows the common procedure of caustics mapping while computing accurate intersections using ray-tracing. This method is then extended in Yang and Ouyang (2021) where the rendering process of the caustics map is improved, resulting in two approaches: one based in photon splatting and another in building an intermediate caustic mesh.

2.5.3 *Small-scale details*

Figure 11: Water surfaces with small-scale details added through normal mapping. Examples from Vlachos (2010), Yu et al. (2011) and Ojeda Contreras (2013), from left to right respectively.

The heightfield simulation cannot resolve waves with wavelengths smaller than the grid resolution Δx. Decreasing Δx is not feasible past a certain point as it implies more cells and requires a smaller time-step to maintain stability, which would break the real-time constraint. Instead, this high frequency detail can be added through normal mapping (Thurey et al., 2007). The idea is to perturb the surface normals per pixel using an animated normal map texture, mimicking wave effects at a smaller scale (examples presented in Figure 11).

A method for advecting texture coordinates was introduced in Max and Becker (1995), which periodically reset the texture coordinates to their original values to avoid too much distortion. To counter the popping caused by the coordinates reset, further sets of textures were used that were smoothly faded-in and out. A drawback is that the reset frequency is dependent on the fluid velocity of the scene used and therefore requires fine-tuning.

An attempt to fix this problem was presented in Neyret (2003) by blending three layers (with different reset times) of three phase-shifted textures, and keeping track of the accumulated deformation of each. The resulting textures were combined using trilinear interpolation based on a maximum allowed deformation. This method however, besides being more expensive, has the downside of having large texture storage requirements in scenes with a big variety of flow speeds.

The method in Max and Becker (1995) was used in Chentanez and Mueller (2010) for normal map texture coordinate regeneration and texture blending, while using an Fast Fourier Transform (FFT) based wave simulation (Tessendorf, 2001) to generate the wave texture normal map. The same method (Max and Becker, 1995) was also applied in Vlachos (2010) to a normal map by utilizing two textures offset by half a phase for blending. The problem of pulsing artifacts was addressed by using a noise texture to change the phase of the pulsing. A disadvantage is that it only works for non-directional wave patterns, i.e, using a normal map with directional waves can result in them appearing to move sideways.

An alternative Lagrangian technique for the advection of textures was presented in Yu et al. (2009) and Yu et al. (2011). The technique ensures that the moving texture follows the velocity field of the fluid, while preserving several key properties of the original texture. A set of deformable textured grids are advected as particles with the flow. At each time step a uniform distribution of particles is maintained by removing and creating particles

as necessary. Particles whose grid is too distorted are also removed. A method was provided for ensuring both spatial and temporal continuity when blending these grids.

Noise functions are a common way of adding detail to surfaces (Lagae et al., 2010), and have been used to simulate water. In Ojeda Contreras (2013) instead of advecting a normal map at the surface, one is generated every frame and applied directly. The texture is generated/updated using 3D Perlin noise (Perlin, 2002) (and going through one dimension over time) or FFT based wave simulation (Tessendorf, 2001). The use of noise to add detail to a water surface is also common in games (e.g., Grujic and Cutocheras (2018)).

Part II

CORE OF THE DISSERTATION

3

SIMULATION

This chapter presents the method used for the numerical simulation of the water's behavior. The chapter starts by presenting a mathematical description of the chosen scheme, followed by the details of its GPU implementation.

A grid-based approach was chosen as it allows for better performance over the particle approach, since previous SWE SPH methods performed significantly worse than its grid-based counterparts (Kellomäki, 2017), and newer SPH methods (Koschier et al., 2019) have yet to be applied to the SWE case. Additionally, constructing a surface is not as straightforward as in the grid case (Kellomäki (2017), Parna (2020)), requiring further processing to avoid a bumpy appearance ((Chladek and Durikovic, 2015), (Fujisawa et al., 2017)).

From within the grid-based methods, the finite difference scheme introduced in Parna et al. (2018) was chosen as it provides higher quality solutions over the commonly used splitting scheme (Parna, 2020). Since it is a higher order shock capturing scheme, it provides better physical accuracy and avoids extra unphysical tweaking that other lower-order schemes rely on (Chentanez and Mueller, 2010). A downside of this scheme is its increased computational complexity, however, this is outweighed by the power of modern GPUs (Parna, 2020).

3.1 NUMERICAL METHOD

The implemented scheme, as previously mentioned, is the PIFWENO scheme introduced in Parna et al. (2018). The scheme solves the conservation law form of the SWE (Section 2.2.2), and is based on the Picard integral formulation (PIF) of conservation laws combined with WENO reconstruction. It consists of two main steps: Approximating the time-averaged fluxes (defined in the PIF), and inserting this result into the finite difference WENO reconstruction, used to approximate the spatial derivatives.

As an explicit scheme, the state of the simulation at the following time instant is computed from the current state. All of the stored quantities (conserved variables $\boldsymbol{U}$ and bathymetry b) are located at the grid cell centers. However, it should be noted that values at cell's edges (with fractional indices) are used during computations, but are never actually stored in the input/output grid.

3.1.1 *Picard integral formulation for SWE*

The PIF of the SWE is defined by integrating Equation 7 over the interval $t \in [t^n, t^{n+1}]$ Christlieb et al. (2015), resulting in Equation 20 (subscripts denote derivatives and superscripts denote the timestep). $\boldsymbol{U}$, $\boldsymbol{F}$ $\boldsymbol{G}$ and $\boldsymbol{S}$ are defined in Equation 8.

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n - \Delta t \tilde{\boldsymbol{F}}_x^n - \Delta t \tilde{\boldsymbol{G}}_y^n + \Delta t \boldsymbol{S}^n \tag{20}$$

$\tilde{\boldsymbol{F}}^n$ and $\tilde{\boldsymbol{G}}^n$ are the time-averaged fluxes and defined as in Equation 21.

$$\begin{aligned} \tilde{\boldsymbol{F}}^n &= \frac{1}{\Delta t} \int_{t^n}^{t^{n+1}} \boldsymbol{F}^n \, dt \\ \tilde{\boldsymbol{G}}^n &= \frac{1}{\Delta t} \int_{t^n}^{t^{n+1}} \boldsymbol{G}^n \, dt \end{aligned} \tag{21}$$

Henceforth, the superscripts for time level n are dropped for simplicity. The time averaged fluxes are approximated as in Equation 22 (Parna et al., 2018), derived from a second order Taylor expansion of the fluxes centered at $t = t^n$ (Christlieb et al., 2015). Here, $\partial \boldsymbol{F}/\partial \boldsymbol{U}$ and $\partial \boldsymbol{G}/\partial \boldsymbol{U}$ are the flux Jacobians, as in Equation 23 (see Appendix A for definition). All derivatives that appear in Equation 22 are evaluated using simple central finite differences ($\boldsymbol{F}_x$, $\boldsymbol{G}_y$ and those in $\boldsymbol{S}$).

$$\begin{aligned} \tilde{\boldsymbol{F}} &= \boldsymbol{F} + \frac{\Delta t}{2} \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}} \left(\boldsymbol{S} - \boldsymbol{F}_x - \boldsymbol{G}_y \right) \\ \tilde{\boldsymbol{G}} &= \boldsymbol{G} + \frac{\Delta t}{2} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{U}} \left(\boldsymbol{S} - \boldsymbol{F}_x - \boldsymbol{G}_y \right) \end{aligned} \tag{22}$$

$$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}} = \begin{pmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{pmatrix} \qquad \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{U}} = \begin{pmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2 + gh & 0 & 2v \end{pmatrix} \tag{23}$$

3.1.2 *WENO reconstruction*

Having $\tilde{\boldsymbol{F}}$ and $\tilde{\boldsymbol{G}}$ approximated at each grid point (Equation 22), the values at the cell's edges, $\hat{\boldsymbol{F}}$ and $\hat{\boldsymbol{G}}$, are computed using the third-order WENO reconstruction (Parna et al., 2018), as in Equation 24. This procedure

takes as input a stencil[1] of the 4 surrounding time-averaged fluxes ($\tilde{\boldsymbol{F}}$ and $\tilde{\boldsymbol{G}}$). The subscripts denote the position in the grid and a description of the $WENO3$ procedure can be found in Appendix B.

$$\begin{aligned}\hat{\boldsymbol{F}}_{i+1/2,j} &= WENO3(\tilde{\boldsymbol{F}}_{i-1,j}, \tilde{\boldsymbol{F}}_{i,j}, \tilde{\boldsymbol{F}}_{i+1,j}, \tilde{\boldsymbol{F}}_{i+2,j}) \\ \hat{\boldsymbol{G}}_{i,j+1/2} &= WENO3(\tilde{\boldsymbol{G}}_{i,j-1}, \tilde{\boldsymbol{G}}_{i,j}, \tilde{\boldsymbol{G}}_{i,j+1}, \tilde{\boldsymbol{G}}_{i,j+2})\end{aligned} \tag{24}$$

The spatial derivatives $\tilde{\boldsymbol{F}}_x$ and $\tilde{\boldsymbol{G}}_y$ (in Equation 20), of $\tilde{\boldsymbol{F}}$ and $\tilde{\boldsymbol{G}}$ respectively, are approximated by a conservative flux difference, as in Equation 25, using the values at the cell's edges, $\hat{\boldsymbol{F}}$ and $\hat{\boldsymbol{G}}$. Δx and Δy are the cell's dimensions in the x and y direction respectively.

$$\begin{aligned}\tilde{\boldsymbol{F}}_{x;i,j} &= \frac{1}{\Delta x}(\hat{\boldsymbol{F}}_{i+1/2,j} - \hat{\boldsymbol{F}}_{i-1/2,j}) \\ \tilde{\boldsymbol{G}}_{y;i,j} &= \frac{1}{\Delta y}(\hat{\boldsymbol{G}}_{i+1/2,j} - \hat{\boldsymbol{G}}_{i-1/2,j})\end{aligned} \tag{25}$$

Inserting Equation 25 into Equation 20 gives its finite difference discretization, as in Equation 26 (Parna et al., 2018).

$$\boldsymbol{U}_{i,j}^{n+1} = \boldsymbol{U}_{i,j}^{n} - \frac{\Delta t}{\Delta x}(\hat{\boldsymbol{F}}_{i+1/2,j}^{n} - \hat{\boldsymbol{F}}_{i-1/2,j}^{n}) - \frac{\Delta t}{\Delta y}(\hat{\boldsymbol{G}}_{i,j+1/2}^{n} - \hat{\boldsymbol{G}}_{i,j-1/2}^{n}) + \Delta t \boldsymbol{S}_{i,j}^{n} \tag{26}$$

3.1.3 *Well-balanced treatment of the source term*

Steady state solutions (e.g., a still flat water surface) in non-flat bottoms have it's flux gradients exactly balanced by the source term S. This balance will fail to be preserved if a straightforward treatment of the source term is used. This issue is addressed by the inclusion of the method described in Xing and Shu (2005) (Parna et al., 2018).

One of the requirements of the method is to modify and split the source term into two parts as in Equation 27. The central finite difference approximations used in Equation 22 must follow this splitting.

$$\boldsymbol{S} = \begin{pmatrix} 0 \\ (\frac{1}{2}gb^2)_x \\ (\frac{1}{2}gb^2)_y \end{pmatrix} + \begin{pmatrix} 0 \\ -(h+b)gb_x \\ -(h+b)gb_y \end{pmatrix} \tag{27}$$

1 A stencil is a set of grid points that relate to the point of interest, when using a numerical approximation routine.

The other requirement is to then approximate the split source term derivatives using the WENO reconstruction (Xing and Shu, 2005), similarly to the fluxes. Therefore, the final source term $\boldsymbol{S}$ (used in Equation 26) is given by Equation 28, where $s = (\frac{1}{2}gb^2)$, and $\hat{s}$ and $\hat{b}$ are the values of s and b at the cells' edges.

$$\boldsymbol{S}_{i,j} = \begin{pmatrix} 0 \\ \left(\frac{1}{\Delta x}(\hat{s}_{i+1/2,j} - \hat{s}_{i-1/2,j})\right) - (h+b)g\left(\frac{1}{\Delta x}(\hat{b}_{i+1/2,j} - \hat{b}_{i-1/2,j})\right) \\ \left(\frac{1}{\Delta y}(\hat{s}_{i,j+1/2} - \hat{s}_{i,j-1/2})\right) - (h+b)g\left(\frac{1}{\Delta y}(\hat{b}_{i,j+1/2} - \hat{b}_{i,j-1/2})\right) \end{pmatrix} \tag{28}$$

The values at the cells' edges, $\hat{s}$ and $\hat{b}$, are computed using the WENO reconstruction, sharing the non-linear weights of the fluxes (as described in Appendix B). The input stencil contains the required values of b and s.

$$\begin{aligned} \hat{s}_{i+1/2,j} &= WENO3(s_{i-1,j}, s_{i,j}, s_{i+1,j}, s_{i+2,j}) \\ \hat{b}_{i,j+1/2} &= WENO3(b_{i,j-1}, b_{i,j}, b_{i,j+1}, b_{i,j+2}) \end{aligned} \tag{29}$$

3.1.4 *Handling of the wetting/drying processes*

The computation of the mass-conservation equation, given by the first line of the conservative SWE, must be made positivity preserving for the scheme to allow wetting and drying processes (Parna et al., 2018), that is, the values of h need to remain non-negative. This is achieved by incorporating the maximum principle preserving method in Liang and Xu (2014) for the mass computation equations.

The final fluxes for the mass, namely the first components of $\hat{\boldsymbol{F}}$ and $\hat{\boldsymbol{G}}$, are re-defined as a linear combination of low- and high-order fluxes, as in Equation 30. Here, the (1) superscript denotes the first entry of a vector.

$$\begin{aligned} \hat{F}^{(1)} &\leftarrow \theta\hat{F}^{(1)} + (1-\theta)\tilde{f} \\ \hat{G}^{(1)} &\leftarrow \theta\hat{G}^{(1)} + (1-\theta)\tilde{g} \end{aligned} \tag{30}$$

The $\tilde{f}$ and $\tilde{g}$ are first-order positivity preserving flux approximations. These are evaluated using the Lax-Friedrichs flux (Seal et al., 2016) as in Equation 31, where α is given by Equation 61.

$$\begin{aligned} \tilde{f}_{i+1/2,j} &= \frac{1}{2}\left(F^{(1)}_{i,j} + F^{(1)}_{i+1,j} - \alpha_i(U^{(1)}_{i+1,j} - U^{(1)}_{i,j})\right) \\ \tilde{g}_{i,j+1/2} &= \frac{1}{2}\left(G^{(1)}_{i,j} + G^{(1)}_{i,j+1} - \alpha_j(U^{(1)}_{i,j+1} - U^{(1)}_{i,j})\right) \end{aligned} \tag{31}$$

The coefficients of the linear combination, $\theta_{i+1/2,j}$ and $\theta_{i,j+1/2}$, are determined in a way that ensures the interpolation between the high and low-order fluxes is of the highest order possible, while preserving the positivity of the solution. This is achieved by solving a simple optimization problem, described in Appendix C.

Wet/dry interface velocities

Since finding the velocities u and v require divisions by h (as only the momenta hu and hv are stored in $\boldsymbol{U}$), high velocities can develop near the wet/dry interface where h is very small or even 0, which would then lead to instabilities (Parna et al., 2018). To avoid this, the method in Kurganov and Petrova (2007) is used, where u and v are recalculated using Equation 32 in regions where h is smaller than a specific threshold ϵ. The determination of ϵ is problem specific but a value of 0.01 as used in Parna et al. (2018) was found to be suitable for the scenes used.

$$\begin{aligned} u' &= \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + max(h^4, \epsilon)}} \\ v' &= \frac{\sqrt{2}h(hv)}{\sqrt{h^4 + max(h^4, \epsilon)}} \end{aligned} \tag{32}$$

This is applied to $\boldsymbol{U}$ at the start of the simulation step, which is updated with the new velocity values u' and v', as in Equation 33. Additionally, the water depth h is clamped to zero as it can become negative due to numerical round-off errors (Parna et al., 2018).

$$\boldsymbol{U} = \begin{pmatrix} \max(h, 0) \\ hu' \\ hv' \end{pmatrix} \tag{33}$$

3.2 IMPLEMENTATION

The implementation was tackled using GPU programming, more specifically OpenGL compute shaders. The GPU has been shown to be highly effective for this kind of simulations as it is able to handle large amounts of parallel data and maps well to the Eulerian (grid) approach (Parna et al. (2018), Dagenais et al. (2018), Horváth et al. (2020)). Additionally, storage of the simulation results on the GPU means that rendering is straightforward as there's no need in copying data between the CPU and GPU. For the solver step, the single kernel GPU implementation of Parna et al. (2018) is followed, which aims at decreasing the costly memory read/writes at the expense of arithmetic operations. The scheme is compressed into a one pass solution by making heavy use of local group setup combined with local synchronization and abstract Group Shared Memory (GSM) usage.

3.2.1 *Timestepping*

The first step of the simulation is determining the timestep, which is limited by the stability restrictions of the scheme. Dynamic timestepping with a maximum fixed timestep is used as it can advance the simulation in real-time, while also guaranteeing stability.

For the simulation to run at real time, every frame of the animation should advance the simulation roughly the same amount forward in time as the frame's duration. This can be approximated by using the previous frame's duration, $frameTime$, as a reference for how much the simulation should advance, so $\Delta t = frameTime$. However, the timestep Δt has to be restricted, in order to ensure stability, by defining an upper bound $max\Delta t$. Since no unexpected accelerations are introduced in this work, e.g. via the user's input, this $max\Delta t$ can be determined empirically. Considering that the frame duration can be longer than the defined maximum timestep, advancing the simulation can require multiple steps, which are performed iteratively while consuming the frame time. For example, with $max\Delta t = 12ms$ and $frameTime = 30ms$, three solver steps would be performed, each with $\Delta t = 10$.

3.2.2 *Data storage*

The three components of $\boldsymbol{U}$ (the height, and momentum terms in the x and y directions) need to be stored in between simulation steps. This is done using two textures, where the data is read from one and written to the other, swapping after each simulation step. These have 4 channels of 32 bit floating point data, 3 for the components of $\boldsymbol{U}$, the last being used to encode boundary information (its use is described in Section 3.2.4). As the algorithm uses a co-located grid, the mapping to a texture is straightforward, where each texel corresponds to a grid cell. A single channel texture is also used to store the bathymetry data.

The height and normal of the water are also required to be stored so they can then be used when rendering the water surface (chapter 4). A 32 bit 4 channel texture is used for this, where the first 3 channels are used to store the normalized normal vector of the water surface, and the last channel is used to store the water surface height (given by the water depth plus the bathymetry).

The grid that constitutes the simulation domain has dimensions of W and H in the x and y directions respectively. The simulation data textures have dimensions equal to $(W + B \times 2, H + B \times 2)$, where (W, H) is the simulation domain and B is the number of boundary cells (see Section 3.2.3). Although not required, the bathymetry and the "normal and height" textures have the same dimensions so they can be accessed using the same coordinates.

3.2.3 *Threading scheme*

The concurrent computation work is managed by decomposing the simulation domain into fixed size smaller groups, where each thread corresponds to a cell. Each one of these groups will require their own set of boundary cells in order to correctly process the values at the edge of the group's domain.

The scheme has a series of computations that require accessing the neighboring cells' data, which is stored in the GSM (further information in Section 3.2.6). The computed data in a cell can be invalid if it is the result of computations that involve values outside the range of the local GSM. If other cells access this invalid data, then their computations derived from it will also become invalid. Tracking the validity of the computed data in each

cell, the scheme is found to require 4 boundary cells (Parna et al., 2018), as illustrated in Figure 12. Cells where the results are valid are referred to as inner domain cells.

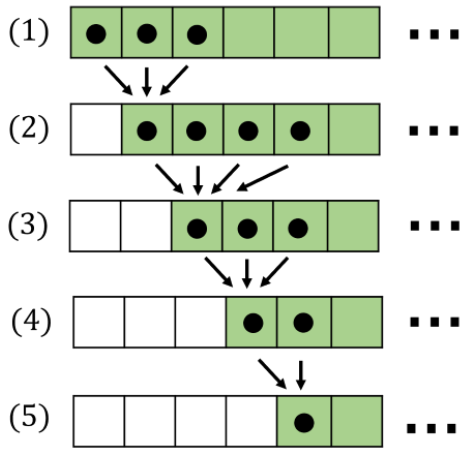

Figure 12: Access of neighboring cells that leads invalid values (example for left boundary of the x direction, same applies for y). Valid cells colored in green and invalid cells in white. The valid cells are determined as follows: (1) initial group domain; (2) cells where $\tilde{\boldsymbol{F}}$ is valid (Equation 22); (3) cells where the WENO reconstructions are valid (Equation 29); (4) cells where θ is valid (Equation 73); (5) cells where the final $\boldsymbol{U}_{n+1}$ is valid (Equation 26)

These boundary cells need to be overlapped with valid cells of other groups, so all the simulation domain is covered. This is illustrated in Figure 13 where a grid point is involved in calculations once as a boundary cell, where the results are not stored, and once as an inner domain cell, where the results are written to the textures.

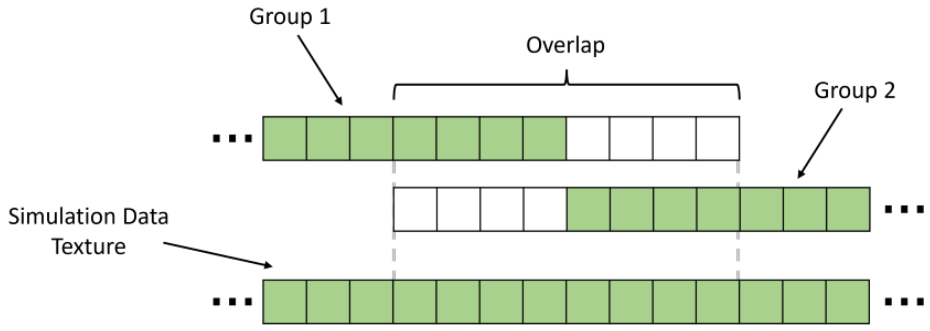

Figure 13: Illustration of overlapping threads between neighboring groups in a single dimension. Inner domain cells are colored in green and boundary cells in white.

As some grid points are involved in calculations more than once, more threads need to be dispatched than the total number of inner domain cells. The total number of groups to be dispatched for each dimension, in order

to accommodate for the boundary cells, is given by Equation 34. K_x and K_y are the number of elements in each work group for the x and y dimensions respectively.

$$\begin{aligned} G_x &= \left\lceil \frac{W}{K_x - 2 \times B} \right\rceil \\ G_y &= \left\lceil \frac{H}{K_y - 2 \times B} \right\rceil \end{aligned} \tag{34}$$

Since the number of groups is rounded up (denoted by $\lceil\ \rceil$) to encompass all the simulation domain, more threads than required can be dispatched. To account for this, global boundary cells are required in addition to the local group boundary cells (which can overlap), since a thread can be inside a group's inner domain, but outside the simulation domain. An illustration of the simulation domain within the dispatched compute groups is presented in Figure 14. The local computation results are written to the output textures only for cells where they are valid, i.e., that are both part of the local inner domain and global inner domain.

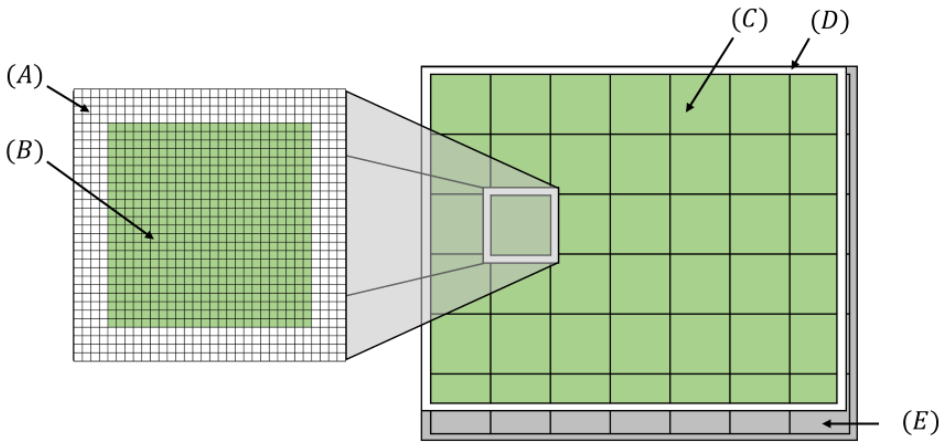

Figure 14: Illustration of the simulation domain within the dispatched compute groups, with a group highlighted. Each group is composed of its local inner domain (B) and local boundary cells (A). Similarly, the whole simulation domain is composed of the global inner domain (C) and global boundary cells (D). There can be threads outside the global domain (E).

Indexing into the textures has to consider the boundary cells and overlapping threads. The global indices i_g and j_g in the x and y directions can be found as in Equation 35, where G denotes the group ID, T the thread ID in the given group and the x/y subscripts the dimension.

$$\begin{aligned} i_g &= G_x \times (K_x - 2 \times B) + T_x \\ j_g &= G_y \times (K_y - 2 \times B) + T_y \end{aligned} \tag{35}$$

These indices are used only when interacting with the simulation data textures, that is, the initial read and the final write. All other intermediate computations that require accessing the neighbors data through shared memory

use the local group thread ID T_x and T_y. The final write to the textures is only performed if the conditions given in Equation 36 are true, which guarantee that the cell is both in the global inner domain (C_g) and local inner domain (C_l).

$$\begin{aligned} C_g &= (i_g \geq B) \wedge (i_g < W + B) \wedge (j_g \geq B) \wedge (i_g < H + B) \\ C_l &= (T_x \geq B) \wedge (T_x < K_x - B) \wedge (T_y \geq B) \wedge (T_y < K_y - B) \end{aligned} \tag{36}$$

3.2.4 *Boundary conditions*

Reflection boundary conditions are used, in the form of "no-stick" boundaries, i.e, the velocity in the normal direction should be zero at the boundaries (Section 2.3.3). These are implemented using the global boundary cells, by making them mirror the 4 nearest inner domain cells, while changing the sign of the velocity in the normal direction to the respective boundary.

At the end of each solver step, a cell writes its computation results to their position in the output texture. If the cell is in the 4 nearest to the boundary, it also writes a copy to the boundary cells that should mirror it. This copy has the sign of its velocity/momentum changed depending on which dimension the cell is mirrored, i.e., the copy is multiplied by a specific boundary vector ($(1, -1, 1)^T$ and $(1, 1, -1)^T$ for x and y propagation respectively). This process is demonstrated in Figure 15.

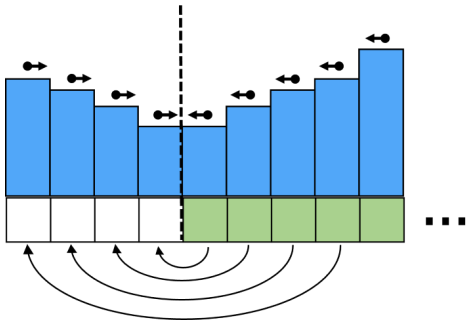

Figure 15: One dimension view of the simulation domain near the left boundary. The 4 inner domain cells closest to the boundary write their data to the mirrored global boundary cells, denoted by the bottom arrows. The cells' depths are represented by the blue columns, and the arrows above it denote the velocity in the displayed dimension. The signal of this directional momentum/velocity component is changed when written to the boundary cells.

