Contents

List of Figures

1 Introduction

Early attempts at aquatic scenes in computer games typically treated water as planar surfaces with authored textures. In today's real-time applications, simple bump mapped planes are still an encountered approximation of watery backdrops. Although this may be an adequate solution in particularly calm cases, the shortcomings become obvious when the scene is a large oceanscape agitated by wind forces, or when the water is a subject of interaction.

This work is focused on the problem of graphically reproducing typical offshore environments. More specifically, the aim is to realistically simulate and visualize the surface of open seas at real-time rendering rates, using current consumer range hardware. This implies very large water surfaces dominated by unhindered wind-driven waves, but also influenced by the presence of watercraft activity and offshore installations. The discourse begins with a look at potential existing approaches, and proceeds to discuss fruitful combinations and possible extensions.

The ocean is an important arena in computer visualization. One example is from the offshore industry, where virtual environments have become important tools in monitoring, planning, training, etc. In later years, such tools have included more and more visual elements, and featured increasingly realistic settings.

The system implemented here tackles unbounded ocean surfaces, with realistic distributions of wind-driven waves. The surfaces are treated as periodic elevation fields, and synthesized from statistically sampled frequency spectra. Obvious repeating structures across a surface, due to this periodic nature, are avoided by decomposing the elevation field synthesis, using two or more discrete spectra with different frequency scales.

To enable responsive water surfaces, with opportunities for boat wakes, surface obstacles, etc., a GPU-based water solver is also included. Its implementation features a convenient input interface, which exploits hardware rasterization both for efficiency and to provide smooth surface deflections, connected deflective paths, etc.

Finally, polygonal representations of visible ocean regions are obtained using a GPU-accelerated tessellation scheme suitable for wave fields. This scheme provides view-dependent resolutions, with very little geometry ending up outside the view volume, and is highly economic with regards to data transfer.

2 Prior work

Water and ocean simulation has long been a popular research topic in the computer graphics community. Today, computer animated water may well fool the eye. With the current advances in graphics hardware, realism is also taking root in real-time water simulation. Wielding modern programmable graphics processors, researchers are able to capture a wider range of phenomena at increasing detail, while keeping interactive rendering rates.

This chapter begins by looking at prevalent approaches to modeling the structure of ocean water. The discussion is divided into *geometrical*, *statistical*, and *physical* models. These categories are more or less connected, but denote different abstractions of the problem, typically diverging in generality, realism and perfomance. Geometric models focus on modeling individual surface waves, with which more complex shapes and dynamics are composed. Statistical models rely on empirical data from oceanographic research for a natural distribution of wave parameters, or may assume statistical self-similarity in the surface structure at different scales. Physical models typically see water as a system of particles or regions that physically interact, and that are influenced by forces acting on the system.

The optical behaviour of water is discussed next, concentrating on adaptations for computer graphics and issues regarding rendering. The concluding topic in this chapter is considerations in real-time ocean simulation, such as economizing the resolution of the model and balancing the use of available hardware resources.

2.1 Geometrical models

A popular method for modeling water surfaces is constructing an animated height field by linear combination of traveling periodic functions. The choice of functions and parameters, and the number of terms used, are subject to variation in literature. Notably, the number of affordable terms is still limited

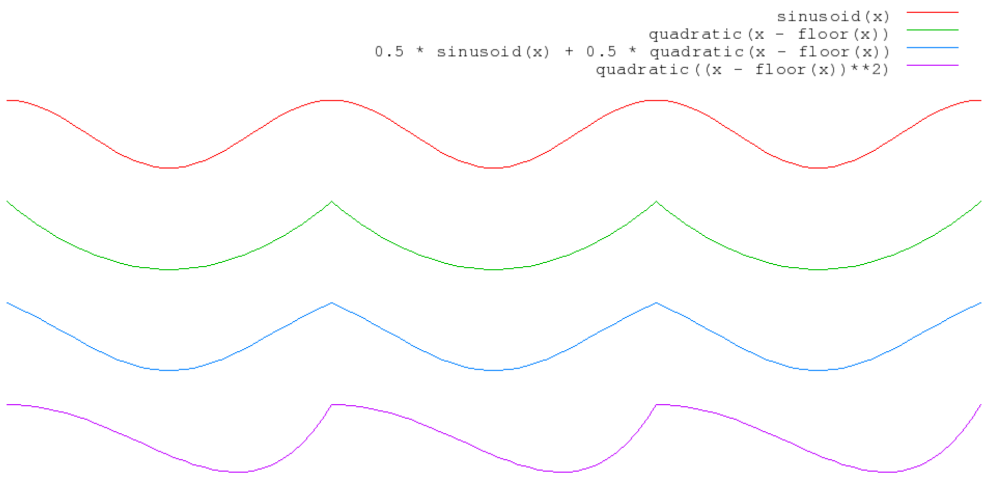


Figure 1: Various wave profiles. From top to bottom: sinusoid wave, piecewise quadratic wave, blended wave, quadratic wave with exponentiated argument. The functions used here are: $sinusoid(x) = \cos(2\,\pi\,x)$ and $quadratic(x) = 8\,(x - 0.5)^2 - 1$.

in real-time computation, with regard to modeling of large oceanic regions.

Before the days of 10^8 transistor graphics chips in personal computers, Max used a Cray-1 supercomputer to produce ray-traced animations of water surfaces [Max81]. In his implementation, the surfaces are generated by superposing a set of 2D sinusoids. Low amplitude sinusoids are used to produce detailed ripples near the eye. Higher amplitude waves, which have visibly wider troughs and narrower crests in nature than do sinusoids, are modeled by approximating the Fourier expansion of cycloid curves. A full temporal period of resulting height fields is rendered and recorded onto film, which can then be seamlessly looped. For a periodic cycle of frames, the sinusoid frequencies are restricted to multiples of some reasonable fundamental frequency.

Ocean surface waves can take on shapes that are problematic, or even impossible, to represent by a sum of sinusoids, such as sharp crests or breaking fronts. To produce waveforms more closely resembling natural ocean waves

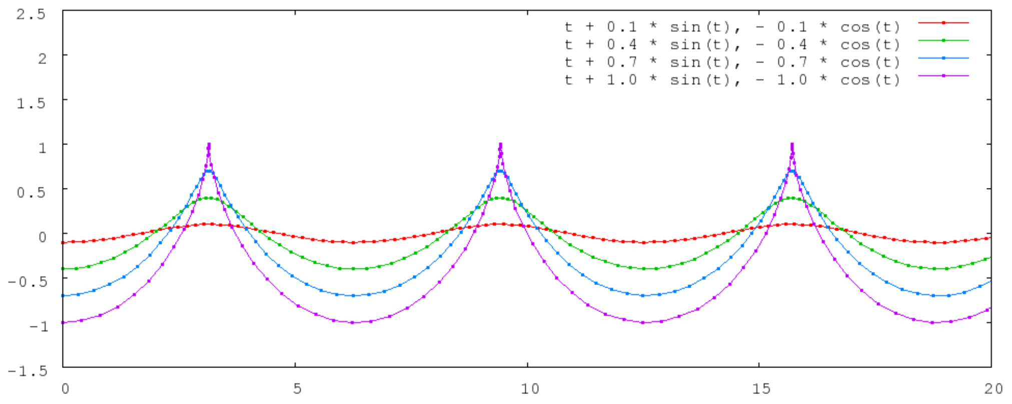


Figure 2: Gerstner waves

under different conditions, Peachey uses a linear blend of sinusoid and cycloid terms [Pea86]. For a more convenient formulation, the cycloids are approximated by piecewise quadratic functions. Since steep waves naturally form narrow crests and wide troughs, steepness is used to control the shift from a sinusoidal shape towards a cycloidal shape. Now, as waves move onto shallower water they gradually steepen, and undergo a realistic change in appearance from long smooth swells to shorter and choppier waves. Waves moving onto shore also grow steeper in the front, and assume an asymmetric profile. This is achieved by exponentiating the argument to the wave function, which is in the range [0, 1), so values are shifted towards the low end of the interval. See figure 1. Peachey further accounts for wave refraction, i.e. directional changes due to seabed topography, and implements a particle system to simulate spray from breaking waves.

Fournier and Reeves adapt the *Gerstner wave* model, a parametric representation of trochoidal waves [FR86]. Using this representation, samples are displaced laterally towards wave peaks, conveniently resulting in higher sample rates where the magnitude of the gradient is greater, i.e. where the geometric error would be greater with a uniform sample rate. See figure 2. In their formulation, Fourier and Reeves extend the Gerstner model, taking wave direction and depth into account, to enable asymmetric wind-driven waves and waves breaking on the shore. They further simulate wave refraction, as well as spray and foam, which is realized using particle systems.