As a way to avoid potentially reduced performance caused by thread divergence, branching based on thread ID for boundary conditions is avoided (Parna et al., 2018). As an alternative, each cell of the inner domain that is to be mirrored in the boundary cells contains the offsets to those boundary cells. The offsets then can be added to the previously computed global indices in order to find the ID of the boundary cell. As the grid is 2D, the value of a cell can be propagated up to a maximum of two boundary cells, one in the x direction and one in the y direction, so the offsets can be defined as a pair of values. To exemplify, if a cell at $(5, 5)$ should copy its data to

the boundary cells at $(4,5)$ and $(5,4)$, then it would contain the offsets $(-1,-1)$. For cells that do not copy their content to boundary cells, a large offset (k) to out-of-bounds cells is assigned [2]. Since offsets provided by 16bit integers are enough, both offsets can be packed into the single 4th component of the simulation data texture.

Summing up, all inner domain cells write their computation results to their corresponding cell, as well as a copy multiplied by a specific boundary vector into two boundary cells, defined by the offsets. Some example scenarios are presented in Figure 16.

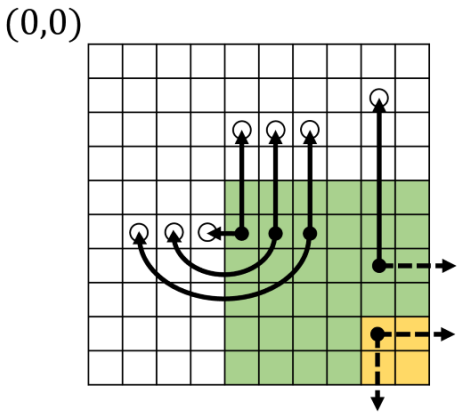

Figure 16: Propagation of data from inner domain cells to boundary cells. Example scenario of top left corner of a group. The inner domain cells that copy their data to boundary cells are highlighted in green, the ones that do not are in yellow. The solid arrows indicate the direction of the packed offsets and the dashed-arrows indicate offsets that lead to out-of-bounds accesses. At $(4,5)$ the offsets are $(-1,-3)$, at $(5,5)$ are $(-3,-3)$, at $(6,5)$ are $(-5,-3)$, at $(8,6)$ are $(k,-5)$ and at $(8,8)$ are (k,k), with k being an arbitrarily large number that always leads to out-of-bounds.

Initial setup

The packed offsets are constant during the simulation so they only need be to set up once at the start. Only the 4 inner domain cells nearest to the boundary (highlighted in green in Figure 15 and Figure 16) are copied to boundary cells, consequently, only they require specific packed offsets. The offsets to be packed at each cell can be determined as in Algorithm 1, where $nWidth$ and $nHeight$ are the width and height of the inner domain,

2 Writes to out-of-bounds texels are guaranteed to do nothing by the OpenGL specification.

OOB_OFFSET is a large offset that leads to out-of-bounds accesses and $B = 4$ is the number of boundary cells.

Algorithm 1: Offsets initialization

Input: Cell coordinates p
$ox \leftarrow OOB\_OFFSET$;
$oy \leftarrow OOB\_OFFSET$;
for $i \leftarrow 0;\ i < B;\ i \leftarrow i + 1$ **do**
  $offset \leftarrow 1 + 2 \times i$;
  if $p.x = i$ **then**
    $ox \leftarrow -offset$;
  if $p.x = nWidth - i - 1$ **then**
    $ox \leftarrow offset$;
  if $p.y = i$ **then**
    $oy \leftarrow -offset$;
  if $p.y = nHeight - i - 1$ **then**
    $oy \leftarrow offset$;
end
return packBCData(ox, oy);

Packing and unpacking

As the texture initialization is done in the GPU where bit manipulation is limited, an example of the packing and unpacking of offsets is described here. When packing the offsets, the `packHalf2x16` function is used to pack them into a single 32 bit integer followed by `uintBitsToFloat` to access the integer as a float. Subsequently, `floatBitsToUint` followed by `unpackHalf2x16` is used to extract the packed offsets. This allows the representation of a range of values up to 65536 which is more than enough for its use.

Flux boundaries

The previously described boundaries apply only to the final write of the computation results to the output texture. However, when computing the WENO reconstructions (Equation 64), the stencil of time-averaged fluxes should also consider the boundary conditions (Parna et al., 2018). These intermediate flux boundaries need to be handled manually as out-of-bounds writes causes the invalidation of shared memory. Given this, the directional flux boundaries are only set if the corresponding boundary cells are inside the current group, as defined in the condition in Equation 37 for the x direction, where O_x is the x offset and K_x the group dimension. This is set similarly for the y direction.

$$C_f = (i + O_x \geq 0 \wedge i + O_x < K_x) \tag{37}$$

To be consistent with the reflective boundary condition applied previously, the boundary cells receive copies of their respective inner domain cell time-averaged flux values multiplied by a boundary-vector that reflects the horizontal velocity components ($(-1, 1, 1)^T$).

3.2.5 *Algorithm loop*

Each solver step takes as input a simulation data texture, resulting from the previous timestep's simulation, and the bathymetry texture. During its execution, it updates the output simulation data texture, as well as the normal and height texture. The two simulation data textures are swapped out after each render step to create a loop (also called the ping-pong technique). Initially, one texture is filled with the simulations starting conditions, and then compute calls are chained that perform operations on this data, swapping out the textures in between.

A diagram of the process is presented in Figure 17. Before this loop, an initial setup step is performed that fills the required textures: first input simulation data texture, bathymetry texture and height and normal texture.

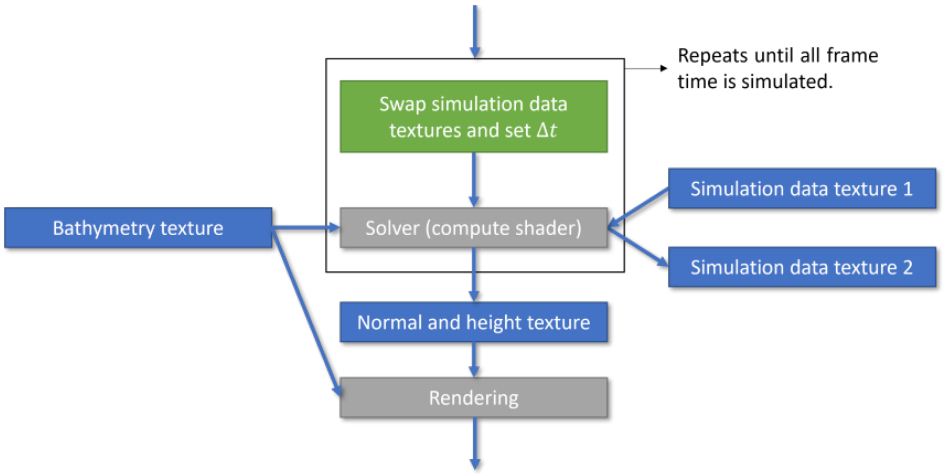

Figure 17: Diagram of the pipeline of the main rendering loop.

3.2.6 *Algorithm step*

A solver step consists of a single compute shader call which implements the previously described numerical scheme (Section 3.1). Each thread computes the required values at its corresponding cell (i, j), and intermediate values at the cell's edges at $(i + 1, j)$ and $(i, j + 1)$.

The implementation of the step follows the description and use of shared memory presented in Parna et al. (2018), adding the steps required for filling the height and normal texture. An abstract approach to GSM management is used in order to minimize the total GSM block size. GSM is treated as arrays of raw memory which are used as needed by the scheme. Each element of the arrays corresponds to a thread in the group, consequently, the arrays' length is equal to the group's size. Four arrays are used in total: one of type `float` (denoted as

GSM0) and three of type `vec3` (denoted as GSM1-3). The individual components of the elements are denoted using x, y, z, w subscripts, e.g. the first component of GSM1 is GSM1_x.

To guarantee a correct order of execution, both group memory barriers and thread synchronization barriers are used (denoted as BARRIER). These are used between accesses to the same GSM array, for example: a thread stores a value in the GSM, then the barriers are applied, then the neighboring threads read the stored value.

1. **Load the previous timestep's results**

   Compute global indices (Equation 35). Load the previous timestep's output and bathymetry values from textures. Extract boundary offsets from 4th component of simulation data. Process velocities if h is below threshold ϵ (Equation 32) and update $\boldsymbol{U}$ accordingly (Equation 33).

2. **Compute α for flux splitting, and low-order positivity preserving fluxes**

   Compute maximum eigenvalues of the flux Jacobians in the x and y direction. Store current water height h and maximum eigenvalues in GSM1. Store bathymetry b in GSM0 Compute the fluxes $\boldsymbol{F}$ and $\boldsymbol{G}$ (Equation 8) and store in GSM2 and GSM3.

   BARRIER

   Compute α for the flux splitting (Equation 61). Compute the the low-order positivity preserving fluxes $\tilde{f}$ and $\tilde{g}$ (Equation 31).

3. **Compute the time-averaged fluxes**

   Compute flux derivatives $\boldsymbol{F}_x$ and $\boldsymbol{G}_y$ and source term $\boldsymbol{S}$, using central differences for the derivatives. Compute the flux Jacobians $\frac{\partial F}{\partial U}$ and $\frac{\partial G}{\partial U}$ (Equation 23). Compute time-averaged fluxes $\tilde{\boldsymbol{F}}$ and $\tilde{\boldsymbol{G}}$ (Equation 22). Store low-order fluxes $\tilde{f}$ and $\tilde{g}$ in GSM1.

   BARRIER

   Compute Γ (Equation 75). Store the modified conserved quantities (η, hu, hv) for the flux splitting in GSM3. Store x directional time averaged flux $\tilde{\boldsymbol{F}}$ in GSM2.

   BARRIER

4. **Do the WENO reconstruction in the x direction**

   Store $\tilde{\boldsymbol{F}}$ multiplied by the boundary vector in the boundary cells (in GSM2) to set the flux boundary conditions in the x direction (Equation 37).

   BARRIER

   Do the WENO reconstruction (Equation 24, described in Appendix B)..

5. **Do the WENO reconstruction in the y direction**

   Repeat the previous step but for the y direction.

6. **Enforce positivity preservation in the h component of the flux**

   Compute $\mathcal{F}_{i+1/2,j}$ and $\mathcal{F}_{i,j+1/2}$ (Equation 76), and store in GSM2$_{\text{xy}}$.

   BARRIER

   Compute bounding Λ values (Equation 79, Equation 77 and Equation 78). Store Λ_L and Λ_D in GSM1.

   BARRIER

   Compute the interpolation coefficients $\theta_{i+1/2,j}$ and $\theta_{i,j+1/2}$ (Equation 73). Replace the first component of $\hat{F}$ and $\hat{G}$ with the respective positivity preserving term (Equation 30). Store $\hat{F}$ and $\hat{G}$ in GSM2 and GSM3.

7. **Compute final result U_{n+1}**

   Store second source term reconstructions $\hat{S}$ in the x and y directions in GSM0 and GSM1$_{\text{z}}$.

   BARRIER

   Determine U_{n+1} as in Equation 26 (with S as in Equation 28).

8. **Compute surface height and normal**

   Compute updated total height ($\eta = h + b$) and store in GSM0.

   BARRIER

   Compute surface normal using central differences of the heights.

9. **Store results**

   Write results to corresponding textures based on conditions in Equation 36. To the height and normal texture: write normal and height to current cell. To the active simulation data texture: write U_{n+1} to current cell and write U_{n+1} multiplied by the corresponding boundary vector the boundary cells specified by the offsets.

3.2.7 *Results*

Examples of the results produced by the scheme are shown in Figure 18 and Figure 19, as a set of 3D snapshots. Geometry is rendered as described in Section 4.1, with simple diffuse shading. The first example (Figure 18) highlights the boundary reflections in a fully wet simulation, while the second example (Figure 19) highlights the advance of a wet/dry contact line.

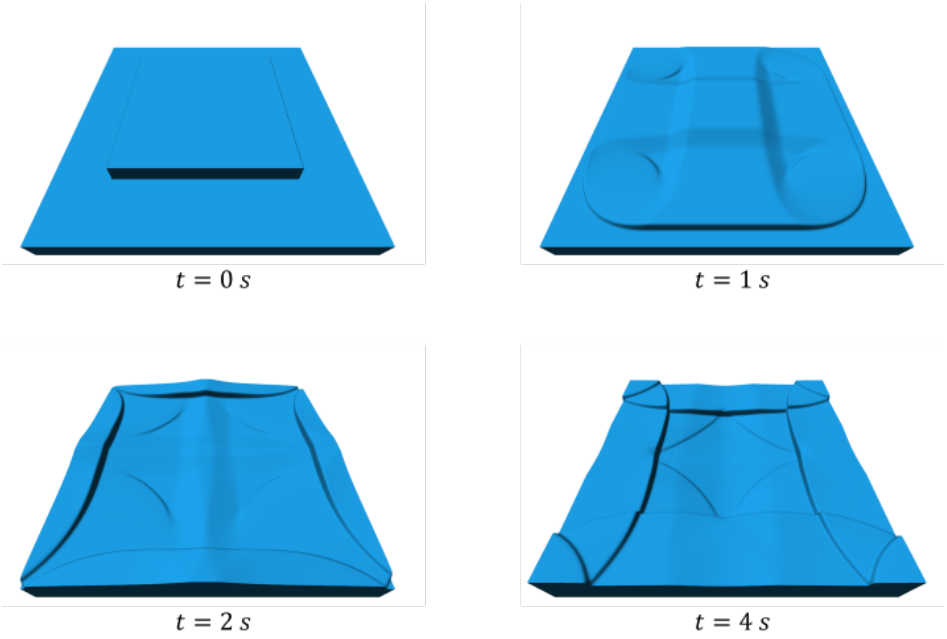

Figure 18: 3D snapshots of a simulation at $t \approx 0, 1, 2, 3s$. The bathymetry is flat and the water height is set to $8m$ in the center and $5m$ everywhere else. Initial velocities are set to 0. Grid size is $512x512$ (including boundary cells) with $\Delta x = \Delta y = 0.1m$ and maximum timestep $t = 0.01s$.

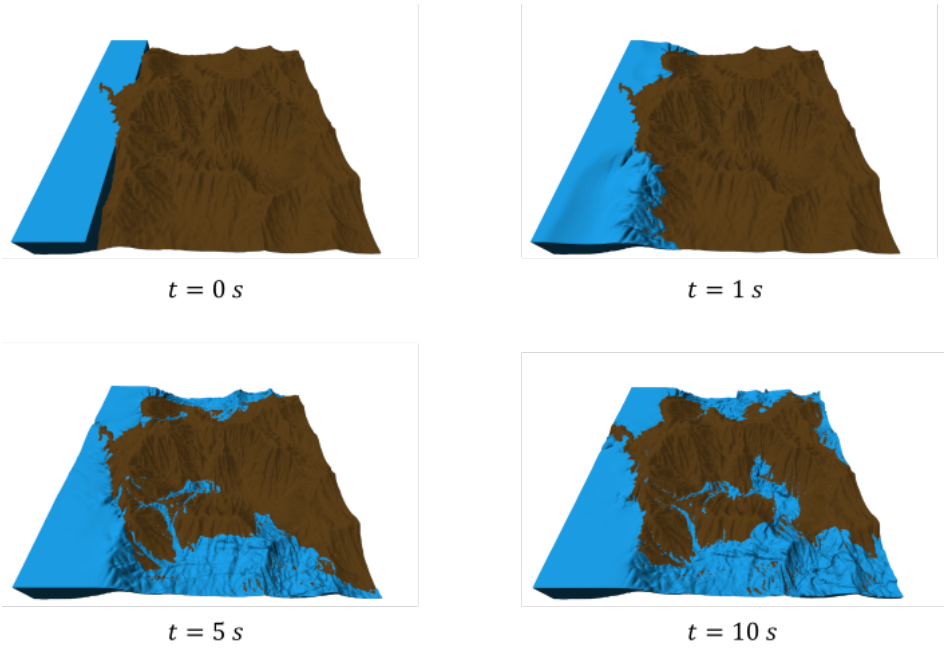

Figure 19: 3D snapshots of a simulation at $t \approx 0, 1, 5, 10s$. The bathymetry is a complex heightfield and the water height is set to $8m$ in the left column and 0 depth everywhere else. Initial velocities are set to 0. Grid size is $512x512$ (including boundary cells) with $\Delta x = \Delta y = 0.1m$ and maximum timestep $t = 0.01s$.

4

RENDERING

This chapter deals with the visualization of the water simulation framework developed in the previous chapter. First, the geometric representation of the water surface and bottom terrain is addressed (Section 4.1). This is followed by the description of the water surface color computations (Section 4.2). The extension of this result is discussed next with the addition of caustics (Section 4.3). Finally, a method for adding small-scale surface details is presented (Section 4.4).

4.1 RENDERING GEOMETRY

Both the bottom terrain and water surface are rendered as an uniform grid-shaped mesh. In both cases only the inner domain is considered, excluding the boundary cells which are required for the solver. The vertex buffers are filled with the terrain's positions and normals.

When rendering the water surface mesh, at the vertex shader, its height and normal are sampled from the corresponding texture. Since the depth can be very close to 0, but not 0, a cell is considered dry if its depth is under a certain threshold. Dry cells then have their height set to slightly lower than the terrain in order to avoid z-fighting and to have a clearly defined shoreline.

The water volume's sides are setup in a similar way, where the vertex buffers are filled with the terrain's positions and normals. When rendering, the upper vertices set their height to match the water surface height, and the bottom vertices set the height to 0. An example of the geometry used is shown in Figure 20, rendered as a wireframe.

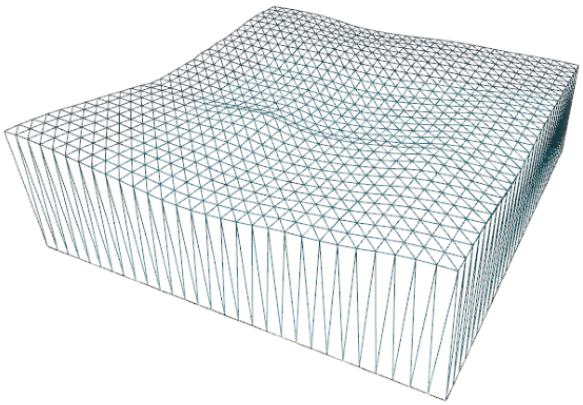

Figure 20: Water volume surface and sides rendered as a wireframe

For dry cells in a dry-wet boundary, its height can be extrapolated by averaging the neighboring wet cells values. The goal is to avoid water climb up artifacts that can happen in these boundaries, as illustrated in 21. This is less of a problem when the water is clear and in denser grids, so its use can be scene dependent. The required neighboring cells data is sampled from the simulation data and bathymetry textures.

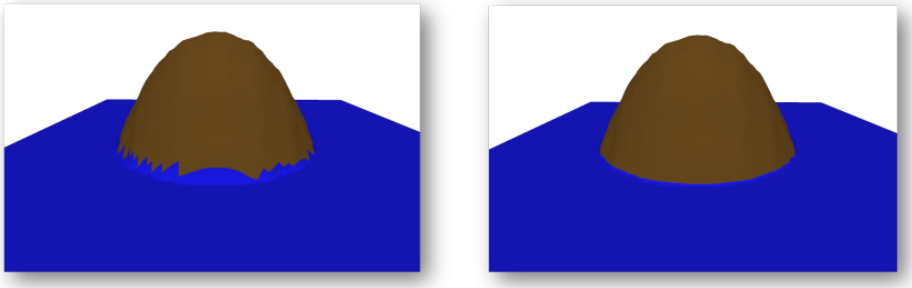

Figure 21: Water climb up artifacts visible in dry-wet boundary (left) and same scene with height averaged from neighbors (right)

The drawing passes are executed before the solver pass, so the first frame contains the initial simulation state. The water surface is drawn after the bottom terrain as the fragment calculations are expected to be more expensive and therefore can be skipped in case the water is covered by the terrain. All geometry is scaled by Δx in the x and y dimensions so it keeps the correct proportions with the height.

4.2 WATER SURFACE COLOR

This section presents the methods used to compute the water's surface color. Similarly to most water rendering methods (Ma et al. (2016), Bruneton et al. (2010)), three components are considered: the Sun's specular reflection C_{sun}, the environment's reflection C_{env} and the transmitted light C_{trans} which encompasses the bottom terrain reflection and scattered light. These components are described separately, with an overview and examples of the results obtained at the end of the section.

4.2.1 *Sun reflection*

The Sun's reflection is computed by using a microfacet BRDF to describe the surface and a directional light to represent the Sun. Certain models have been used in ocean rendering but these rely on the statistical properties of the deep ocean waves (Bruneton et al., 2010) so a more general BRDF is required. The Sun's specular reflection can therefore be given by the Cook-Torrance specular term (Guy and Agopian, 2019), as in Equation 38.

$$f_{spec}(\boldsymbol{p}, \boldsymbol{v}, \boldsymbol{l}) = \frac{D(\boldsymbol{h})F(\boldsymbol{l}, \boldsymbol{h})G(\boldsymbol{l}, \boldsymbol{v}, \boldsymbol{h})}{4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})} \tag{38}$$

By considering the sun as a directional light, this BRDF can then be used to compute the Sun's contribution to radiance as in Equation 39.

$$L_{sun} = \pi f(\boldsymbol{p}, \boldsymbol{l}, \boldsymbol{v}) \circ C_{sun}(\underline{\boldsymbol{n} \cdot \boldsymbol{l}}) \tag{39}$$

The $D(\boldsymbol{h})$ term follows the GGX distribution described in Walter et al. (2007), which is one of the most widely used in the rendering industry (Heitz, 2018). The underline denotes a function such that $\underline{a} = \max(a, 0)$ and is used to avoid negative numbers if the angle in a dot product is higher than $90\,\mathrm{deg}$.

$$D(\boldsymbol{n}, \boldsymbol{h}) = \frac{\alpha^2}{\pi((\underline{\boldsymbol{n} \cdot \boldsymbol{h}})^2(\alpha^2 - 1) + 1)^2} \tag{40}$$

This microfacet distribution is commonly used with the matching Smith shadowing function (as the $G(\boldsymbol{v}, \boldsymbol{l}, \alpha)$ term), derived in the same paper (Walter et al., 2007). The geometry term term can be replaced with a visibility term $V(\boldsymbol{l}, \boldsymbol{v}, \boldsymbol{h})$ which is a simplification of the BRDF by merging the denominator $4(\boldsymbol{n} \cdot \boldsymbol{l})(\boldsymbol{n} \cdot \boldsymbol{v})$ with the geometric term (Hoffman, 2013). The final BRDF is therefore given by Equation 41 where the visibility term is given in Equation 42 (Guy and Agopian, 2019).

$$f_{spec} = D(\boldsymbol{h})V(\boldsymbol{l}, \boldsymbol{v}, \boldsymbol{h})F(\boldsymbol{l}, \boldsymbol{h}) \tag{41}$$

$$
\begin{aligned}
V(v,l,\alpha) &= \frac{G(\boldsymbol{v},\boldsymbol{l},\alpha)}{4(\boldsymbol{n}\cdot\boldsymbol{l})(\boldsymbol{n}\cdot\boldsymbol{v})} = V_1(\boldsymbol{l},\alpha)V_1(\boldsymbol{v},\alpha) \\
V_1(v,\alpha) &= \frac{1}{\underline{\boldsymbol{n}\cdot\boldsymbol{v}} + \sqrt{\alpha^2 + (1-\alpha^2)(\boldsymbol{n}\cdot\boldsymbol{v})^2}}
\end{aligned}
\tag{42}
$$

Heitz (2014) notes however that taking the height of the microfacets into account to correlate masking and shadowing leads to more accurate results. The height correlated visibility function thus becomes Equation 43, as in Guy and Agopian (2019).

$$
V(v,l,\alpha) = \frac{0.5}{\underline{\boldsymbol{n}\cdot\boldsymbol{l}}\sqrt{\alpha^2 + (1-\alpha^2)(\boldsymbol{n}\cdot\boldsymbol{v})^2} + \underline{\boldsymbol{n}\cdot\boldsymbol{v}}\sqrt{\alpha^2 + (1-\alpha^2)(\boldsymbol{n}\cdot\boldsymbol{l})^2}} \tag{43}
$$

4.2.2 *Intersecting the heightfield*

Reflected and refracted rays need to be intersected with the environment in order to compute the light going in its direction and effects along its path. Only intersections with the terrain are considered, and therefore, only one intersection test is needed for either refractions or reflections. As the terrain is represented by a heightfield, a heightfield ray-casting approach is followed for both refractions and reflections.

The maximum mipmaps technique is used as it provides a fast and scalable solution for intersecting heightfields, with negligible precomputing (Dick et al. (2009), Silvestre (2017)). The ray-casting algorithm relies on a precomputed maximum mipmap as an acceleration structure to skip empty space when advancing the ray. A limitation of this algorithm is that, due to its quadtree structure, it can only be used with heightfields in power-of-two resolutions.

Maximum mipmap

Texels in the maximum mipmap contain the maximum height value over a certain extent of the heightfield depending on the mipmap level. The maximum mipmap follows a quadtree structure, where a texel in given mipmap level contains four children texels in the next finer level. The finest level (0), is the original heightfield texture, and the coarser levels are associated with ascending level numbers. Each mipmap level is, therefore, computed by determining the maximum value of the four children texels in the level below and storing it in the parent texel. This process starts at the finest level, and is applied recursively on each level until the coarsest level is reached, which consists of a single texel. A visual representation of the maximum mipmap is presented in Figure 22. The mipmap is generated at the application's start, and introduces additional GPU memory requirements of $1/3$ of the memory needed to store the original heightfield.

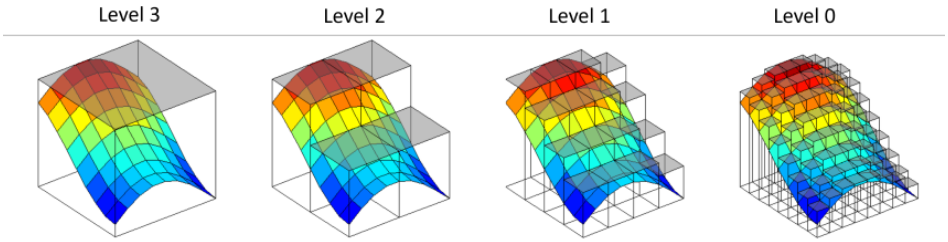

Figure 22: Visualization of different levels in a maximum mipmap representing an heightfield. Each texel can be seen as a bounding box over a certain extent of the heightfield (from Tevs et al. (2008)).

Ray traversal algorithm

The ray-traversal method described in Dick et al. (2009) is followed, using a linear approximation for the final intersection test (Drobot, 2018).

Initially, the ray's starting position and direction are converted to the heightfield's texture size coordinates from world coordinates. After the final hit point is determined, the position is converted back to world coordinates.

The following description of the algorithm applies to an example ray direction dir with $dir.x \geq 0$ and $dir.z \geq 0$. To reduce the number of conditional branches within the ray-casting loop, the sign ($\geq 0, < 0$) of each $dir.x$, $dir.z$ and $dir.y$ is tested at the beginning, and the ray-casting loop is replicated for each of the eight branches. The differences between branches are explained when relevant.

The traversal algorithm steps the ray through any of the mipmap levels by recalculating the ray entry and exit points of each texel. The entry point of the current heightfield texel is denoted by $texEntry$ and is initially equal to fragment's position. The ray-casting loop is run while the ray does not intersect the heightfield and does not leave the domain, i.e, $texEntry.x < N \wedge texEntry.z < N$. For other branches these conditions are adjusted accordingly, for example if $dir.x < 0$, the x boundary condition is instead $texEntry.x \geq 0$. If $dir.y > 0$, an additional boundary is set at the highest level of the heightfield, which can be sampled from the coarsest level of the mipmap ($texEntry.y \leq maxHeight$). Let l denote the current level of the mipmap which is used to test for ray intersection. Initially, the second coarsest level is used, so $l = maxLevel - 1$.