Ts'o and Barsky achieve wave refraction by wave-tracing, which can be thought of as ray-tracing over a subsea topography [TB87]. In their formulation, waves are traced in a similar manner as a ray would be traced through refractive media. The contribution of the different waves are subsequently superposed on a height field, using sinusoidal shapes. This height field is finally represented by Beta-splines, whose tension parameters can be used to tweak the appearance of the final surface.

2.2 Statistical models

While realistic looking scenes can be achieved by the above methods, using enough components, they do not solve the problem of selecting an appropriate set of parameters for the model. Tuning these manually may turn into a cumbersome task as more components are thrown in. Rather than manual tuning, a model that provides a natural distribution of components based on meaningful governing parameters like wind and gravity would be valuable. Moreover, sum evaluation intensifies as the number of terms grows, becoming an important point of optimization.

Loosely qualifying as a statistical approach, time-varying stochastic fractals have been used to simulate a variety of natural phenomena. Perlin used stochastic fractals, essentially summations of a noise function at different scales, to generate images of fire, water, clouds and more [Per85]. See figure 3. Musgrave makes extensive use of fractals in landscape imaging, therein ocean simulation [Mus93]. While such models are attractively simple, water does not generally exhibit fractal shapes. Plausible results can be achieved for calm waters, but the physical basis seems too weak to realistically portray natural propagation, wind-driven scenarios, etc.

Mastin et al. use an empirical frequency spectrum of wind-driven ocean, given by the *Pierson-Moskowitz filter*, to sample an appropriate set of frequency components [MWM87]. In their formulation, sampling is done by

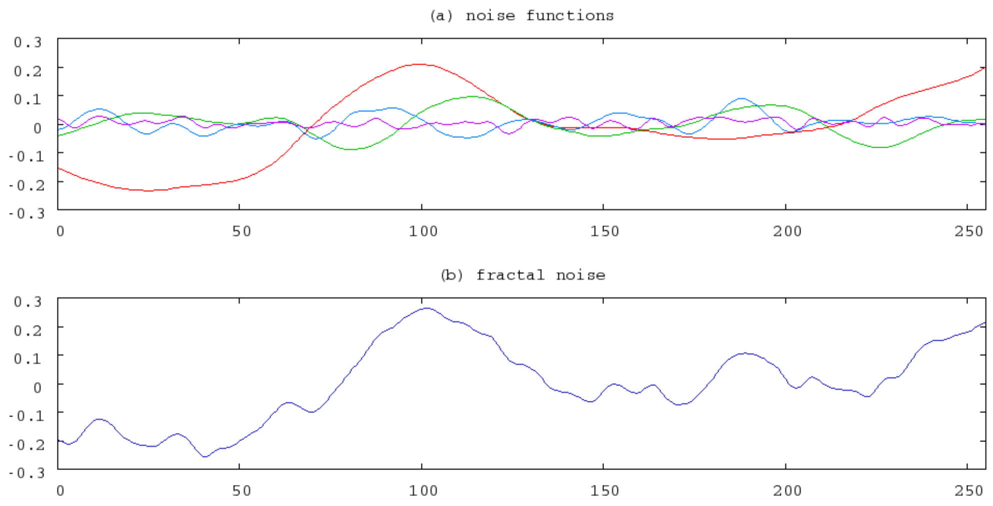


Figure 3: Perlin noise. (a) shows four smoothly interpolated noise sets, with different amplitudes and frequencies. (b) shows the sum of these functions forming a stochastic fractal, made to resemble rippled water waves.

transforming a white noise image, by an FFT algorithm, and applying the filter to the resulting spectrum. The filtered frequency data is further used to synthesize a discrete height map of an ocean region, by the inverse FFT. See figure 6 for a visual example of synthesis from an oceanic spectrum. The height maps can be animated by phase manipulation in the frequency domain, and in this regard, two different schemes are explored. The proposed model applies to fully developed wind-driven seas, and does not account for shallow water phenomena. Notably, only a few parameters, e.g. wind direction and speed, need be specified to generate a fairly realistic ocean surface.