At the start of each ray-casting step, the current heightfield texel is fetched from the current level. The texel's coordinates are given by $(\lfloor \frac{texEntry.x}{2^l} \rfloor, \lfloor \frac{texEntry.z}{2^l} \rfloor)$. For other branches, for example if $dir.x < 0$, the texel's x coordinate would be instead $\lceil \frac{texEntry.x}{2^l} \rceil - 1$. This also applies to any other texel's coordinates, such as in texel's edges.

The ray's exit point $texExit$ is then computed from that texel. For this, the ray is intersected with the texel's edges, as illustrated in Figure 23. Of the two intersections, $texExit$ becomes the closest one, given by the smaller ray parameter Δt.

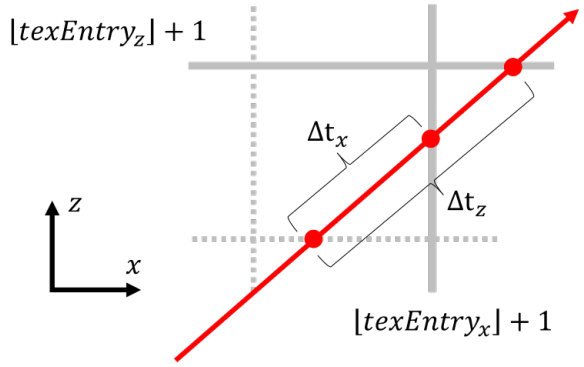

Figure 23: The exit point of a ray from a texel is determined by considering two of its edges. Of the ray's intersection points with these edges, $texExit$ is the one closest to $texEntry$, i.e., with the smaller Δt.

To avoid infinite looping due to rounding errors, the coordinates are set to the intersected texel edge, for example, if $\Delta t = \Delta tx$ then $texExit.x = \lfloor texEntry.x \rfloor + 1$. The full process for determining $texExit$ is given in Algorithm 2.

Algorithm 2: Determining the exit point of a heightfield texel

$texEdges.xz \leftarrow (\lfloor \frac{texEntry.xz}{2^l} \rfloor + 1) \times 2^l$
$\Delta t.xz \leftarrow \frac{texEdges.xz - texEntry.xz}{dir.xz}$
$\Delta t \leftarrow min(\Delta tx, \Delta t_z)$
$texExit \leftarrow texEntry + \Delta t \times dir$
if $\Delta t = \Delta tx$ **then**
  $texExit.x \leftarrow texEdges.x$
else
  $texExit.z \leftarrow texEdges.z$
end

Having determined $texExit$, the intersection test is performed. This test depends on the ray's vertical direction: If the ray is going upwards ($dir.y \geq 0$) the ray intersects the texel if $texEntry.y < height$; if the ray is going downwards ($dir.y < 0$) the ray intersects the texel if $texExit.y < height$. If an intersection is detected in the first case, the intersection point is equal to $texEntry$. In the second case, the intersection point is computed as in Equation 44. Both outcomes are illustrated in Figure 24.

$$intersection = texEntry + max\left(\frac{height - texEntry.y}{dir.y}, 0\right) \times dir \tag{44}$$

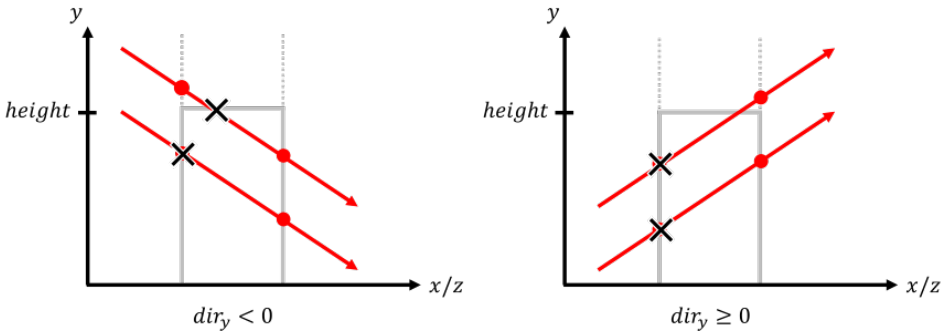

Figure 24: The ray's entry and exit points at the texel are denoted by a red point, and the intersection points by a black cross. When testing for intersection, the height value is compared to either the entry or exit point, depending on whether the ray is going upwards or downwards, respectively.

If the ray intersects the texel, the ray is advanced to the intersection point, so $texEntry$ becomes $texExit$. Additionally, if $l > 0$, traversal must proceed to the next finer level to test intersections more accurately on a child texel ($l \leftarrow l - 1$).

Once at the lowest level ($l = 0$), instead of a descending operation, an intersection test is performed with the heightfield. Here, the surface is approximated as a linear interpolation (Drobot, 2018). The heightfield is sampled at the entry and exit positions of the cell, $texEntry$ and $texExit$. Then, the final point is given as the intersection between the ray and linearly approximated curve created by the sampled points, as illustrated in Figure 25.

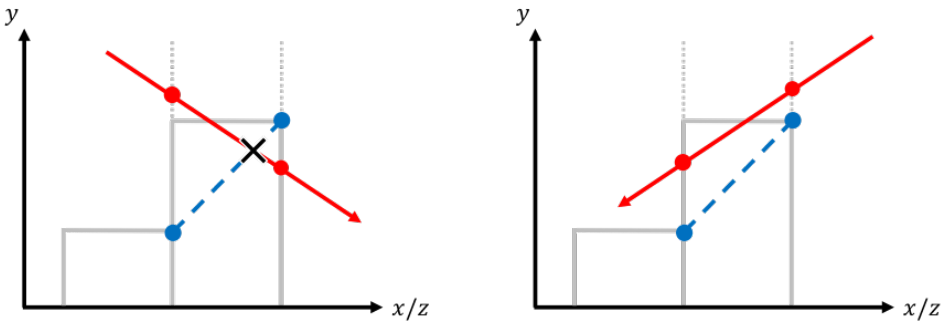

Figure 25: At the finest level, the ray (denoted in red) is intersected with a linear approximation of the heightfield (denoted in blue). Even if the ray intersects the texel it can miss the linear intersection test (left).

If the ray does not intersect the heightfield within the texel, $texEntry$ becomes $texExit$ and the loop advances to the next step. If the texel that would be tested in the next traversal iteration has a different parent texel than the current texel, the level is increased ($l \leftarrow l + 1$). This process is given in Algorithm 3.

Algorithm 3: Increasing the mipmap level if necessary

if $\Delta t = \Delta tx$ **then**
  $edge \leftarrow \lfloor \frac{texExit.x}{2^l} \rfloor$
else
  $edge \leftarrow \lfloor \frac{texExit.z}{2^l} \rfloor$
end
$l \leftarrow min(l + 1 - (edge \pmod 2), maxLevel - 1)$

If it leaves the domain without intersecting the heightfield, the ray is intersected with the boundary. This is performed similarly to Algorithm 2, but intersecting the domain boundaries instead of the texel's edge.

An overview of the full algorithm is given in Algorithm 4, for heightfield size $N \times N$. The return values are a boolean which indicates if an intersection with the terrain was found, and the intersection point position.

Algorithm 4: Maximum mimpmap heightfield intersection

Input: Ray direction and initial position.

$l \leftarrow maxLevel - 1$

$texEntry \leftarrow rayOrigin$

while $texEntry.x < N \wedge texEntry.z < N$ **do**

  Fetch $height$ from current texel and level.

  Determine $texExit$ (Algorithm 2).

  if $texExit.y \leq height$ **then**

    Advance $texEntry$ to intersection point (Equation 44).

    if $l > 0$ **then**

      Descend one level.

    else

      Check intersection with heightfield (Figure 25).

      if *intersects heightfield* **then**

        return true and hit position.

      else

        Advance $texEntry$ to $texExit$.

        Ascend one level if needed (Algorithm 3).

      end

    end

  else

    Advance $texEntry$ to $texExit$.

    Ascend one level if needed (Algorithm 3).

  end

end

Compute accurate intersection with domain boundaries.

return false and hit position.

4.2.3 *Environment reflection*

In addition to the Sun's reflection, the environment's reflection is also considered. These include the parts of the terrain that are above the water surface, and the far-away environment, namely the sky. A simplified approach is used where the water surface is assumed as perfectly specular, so only one direction needs to be sampled

to obtain the reflection color. This value is then multiplied by the Fresnel reflectance about the surface normal, resulting in Equation 45.

$$C_{env} = F(\underline{\boldsymbol{v} \cdot \boldsymbol{n}})C_{env} \tag{45}$$

The reflection color C_{env} is determined by casting a ray to intersect the terrain, using the previously described method. If it intersects the terrain, its color and normal are fetched from the respective textures and its lighting is computed using a perfectly diffuse BRDF. For rays that do not intersect the terrain, a color is fetched from a cubemap that depicts the far-away environment. Both outcomes are exemplified in Figure 26.

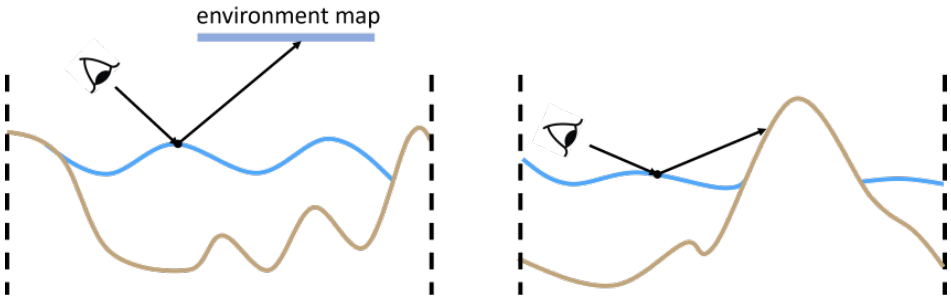

Figure 26: A reflected ray can intersect the terrain (right), or sample from an environment map (left).

4.2.4 *Transmission*

The transmitted color C_{trans} is composed of light reflected on the bottom C_{bottom} and scattered light (due to in-scattering) $C_{scatter}$, while subject to attenuation. Light that is not attenuated is determined by the transmittance $T(d)$, as given by Equation 19, where d is the distance that light traveled while inside the medium. This is under the assumption that the water volume is homogeneous, i.e. it has the same extinction coefficient in all its volume. Scattering is simply approximated as an added constant diffuse color, $C_{scatter}$.

An additional intensity modifier I_c is fetched from a caustics map texture and multiplied by the bottom color. The caustics map generation process is described in the following section (Section 4.3). The final transmission color computation is given by Equation 46.

$$C_{trans} = (1 - F(\underline{\boldsymbol{v} \cdot \boldsymbol{n}})) \times (I_c C_{bottom} T(d) + C_{scatter}) \times C_{sun} \tag{46}$$

Refracted ray

The bottom reflection color C_{bottom} and transmission distance d_v are determined by intersecting a refracted ray with the bottom terrain. Most water bodies are bounded by terrain, such as lakes and rivers, so a ray that enters the water is expected to hit the ground further on. This, however, is not true for transparent enclosing surfaces, such as in an aquarium, or terrain sections with invisible boundaries. When hitting one of these surfaces, the ray

can then be refracted again and used to fetch a color from an environment map (as used for reflections). Both cases can be visualized in Figure 27.

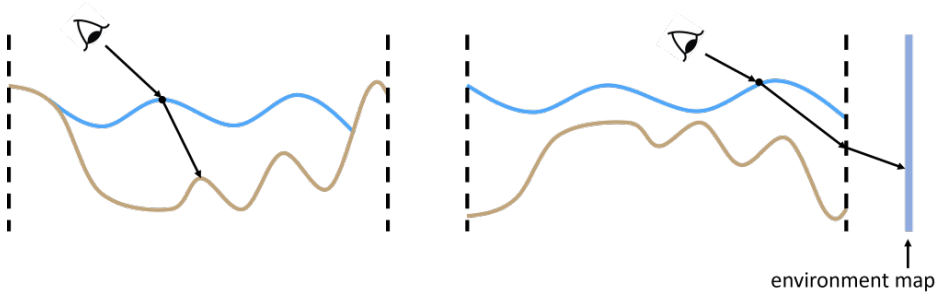

Figure 27: A refracted ray can intersect the terrain (left), or a bounding box where it is refracted again and samples from an environment map (right).

As with the reflections, the bottom color and normal are fetched from the corresponding textures using the intersection point's position. Similarly, a perfectly diffuse BRDF is used for the lighting computation.

Light Path

When computing the transmittance of the refracted ray during the light's path inside the water volume, the whole path should be considered, that is, both before and after the light is reflected at the bottom. The former is given by the intersection algorithm, while the latter can be approximated as coming from a flat surface at the same height as the entry point. The total distance is given by Equation 47 and is illustrated in Figure 28, where d_v is the distance determined during ray intersection and $\cos\theta = \boldsymbol{n} \cdot \boldsymbol{l}_r$. The water height h is given by the difference between the height of the current fragment and of the intersection point.

$$d = d_v + \frac{h}{\cos\theta} \tag{47}$$

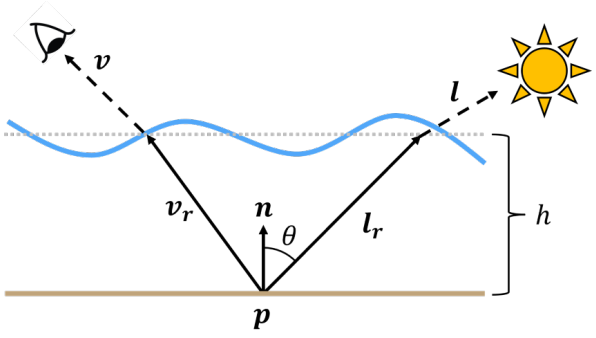

Figure 28: Path traveled by the light inside the water body before it reaches the viewer, represented by the solid vectors $\boldsymbol{v}_r$ and $\boldsymbol{l}_r$.

A more accurate estimate for this distance can also be retrieved from a texture. This distance is computed during the caustics simulation and stored in the texture in a previous pass, as is described in the following section (Section 4.3). The previous approximation has the worst results when the light ray enters and leaves the water surface at considerably different heights. As such, when using the more accurate path obtained from the caustics simulation, the difference in results will be visible where these heights most differ. This difference is exemplified in the comparison presented in Figure 29, where $\boldsymbol{l}_r$ is the approximated light path, and $\boldsymbol{l}_{rc}$ the one determined during caustics simulation.

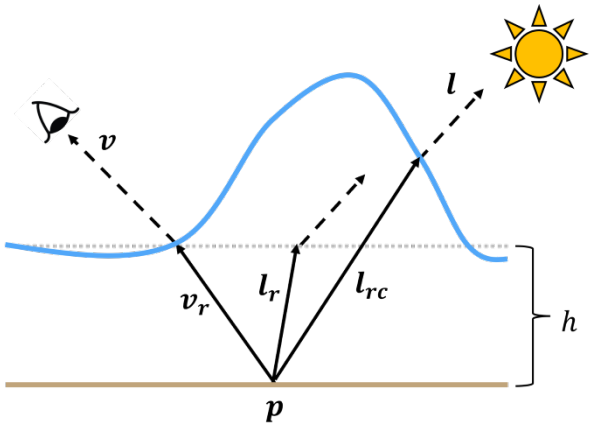

Figure 29: Comparison between an approximated light path, $\boldsymbol{l}_r$, and a more accurate path computed during caustics simulation, $\boldsymbol{l}_{rc}$.

A comparison of results is presented in Figure 30. The difference is particularly noticeable if the water has low transmittance, as the distance traveled by the light inside the water volume will have a greater impact on its final color.

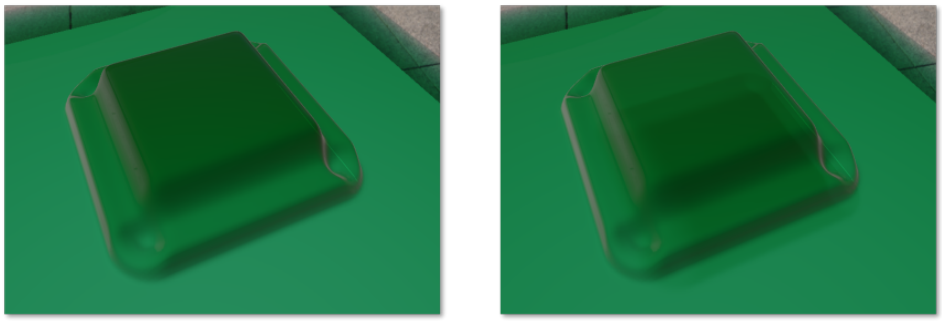

Figure 30: Comparison between an approximated light path (left) and a more accurate path computed during caustics simulation (right). A high absorption coefficient is used, and the caustics' intensity is not considered for better clarity.

4.2.5 *Overview*

The water's final color is defined as a mix of reflected and transmitted light (Equation 48) where the Fresnel term is computed using Schlick's approximation (Equation 16). To be noted that C_{Sun} already includes the Fresnel term in its computation.

$$C = C_{Sun} + F(\theta_i)C_{env} + (1 - F(\theta_i))C_{trans} \tag{48}$$

An overview of the full color computation is presented in Algorithm 5, which is implemented in a fragment shader. All calculations are done in world space, with the appropriate conversions to texture space performed when needed. Only the inner domain is considered when accessing simulation data textures. A normal map can also be applied to the water surface, which is discussed in Section 4.4.

Algorithm 5: Water surface shading

Input: eye, normal, light

$normal \leftarrow applyNormalMap(normal)$
$reflectDir \leftarrow reflect(eye, normal)$
$hit \leftarrow intersectTerrain(reflectDir)$
if *hit* **then**
  $C_{bottom}, tNormal \leftarrow$ Sample terrain color and normal from textures
  $C_{env} \leftarrow diffuse(C_bottom, tNormal)$
else
  $C_{env} \leftarrow$ Sample environment map color from texture
end
$refractDir \leftarrow refract(eye, normal)$
$hit, pathLength \leftarrow intersectTerrain(refractDir)$
if *hit* **then**
  $C_{bottom}, tNormal \leftarrow$ Sample terrain color and normal from textures
  $C_{refr} \leftarrow diffuse(C_bottom, tNormal)$
  $causticsIntensity, causticsPathLength \leftarrow$ Sample caustics data from textures
  $C_{refr} \leftarrow C_{refr} \times causticsIntensity$
  $pathLength \leftarrow pathLength + causticsPathLength$
else
  $C_{refr} \leftarrow$ Sample environment map color from texture
end
$fresnel \leftarrow fresnelSchlick(eye, normal)$
$C_{sun} \leftarrow cookTorranceSpecular(normal, eye, lightDir) \times normal \cdot lightDir$
$C_{env} \leftarrow C_{env} \times fresnel \times normal \cdot lightDir$
$C_{trans} \leftarrow (1 - fresnel) \times C_{refr} \times (transmittance(pathLength) + C_{scatter})$
return $C_{sun} + C_{env} + C_{trans}$

4.2.6 *Results*

Examples showcasing different parameters are presented in Figure 31, based on varying extinction coefficients and diffuse scattering colors. These are what mainly affect the water tone and bottom visibility. Another example of the rendering results is shown in Figure 32, showcasing the contributions of the various components in separate.

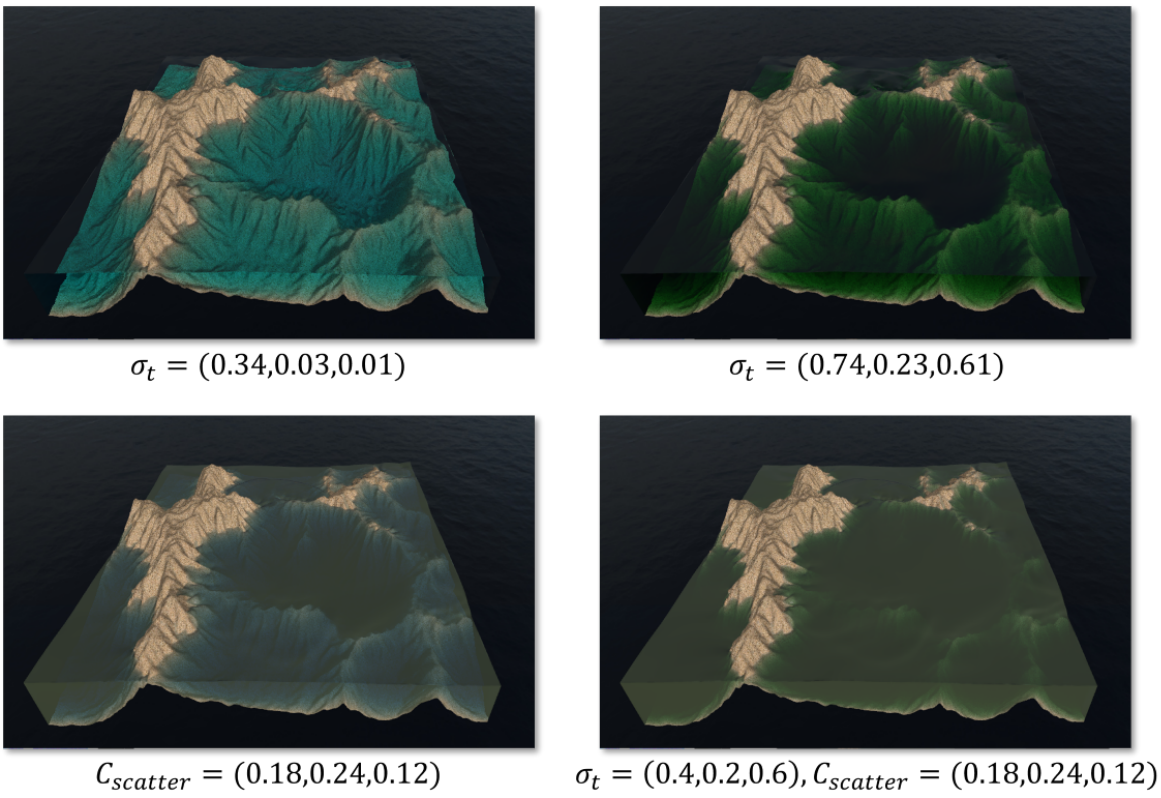

Figure 31: Different water tones obtained by varying the extinction coefficient (σ_t) and diffuse scattering color ($C_{scatter}$).

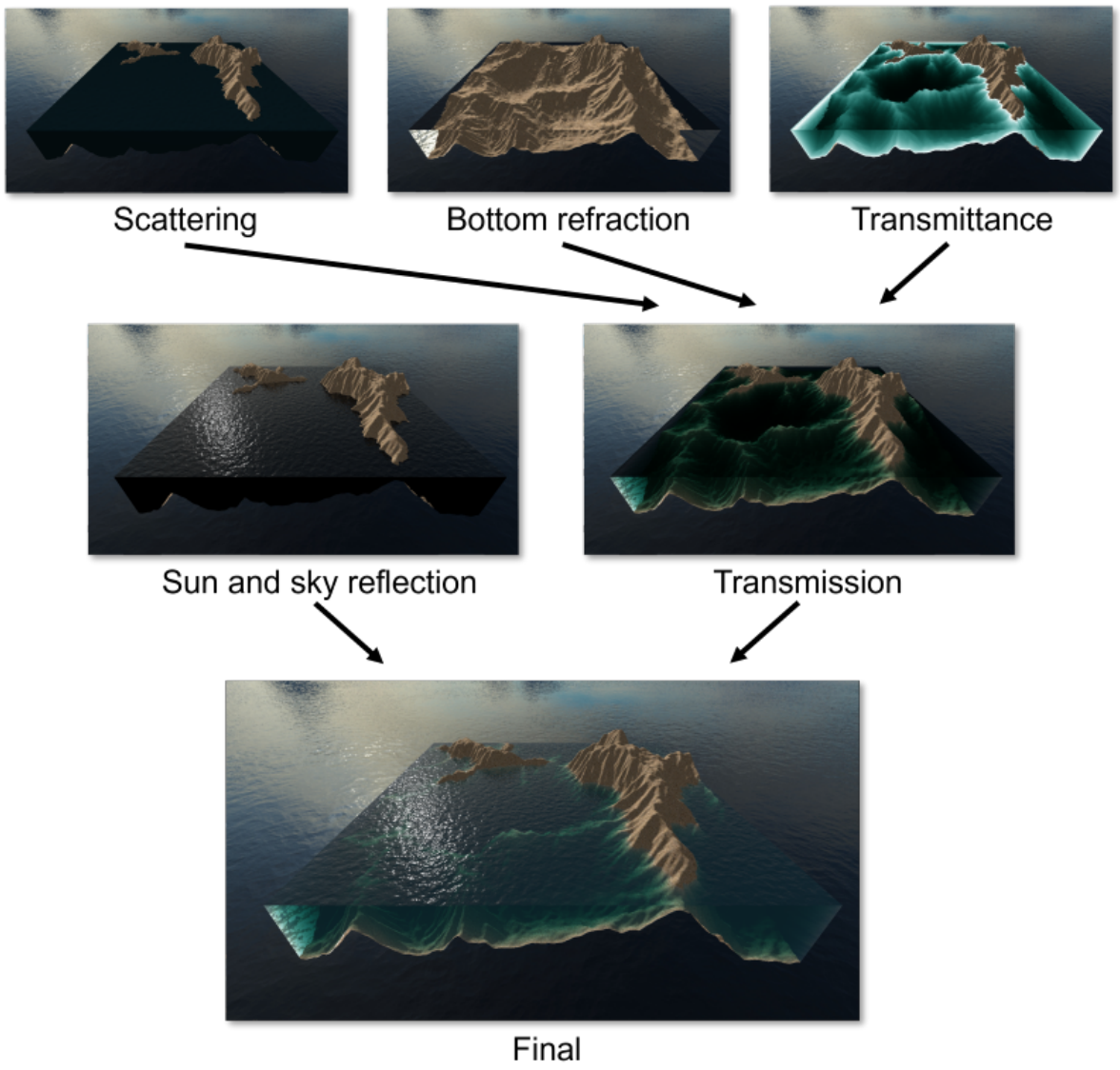

Figure 32: Contribution of each component to the water's final color. Reflection and transmission include the Fresnel factor. A normal map is applied to the surface as a way to add small-scale detail (Section 4.4).

4.3 CAUSTICS

This work will consider caustics generated due to refraction, as they are projected on the bottom terrain and visible when viewing the water surface from the outside. The approach in Parna (2020) for generating a caustics map is followed as it is particularly suited for heightfield surfaces and allows improved intensity computations. Some further modifications are done to take advantage of the heightfield bottom terrain, and an additional light path length output is generated. The algorithm consists of three main steps: computing the caustics vertex buffer (where the bulk of the complexity is), rendering the caustics map, and application of the caustics map to

a receiver diffuse surface, namely the bottom terrain. This section will go through each of the steps and present the obtained results.

4.3.1 *Simulation*

Caustics are formed when multiple rays of light converge at a single point. This occurs in water as the refraction at the water surface causes the light rays to deviate from their original path and converge to a common region. The core of the simulation emulates the photon paths from the light source through the water surface (Figure 33). This is often modeled using a light view-space grid (Shah et al. (2007), Papadopoulos and Papaioannou (2010)) where each vertex of the grid constitutes a single photon to be traced. Since this work uses a heightfield representation for the water surface which has a grid structure already, this can be exploited by considering the world space vertices as reference for the photons to be traced (Parna, 2020). This also removes the dependency on the light view resolution (as in Shah et al. (2007)) and instead couples it to the fluid simulation resolution.

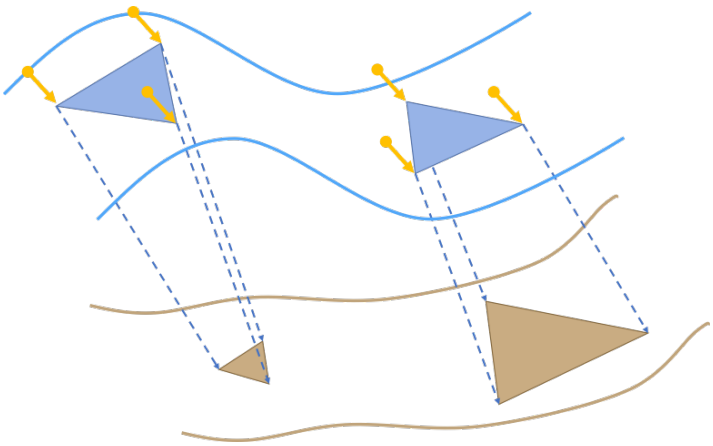

Figure 33: Overview of caustics simulation, vertices of a water surface grid emulate photons and are intersected with the bottom terrain. The typical caustic patterns would be formed by the triangles such as the one on the left where the rays converge, which results in a higher light intensity.

The entire caustics simulation is performed with a single compute kernel where each thread matches a vertex in the caustics grid. The threading scheme is similar to the fluid simulation (Section 3.2.3), but only a single-cell boundary layer is required. The global indexes are given by Equation 35 and the number of compute groups to be dispatched is given by Equation 34. The output of this step is stored in a vertex buffer that has 4 floats per vertex, 2 for the x and z world space position of the hit points, 1 for the corresponding light intensity and 1 for the light path length.

Initial Positions

The simulation pass starts with determining the vertex position assigned to each thread, which is sampled from the simulation heights and normals texture. This data is the starting point for the refracted ray used in the

following simulation steps. Each vertex is also considered to be associated with a grid cell that consists of two triangles, which will be referred to as the vertex's associated triangles.

A caustics grid of size $NW \times NH$ is used where N defines the grid density increase (W/H are the simulation grid width/height), i.e., $N = 1$ results in the caustics grid matching the simulation grid and $N = 2$ results in a 4 times denser caustics grid compared to the simulation grid. The scaled grid spacing is given by Equation 49 and the caustics grid initial vertex position is therefore given by Equation 50, where (i, j) are the global indexes and η is the surface height sampled from the simulation data texture at the specified position.

$$\begin{aligned} \beta_x &= \frac{W-1}{NW-1}\Delta x \\ \beta_y &= \frac{H-1}{NH-1}\Delta x \end{aligned} \tag{49}$$

$$P = (i\beta_x, \eta, j\beta_y) \tag{50}$$

Post-refraction hit-point detection

The intersection point of the refracted ray with the receiver geometry must be computed in order to output the final vertex position. Since the intersections are not required to be completely accurate, as the patterns generated are the most important feature, a cheaper but less accurate intersection algorithm is used (compared to the one described in Section 4.2). A similar algorithm to the one described in Shah et al. (2007) is used, which is an iterative estimation process that is derived from the Newton-Raphson root finding method. The original algorithm requires a previous pass to render the positions of the receiver geometry. As only the bottom heightfield is considered as receiver geometry in this case, the bathymetry texture can be used instead.

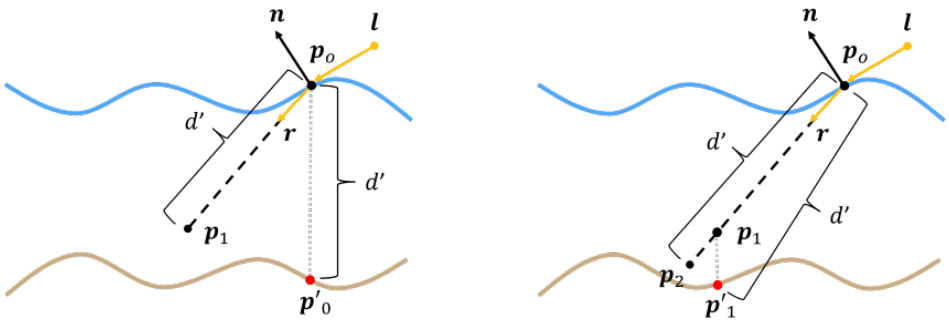

Figure 34: Diagram of two iterations of the intersection estimation algorithm (left to right). The light gray dotted line correspond to lookups to the bathymetry texture.

Let p_0 be the position of the current vertex with surface normal $\boldsymbol{n}$, the light direction be $\boldsymbol{l}$ and the normalized refracted light vector be $\boldsymbol{r}$, as illustrated in Figure 34. Points along the refracted ray are thus defined as in Equation 51, where d is the distance from the vertex $\boldsymbol{p}_0$.

$$\boldsymbol{p} = \boldsymbol{p}_0 + d \times \boldsymbol{r} \tag{51}$$

Estimating the point of intersection comes down to estimating the value of d, the distance between $\boldsymbol{p}_0$ and the receiver geometry along $\boldsymbol{r}$. Let $\boldsymbol{p'}$ be the project texture sampling of $\boldsymbol{p}$ on the bathymetry texture with a top-down view, therefore sharing the same coordinates. Point $\boldsymbol{p}_0$ is initially projected into the bathymetry and the distance d' between $\boldsymbol{p}_0$ and the looked up position $\boldsymbol{p}'_0$ is used as an estimate value for d in Equation 51 to obtain a new point, $\boldsymbol{p}_1$.

This process composes a single iteration of the algorithm. For example, in the next iteration $\boldsymbol{p}_1$ is projected into the bathymetry to obtain $\boldsymbol{p}'_1$ and the distance between $\boldsymbol{p}'_1$ and $\boldsymbol{p}_0$ is used as a new estimate for d. This value is plugged into Equation 51, returning a new estimated intersection point $\boldsymbol{p}_2$.

The estimate of the intersection point improves with each iteration as it tends to converge at the true intersection point (Shah et al., 2007). The magnitude of error and the number of iterations to convergence will depend on the terrain topology. A constant value of 5 iterations was used, as suggested in Shah et al. (2007), and found to produce good results.

The full process can be seen in Algorithm 6.

Algorithm 6: Intersection estimation

Input: Initial point of the ray $p0$ and normalized refracted light direction $refractDir$

$p \leftarrow p0$

for $i \leftarrow 0;\ i < intersectIters;\ i \leftarrow i + 1$ **do**

```
// Project p into bathymetry
```

$b \leftarrow \text{texture}(bathymetry, p).r$

$pProj = (p.x, b, p.z)$

```
// Use distance for next approximation
```

$d \leftarrow distance(p0, pProj)$

$p \leftarrow p0 + d \times refractDir$

end

return p

With the intersection point determined, the path length is simply computed as the distance between its position and the initial water surface position.

Caustics intensity

Since the vertices went through refraction, the associated triangles areas may differ between the original/pre-refraction vertices and the post-refraction vertices. For focusing rays, the area decreases, and for dispersive rays it increases (see Figure 33). This difference in areas is used as a way to define a per vertex intensity (Parna, 2020).

The final intensity I_i is determined as a ratio of the sum of the areas of the pre-refraction triangles around vertex p_i and the post-refraction triangles around $R(p_i)$, multiplied by the light intensity at the water surface. This can be seen in Equation 52, where $R(p_i)$ is the post refraction hit point of p_i and $N(p)$ is the set of triangles neighboring a given vertex p. $A(\triangle)$ is the area of a triangle, and $I(\triangle)$ is the light intensity as defined in Equation 53, where $\boldsymbol{n}_\triangle$ is the triangle's normal and $\boldsymbol{l}$ is the normalized light direction. Considering that the normal of the triangle is computed using a cross product, the triangle's area can be given by the length of the normal vector[1].

$$I_i = \frac{\sum_{\triangle_i \in N(p_i)} A(\triangle_i) I(\triangle_i)}{\sum_{\triangle_j \in N(R(p_i))} A(\triangle_j)} \tag{52}$$

$$\begin{aligned} A(\triangle) &= \|\boldsymbol{n}_\triangle\| \\ I(\triangle) &= \text{normalize}(\boldsymbol{n}_\triangle) \cdot \boldsymbol{l} \end{aligned} \tag{53}$$

Following this idea, the steps required to compute the caustics' intensity are:

1. Compute the pre- and post-refraction areas and intensities associated with each triangle.
2. Sum the values associated with each vertex by adding the surrounding triangles' values.
3. Compute the ratio between both values.

Although the values used for the intensity ratio computation are the triangles areas multiplied by the light intensity, they will still be referred to as areas for simplicity's sake.

The regular nature of the water surface grid allows the neighboring triangle set of a vertex to be easily determined. Considering that refraction does not alter vertex connectivity, the neighboring region stays the same both pre- and post-refraction.

All data related to the vertices positions required for the triangle normal calculations is stored in GSM. As the grid spacing is uniform (β_x and β_y), only height needs to be shared for the pre-refraction area computation. For the post-refraction areas the equal grid-spacing is not preserved and as such the areas are computed using the vertices positions.

With the pre- and post-refraction areas computed for all triangles, they are also stored in GSM so they can be accessed for the total neighboring triangle area calculation. The neighborhood of triangles for a vertex $p_{i,j}$ to be summed is illustrated in Figure 35). The final area is defined as in Equation 54, where a_t is the total area

1 The actual area of the triangle would be half of the cross-product, but since only the ratio of areas is of interest, the division is dropped.

and $area_u(i,j)$ and $area_l(i,j)$ are functions that read from GSM the area of the upper and lower associated triangles of vertex (i,j). The use of GSM is further explained below.

$$
\begin{aligned}
a_1 &= area_u(i-1,j) \\
a_2 &= area_l(i-1,j) \\
a_3 &= area_u(i,j) \\
a_5 &= area_l(i-1,j-1) \\
a_5 &= area_u(i,j-1) \\
a_6 &= area_l(i,j-1) \\
a_t &= a_1 + a_2 + a_3 + a_4 + a_5 + a_6
\end{aligned}
\tag{54}
$$

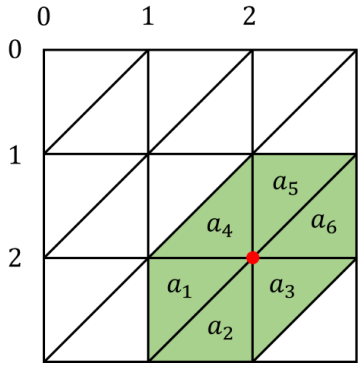

Figure 35: Neighborhood of triangles around vertex (2, 2) on a 4×4 caustics grid.

Simulation overview

An overview of the steps of the caustics simulation is provided in Figure 36. Starting at step (1), the intersection point of the refracted ray with the bottom terrain is estimated. At step (2), the vertex's associated triangles areas and intensities are computed. Step (3) takes the previously computed results of the vertex's triangle neighborhood, and adds them. Finally at step (4), a ratio of the neighborhood sums is computed by dividing the pre-refraction area by the post-refraction area, resulting in the final caustics intensity. Both the hit point position computed at step (1) and the caustics intensity at step (4) are stored in a buffer to be later used to render the caustics map.

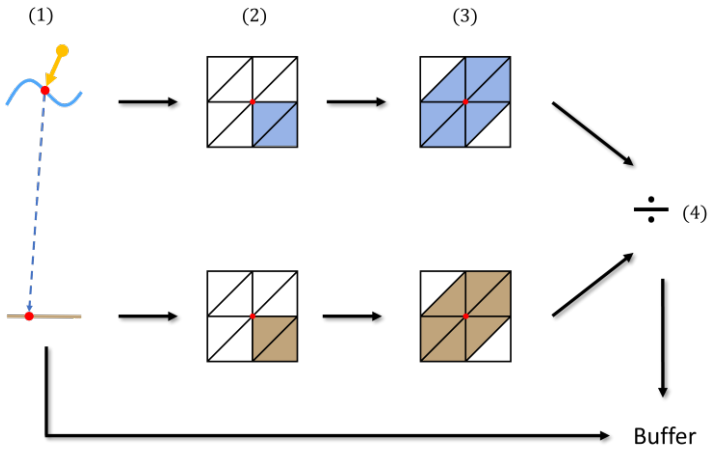

Figure 36: Steps of the caustics simulation performed at each thread.

During this compute shader pass, GSM is used to avoid doing redundant processing and texture accesses. The initial vertex height (1 float) and hit point position (3 floats) are stored in GSM so they can be later accessed for area calculations. Then, pre- and post-refraction triangle areas are also stored in GSM for triangle neighborhood accesses (4 floats in total). As only 4 floats need to be stored at the same time, a single GSM `vec4` array is used.

An overview of the caustics simulation process where the order of execution and use of GSM can be seen is in Algorithm 7. Memory barriers with thread synchronizations are denoted as **barrier** and used between reads

and writes in the GSM to guarantee a correct order of execution. The input resources to this pass are the heights and normals texture, bathymetry texture and normal map.

Algorithm 7: Caustics simulation

```
// Compute and sample initial data
```
Compute global indices (Equation 35).

Sample $height$ and $normal$ from simulation data texture.

Sample and apply normal map to perturb $normal$.

Compute new grid spacing and vertex position $waterPos$ (Equation 49 and Equation 50).

```
// Find hit point
```
$refractDir \leftarrow$ normalize(refract($lightDir, normal, eta$))

Estimate intersection point ($hitPoint$) with receiver using $waterPos$, $refractDir$ and bathymetry texture (Algorithm 6).

$pathLength \leftarrow$ distance($waterPos, hitPoint$)

Store $height$ and $hitPoint$ in GSM.

barrier

```
// Compute associated triangle areas and intensities
```
Read $heights$ and $hitPoints$ of vertices in associated triangles from GSM.

Compute normals for pre- and post-refraction associated triangles using $heights$ and $hitPoints$.

Compute triangle areas and intensities for the required triangles using their normals (Equation 53).

barrier

Store triangle areas in GSM.

barrier

```
// Read and sum values from triangle neighborhood (Equation 54)
```
Read vertex triangle neighborhood intensities from GSM, $preAreas$ and $posAreas$ for pre- and post-refraction areas respectively.

$totalPre \leftarrow \sum_{x \in preAreas} x$

$totalPos \leftarrow \sum_{x \in posAreas} x$

$intensity \leftarrow \frac{totalPre}{totalPos}$

```
// Write results to buffer
```
if *this vertex is in inner local and global domains (Equation 36)* **then**

  Write $intensity$, $pathLength$ and $hitPoint$ x and z components to buffer.

4.3.2 *Rendering caustics map*

The previous pass filled the caustics vertex buffer with the positions and intensity data, which will now be used to render the caustics map texture.

There are two main rendering methods: splatting based (Shah et al., 2007) and triangle based (Ernst et al., 2005). Both methods output a caustics map texture that is then projected onto the caustics receiver. Parna

(2020) compared the methods, suggesting the use of caustics triangle rendering as it provided significantly sharper results while also being slightly faster to render and not requiring additional parameter tuning. As such, it is the method implemented in this work.

The caustics map texture is obtained by rendering the triangles of the caustics grid and using the fragment shader to smoothly interpolate the vertex intensities across the rasterized pixels. Since triangles can end up overlapping or have a reversed winding order due to refraction (Figure 37), the rendering pass is performed with both culling and depth-testing disabled and additive blending enabled.

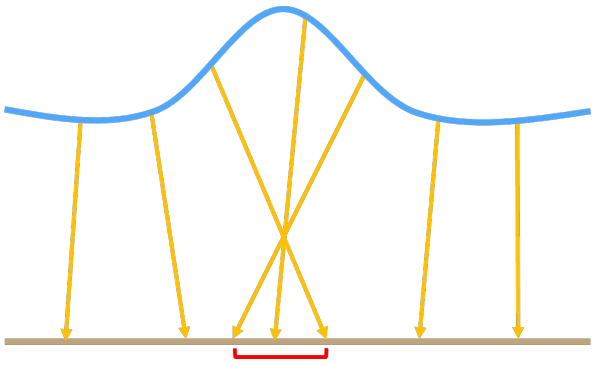

Figure 37: Example scenario that leads to overlapping triangles and reversed winding order (highlighted in red).

The path length requires different blending, so an additional texture is rendered in the same way, but using the path length as the color and minimum blending instead of additive.

Parna (2020) rendered the caustics map from camera's view which required an additional pass to render the depth so it can be later used to avoid incorrect caustics when the viewing ray crosses the caustics mesh more than once. As in this work only heightfields are to be rendered underwater, a top-down camera was chosen instead, therefore guaranteeing that the mesh is intersected only once and avoiding the previously described problem. Additionally, only the x and z coordinates of the hit point are needed in the caustics buffer and the projection of the caustics map to the receiver is trivial.

4.3.3 *Applying the caustics map*

When rendering the water surface, the intersection point between the viewing ray and the bottom terrain is computed (Section 4.2.2). The x and z coordinates of this point are converted to texture space and used to access the caustics map and path length textures. The intensity modifier sampled from the caustics map texture is then multiplied by the refracted color which is used in the remainder color computations. The path length replaces the approximate path length when computing the transmittance as described in the previous section.

4.3.4 *Algorithm overview*

An high level overview of the rendering pipeline is provided in Figure 38. Since the algorithm runs entirely on the GPU, it is presented in terms of render passes performed.

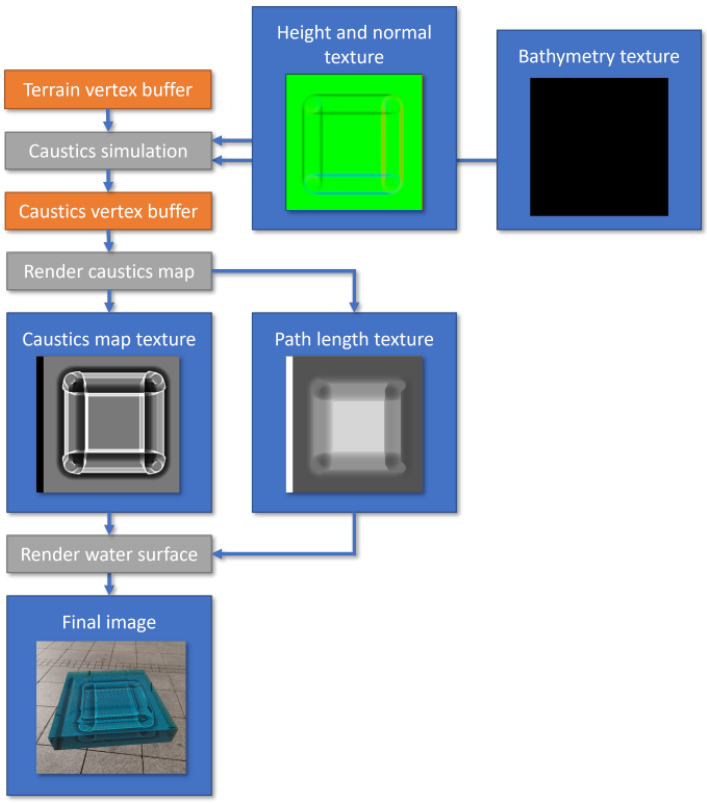

Figure 38: Overview of the caustics algorithm. Render passes are represented by gray boxes, and the resources used and produced by these passes are represented as orange (buffers) and blue (textures/images) boxes. The texture's colors are scaled for better visualization.

4.3.5 *Results*

Some example scenarios with caustics are provided in Figure 39 and Figure 40. In both examples the caustics distinctive brighter zones are clearly visible due to the convergence of light rays.

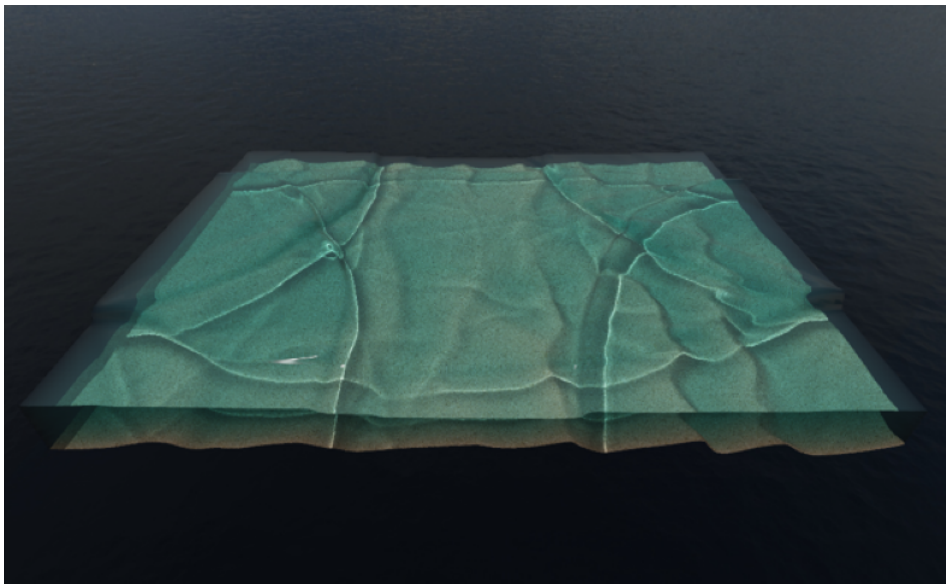

Figure 39: Caustics created by multiple waves on the water surface.

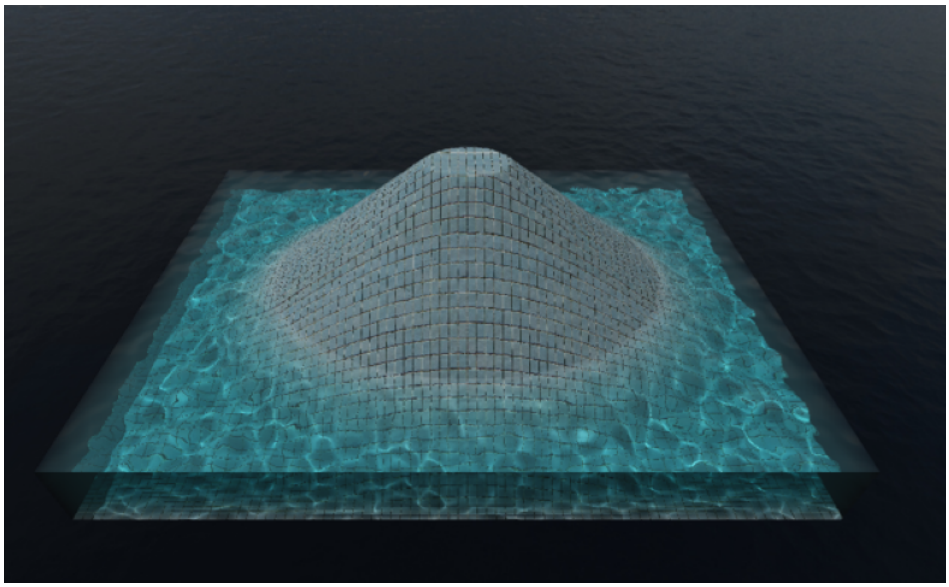

Figure 40: Caustics created by a normal map (Section 4.4) applied on a flat water surface pierced by the bottom terrain.

A limitation of the method used is that only light incoming from the water surface is considered, as if the scene was contained inside a box, which produces shadows near the borders (Figure 41). This could be fixed by simply extending the caustics generating grid to outside the domain of the simulation as necessary.

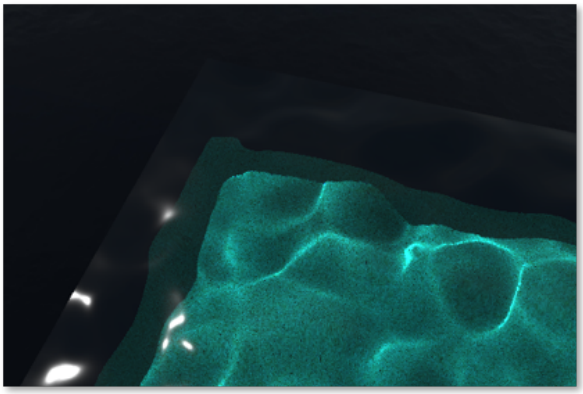

Figure 41: Shadowed borders produced by the caustics algorithm, since it only considers incoming light from the water surface.

4.4 SMALL SCALE DETAILS

A procedurally generated normal map is used to provide higher frequency details for surface rendering. Procedural noise functions are widely used in Computer Graphics (Lagae et al., 2010) and for a diverse range of purposes including varied natural scenery, water being one of them. This section explores the use of Perlin and cellular noise as a way of adding detail to the water surface and the caustics it produces.

Although only a 2D texture is needed to generate the normal map, the 3D version of the noise functions are used to animate the texture. By fixing two coordinates and moving through the other one (simulating time), as if a plane was slicing a 3D volume, a smooth animation can be achieved.

4.4.1 *Perlin Noise*

Perlin noise (Perlin, 2002) is probably the most famous noise algorithm and is commonly used to generate patterns and shapes found in nature. It is an implementation of gradient noise, which consists in the creation of a lattice of random gradients at integer locations, interpolated to obtain values between the lattices. The version used here is Perlin simplex noise, an improved version of classic Perlin noise (Perlin, 2002), but will be referred to simply as Perlin noise. The GLSL implementation presented in Gustavson and McEwan (2022) is used for the noise generating function. The normal map is obtained from the analytical derivatives of the noise function.

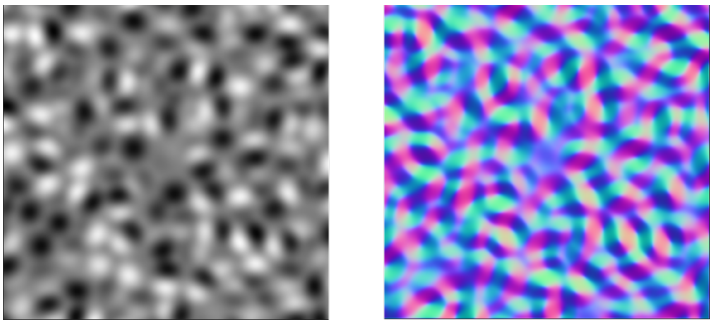

Figure 42: Perlin noise, height map values on the left, normal map on the right.

4.4.2 *Cellular Noise*

Cellular noise, also known as Worley noise (Worley, 1996), was explored with the goal of producing more convincing and diversified caustics patterns. Cellular Noise is based on distance fields, more specifically, of the distance to the closest point in a set of feature points. The space is subdivided into tiles, each one containing a feature point in a random position. At each pixel, the distance between the point in their own tile and the points in the surrounding tiles is checked and the shorter distance is stored. The result is a distance field as in Figure 43. The open source GLSL implementation in Gustavson (2021) is used for the noise generating function. The normal map is computed from the noise in an additional pass, using a finite difference approximation of the derivatives.

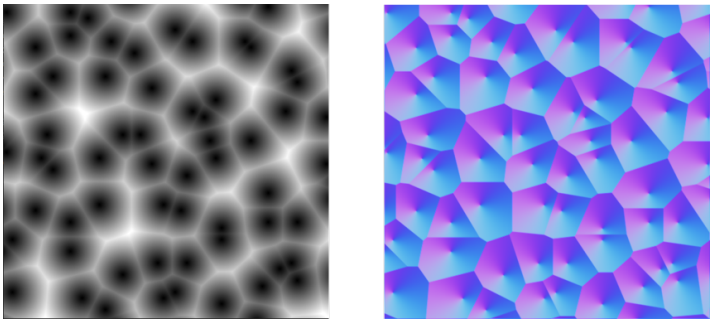

Figure 43: Cellular noise, height map values on the left, normal map on the right.