The method used by Mastin et al. produces discrete height fields, due to the IFFT, which may introduce issues like aliasing, depending on the field of view. Moreover, the underlying components are sinusoidal, and do not readily compose agitated seas exhibiting cycloidal waveforms. Thon et al. also use the Pierson-Moskowitz spectrum, but do not transform sampled data into spatial images [TDG00]. Instead, they select a representative set of frequency components from the sampled spectrum, and assign corresponding parameters to trochoid waveforms, as given by the Gerstner model. The result is a continuous surface, defined by a sum of trochoids, which can be evaluated as needed upon rendering. Since no optimized algorithm is applied in computing the superposition, such as the FFT, the number of affordable components is reduced. To compensate for this, the main structure of trochoid waveforms is perturbed using a three-dimensional turbulence function, thereby adding a finer detail level to the surface.

Tessendorf replaces the Pierson-Moskowitz filter with the *Phillips spectrum*, which is directly applicable to noise in the Fourier domain [Tes04]. In his course notes, a number of modifications is introduced to allow more control over the model, like supression of small wavelengths and waves with directionality dissimilar to that of the wind. The IFFT is applied to the generated data, and a discrete height map is obtained. To allow choppier waves, a field of horizontal displacement vectors is computed, based on the gradient of the height map. Horizontal displacements are applied to grid points along with height displacements, pushing samples towards peaks in a similar manner as Gerstner waves. The height field is animated by frequency domain phase manipulation, accounting for dispersion of water surface waves, i.e. the relation between wavenumber and propagative speed.

2.3 Physical models

The motion of fluids is described by a set of nonlinear partial differential equations, called the *Navier-Stokes equations*, or NSE for short. Equation 1 shows one formulation of the NSE, for an incompressible Newtonian fluid. Here, ρ and μ are measures of the fluid's density and viscosity, respectively. $\mathbf{v}$ is the continuous velocity field within the fluid, ∇p is a pressure gradient, and $\mathbf{f}$ represents other forces, like gravity. The latter equation states the conservation of volume, in other words incompressibility. Given well formed boundary conditions, this set of equations seems to accurately model the motion of fluid volumes, such as water. Most problems based on the NSE are too complex to lend themselves to analytical solution, and must be solved numerically. Numerical methods for solving such problems are well adapted for scientific simulation, but typically unsuitable for real-time purposes, due to computational intensity. If superficial realism can be settled for, however, a number of simplifications can be considered.

$$\overbrace{\rho\,(\underbrace{\partial\mathbf{v}/\partial t}_{\substack{\textit{unsteady}\\ \textit{acceleration}}} + \underbrace{\mathbf{v}\cdot\nabla\mathbf{v}}_{\substack{\textit{convective}\\ \textit{acceleration}}})}^{\textit{inertia}} = \underbrace{-\nabla p}_{\substack{\textit{pressure}\\ \textit{gradient}}} + \underbrace{\mu\,\nabla^2\mathbf{v}}_{\textit{viscosity}} + \underbrace{\mathbf{f}}_{\substack{\textit{other}\\ \textit{forces}}} \qquad (1)$$
$$\nabla\cdot\mathbf{v} = 0$$

Numerous papers have addressed computational fluid dynamics in the context of computer graphics. Early, in this respect, Kass and Miller turn to a substantially simplified set of equations called the 2D *shallow water equations*, which models the surface of water [KM90]. In their formulation, a linear approximation of the shallow water equations is used, and solved on a uniform finite difference grid. This approximation can be stated as a second order differential equation, with the form:

$$\frac{\partial^2 h}{\partial t^2} = g\,d\left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2}\right) \qquad (2)$$

Here, h is the surface height at position (x, y), g is the gravitational acceleration, and d is the varying depth of the water. Kass and Miller's model makes assumptions (e.g. low fluid velocities, height field representation) that can only result in good approximations for relatively calm cases, when gentler forces are at work. In addition to wave refraction, which was accounted for in less physically based models discussed in section 2.1, this model also reproduces the reflection of waves off objects in the water.