4.4.3 *Fractal Brownian Motion*

To obtain a finer granularity in the noise and get more fine detail, a technique called fractal Brownian motion (fBm) can be used (Vivo and Lowe, 2021), also referred to simply as fractal noise. This consists in adding different iterations of noise (octaves), where the frequencies are successively incremented in regular steps (lacunarity) and the amplitude decreased (gain). The derivatives/gradient are accumulated in the same way as the noise,

added and multiplied by the amplitude at each iteration. A fractal noise function is provided in Algorithm 8, and example generated textures in Figure 44.

Algorithm 8: fBm function

Input: 2D coordinates p

$frequency \leftarrow 1$

$amplitude \leftarrow 1$

$total \leftarrow 0$

for $i \leftarrow 0$; $i < octaves$; $i \leftarrow i + 1$ **do**

  $total \leftarrow total + amplitude \times noise(frequency \times p)$

  $frequency \leftarrow frequency \times lacunarity$

  $amplitude \leftarrow amplitude \times gain$

end

return $total$

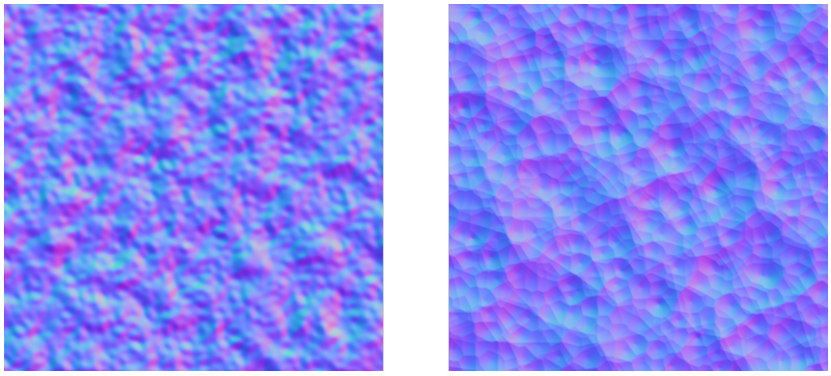

Figure 44: Perlin (left) and cellular (right) 3 octaves fractal noise normal maps.

4.4.4 *Domain Warping*

With cellular noise, the edges of the cells create sharp peaks and lead to the typical cell shaped caustics. However, the cell's straight lines give it a unnatural appearance which needs further tweaking. Domain warping can be used to add some distortion to the generated noise. With a image defined as a function of space, as is the case of the noise function used so far, warping simply means distorting the domain with another function, i.e, replacing $f(p)$ with $f(g(p))$. As the goal is only to slightly distort the image, the domain warping function $g(p)$ can be defined as a small arbitrary distortion $h(p)$, or in other words, $g(p) = p + h(p)$. The final noise function will therefore be computed as $f(p + h(p))$, where $h(p)$ can be another noise function (example of use in Algorithm 9).

The final noise is generated using a cellular noise fractal sum and an additional Perlin noise fractal sum used for domain warping. Examples of generated noise using this configuration are presented in Figure 45.

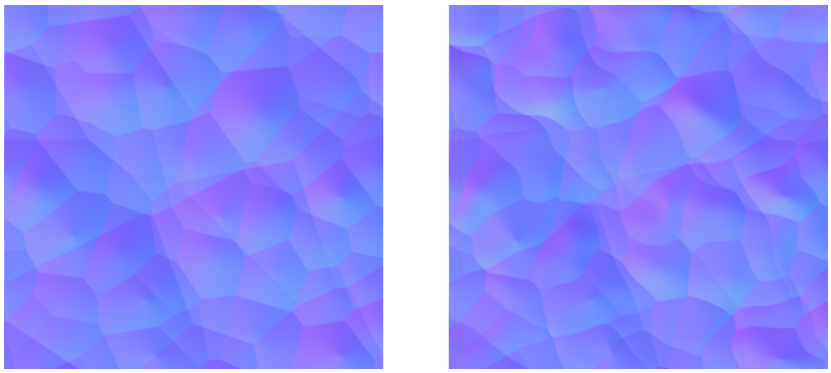

Figure 45: Cellular fractal noise normal maps, on the right with domain warping, and on the left without.

Algorithm 9: Noise function using fractal sums and domain warping

Input: 2D coordinates p

$q = fBm(p)$

return $fBm(p + 0.1 \times (q, q))$

4.4.5 *Using the normal map*

When rendering the water surface, the normal map is applied as usual by distorting the water surface normals. When the generated normal map is accessed it can also optionally be stretched and slid over the surface over time to simulate small wind waves.

Examples of scenes with noise generated normal maps can be seen in Figure 46, Figure 47 and Figure 48. The first example has clear water and clearly visible caustics patterns. In this case only a small number of octaves are used, but domain warping is required in order to distort the straight lines of cellular noise. The second example has turbid water where the bottom terrain barely contributes to the water's color and the sun's specular highlights are visible. Here 4 octaves are used as the finer detail results in the distinct sparkling look created by the water's specular reflections. The final example shows small directional waves, resulting from stretching the normal map in one direction. In this example the normal map would also be slid over the surface over time, in the direction perpendicular to which it was stretched.

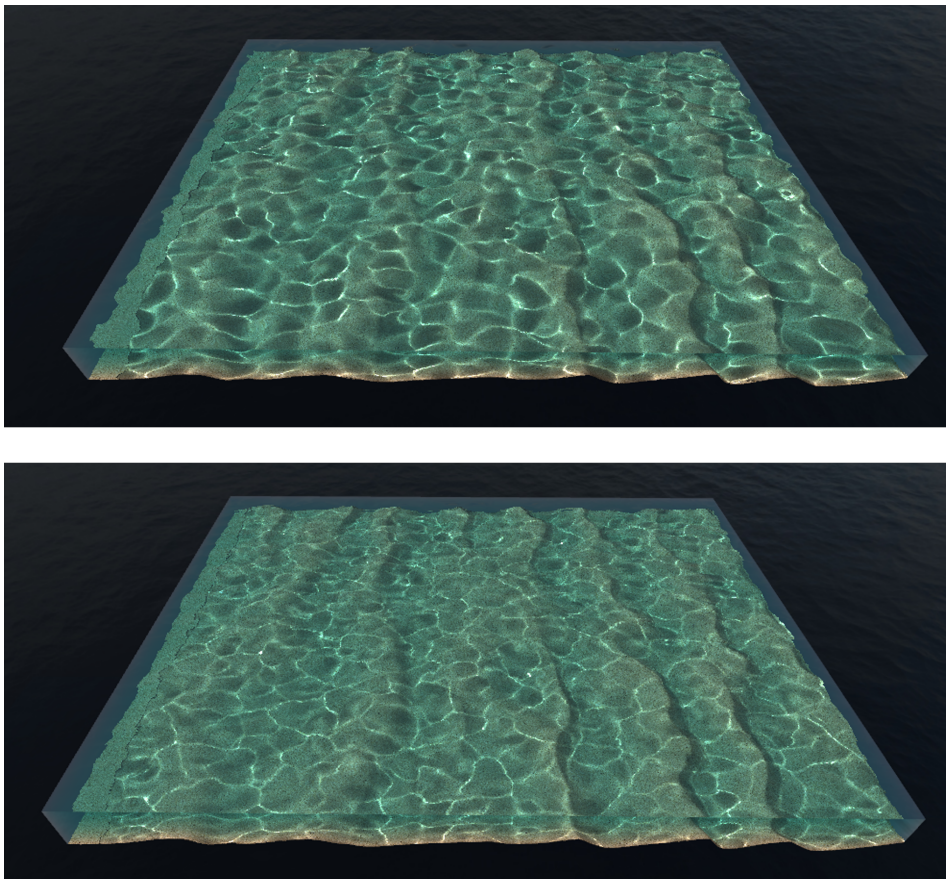

Figure 46: Perlin (top) and cellular (bottom) fractal noise applied as a normal map to clear water. The noise is composed of 2 octaves and the cellular noise has domain warping caused by 2 octaves Perlin fractal noise.

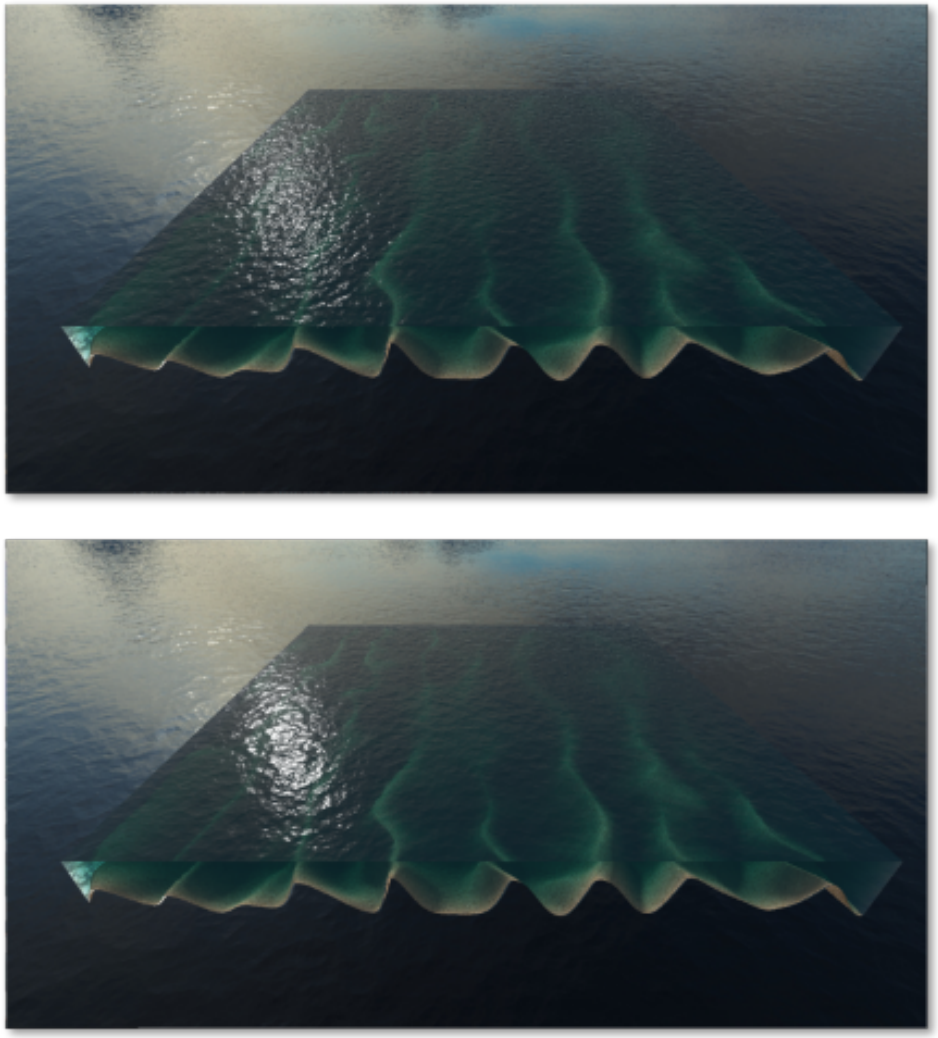

Figure 47: Perlin (top) and cellular (bottom) fractal noise applied as a normal map to turbid water. The noise is composed of 4 octaves. The cellular noise has domain warping caused by 4 octaves Perlin fractal noise.

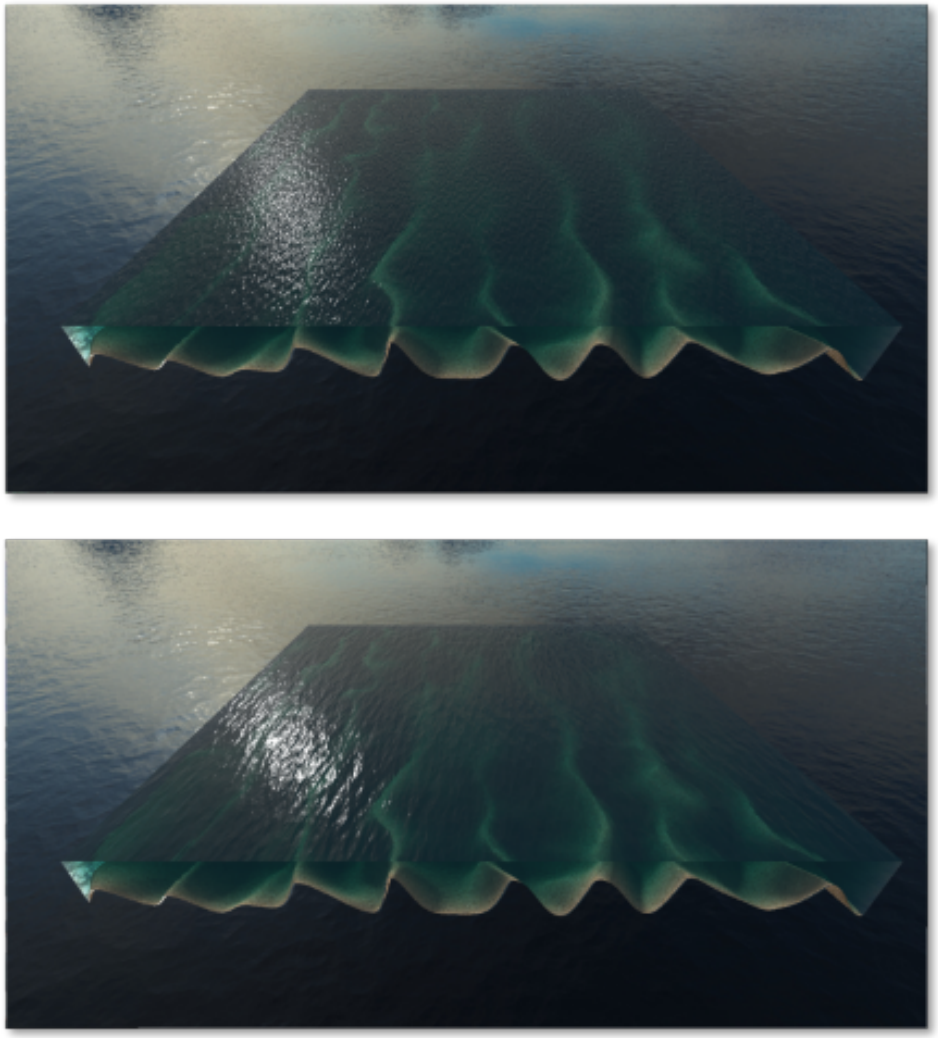

Figure 48: Perlin (top) and cellular (bottom) fractal noise applied as a normal map to turbid water. The noise is composed of 4 octaves and stretched in one dimension. The cellular noise has domain warping caused by 4 octaves Perlin fractal noise.

5

EVALUATION

This chapter discusses the results of the developed framework in various scenarios and provides performance benchmarks.

Initially, an overview of the developed application is provided (Section 5.1), where a brief review of the pipeline is presented, as well as common performance benchmarks for the main components. The following section (Section 5.2) provides a more in-depth review of the solver, analyzing its capabilities for modeling different scenes. The final section (Section 5.3) performs a similar analysis for the various rendering components, exploring different options and providing additional performance benchmarks.

All timings are obtained with OpenGL queries and using a NVIDIA GeForce GTX 1060 with 6 GB of VRAM. The timings are averaged over at least 1000 frames.

5.1 OVERVIEW

This section presents a brief review of the full application and performance benchmarks of its various components.

A diagram of the application pipeline can be seen in Figure 49 where the separate components are highlighted. Additional passes such as for the terrain and skybox were omitted. The solver step is the last step as it updates the state for following frame, with the first frame displaying the initial state. Multiple solver steps may need to be performed in a single frame due to stability restrictions (Section 3.2.1).

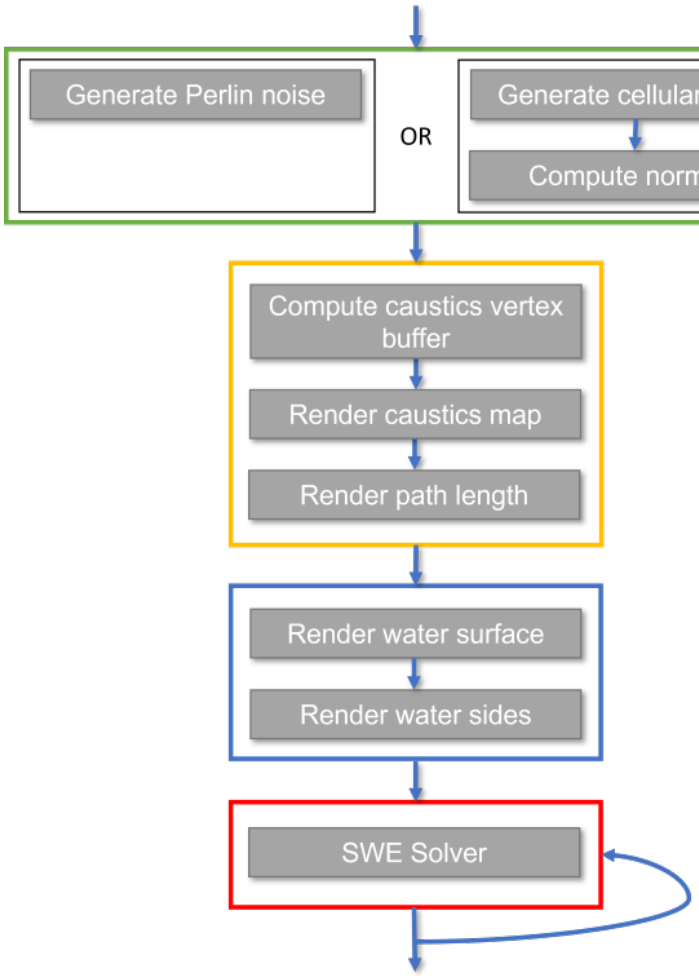

Figure 49: Pipeline of the full application. Each gray box is a pass, and the surrounding boxes highlight the main components. At the noise generating component, the passes performed depend on the type of noise used. The solver pass can be repeated several times if needed due to a limiting timestep.

When rendering a certain simulation grid, the water surface vertex grid is actually slightly smaller due to excluding the boundary cells, e.g., for a simulation grid size of $1024x1024$, the corresponding vertex grid is of size $1016x1016$. However, for simplicity's sake, when referring to grid sizes only the simulation size will be mentioned.

The timings of the full application for various combinations of resolutions and grid sizes are presented in Table 1. Graphs of these timings are presented in Figure 52 and Figure 51, and the scene used for these tests is displayed in Figure 50. A chart with a more detailed breakdown into the various components is presented in Figure 53 for some of the grid size and resolution pairs.

The timings for rendering include the water volume rendering and the previous components that generated textures that are used in it (the first three components in Figure 49). A single step of the solver is performed per frame in all cases, and the cell's size is adjusted so that the physical dimensions of the scene remain the same

for all grid sizes. The caustics and normal map textures have a resolution equal to the grid's size. The normal map is generated using two octaves of cellular noise.

Other parameters that influence the performance of the application, such as the work group size in a compute shader, are set as the ones that yield the best performance, with the analysis regarding them being presented further ahead (Section 5.2.3, Section 5.3.2).

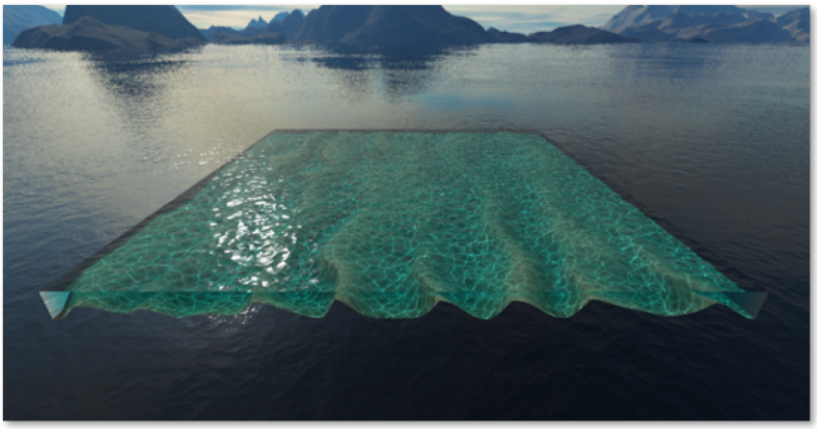

Figure 50: Example scene used to obtain the timings displayed in Table 1.

Resolution \ Grid size	256 x 256	512 x 512	1024 x 1024	2048 x 2048
480 x 270	0.09; 0.73; 0.82	0.33; 1.68; 2.01	1.33; 4.23; 5.56	5.42; 13.89; 19.31
960 x 540	0.09; 1.32; 1.41	0.33; 2.96; 3.29	1.33; 6.30; 7.63	5.42; 16.50; 21.92
1920 x 1080	0.09; 2.00; 2.09	0.33; 4.82; 5.15	1.33; 11.80; 13.13	5.42; 27.22; 32.64

Table 1: Frame time breakdown of the full application for different combinations of grid sizes and resolutions. Each cell of the table has 3 timings which from left to right correspond to: simulation, rendering and total. Timings are in milliseconds.

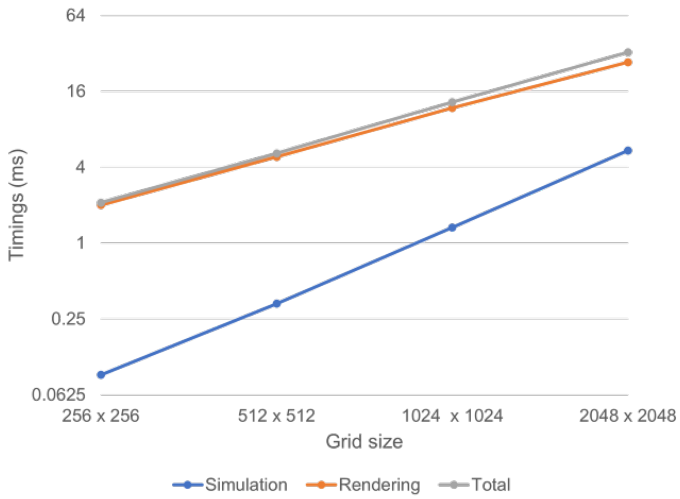

Figure 51: Frame time breakdown of the full application for different grid sizes, and resolution 1920x1080. Both axes are scaled logarithmically. Values in Table 1.

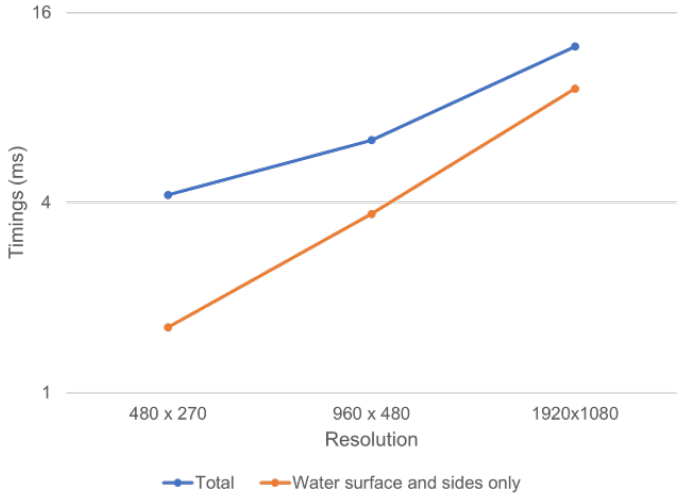

Figure 52: Rendering time for different resolutions, and grid size 1024x1024. The time is also displayed without the caustics and noise passes, that is, only the water surface and sides drawing passes. Both axes are scaled logarithmically. Values in Table 1.

As seen on Figure 51, frame time scales linearly with the grid size. The rendering time also scales linearly with the resolution when considering only the surface and and sides rendering passes (Figure 52), since in these examples the caustics and noise texture sizes were defined based on the grid size, and therefore are equal for all resolutions.

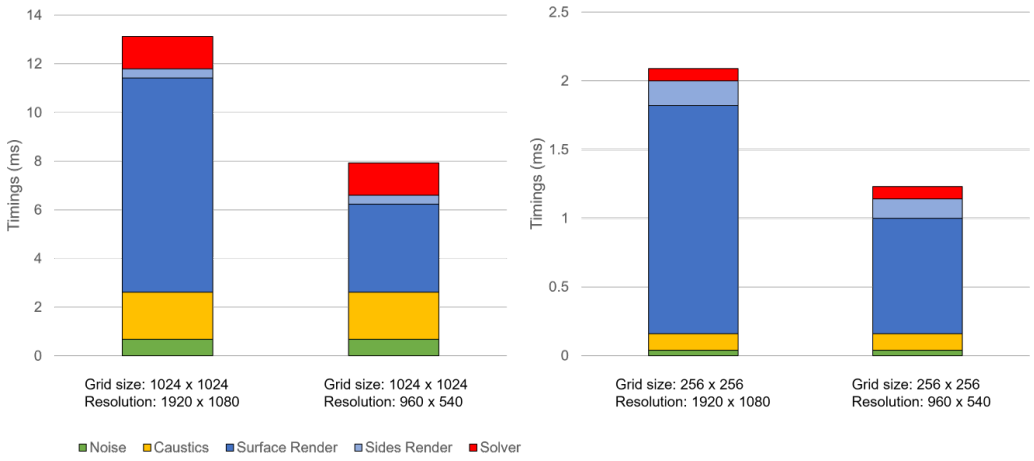

Figure 53: Frame breakdown with all components for some combinations of resolution and grid size. Values in Table 1.

Rendering takes the bulk of computation time, this difference being accentuated when a small grid size is used with a larger resolution, e.g, in Figure 53 comparing the two middle examples. However, the examples given had only one solver step per frame, while due to stability restrictions more can be required, an option that is reviewed at Section 5.2.1. From within the different rendering components, drawing the water surface takes the most time. Even when the texture resolution used for caustics and noise is larger than the final rendering resolution, rendering the water surface takes more than half of the rendering time (second example from the right in Figure 53), with noise generation taking the least.

5.2 SHALLOW WATER EQUATIONS SOLVER

This section presents a review of the SWE solver for different scenarios. Initially, the impact of the grid and cells size is evaluated for different scenarios, providing a visual showcase of the water behavior modeled by the solver. Then, different possible work group configurations for the compute shader are evaluated. Finally, the option of adjusting the solver for fully water covered scenes is analyzed.

5.2.1 *Grid and cell size comparison*

The simulation has two main parameters that define the physical domain it models, the grid size and the cell size. While keeping the total physical dimension constant, increasing the grid size and reducing the cell size will increase the simulations detail level, while doing the opposite decreases the detail level. For a scene with the same physical dimensions and initial conditions, several combinations of grid and cell sizes are tested with rendered results shown in Figure 54.

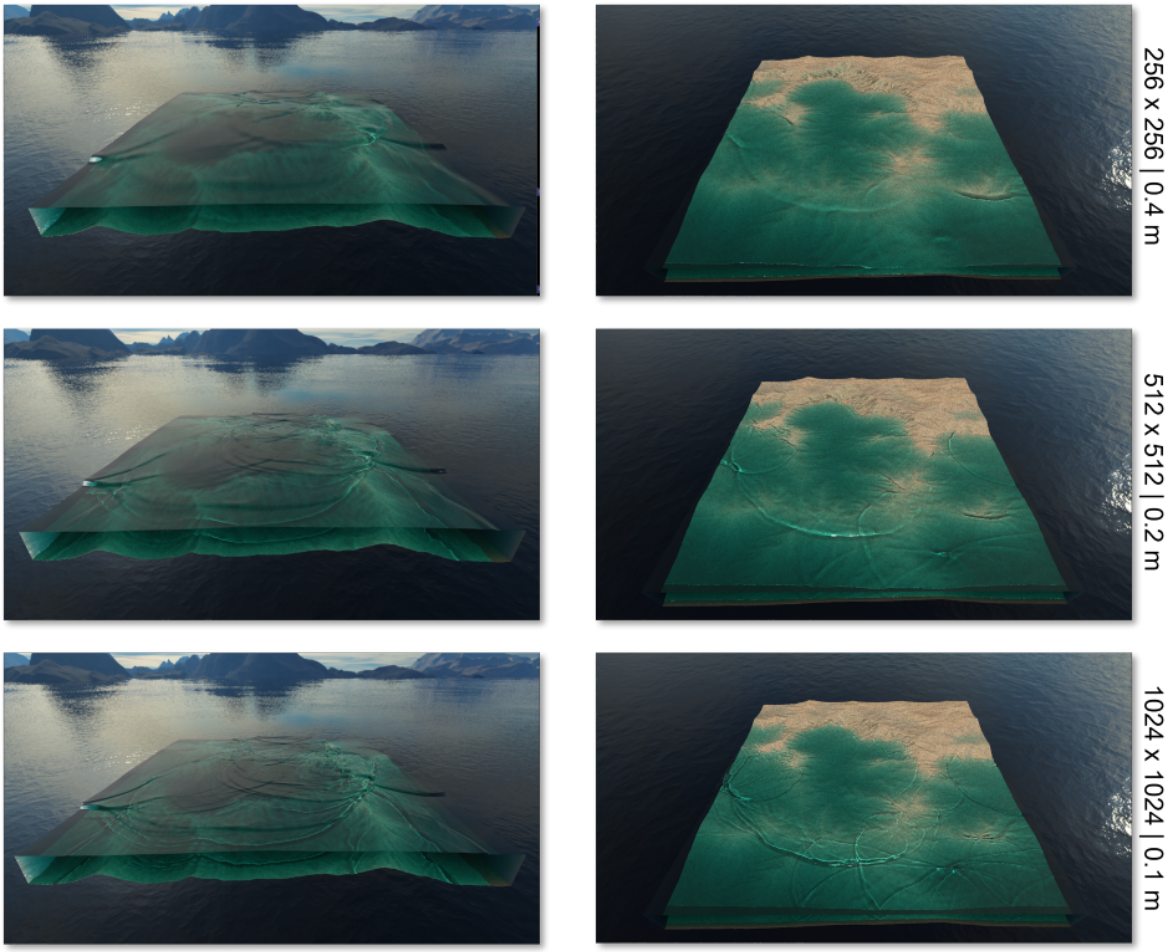

Figure 54: Scenes with the same physical dimensions ($\approx 200m$) but different combinations of grid and cell sizes. From top to bottom the scenes have increasingly smaller cells and larger grids.

As seen in Figure 54, lowering the cell size and increasing the grid size accordingly leads to a more detailed simulation, with sharper features and more complex waves and caustics patterns.

As all cells are treated the same in the solver, and as confirmed during testing, the state of the scene (e.g., the wet to dry cell ratio) has no impact in the performance of the solver. That leaves the size of the simulated grid as the performance defining parameter. Increasing the detail level implies increasing the grid size, which will in consequence affect performance.

However, due to stability requirements (Section 3.2.1), more than one solver step can be required per frame since there is a limit on the timestep size. Stability will mostly depend on the velocity of the water flow, and on the dimension of the cells relative to that velocity: the smaller the cell size, or the higher the velocities are, the smaller the maximum timestep allowed will be. So for a certain grid size, reducing the cell size might require additional solver steps in order to ensure stability.

For a grid size of $1024x1024$, the same scenario was tested with different cell sizes, and consequently, different total physical domains. The timings for the tested scenes as well as the maximum timesteps (determined

as to ensure stability in the simulation) and number of solver steps are provided in Table 2. The maximum timestep (Max $Delta$ t) was determined through trial and error as the lowest value that would allow for a stable simulation.

Cell size (m)	0.4 x 0.4	0.2 x 0.2	0.1 x 0.1
Max Δ t (ms)	20	10	5
Solver steps per frame	1	2	4
Render time (ms)	9.09	10.59	11.95
Solver time (ms)	1.25	2.49	4.98
Total (ms)	10.34	13.98	16.93

Table 2: Data about scenes with grid size $1024x1024$, resolution $1920x1080$ and varying cell size.

As seen in Table 2, decreasing the cell's size to a quarter resulted in requiring to halve the maximum timestep in order to ensure stability. This leads to more solver steps being needed per frame, increasing its contribution to frame time in relation to rendering.

Similarly, increasing the velocities in the scene, for example by increasing the water depth which causes the waves to travel faster, will lead to more strict stability requirements, in the form of a lower maximum timestep. Two scenarios with the same parameters but different water depths were tested, with the results shown in Table 3. The scenes tested are depicted in Figure 55.

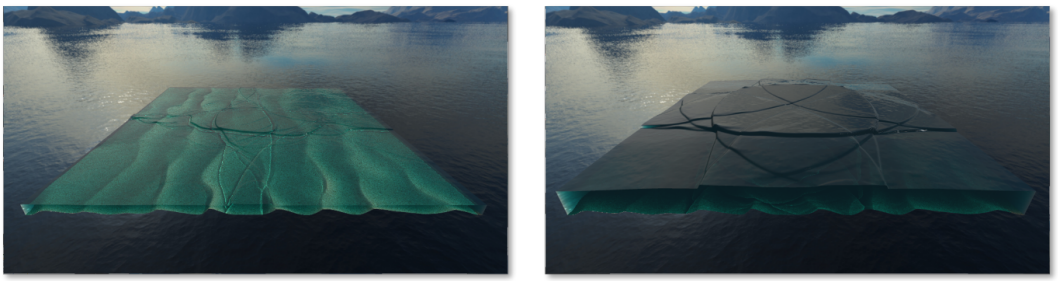

Figure 55: Scenes with different starting conditions but all other parameters equal. The scene on the right (B) has a higher average depth which results in faster waves.

Scene	A	B
Max Δ t (ms)	10	7
Solver steps per frame	2	3
Render time (ms)	10.76	11.31
Solver time (ms)	2.49	3.75
Total (ms)	13.25	15.06

Table 3: Data about the scenes depicted in Figure 55, with grid size $1024x1024$, resolution $1920x1080$ and cell size $0.2x0.2$.

As seen in Table 3, a lower maximum timestep required to ensure stability will lead to a larger number of solver steps per frame.

Although depending on several different parameters, the simulation proved viable (when aiming for around 60 frames per second, or 0.017 ms per frame) for the examples provided for a resolution of $1920x1080$, grid of size $1024x1024$ and cell size $0.2x0.2m$, which accounts for a total physical domain of $\approx 200m$ (Table 2, Table 3). Covering larger domains can be achieved by increasing the cells' size, having the drawback of lower detail. For scenes with higher rendering or stability requirements, other parameters would have to be adjusted accordingly, such as lowering the grid resolution or increasing the cells' size.

5.2.2 *Limitations*

A limitation of the heightfield representation is that when creating a larger/faster wave, if the wave were to break, an unnatural wall-like wave is formed instead. An example of this is shown in Figure 56.

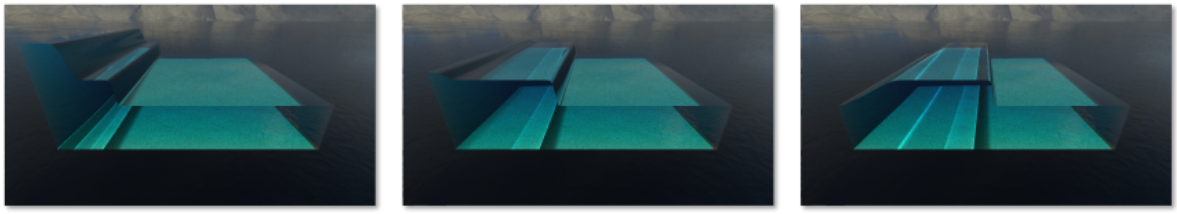

Figure 56: Several frames of a wave in a simulation, advancing from left to right. Shallow waters can not model breaking waves, so instead a unnatural wall-like wave can be formed.

5.2.3 *Computation work group sizes*

As there are various different conflicting optimization parameters, which make finding the optimal work group size a difficult task (Brodtkorb et al., 2012), a set of different possible configurations is tested. Still, there are some aspects that can be considered. Firstly, there are some restrictions set by OpenGL, namely, for each work group there is a maximum number of threads of 1536, and a maximum size of the shared memory of 49152 Bytes.

The compute kernel uses 10 floats per thread, therefore requiring 40 bytes in total. Considering the limit of $49152B$, a maximum number of threads per work group of $49152/40 \approx 1229$ would be expected. However, when compiling the shader, the shared memory limit was reached at a lower value of 928 threads, possibly due to how OpenGL internally handles the memory.

Another aspect to consider is that having a bigger and square work group maximizes the ratio of internal valid cells to boundary cells (see Figure 14). Finally, since the warp size is 32, it is a common practice to keep the number of threads to an integer multiple of it (Brodtkorb et al., 2012). During testing, only the configurations with either 16 or 32 threads on the x dimension produced correct results, possibly due to synchronization issues. Due to the shared memory limit, that makes the maximum work group sizes to be $32x29$ or $16x58$.

Considering these guidelines and restrictions, a variety of possible thread configurations were tested. A selection of the computation times of these configurations are shown in Table 4 and Figure 57. More configurations than those shown were tested, which followed a similar distribution to the one presented in Figure 57. Smaller groups than those shown had increasingly worse results.

Threads (XxY)	16x16	16x20	16x24	16x28	16x32	16x36	16x40	16x44	16x48
Timings (ms)	0.54	0.44	0.40	0.42	0.38	0.38	0.37	0.46	0.46
Threads (XxY)	32x12	32x14	32x16	32x18	32x20	32x22	32x24	32x26	32x28
Timings (ms)	052	0.45	0.39	0.36	0.33	0.42	0.41	0.38	0.38

Table 4: Performance of the SWE solver with different work group thread configurations. Simulation grid size is $512x512$.

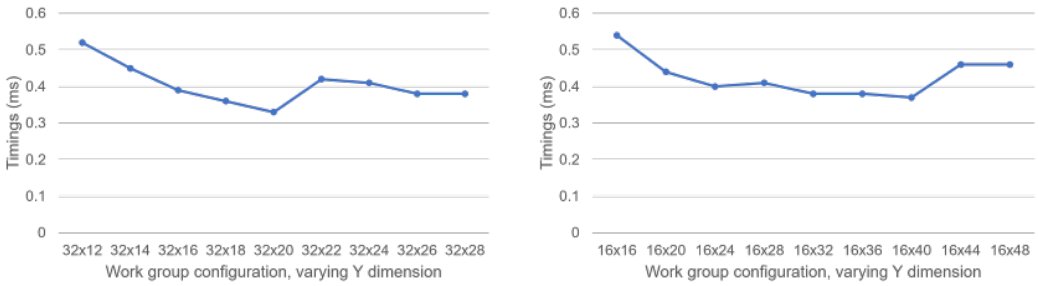

Figure 57: Performance of the SWE solver with different work group thread configurations. Values in Table 4.

As expected, the smallest configurations had the worst results, probably due to the low ratio of inner to boundary cells, which then requires more groups to be dispatched to cover the whole domain. However, increasing the size of the group only yields better results up until a certain point, with the largest groups having worse results. The best configuration can be seen as a "dip" in both charts (Figure 57), which corresponds to a size of $640 = 32x20 = 16x40$ threads. Within this size, the configuration of $32x20$ had the best results, so it is used for all other tests.

5.2.4 *Wet only solver*

Since the solver has specific computations to handle wetting/drying processes (Section 3.1.4), an alternative version of the solver without these computation was also tested, which will be referred to as the wet only solver. This solver could be used in scenes where it is guaranteed that every cell is covered in water, as a way to improve performance. The computation times for both solvers are presented in Table 5 for varying grid sizes.

Grid size	128 x 128	256 x 256	512 x 512	1024 x 1024	2048 x 2048
Standard	0.03	0.09	0.33	1.33	5.42
Wet only	0.03	0.11	0.26	1.26	5.29

Table 5: Performance of the SWE solver with different grid sizes and solvers, timings in milliseconds. The wet only solver has no computations for handling wetting/drying processes.

The wet only solver provides a slight increase in performance which can justify its use in scenes that allow it. Still, since the solver usually amounts for a small part of total computation time, it may only be noticeable in scenes that rely on multiple solver passes.

5.3 WATER RENDERING

This section presents a more in-depth review of some parameters and options of the rendering components, as well as additional performance benchmarks.

5.3.1 *Terrain intersection*

As part of the water color computations, when determining the intersection of the refracted/reflected viewing ray with the terrain, a heightfield intersection algorithm is used. This algorithm is tested with bathymetries of varying steepness and its computation times are shown in Table 6. The scenes tested and heatmaps of the number of iterations required for intersection are provided in Figure 58.

Bathymetry type	A	B	C
Timings (ms)	2.95	2.30	1.99

Table 6: Rendering timings for different bathymetries of varied steepness (shown in Figure 58). Only the water surface and sides passes are considered. Grid size is $512x512$ and resolution is $960x540$.

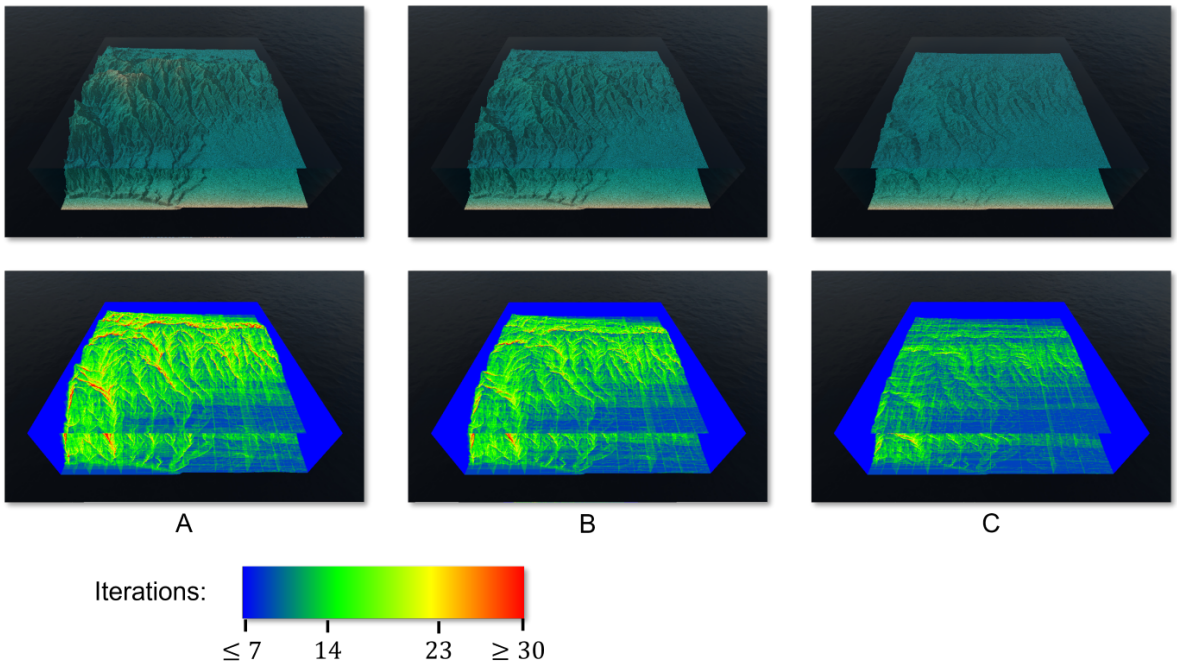

Figure 58: Scenes with different bathymetries of decreasing steepness from left to right. The bottom row depicts a heatmap of iterations of the intersection algorithm, corresponding to the scenes in the top row. Timings in Table 6.

As seen in Table 6 and Figure 58, the number of iterations required o intersect the terrain depends on its steepness, with steeper terrain requiring more iterations. This is due to the high variation in heights of the steep terrain causing the ray to descend levels too quickly, which then requires more iterations to advance through the lower levels (as seen in the red hotspots in Figure 58). On the other hand, on flatter terrain, the ray is more likely to only descend levels when actually getting near the intersection point.

5.3.2 *Caustics*

The caustics generation step (yellow outline in Figure 49) consists of three passes, the first one being a compute shader that creates a vertex buffer filled with the caustics data. Then, there are two rendering passes that utilize the generated vertex buffer to create textures, one drawing the caustics intensity values, and the other drawing the incident light path length.

Performance results for the caustics map generation with varying grid sizes are given in Table 7 and Figure 59, and with varying resolutions in Table 8. Both rendering passes have similar performance so a single value is displayed for both. The total includes the three passes, namely the initial compute pass and the following two rendering passes.

Caustics grid size	256 x 256	512 x 512	1024 x 1024	2048 x 2048
Computer vertex buffer	0.07	0.16	0.52	1.97
Render textures	0.26	0.38	1.44	6.80
Total	0.33	0.54	1.96	8.77

Table 7: Performance of the various caustics simulation passes with different caustics grid sizes. Caustics textures resolution is $1024x1024$ and timings are in milliseconds. The total accounts for a complete step, which includes the compute pass and two render passes.

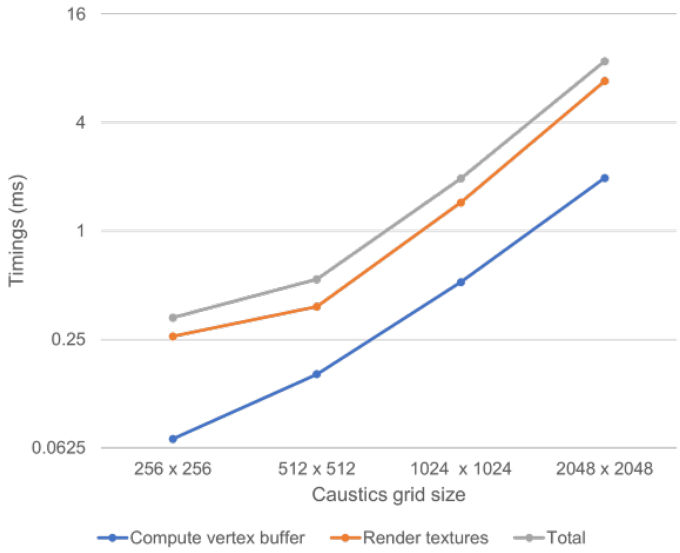

Figure 59: Performance of the various caustics simulation passes with different caustics grid sizes. Both axes are scaled logarithmically. Values in Table 7. The total accounts for a complete step, which includes the compute pass and two render passes.

Grid size \ Resolution	256 x 256	512 x 512	1024 x 1024	2048 x 2048	4096 x 4096
256 x 256	0.05	0.05	0.08	0.28	0.98
512 x 512	0.20	0.18	0.18	0.52	1.17
1024 x 1024	0.88	0.86	0.72	0.76	1.47
2048 x 2048	3.58	3.54	3.40	2.90	3.03

Table 8: Performance of a caustics rendering pass with different resolutions. The grid size refers to the caustics grid and the resolution to the generated caustics textures.

As shown in Table 7 and Figure 59, the bulk of the computation time is spent drawing the caustics textures and not computing the vertex buffer that they are drawn from. Since computing the path length for the attenuation is only useful in certain scenes, it can be omitted to improve performance. Reducing the resolution to below the

grid size resulted in similar or worse timings, as shown in Table 8, so this should be avoided (considerably worse results are also produced, as later seen in Section 5.3.2).

Caustics work group size

Similarly to the SWE solver (Section 5.2.3), the caustics simulation is tested with various work group sizes. Since the compute kernel uses only 4 floats per thread (16 bytes) of shared memory, a maximum number of threads per work group of $49152/16 \approx 3072$ would be expected to reach the limit. The upper bound for work group size is instead set by the maximum number of threads, 1536.

A selection of computation times of this pass with some different possible work group configurations are presented in Table 9 and Figure 60, with a caustics grid of size $1024x1024$.

Threads (XxY)	4x4	8x8	12x12	16x16	20x20	24x24	28x28	32x32	36x36
Timings (ms)	1.90	0.93	0.52	0.72	0.52	0.77	0.58	0.87	0.62

Table 9: Performance of the caustics simulation pass with different work group thread configurations. Caustics grid size is $1024x1024$.

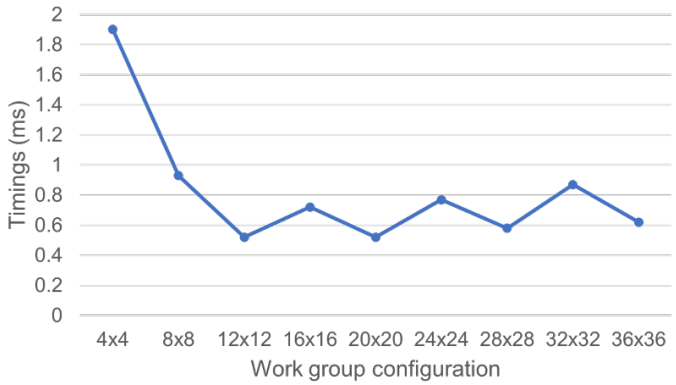

Figure 60: Performance of the caustics vertex buffer compute pass with different work group thread configurations. Values in Table 9.

Similarly to the results obtained in Section 5.2.3, a trend can be seen where the smallest sized groups performed the worst, but increasing the size will only provide improved results up until a certain point, where a "dip" in the chart can be seen (Figure 60). The major difference here is that further increasing the group's size results in an oscillating increase in the timings rather than a steady one. Looking at the previously mentioned "dip", one of the work group configurations that yielded the best results is $12x12$, and therefore, was used for all other tests.

Caustics grid size and resolution comparison

Example of scenes rendered with caustics of varying grid sizes and texture resolutions are presented in Figure 61. As can be seen, increasing the caustics grid size results in better defined and sharper caustics. This, however, requires a matching increased resolution, with the best results obtained when the resolution is greater than the grid size. This supports the previous conclusion that using a lower resolution than the caustics grid size is detrimental (Section 5.3.2).

The caustics grid size is not coupled to the simulation's grid size and therefore can be increased for added detail. Since increasing the whole simulations grid size has a larger impact on performance than just increasing the caustics grid size, the latter option could be used as an intermediate way of increasing visual quality.

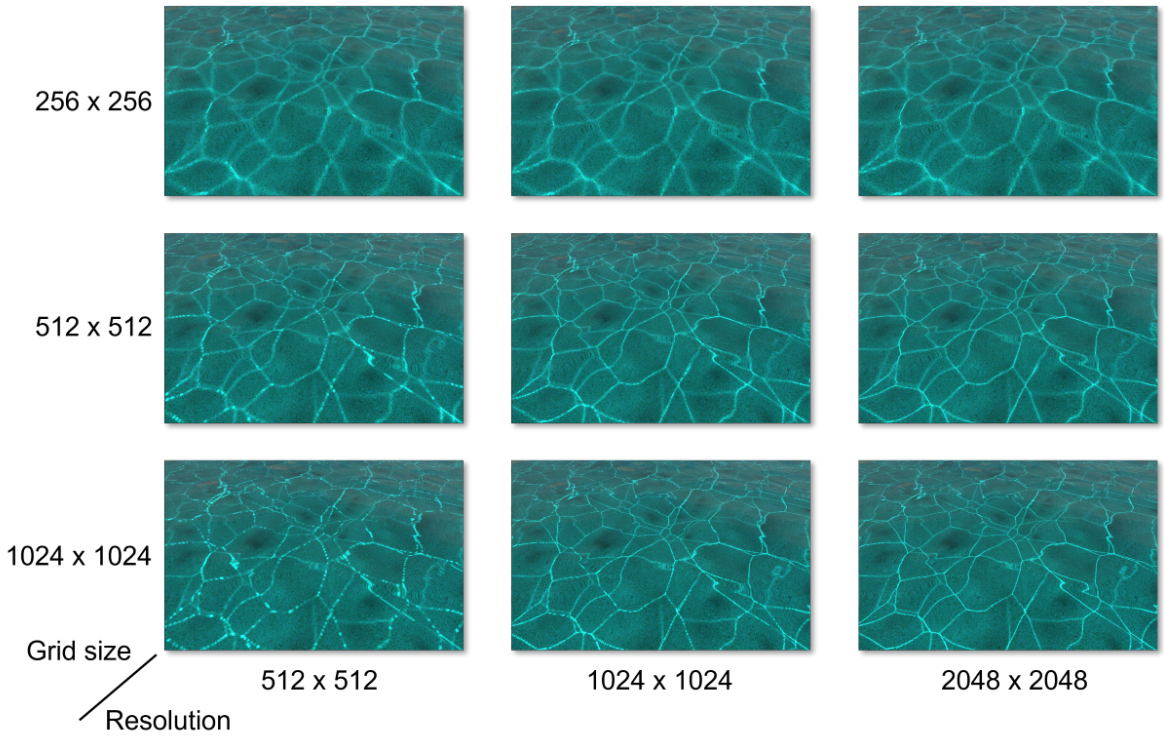

Figure 61: Close-ups of different caustics obtained by varying the grid size and texture resolution. Patterns created by applying a cellular noise normal map to the surface.

5.3.3 *Small scale details*

The generation of the normal map for small scale details (green outline in Figure 49) requires either one or two passes depending on the type of noise used. Perlin requires only one pass as the analytical derivatives/normals can be computed directly, while cellular noise requires an additional pass to compute the normals from the noise values.

The computation times of the normal map generation passes are presented in Table 10 and Figure 62. The timings for cellular noise include both the noise generation pass, and the additional one for computing the normals. Cellular noise timings are provided both with and without warping, with the first case requiring the computation of an additional Perlin noise value. The results are for 1 octave of noise for both noise types, including the one used for domain warping.

Texture Resolution	256 x 256	512 x 512	1024 x 1024	2048 x 2048
Perlin	0.02	0.07	0.30	1.22
Cellular w/o warping	0.03	0.14	0.37	1.53
Cellular	0.06	0.18	0.51	2.12

Table 10: Performance of the details normal map generation with different resolutions and noise types (one octave), timings in milliseconds.

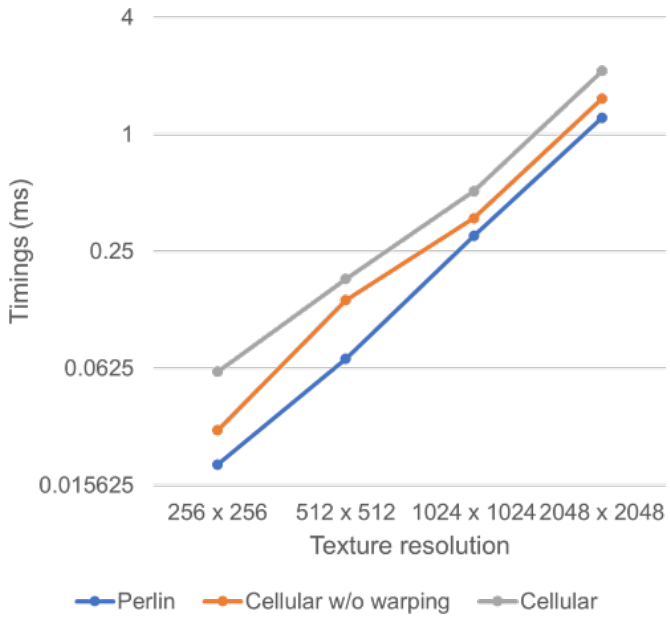

Figure 62: Performance of the details normal map generation with different resolutions and noise types (one octave). Both axes are scaled logarithmically. Values in Table 10.