Modeling water as a surface, and not a volume, limits the range of phenomena that are readily captured. Indeed, a lot goes on under the surface that contributes to the motion of the surface itself. The 2D grid model used by Kass and Miller, for example, does not reproduce the swirling motions often seen in fluids. Stam addresses the full NSE, to produce both 2D and 3D fluid animations [Sta99]. In his formulation, the problem is solved on a grid, incorporating a semi-Lagrangian scheme to solve for advection in the fluid. This involves interpolation between cells in the velocity grid, simulating particles moving within the fluid. The result is a fast and numerically stable fluid solver that would allow the user to interact with fluid volumes in real-time on a graphics workstation at the time of publication. The model suffers from numerical dissipation, however, which introduces issues like increased rotational damping and mass dissipation. Damping can to some extent be remedied by having the animator add external forces to keep flows alive longer. Though the model is not accurate within engineering standards, it is capable of creating realistic looking scenarios with nice swirling flows.

Foster and Fedkiw adapt the semi-Lagrangian method introduced by Stam in their modeling and animation of liquids [FF01]. In their formulation, mass dissipation is addressed by tracking the motion of the liquid surfaces, using a hybrid representation of inertialess particles and a level set. Particles are used in sparse regions of the fluids, where explicit details such as splashes can be seen. The level set is preferred in more well-resolved regions, where a smooth surface is desired. Keeping track of particles in cases like splashing prevents the loss of mass when regions of liquid are too small to be resolved by the level set.

Premože et al. borrow from a technique called smoothed particle hydrodynamics, originally from astrophysics [PTB+03]. This is a Lagrangian approach, where fluid regions are modeled as particles that can move about in the fluid, as opposed to Eulerian approaches where fluids are modeled using fixed grids. Smoothed particle hydrodynamics is primarily applicable to compressible fluids, so Premože et al. adopts a similar method that solves the NSE for incompressible fluids. Particle-based methods are particularly useful when situations like splashing can be expected, as discussed by Foster and Fedkiw, but one challenge is constructing a smooth surface, both spatially and temporally coherent, from the particle representation.

On larger scales, the discussed fluid solvers do not nearly handle real-time, most distantly the full volume-of-fluid solvers. The computational complexity of these methods puts strict limits on affordable scale and resolution, even offline. Recently, Thürey et al. presented a hybrid method that couples 2D and 3D fluid simulation [TRS06]. In their formulation, the full fluid flow is computed in a region of interest, and a faster two-dimensional shallow water simulation is used for the surrounding surface. Both models are solved using the lattice Boltzmann method, which approximates the NSE by relaxing the fluid's incompressibility constraint. The 3D region can be moved within the 2D region during the course of simulation, allowing the animator to add complex flow around a moving boat, for example.

2.4 Optics and rendering

In addition to models describing the surface geometry of oceans, a suitable model to describe the interaction between water and light is needed to give the surface a realistic appearance. Fortunately, the optical behaviour of water is well theorized. As in the modeling of a physical structure, however, the scope of the simulation must be limited to stay within reasonable computational bounds.

The ocean is a near perfect specular reflector, with varying translucency. Essential in its visual characteristics is the relation between incoming light and the light that is reflected away from and refracted into the water/air at different incidences. The intensity fraction of reflected rays at the interface of two media, at different angles of incidence, is given by the *Fresnel equations.* Figure 24 illustrates the relation between the angles of incident light and refracted light, and 25 shows the Fresnel reflectance at difference incidence angles. Since the Fresnel term tends to change rapidly over rippled waters, it is preferrable to evaluate this term per pixel. At this rate the evaluation makes a target for optimization, and an approximation is often used, e.g. reciprocals (Jensen and Goliás [JG01]), a 1D lookup texture (Heidrich and Seidel [HS99]), or a 2D lookup texture (Hu et al. [HVT$^+$06]). Fresnel reflectance is explained more in detail in appendix C.

Calculating the contributions from reflected and refracted light is a global illumination problem that can be solved by methods such as ray-tracing. This process can be particularly time consuming, with translucency contributing for the worse, so in real-time, its sophistication must typically be reduced substantially, e.g. by only taking first-order rays into account. For reflections and refractions from the global environment, e.g. the sky and the sea bottom, a common approach is to use cube maps. Environment mapping of water surfaces is explained further in appendix D. For local reflections and refractions, e.g. from objects in the water, one possible technique is to render the scene as reflected by a flat mirror into a projective texture, and perturb texture coordinates according to the sea topography on lookup (Jensen and Goliás). Conversely, the water surface casts light onto other surfaces, creating caustics. Caustics are patterns formed by the focusing and defocusing of reflected and refracted rays on receiving surfaces. This phenomenon is also addressed by Jensen and Goliás, who generate caustic texture maps, approximating receivers as planar.