While cellular noise is slightly more expensive than Perlin noise, both cases are fairly fast, having a minor contribution to the total frame time (Figure 53).

Noise octaves comparison

While there are several parameters that can be tweaked for the noise, such as the frequency and amplitude, these have no impact in the performance and can be changed as necessary. The number of octaves, however, does have an impact in performance and is also one of the main characteristics of the noise produced. Computation

times for different octave counts are provided in Table 11 and Figure 63. The timings for cellular noise include the generation of the Perlin noise used for warping (with the number of octaves matching the one of the cellular noise) and include the additional pass required to compute the normals.

Octaves	1	2	3	4	5	6
Perlin	0.07	0.13	0.18	0.24	0.30	0.37
Cellular	0.18	0.24	0.34	0.42	0.52	0.62

Table 11: Performance of the details normal map generation with different noise types and octaves ($512x512$ resolution). The cellular noise includes warping and an additional pass to compute the normals. Timings in milliseconds.

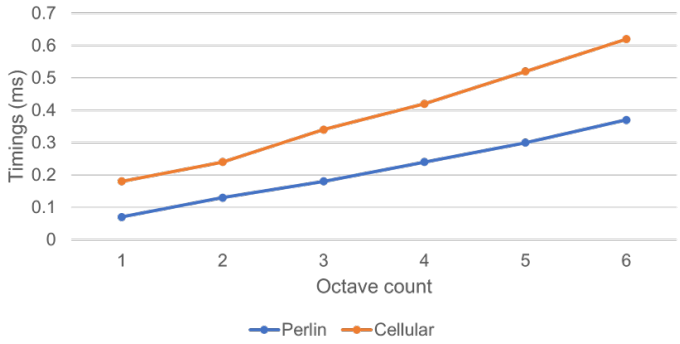

Figure 63: Performance of the details normal map generation with different noise types and octaves ($512x512$ resolution). Values in Table 11.

Examples of scenes rendered using the generated noise with varying octave counts (Table 11) are shown in Figure 64, Figure 65 and Figure 66. When considering water surface detail (Figure 64, Figure 65), adding more octaves increases the detail, which is especially noticeable in the Sun's specular glitter. However, the impact of the added octaves is also increasingly smaller, being barely noticeable at above 4 octaves. When aiming only for generating caustics patterns, lower octave counts are enough (Figure 66). Both noise types can produce the intended level of detail, with cellular noise producing more diverse caustics patterns.

As mentioned in Section 5.3.3 and confirmed by the values in Table 11, for the desired resolutions and octave counts the noise generating step should have a minimal contribution to total frame time.

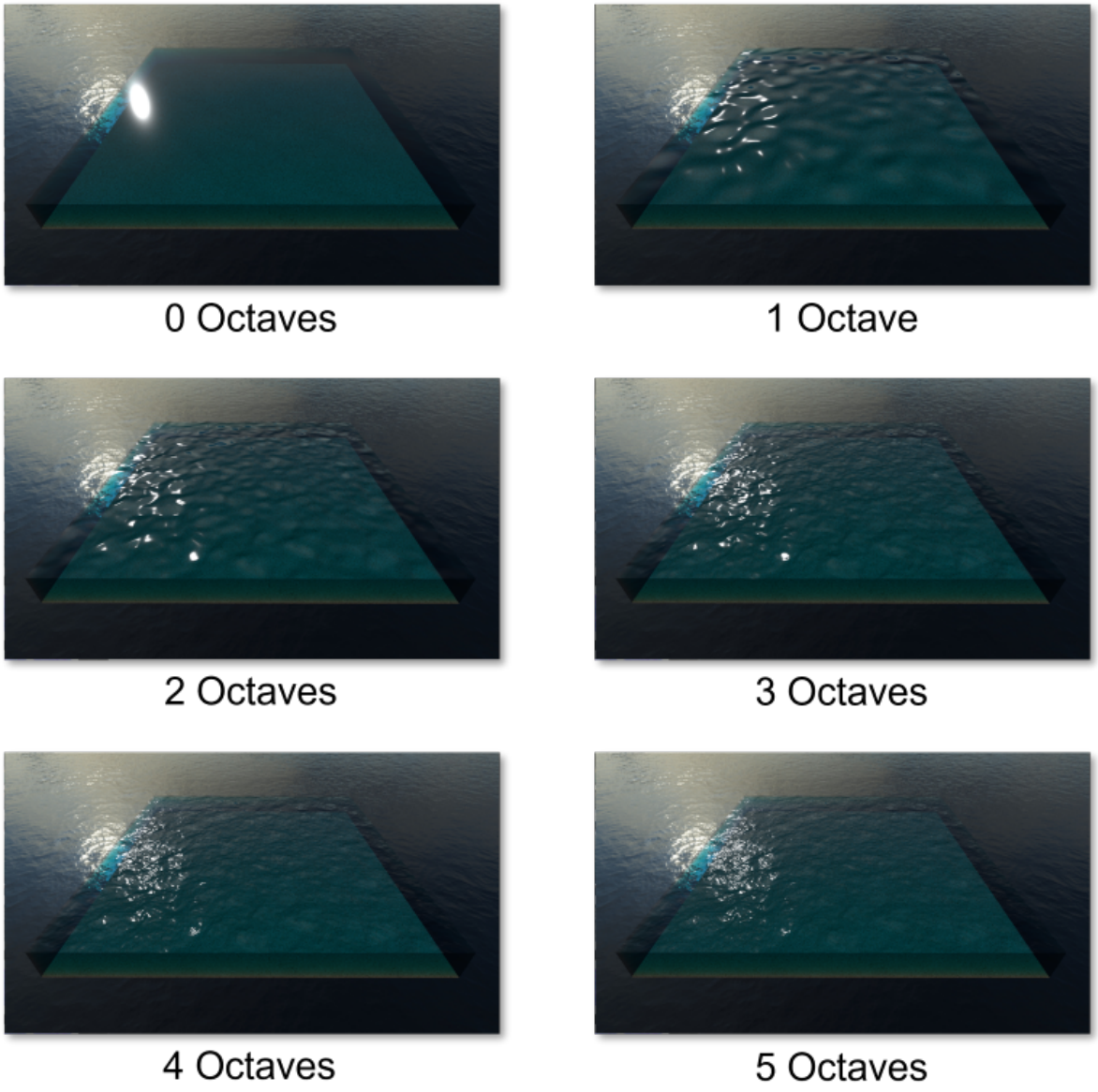

Figure 64: Different surface detail levels obtained by varying the number of octaves of Perlin noise used to generate the normal map.

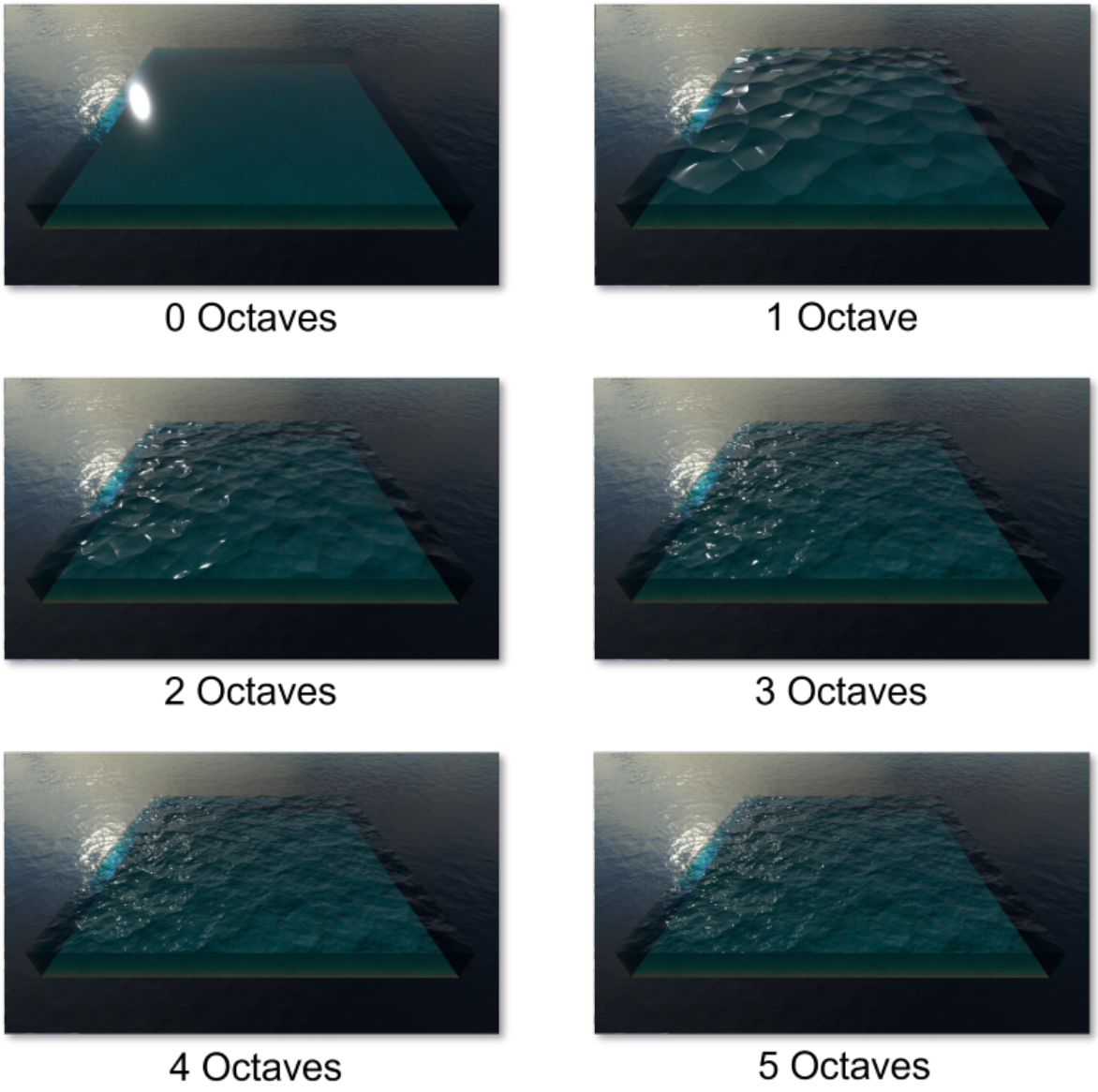

Figure 65: Different surface detail levels obtained by varying the number of octaves of cellular noise used to generate the normal map.

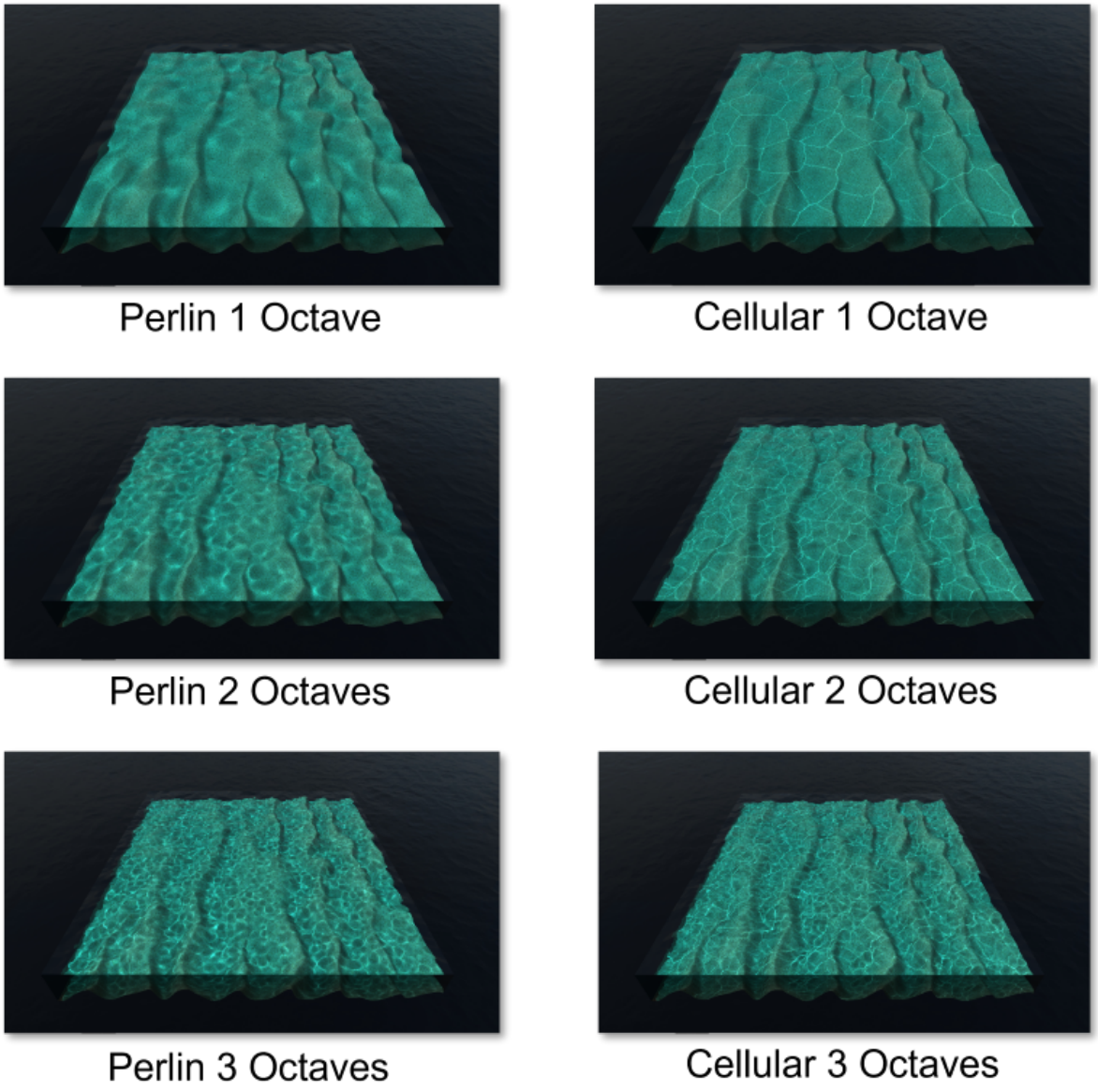

Figure 66: Different caustics patterns obtained by varying the number of octaves and type of noise used to generate the normal map.

6

CONCLUSIONS AND FUTURE WORK

The domain of application for fluid simulation is vast, and there are numerous specializations based on the requirements. When the goal is water simulation on real-time applications, certain aspects have to be ignored in order to meet the performance constraints. Shallow water simulations are a commonly used approximation for applications with real-time requirements as they allow for both faster simulation and rendering.

A framework for water simulation in real-time using the SWE and a heightfield surface representation was presented in this work. The water's behavior was simulated by using an Eulerian SWE solver which computes a grid of values at each frame in an efficient GPU implementation. The grid generated by the simulation is then used to render a mesh of the water surface. The color of the water surface is determined by a combination of reflected and transmitted light, using a physically based BRDF to describe the Sun's reflection. The terrain's contribution is obtained by intersecting a reflected and a refracted ray. Transmitted light also takes into consideration the water's attenuation and simplified scattering. This result is then enhanced by a separate pass which generates caustics and improves the attenuation computations. Finally, small-scale details were added to the surface as a normal map to account for the limit in resolution and scope of the simulation, as well as aiding in creating interesting caustics patterns. The developed application was tested and performance benchmarks as well as insights into some parameters and options were provided.

The resulting simulation accurately captures the intended shallow water wave behaviors and interaction with terrain, and its rendering provides the essential identifiable features of a water surface. The full system was shown to be able to achieve a desirable 60 frames per second on consumer hardware.

6.1 FUTURE WORK

There are several areas where the developed work can be improved, either by addressing the existing limitations, adding new functionality or simply improving on the various components. A few options are proposed here.

Starting with the inherent limitations of shallow water simulations, since the surface is represented by an heightfield, behavior such as breaking waves, splashing and waterfalls can not be modeled. To address this, the system could be combined with a particle system (e.g., Ojeda and Susín (2014)) or volumetric techniques (e.g., Parna (2020)). Similarly, in order to support a larger area and deep ocean regions where the water is no longer "shallow", the system could be combined with a procedural simulation.

Another approach could be to add rigid-body coupling, to add a higher degree of interactivity with the scene.

When considering the simulation, since dry regions are treated just as wet regions (Section 5.2.1), a clear option would be to have a sparse simulation where dry regions are not computed. Another option would be to explore other methods of simulation, some mentioned in Section 2.4, such as LBM and 2D SPH, and do a comparison in the context of computer graphics as this appears to be lacking recent data.

As for rendering, since hardware supported ray-tracing is getting increasingly better and widespread, it would make sense to explore a ray-tracing solution for reflections, refractions and caustics generation. One more option would be to explore underwater visualization, which also pairs well with an improvement of the scattering computations, including the addition of volumetric caustics.

BIBLIOGRAPHY

Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018. ISBN 0134997832.

R. Angst, N. Thürey, M. Botsch, and M. Gross. Robust and efficient wave simulations on deforming meshes. *Computer Graphics Forum*, 27, 2008.

D. Arumuga Perumal and Anoop K. Dass. A review on the development of lattice boltzmann computation of macro fluid flows and heat transfer. *Alexandria Engineering Journal*, 54(4):955–971, 2015. ISSN 1110-0168. doi: https://doi.org/10.1016/j.aej.2015.07.015. URL http://www.sciencedirect.com/science/article/pii/S1110016815001362.

Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, 1994. ISSN 0021-9991. doi: https://doi.org/10.1006/jcph.1994.1159. URL http://www.sciencedirect.com/science/article/pii/S0021999184711594.

Robert Bridson. *Fluid simulation for computer graphics*. A K Peters/CRC Press, 2nd edition, 2015. ISBN 9781351968843.

André R. Brodtkorb, Martin L. Sætra, and Mustafa Altinakar. Efficient shallow water simulations on gpus: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55:1–12, 2012. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2011.10.012.

Eric Bruneton, Fabrice Neyret, Nicolas Holzschuch, Eric Bruneton, Fabrice Neyret, Nicolas Holzschuch, Real time Realistic, Ocean Lighting, Hal Id Inria, Eric Bruneton, Fabrice Neyret, and Nicolas Holzschuch. Realtime realistic ocean lighting using seamless transitions from geometry to brdf. *Comput. Graph. Forum*, pages 487–496, 2010.

Brent Burley. Physically based shading at disney. In *ACM SIGGRAPH 2012 Courses*, 2012.

J. C. Butcher. *The Numerical Analysis of Ordinary Differential Equations: Runge-Kutta and General Linear Methods*. Wiley-Interscience, USA, 1987. ISBN 0471910465.

Nuttapong Chentanez and Matthias Mueller. Real-time Simulation of Large Bodies of Water with Small Scale Details. In MZoran Popovic and Miguel Otaduy, editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. The Eurographics Association, 2010. ISBN 978-3-905674-27-9. doi: 10.2312/SCA/SCA10/197-206.

Wang Chi-Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. Technical report, Institute for Computer Applications in Science and Engineering (ICASE), 1997.

Michal Chladek and Roman Durikovic. Particle-based shallow water simulation for irregular and sparse simulation domains. *Computers & Graphics*, 53:170–176, 2015. ISSN 0097-8493. doi: https://doi.org/10.1016/j.cag.2015.04.002.

Andrew J. Christlieb, Yaman Güçlü, and David C. Seal. The picard integral formulation of weighted essentially nonoscillatory schemes. *SIAM Journal on Numerical Analysis*, 53(4):1833–1856, 2015. doi: 10.1137/140959936.

R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, jan 1982. ISSN 0730-0301. doi: 10.1145/357290.357293.

R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1):32–74, Dec 1928. ISSN 1432-1807. doi: 10.1007/BF01448839. URL https://doi.org/10.1007/BF01448839.

Keenan Crane, Ignacio Llamas, and Sarah Tariq. *Real Time Simulation and Rendering of 3D Fluids*, chapter 30. Addison-Wesley, 2007.

I. Cravero and M. Semplice. On the accuracy of weno and cweno reconstructions of third order on nonuniform meshes. *Journal of Scientific Computing*, 67(3):1219–1246, Jun 2016. ISSN 1573-7691. doi: 10.1007/s10915-015-0123-3. URL https://doi.org/10.1007/s10915-015-0123-3.

F. Dagenais, J. Guzmán, V. Vervondel, A. Hay, S. Delorme, D. Mould, and E. Paquette. Real-time virtual pipes simulation and modeling for small-scale shallow water. In *Proceedings of the 14th Workshop on Virtual Reality Interactions and Physical Simulations*, VRIPHYS '18, page 45–54, Goslar, DEU, 2018. Eurographics Association.

Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU Ray-Casting for Scalable Terrain Rendering. In D. Ebert and J. Krüger, editors, *Eurographics 2009 - Areas Papers*. The Eurographics Association, 2009. doi: 10.2312/ega.20091007.

Michał Drobot. *Quadtree Displacement Mapping with Height Blending*, chapter 1. Taylor & Francis, 2018.

B. Düz, M.J.A. Borsboom, A.E.P. Veldman, P.R. Wellens, and R.H.M. Huijsmans. An absorbing boundary condition for free surface water waves. *Computers & Fluids*, 156:562–578, 2017. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2017.05.018. URL https://www.sciencedirect.com/science/article/pii/S0045793017301871. Ninth International Conference on Computational Fluid Dynamics (ICCFD9).

Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In *Proceedings of Graphics Interface 2005*, GI '05, page 87–96, Waterloo, CAN, 2005. Canadian Human-Computer Communications Society. ISBN 1568812655.

Nick Foster and Dimitris Metaxas. Realistic animation of liquids. *CVGIP: Graphical Model and Image Processing*, 58:471–483, 01 1996.

Makoto Fujisawa, Takuya Nakada, and Masahiko Mikawa. Particle-based shallow water simulation with splashes and breaking waves. *Journal of Information Processing*, 25:486–493, 07 2017. doi: 10.2197/ipsjjip.25.486.

Cristian García Bauza, Juan D'Amato, Gustavo Boroni, Marcelo Vénere, and Alejandro Clausse. Real-time interactive animations of liquid surfaces with lattice-boltzmann engines. *Australian Journal of Basic and Applied Sciences*, 08 2010.

Holger Gruen. *Ray-Guided Volumetric Water Caustics in Single Scattering Media with DXR*, pages 183–201. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4427-2. doi: 10.1007/978-1-4842-4427-2_14. URL https://doi.org/10.1007/978-1-4842-4427-2_14.

Branislav Grujic and Cristian Cutocheras. Water Rendering in FarCry 5. https://www.gdcvault.com/play/1025555/Advanced-Graphics-Techniques-Tutorial-Water, 2018. [Online; accessed 28-June-2022].

Juan Guardado and Daniel Sanchez-Crespo. *Rendering water caustics*, chapter 2. Addison-Wesley, 2007.

Stefan Gustavson. Noise for glsl 1.20. https://github.com/stegu/webgl-noise, 2021. [Online; accessed 28-June-2022].

Stefan Gustavson and Ian McEwan. Tiling simplex noise and flow noise in two and three dimensions. *Journal of Computer Graphics Techniques (JCGT)*, 11(1):17–33, February 2022. ISSN 2331-7418. URL http://jcgt.org/published/0011/01/02/.

Romain Guy and Mathias Agopian. Physically based rendering in filament. https://www.gamedeveloper.com/programming/deep-water-animation-and-rendering, 2019.

Eric Heitz. Understanding the masking-shadowing function in microfacet-based brdfs. *Journal of Computer Graphics Techniques (JCGT)*, 3(2):48–107, June 2014. ISSN 2331-7418. URL http://jcgt.org/published/0003/02/03/.

Eric Heitz. Sampling the ggx distribution of visible normals. *Journal of Computer Graphics Techniques (JCGT)*, 7(4):1–13, November 2018. ISSN 2331-7418. URL http://jcgt.org/published/0007/04/01/.

Naty Hoffman. Background: Physics and math of shading, 2013.

Zsolt Horváth, Andreas Buttinger-Kreuzhuber, Artem Konev, Daniel Cornel, Jürgen Komma, Günter Blöschl, Sebastian Noelle, and Jürgen Waser. Comparison of fast shallow-water schemes on real-world floods. *Journal of Hydraulic Engineering*, 146(1):05019005, 2020. doi: 10.1061/(ASCE)HY.1943-7900.0001657.

Wei Hu and Kaihuai Qin. Interactive approximate rendering of reflections, refractions, and caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):46–57, 2007. doi: 10.1109/TVCG.2007.14.

Stefan Jeschke and Chris Wojtan. Water wave packets. *ACM Trans. Graph.*, 36(4), jul 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073678. URL https://doi.org/10.1145/3072959.3073678.

Stefan Jeschke, Tomáš Skřivan, Matthias Müller-Fischer, Nuttapong Chentanez, Miles Macklin, and Chris Wojtan. Water surface wavelets. *ACM Trans. Graph.*, 37(4), jul 2018. ISSN 0730-0301. doi: 10.1145/3197517.3201336. URL https://doi.org/10.1145/3197517.3201336.

Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. *SIGGRAPH Comput. Graph.*, 24(4):49–57, September 1990. ISSN 0097-8930. doi: 10.1145/97880.97884. URL https://doi.org/10.1145/97880.97884.

Timo Kellomäki. *Large-Scale Water Simulation in Games*. PhD thesis, Tampere University of Technology, 2015.

Timo Kellomäki. Fast water simulation methods for games. *Computers in Entertainment*, 16(1), 2017. doi: 10.1145/2700533. URL https://doi.org/10.1145/2700533.

Timo Kellomäki and Timo Saari. A User Study: Is the Advection Step in Shallow Water Equations Really Necessary? In Eric Galin and Michael Wand, editors, *Eurographics 2014 - Short Papers*. The Eurographics Association, 2014. doi: 10.2312/egsh.20141010.

Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. Smoothed particle hydrodynamics for physically-based simulation of fluids and solids. In *EUROGRAPHICS 2019 Tutorials*. Eurographics Association, 2019.

Alexander Kurganov and Guergana Petrova. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Communications in Mathematical Sciences*, 5(1):133–160, 2007. doi: cms/1175797625.

Frédéric Kuznik, Christian Obrecht, Gilles Rusaouen, and Jean-Jacques Roux. Lbm based flow simulation using gpu computing processor. *Computers & Mathematics with Applications*, 59(7):2380–2392, 2010. ISSN 0898-1221. doi: https://doi.org/10.1016/j.camwa.2009.08.052. URL https://www.sciencedirect.com/science/article/pii/S0898122109006361. Mesoscopic Methods in Engineering and Science.

A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D.S. Ebert, J.P. Lewis, K. Perlin, and M. Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8):2579–2600, 2010. doi: https://doi.org/10.1111/j.1467-8659.2010.01827.x.

Anita Layton and Michiel Panne. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*, 18:41–53, 02 2002. doi: 10.1007/s003710100131.

Hyokwang Lee and Soonhung Han. Solving the shallow water equations using 2d sph particles for interactive applications. *The Visual Computer*, 26:865–872, 06 2010. doi: 10.1007/s00371-010-0439-9.

Richard Lee and Carol O'Sullivan. A fast and compact solver for the shallow water equations. In *Vriphys: Workshop on Virtual Reality Interactions and Physical Simulations*, pages 51–57, Postfach 8043, 38621 Goslar, Germany, 2007. The Eurographics Association;. ISBN 978-3-905673-65-4. doi: 10.2312/PE/vriphys/vriphys07/051-057.