The absorption and scattering of light due to water molecules and impurities in the water contributes to the color seen at the surface, and is responsible for phenomena like godrays, i.e. visible shafts of scattered light. Such water volume effects are addressed by Iwasaki et al., who approximate second-order scattering by solving the *radiative transfer equation* numerically on a number of sampling planes [IDN03]. While relatively thorough, this volumetric approach is not very suitable for real-time computation. Premože and Ashikhmin simplify the radiative transfer problem, using empirical equations and experimental optical parameters to estimate the radiance of scattered light [PA01]. Jensen and Goliás obtain closed formulae for the color contribution from deep water volumes, by ignoring effects like godrays. Refractions can then be looked up in a precalculated cube map, using the direction of refracted rays.

High contrast ratios are typical for ocean scenes, where glittering and glaring reflections of the sun are commonly seen. Memory formats used in digital imaging and rendering have very limited luminosity ranges compared to the

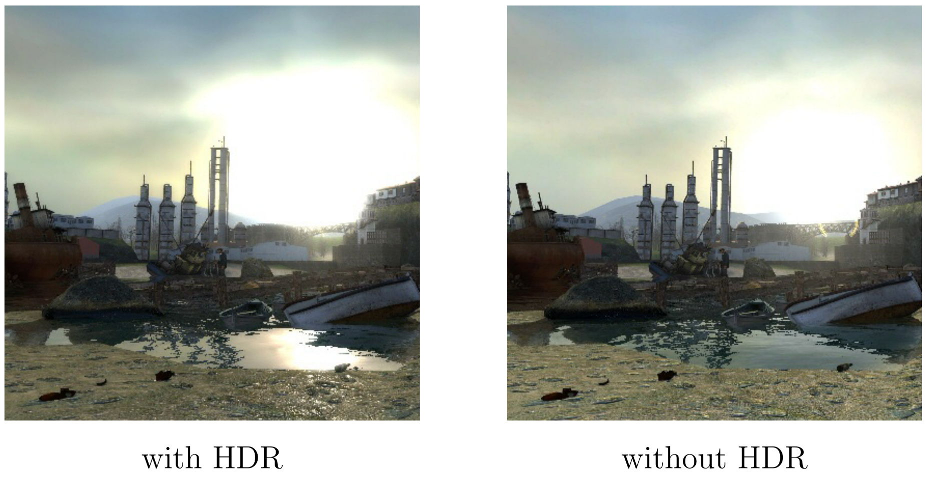


Figure 4: High dynamic range versus low dynamic range. From the game *Half-Life 2: Lost Coast.*

capabilities of human vision, however. Using standard lighting, high contrast scenes are typically rendered with severly truncated luminance, making bright areas like ocean glitter look rather dull. *High dynamic range* (HDR) lighting addresses this by using a higher memory precision, enabling a larger dynamic range in rendering. See figure 4. Newer graphics cards support high dynamic range rendering, trending towards increased precision. Pioneering work on high dynamic range imaging is found in Debevec and Malik's article [DM97], and a recent example of high dynamic range rendering in real-time use is discussed by McTaggart et al. [MGM06].

2.5 Real-time simulation

Most of the literature discussed in the previous paragraphs is primarily concerned with offline rendering. Having an efficient mathematical description of an ocean, however, challenges are still abundant when piecing together a real-time implementation. Firstly, oceanscapes tend to be very big, and large parts of the scenes are often visible at the same time, making a level of detail scheme important to the efficiency of the simulation. Secondly, the geometry of ocean surfaces changes constantly, inciting optimization in the data transfer between CPU and GPU. Since modern day graphics cards have become highly flexible and powerful, parts of the simulation could even be moved to the GPU, in the attempt to achieve a better load balance between the two processing units, and at the same time downscale the CPU-to-GPU data transfer.

Isidoro et al. simulate ocean entirely on graphics hardware, using a single pass per frame [IVB02]. In their implementation, the ocean surface is constructed from four sinusoids, whose sum is evaluated in a vertex shader and used to perturb a stored mesh. The mesh is animated by phase shifting the sinusoids. Further visual detail is added in the pixel shader by bump mapping. The bump map is obtained by combining two predefined texture maps, and animated by scrolling the textures at different rates. Since the bump map describes tangent space perturbations, a transformation into world space is required. Finally, lighting is performed using a cube mapped environment and a 1D texture map to approximate the Fresnel term.

Finch extends this approach, using Gerstner waves to displace the mesh vertices [Fin04]. Further, the tangent space bump map is generated by superposing about 15 higher frequency sinusoids. This implementation is more physically faithful, accounting for the *dispersion relation*, controlling the choppiness of the waves according to their steepness, and using a less provisional bump map. The result is a fast simulation, due to the relatively few wave components and the low bandwidth toll. The quality of the simulation is largely dependent on a good choice of matching parameters. More complex dynamics could be achieved by increasing the number of geometric waves, but evaluating their sum at every vertex would eventually become a performance consideration.

Kryachko exploits *vertex textures*, and uses authored animated height maps both for bump mapping and mesh perturbation [Kry05]. This technique was used in the game *Pacific Fighters*, where four maps with different spatial and temporal scales were combined for fragment shading, and the two larger scales were used to displace the underlying mesh within a vertex shader.

Kryachko applies a simple level of detail scheme, where a radial grid follows the camera, using lower resolution with increasing distance from the camera. Since both the grid and the maps can be stored in video memory, very little data need to be transferred from the CPU.

Rather than authored maps, a more physically rooted means of acquiring height maps might be preferred, like the FFT-based techniques described earlier. Using vertex textures, the maps may either be precalculated and stored in video memory, transferred to the GPU at every update, or calculated directly on the GPU. Moreland and Angel describe an implementation of the FFT running solely on the GPU, with a performance comparable to that of highly optimized CPU libraries [MA03]. Using a GPU algorithm to perform the FFT step can serve to balance the load between the CPU and the GPU, the CPU being a common bottleneck. Another inmportant realization is that both input and output are stored on the GPU-side, which is where the visualization data are often needed. This allows a reduction in bandwidth load, another typical bottleneck in visualization applications.

Mitchell adopts the techniques outlined by Tessendorf, performing Fourier synthesis on graphics hardware by a GPU-implementation of the FFT [Mit05]. In his implementation, a low frequency band is used in the synthesis of a displacement map, whereas a broader frequency band is used in a detailed version of the height field, for normal mapped lighting. Having a low frequency version for geometric displacement allows lower resolution grids without undersampling artifacts from higher frequencies. The two maps are also exploited for damping effects from e.g. plant matter, suppressing higher frequencies by blending between the maps in the lighting stage. This prominent use of the GPU shows promising framerates, with a low toll on both CPU and bandwidth resources.

Chiu and Chang extend Mitchell's GPU-based approach, implementing both GPU-based tessellation and spray simulation [CC06]. Their tessellation scheme consists of adjusting a stored rectangular grid in the view plane so its projection onto the ocean plane covers the currently visible ocean region. The vertices are then projected onto the ocean plane, where the height field is evaluated. See figure 14. This results in a view-dependent level of detail, where far regions are automatically tessellated with a lower resolution. Moreover, the simulated ocean surface is unbounded, using tiled height maps, since the grid continuously follows the visible region of ocean. The spray simulation is handled by a particle system, using floating point textures to store and evolve particle states on the GPU.

Krüger and Westermann present a general framework for solving systems of linear equations on the GPU, using efficient texture-based layouts for vectors and matrices [KW05]. A conjugate gradient solver is implemented, and demonstrated on the 2D *wave equation.* See equation 9, section 4.1.4. This shallow water simulation is run with interactive framerates, at relatively high resolutions, not accounting for 3D representation of the generated height map.

Concluding the section with a note on level of detail schemes, there exists a variety of LOD techniques designed for height fields, being important in the context of large-scale terrain rendering. Without further discussion, the reader is referred to work such as the GPU-based approach of Losasso and Hoppe [LH04].

3 Problem statement

Starting off, the elected problem is clarified, and a scope for the thesis is defined. The outlined scope is further viewed in relation to prior work, and the value of an implementation tackling the problem is discussed.