Randall J LeVeque et al. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002. ISBN 9780511791253.

Chao Liang and Zhengfu Xu. Parametrized maximum principle preserving flux limiters for high order schemes solving multi-dimensional scalar hyperbolic conservation laws. *Journal of Scientific Computing*, 58(1):41–60, Jan 2014. ISSN 1573-7691. doi: 10.1007/s10915-013-9724-x. URL https://doi.org/10.1007/s10915-013-9724-x.

Gabor Liktor and Carsten Dachsbacher. Real-time volumetric caustics with projected light beams. In *Fifth Hungarian conference on computer graphics and geometry*, 2010.

Gábor Liktor and Carsten Dachsbacher. Real-time volume caustics with adaptive beam tracing. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, page 47–54, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305655. doi: 10.1145/1944745.1944753. URL https://doi.org/10.1145/1944745.1944753.

Chunyong Ma, Shu Xu, Hongsong Wang, Fenglin Tian, and Ge Chen. A real-time photo-realistic rendering algorithm of ocean color based on bio-optical model. *Journal of Ocean University of China*, 15(6):996–1006, Dec 2016. ISSN 1993-5021. doi: 10.1007/s11802-016-3037-2. URL https://doi.org/10.1007/s11802-016-3037-2.

Miles Macklin and Matthias Müller. Position based fluids. *ACM Trans. Graph.*, 32(4), jul 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461984. URL https://doi.org/10.1145/2461912.2461984.

N Max and B Becker. Flow visualization using moving textures. In *ICASE/LaRC Symposium on Visualizing Time Vary Data*, 1995.

Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 154–159, Goslar, DEU, 2003. Eurographics Association. ISBN 1581136595.

Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics: Class notes. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

ISBN 9781450378451. doi: 10.1145/1401132.1401245. URL https://doi.org/10.1145/1401132.1401245.

Octavio Navarro-Hinojosa, Sergio Ruiz-Loza, and Moises Alencastre-Miranda. Physically based visual simulation of the lattice boltzmann method on the gpu: a survey. *The Journal of Supercomputing*, 74, 07 2018. doi: 10.1007/s11227-018-2392-8.

Fabrice Neyret. Advected Textures. In D. Breen and M. Lin, editors, *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 147–153, San diego, United States, July 2003. Eurographics Association. URL https://hal.inria.fr/inria-00537472.

J.F. O'Brien and J.K. Hodgins. Dynamic simulation of splashing fluids. In *Proceedings Computer Animation'95*, pages 198–205, 1995. doi: 10.1109/CA.1995.393532.

J. Ojeda and A. Susín. Enhanced lattice boltzmann shallow waters for real-time fluid simulations. In *Eurographics*, 2013.

Jesus Ojeda and Antonio Susín. Real-time lattice boltzmann shallow waters method for breaking wave simulations. In Sebastiano Battiato, Sabine Coquillart, Robert S. Laramee, Andreas Kerren, and José Braz, editors, *Computer Vision, Imaging and Computer Graphics – Theory and Applications*, pages 3–18, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44911-0.

Jesús Ojeda Contreras. *Efficient algorithms for the realistic simulation of fluids*. PhD thesis, Universitat Politècnica de Catalunya, 2013.

Charilaos Papadopoulos and Georgios Papaioannou. Realistic real-time underwater caustics and godrays. *19th International Conference on Computer Graphics and Vision, GraphiCon'2009 - Conference Proceedings*, 08 2010.

P. Parna, K. Meyer, and R. Falconer. Gpu driven finite difference weno scheme for real time solution of the shallow water equations. *Computers & Fluids*, 161:107–120, 2018. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2017.11.012.

Peeter Parna. *Shallow water equations in real-time computer graphics*. PhD thesis, Abertay University, 2020.

Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566636. URL https://doi.org/10.1145/566654.566636.

M. Pharr and G. Humphreys. Physically based rendering: from theory to implementation. https://www.pbr-book.org/3ed-2018/contents, 2018.

R. Salmon. The lattice boltzmann method as a basis for ocean circulation modeling. *Journal of Marine Research*, 57:503–535, 1999.

Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13 (3):233–246, 1994. doi: https://doi.org/10.1111/1467-8659.1330233.

David C. Seal, Qi Tang, Zhengfu Xu, and Andrew J. Christlieb. An explicit high-order single-stage single-step positivity-preserving finite difference weno method for the compressible euler equations. *Journal of Scientific Computing*, 68(1):171–190, Jul 2016. ISSN 1573-7691. doi: 10.1007/s10915-015-0134-0. URL https://doi.org/10.1007/s10915-015-0134-0.

Andrew Selle, Ronald Fedkiw, ByungMoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *J. Sci. Comput.*, 35:350–371, 06 2008. doi: 10.1007/s10915-007-9166-4.

Musawir A. Shah, Jaakko Konttinen, and Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):272–280, 2007. doi: 10.1109/TVCG.2007.32.

André Silvestre. A real-time terrain ray-tracing engine. Master's thesis, Instituto Superior Técnico da Universidade de Lisboa, 2017.

Andreas Söderström and Ken Museth. Non-reflective boundary conditions for incompressible free surface fluids. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588346. doi: 10.1145/1597990.1597994. URL https://doi.org/10.1145/1597990.1597994.

Barbara Solenthaler, Peter Bucher, Nuttapong Chentanez, Matthias Müller, and Markus Gross. SPH Based Shallow Water Simulation. In Jan Bender, Kenny Erleben, and Eric Galin, editors, *Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2011)*. The Eurographics Association, 2011. ISBN 978-3-905673-87-6. doi: 10.2312/PE/vriphys/vriphys11/039-046.

Jos Stam. Random caustics: Natural textures and wave theory revisited. In *ACM SIGGRAPH 96 Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH '96*, SIGGRAPH '96, page 150, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917847. doi: 10.1145/253607.253883. URL https://doi.org/10.1145/253607.253883.

Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 121–128, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0201485605. doi: 10.1145/311535.311548. URL https://doi.org/10.1145/311535.311548.

Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 129–136, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917014. doi: 10.1145/218380.218430. URL https://doi.org/10.1145/218380.218430.

Jerry Tessendorf. Simulating ocean water. *SIGGRAPH 2001 Course Notes*, 2001.

Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, page 183–190, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939838. doi: 10.1145/1342250.1342279. URL https://doi.org/10.1145/1342250.1342279.

N. Thurey, M. Muller-Fischer, S. Schirm, and M. Gross. Real-time breaking waves for shallow water simulations. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)*, pages 39–46, 2007. doi: 10.1109/PG.2007.33.

Nils Thürey. *Physically based animation of free surface flows with the Lattice Boltzmann method*. PhD thesis, University of Erlangen-Nuremberg, 2007.

Nils Thürey, Ulrich Rüde, and Carolin Körner. Interactive free surface fluids with the lattice boltzmann method. Technical report, University of Erlangen-Nuremberg, 2005.

Eleuterio F. Toro. *Some Properties of the Euler Equations*, pages 87–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-49834-6. doi: 10.1007/b79761_3. URL https://doi.org/10.1007/b79761_3.

Kevin R. Tubbs and Frank T.-C. Tsai. Gpu accelerated lattice boltzmann model for shallow water flow and mass transport. *International Journal for Numerical Methods in Engineering*, 86(3):316–334, 2011. doi: https://doi.org/10.1002/nme.3066.

Patricio Gonzalez Vivo and Jen Lowe. Fractal brownian motion. https://thebookofshaders.com/13/, 2021. [Online; accessed 28-June-2022].

Alex Vlachos. Water flow in Portal 2. https://advances.realtimerendering.com/s2010/, 2010. [Online; accessed 28-June-2022].

Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet Models for Refraction through Rough Surfaces. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques*. The Eurographics Association, 2007. ISBN 978-3-905673-52-4. doi: 10.2312/EGWR/EGSR07/195-206.

Huamin Wang, Gavin Miller, and Greg Turk. Solving general shallow wave equations on surfaces. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, page 229–238, Goslar, DEU, 2007. Eurographics Association. ISBN 9781595936240.

T. Weaver and Z. Xiao. Fluid simulation by the smoothed particle hydrodynamics method: A survey. In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications: Volume 1: GRAPP*, GRAPP 2016, page 215–225, Setubal, PRT, 2016. SCITEPRESS - Science and Technology Publications, Lda. ISBN 9789897581755. doi: 10.5220/0005673702130223. URL https://doi.org/10.5220/0005673702130223.

Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 291–294, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917464. doi: 10.1145/237170.237267. URL https://doi.org/10.1145/237170.237267.

Chris Wyman. Hierarchical caustic maps. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, page 163–171, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939838. doi: 10.1145/1342250.1342276. URL https://doi.org/10.1145/1342250.1342276.

Chris Wyman and Scott Davis. Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 153–160, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593295X. doi: 10.1145/1111411.1111439. URL https://doi.org/10.1145/1111411.1111439.

Chris Wyman and Greg Nichols. Adaptive caustic maps using deferred shading. *Computer Graphics Forum*, 28(2):309–318, 2009. doi: https://doi.org/10.1111/j.1467-8659.2009.01370.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01370.x.

Yulong Xing and Chi-Wang Shu. High order finite difference weno schemes with the exact conservation property for the shallow water equations. *Journal of Computational Physics*, 208(1):206–227, 2005. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2005.02.006. URL https://www.sciencedirect.com/science/article/pii/S002199910500094X.

Xueqing Yang and Yaobin Ouyang. *Real-Time Ray Traced Caustics*, pages 469–497. Apress, Berkeley, CA, 2021. ISBN 978-1-4842-7185-8. doi: 10.1007/978-1-4842-7185-8_30. URL https://doi.org/10.1007/978-1-4842-7185-8_30.

Q. Yu, E. Bruneton, N. Holzschuch, and F. Neyret. Lagrangian texture advection: Preserving both spectrum and velocity field. *IEEE Transactions on Visualization & Computer Graphics*, 17(11):1612–1623, nov 2011. ISSN 1941-0506. doi: 10.1109/TVCG.2010.263.

Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. Spectrum-preserving texture advection for animated fluids. Research Report RR-6810, INRIA, 2009. URL https://hal.inria.fr/inria-00355827.

Cem Yuksel and John Keyser. Fast real-time caustics from height fields. *The Visual Computer*, 25:559–564, 05 2009. doi: 10.1007/s00371-009-0350-4.

Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3), 2007. doi: 10.1145/1276377.1276501. URL http://doi.acm.org/10.1145/1276377.1276501.

Jian Guo Zhou. Lattice boltzmann model for the shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 191:3527–3539, 06 2002. doi: 10.1016/S0045-7825(02)00291-8.

Jian Guo Zhou. *Lattice Boltzmann Methods for Shallow Water Flows*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-662-08276-8.

Jian Guo Zhou. Macroscopic lattice boltzmann method for shallow water equations (maclabswe). *Water*, 14 (2065), 2022.

Part III

APPENDICES

A

DEFINING THE FLUX JACOBIANS

The fluxes in the conservative SWE, $\boldsymbol{F}$ and $\boldsymbol{G}$, can be written as in Equation 56, with $\boldsymbol{U}$ as in Equation 55.

$$\boldsymbol{U} = \begin{pmatrix} h \\ hu \\ hv \end{pmatrix} = \begin{pmatrix} A \\ B \\ C \end{pmatrix} \tag{55}$$

$$\begin{aligned} \boldsymbol{F} &= \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix} = \begin{pmatrix} B \\ \frac{B^2}{A} + \frac{1}{2}gA^2 \\ \frac{BC}{A} \end{pmatrix} \\ \boldsymbol{G} &= \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix} = \begin{pmatrix} C \\ \frac{BC}{A} \\ \frac{C^2}{A} + \frac{1}{2}gA^2 \end{pmatrix} \end{aligned} \tag{56}$$

Then, the flux Jacobians $\partial \boldsymbol{F}/\partial \boldsymbol{U}$ and $\partial \boldsymbol{G}/\partial \boldsymbol{U}$ are determined as per the Jacobian definition, resulting in Equation 57 ((LeVeque et al., 2002)). The subscripts denote the entry of the vector, e.g., $F_1 = hu = B$.

$$\begin{aligned} \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{U}} &= \begin{pmatrix} 0 & 1 & 0 \\ -u^2 + gh & 2u & 0 \\ -uv & v & u \end{pmatrix} = \begin{pmatrix} \frac{\partial F_1}{\partial A} & \frac{\partial F_1}{\partial B} & \frac{\partial F_1}{\partial C} \\ \frac{\partial F_2}{\partial A} & \frac{\partial F_2}{\partial B} & \frac{\partial F_2}{\partial C} \\ \frac{\partial F_3}{\partial A} & \frac{\partial F_3}{\partial B} & \frac{\partial F_3}{\partial C} \end{pmatrix} \\ \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{U}} &= \begin{pmatrix} 0 & 0 & 1 \\ -uv & v & u \\ -v^2 + gh & 0 & 2v \end{pmatrix} = \begin{pmatrix} \frac{\partial G_1}{\partial A} & \frac{\partial G_1}{\partial B} & \frac{\partial G_1}{\partial C} \\ \frac{\partial G_2}{\partial A} & \frac{\partial G_2}{\partial B} & \frac{\partial G_2}{\partial C} \\ \frac{\partial G_3}{\partial A} & \frac{\partial G_3}{\partial B} & \frac{\partial G_3}{\partial C} \end{pmatrix} \end{aligned} \tag{57}$$

B

WENO RECONSTRUCTION PROCEDURE

The main idea of Essentially Non-Oscillatory (ENO) and WENO reconstruction procedure is to choose an approximation to the function to be reconstructed such that it is as smooth as possible in the candidate stencil used for the approximation (Chi-Shu, 1997). WENO extends ENO in that, instead of choosing a single approximation, a convex combination of all reconstructions from different candidate stencils is used, assigning larger weights to smoother stencils. A more in-depth description and survey of it's applications can be found in Chi-Shu (1997).

The third-order WENO procedure is used to compute $\hat{\boldsymbol{F}}$ and $\hat{\boldsymbol{G}}$, as in Parna et al. (2018). Generally, the fluxes and conserved variables are projected to the local characteristic fields before applying the WENO reconstruction procedure. However, an alternative without characteristic projection (also called component-by-component) was shown in (Parna et al., 2018) to have better performance, while having similar accuracy, so this approach is followed.

B.0.1 *Flux splitting*

The first step for finite difference WENO methods is to split the fluxes so correct upwinding can be used (Chi-Shu, 1997). Upwinding is the biasing of the stencil to the direction that the flow is coming from. For example, if $\frac{\partial \tilde{\boldsymbol{F}}}{\partial \boldsymbol{U}} \geq 0$, the "wind direction" is positive, so more points would be sampled in this direction. Since $\frac{\partial \tilde{\boldsymbol{F}}}{\partial \boldsymbol{U}}$ and $\frac{\partial \tilde{\boldsymbol{G}}}{\partial \boldsymbol{U}}$ can change sign over the sampled points, the fluxes ($\tilde{\boldsymbol{F}}$ and $\tilde{\boldsymbol{G}}$) are split, as in Equation 58, such that it satisfies Equation 59. From now on, only the x direction is considered, as for flux $\tilde{\boldsymbol{F}}$, but the same applies to $\tilde{\boldsymbol{G}}$.

$$\tilde{\boldsymbol{F}} = \boldsymbol{f}^{+} + \boldsymbol{f}^{-} \tag{58}$$

$$\frac{\partial \boldsymbol{f}^{+}}{\partial \boldsymbol{U}} \geq 0 \qquad\qquad \frac{\partial \boldsymbol{f}^{-}}{\partial \boldsymbol{U}} \leq 0 \tag{59}$$

Local Lax-Friedrichs flux splitting (LeVeque et al., 2002) is used to split the fluxes (Parna et al., 2018), as in Equation 60. A modified $\boldsymbol{U}$ is used where η replaces h, so that $\boldsymbol{U}' = (\eta, hu, hv)^T$. This is part of the well-

balanced treatment of the source term (Section 3.1.3), and is useful because in still water stationary solutions, η stays constant (Xing and Shu, 2005).

$$\boldsymbol{f}^{\pm} = \frac{1}{2}\left(\tilde{\boldsymbol{F}} \pm \alpha \boldsymbol{U}'\right) \tag{60}$$

The α is given by Equation 61, where λ_n is the nth eigenvalue of the flux Jacobian (Equation 23).

$$\alpha = \max_{\boldsymbol{U}\in\mathcal{I}} \max_{1\leq n\leq 3} |\lambda_i| \tag{61}$$

The maximum eigenvalues ($\max_{1\leq n\leq 3} |\lambda_i|$) for the x and y direction flux Jacobians can be computed as $|u| + \sqrt{gh}$ and $|v| + \sqrt{gh}$ respectively. $\mathcal{I}$ is the stencil of values to be considered, which has 4 points around the value being computed, e.g, when computing $\hat{\boldsymbol{F}}_{i+1/2,j}$, $\mathcal{I} \in \{\boldsymbol{U_{i-1,j}}, \boldsymbol{U_{i,j}}, \boldsymbol{U_{i+1,j}}, \boldsymbol{U_{i+2,j}}\}$.

When applying the WENO reconstruction to the source term derivatives (Section 3.1.3), they are simply split as in Equation 62 (example for b).

$$b^{\pm} = \frac{1}{2} b \tag{62}$$

B.0.2 *Applying the WENO reconstruction*

Considering the WENO reconstruction function as presented in Equation 63, after flux splitting, each flux $\boldsymbol{F}$ in the stencil is split into $\boldsymbol{f}^+$ and $\boldsymbol{f}^-$ (Equation 60).

$$\hat{\boldsymbol{F}}_{i+1/2,j} = WENO3(\tilde{\boldsymbol{F}}_{i-1,j}, \tilde{\boldsymbol{F}}_{i,j}, \tilde{\boldsymbol{F}}_{i+1,j}, \tilde{\boldsymbol{F}}_{i+2,j}) \tag{63}$$

The WENO reconstruction is then applied to the split fluxes, using the corresponding biased stencil for each, as in Equation 64. The $WENO3^{\pm}$ function is described further below.

$$\begin{aligned} \hat{\boldsymbol{f}}^{+}_{i+1/2,j} &= WENO3^{+}(\boldsymbol{f}^{+}_{i-1,j}, \boldsymbol{f}^{+}_{i,j}, \boldsymbol{f}^{+}_{i+1,j}) \\ \hat{\boldsymbol{f}}^{-}_{i+1/2,j} &= WENO3^{-}(\boldsymbol{f}^{-}_{i,j}, \boldsymbol{f}^{-}_{i+1,j}, \boldsymbol{f}^{-}_{i+2,j}) \end{aligned} \tag{64}$$

$\hat{\boldsymbol{F}}_{i+1/2,j}$ is then defined as in Equation 65.

$$\hat{\boldsymbol{F}}_{i+1/2,j} = \hat{\boldsymbol{f}}^{+}_{i+1/2,j} + \hat{\boldsymbol{f}}^{-}_{i+1/2,j} \tag{65}$$

B.0.3 *WENO step*

A single third order WENO reconstruction step for a value v is described here (as in Parna et al. (2018)). The x direction is used as an example (j subscript dropped for simplicity), where the procedure is denoted by the functions $WENO3^{\pm}$ in Equation 66. The $\pm$ superscript denotes which bias the stencil uses.

$$\begin{aligned} v^{+}_{i+1/2} &= WENO3^{+}(v^{+}_{i-1,j}, v^{+}_{i,j}, v^{+}_{i+1,j}) \\ v^{-}_{i+1/2} &= WENO3^{-}(v^{-}_{i,j}, v^{-}_{i+1,j}, v^{-}_{i+2,j}) \end{aligned} \tag{66}$$

The steps of the procedure are as follows:

1. Find the approximations on two different stencils (Equation 67).

$$\begin{aligned} f^{+}_{0} &= \frac{1}{2}v^{+}_{i} + \frac{1}{2}v^{+}_{i+1} & \qquad f^{-}_{0} &= \frac{1}{2}v^{-}_{i+1} + \frac{1}{2}v^{-}_{i} \\ f^{+}_{1} &= -\frac{1}{2}v^{+}_{i-1} + \frac{3}{2}v^{+}_{i} & \qquad f^{-}_{1} &= -\frac{1}{2}v^{-}_{i+2} + \frac{3}{2}v^{-}_{i+1} \end{aligned} \tag{67}$$

2. Find the smoothness indicators (Equation 68).

$$\begin{aligned} \beta^{+}_{0} &= (v^{+}_{i+1} - v^{+}_{i})^2 & \qquad \beta^{-}_{0} &= (v^{-}_{i+1} - v^{-}_{i})^2 \\ \beta^{+}_{1} &= (v^{+}_{i} - v^{+}_{i-1})^2 & \qquad \beta^{-}_{1} &= (v^{-}_{i+2} - v^{-}_{i+1})^2 \end{aligned} \tag{68}$$

3. Find the non-linear weights (Equation 69, where $n \in \{0, 1\}$, and the linear weights are $d_0 = \frac{2}{3}$ and $d_1 = \frac{1}{3}$).

$$\omega^{\pm}_{n} = \frac{\zeta^{\pm}_{n}}{\zeta^{\pm}_{0} + \zeta^{\pm}_{1}} \qquad \zeta^{\pm}_{n} = \frac{d_n}{(\epsilon + \beta^{\pm}_{n})^2} \tag{69}$$

   While ϵ was originally chosen in order to avoid the denominator becoming zero, it has been shown that it actually should relate to the grid-spacing, so it is set as $\epsilon = (\Delta x)^2$ (Cravero and Semplice, 2016).

4. Find the third order accurate approximation (Equation 70) by combining the two approximations with the non-linear weights.

$$v^{\pm}_{i+1/2} = \omega^{\pm}_{0} f^{\pm}_{0} + \omega^{\pm}_{1} f^{\pm}_{1} \tag{70}$$

When applying the WENO reconstruction to the source term derivatives (Section 3.1.3), the first source term derivative, $s = (\frac{1}{2}gb^2)$, can simply be absorbed into the computations of the relevant flux to save cost (Xing

and Shu, 2005), e.g., for direction x as given in Equation 71. Consequently, when computing the final source term $\boldsymbol{S}$ Equation 28, the first term is ignored, as it is already included in the flux.

$$\boldsymbol{f}^{\pm} \leftarrow \boldsymbol{f}^{\pm} - \frac{1}{2}\begin{pmatrix} 0 \\ s \\ 0 \end{pmatrix} \tag{71}$$

The second term, b, goes through the usual reconstruction procedure but uses the same smoothness indicators as in the flux reconstruction. That is, for example, after computing the reconstruction of a split flux, $\hat{f}^{+}$, when computing the reconstruction of corresponding $\hat{b}^{+}$, the same smoothness indicators β^{+} are used (Equation 68).

C

HIGH AND LOW-ORDER FLUX INTERPOLATION COEFFICIENTS

When interpolating the low- and high-order fluxes, the coefficients of the linear combination $\theta_{i+1/2,j}$ and $\theta_{i,j+1/2}$ need to be determined in a way that ensures the interpolation is of the highest order possible, while preserving the positivity of the solution. This is achieved by solving the following optimization problem.

Given Λ values that bound θ (with $\Lambda \in [0, 1]$), as in Equation 72, the θ parameters are then given by the minimum case of the bounding values (Seal et al., 2016), as in Equation 73.

$$\begin{aligned} \theta_{i-1/2,j} &\in [0, \Lambda_L] & \theta_{i,j-1/2} &\in [0, \Lambda_D] \\ \theta_{i+1/2,j} &\in [0, \Lambda_R] & \theta_{i,j+1/2} &\in [0, \Lambda_U] \end{aligned} \tag{72}$$

$$\begin{aligned} \theta_{i+1/2,j} &= \min(\Lambda_{R,i,j}, \Lambda_{L,i+1,j}) \\ \theta_{i,j+1/2} &= \min(\Lambda_{U,i,j}, \Lambda_{D,i,j+1}) \end{aligned} \tag{73}$$

The minimal case method proposed in Parna et al. (2018) is used to determine the bounding values Λ, while ensuring that $h_{i,j}^{n+1} \geq 0$, whose steps are as follows:

1. Determine Γ and the $\mathcal{F}$ values (Equation 75 and Equation 76). These are defined by applying Equation 30 to the first line of Equation 26, which becomes as in Equation 74 (which is being solved so that $h_{i,j}^{n+1} \geq 0$).

$$h_{i,j}^{n+1} = -\Gamma_{i,j} + \theta_{i+1/2,j}\mathcal{F}_{i+1/2,j} + \theta_{i-1/2,j}\mathcal{F}_{i-1/2,j} + \theta_{i,j+1/2}\mathcal{F}_{i,j+1/2} + \theta_{i,j-1/2}\mathcal{F}_{i,j-1/2} \tag{74}$$

$$\Gamma_{i,j} = -\left(h - \frac{\Delta t}{\Delta x}(\tilde{f}_{i+1/2,j} - \tilde{f}_{i-1/2,j}) - \frac{\Delta t}{\Delta y}(\tilde{g}_{i,j+1/2} - \tilde{g}_{i,j-1/2})\right) \tag{75}$$

$$\begin{aligned}
\mathcal{F}_{i+1/2,j} &= -\frac{\Delta t}{\Delta x}(\hat{F}^{(1)}_{i+1/2,j} - \tilde{f}_{i+1/2,j}) \qquad & \mathcal{F}_{i-1/2,j} &= \frac{\Delta t}{\Delta x}(\hat{F}^{(1)}_{i-1/2,j} - \tilde{f}_{i-1/2,j}) \\
\mathcal{F}_{i,j+1/2} &= -\frac{\Delta t}{\Delta y}(\hat{G}^{(1)}_{i,j+1/2} - \tilde{g}_{i,j+1/2}) \qquad & \mathcal{F}_{i,j-1/2} &= \frac{\Delta t}{\Delta y}(\hat{G}^{(1)}_{i,j-1/2} - \tilde{g}_{i,j-1/2})
\end{aligned} \tag{76}$$

2. Determine Q (Equation 77) and $\alpha, \beta, \gamma, \delta$ (Equation 78[1] , which are functions of the signs of the values in Equation 76).

$$Q = \min\left(\frac{\Gamma_{i,j}}{\alpha\mathcal{F}_{i+1/2,j} + \beta\mathcal{F}_{i-1/2,j} + \gamma\mathcal{F}_{i,j+1/2} + \delta\mathcal{F}_{i,j-1/2}}\right) \tag{77}$$

$$\begin{aligned}
\alpha &= \begin{cases} 1, & \text{if } \mathcal{F}_{i+1/2,j} < 0 \\ 0, & \text{otherwise} \end{cases} \qquad & \beta &= \begin{cases} 1, & \text{if } \mathcal{F}_{i-1/2,j} < 0 \\ 0, & \text{otherwise} \end{cases} \\
\gamma &= \begin{cases} 1, & \text{if } \mathcal{F}_{i,j+1/2} < 0 \\ 0, & \text{otherwise} \end{cases} \qquad & \delta &= \begin{cases} 1, & \text{if } \mathcal{F}_{i,j-1/2} < 0 \\ 0, & \text{otherwise} \end{cases}
\end{aligned} \tag{78}$$

3. Determine the Λ bounding values (Equation 79).

$$\begin{aligned}
\Lambda_R &= (1-\alpha) + \alpha Q \qquad & \Lambda_L &= (1-\beta) + \beta Q \\
\Lambda_U &= (1-\gamma) + \gamma Q \qquad & \Lambda_D &= (1-\delta) + \delta Q
\end{aligned} \tag{79}$$

1 Note that the branching in Equation 78 can be avoided by directly utilizing the return value of a conditional expression.