3.1 Scope

The aim here is to simulate and visualize water surfaces under typical offshore conditions at real-time rates, without relying on particularly expensive computing hardware. The term “real-time” is used in a strict sense, since the implemented functionality is intented for integration into larger systems that are already computationally intensive. Thus, relatively low CPU-usage and highly interactive framerates are prioritized. Another point with respect to later use is to make the functionality easily accessible in an open scene-graph library.

Most focus is devoted to issues that are not as well resolved for interactive simulation of open seas, e.g. how to reproduce a seemingly infinite ocean region realistically and efficiently. In simulating the surface structure, wind-driven waves are regarded as most important, but it is also interesting to capture the notion of a responsive surface that interacts with moving vessels or stationary offshore installations. For surface rendering, good 3D representations are needed. Thus, a suitable tessellation scheme for wave fields should be identified. Due to the vast extent of ocean surfaces, view-dependent resolutions are needed, and due to the constantly changing geometry, economic data transfer is of high importance.

3.2 Relation to prior work

Among existing modeling approaches, physical models, discussed in section 2.3, seem best equipped to handle the general case, responding realistically to forces applied. It is not clear, however, that such models will allow the user to easily set up complicated scenarios like wind-driven oceanscapes. After all, the forces that create such situations have complex causes themselves. Moreover, NSE solvers are still too expensive for real-time simulation, even at modest scales and resolutions.

Purely geometrical models, discussed in section 2.1, are simple, and have been used to produce interesting shapes, such as asymmetric wind-driven waves. Further, such models have proven themselves in fast GPU-based implementations, as seen in section 2.5, where wave components are evaluated and combined directly on the GPU. The affordable frequency resolution is somewhat limited, however, and the problem of composing a natural wave field under given conditions is not well resolved by these techniques alone.

The statistical FFT-based models, discussed in section 2.2, are capable of generating highly realistic wind-driven wave fields, solving the problem of acquiring a natural wave distribution under given conditions. The wave fields are tileable, which allows for arbitrarily large surfaces. For arbitrary views, e.g. if the camera is allowed to move freely, however, distinct artificial periodicity is a problem with such techniques, as noted by the authors. Another drawback is that they do not provide a means of interacting with the surface, to create boat wakes, etc.

The surface optics of water are well understood, and well adapted to real-time computer graphics. Sophisticated lighting techniques, accounting for water volume scattering and global illumination, have been used to achieve impressive visual results, but are computationally intensive and less suited for real-time rendering. In this work, perfomance is prioritized, and a simple lighting model is opted for, focusing on correct surface optics and realistic global reflections. For techniques approximating local reflections/refractions and caustics, handling fairly well in real-time, the reader is referred to work discussed in section 2.4.

Level of detail and hidden surface removal schemes have been extensively studied, as mentioned in section 2.5, and good solutions exist for elevation data rendering, that could also be used here. It is desirable to look further, however, to see what characteristics separate ocean surfaces e.g. from geological terrains, and how this could be exploited.

In conclusion, realistic ocean simulation at certain scales and under certain conditions has been shown possible in real-time. Realistically simulating large-scale open seas, with scattered boating activity, etc., still has associated problems.

3.3 Value

On a globe covered in around 70% ocean, water seems as good an element as any to focus attention to. In computer visualization, the ocean is an important arena. One example is from the offshore industry, where virtual environments have been adopted to improve cross-disciplinary information sharing, decision making, and more.

Convincing natural backdrops not only serve as a benchmark for computer games these days, but are to a higher degree expected elements in professional applications. Apart from being cosmetic selling features, such elements can improve the user's experience and provide intuitive visual cues, e.g. about weather conditions. It is important, however, that the simulation does not interfere with application workflow, and it may well be expected that the application runs smoothly on a standard issue laptop computer.

4 Real-time simulation and visualization of open seas

This chapter describes theory, problems and proposed solutions falling under the scope of the thesis in detail. The discourse is divided into *modeling*, section 4.1, which is concerned with describing the physical structure and motion of oceanic surface waves, and *tessellation*, section 4.2, where the aim is to prepare an appropriate set of rendering primitives from the model.

The final task in the visualization process is to perform realistical lighting, ensuring that triangles are filled with colors that correspond as closely as possible to the optical properties of water. Appendices C and D briefly explain the theory and techniques used to achieve realistic water surface shading in this work.

Details from the implementation of this theory follow in section 4.3, where both an overview of the implemented system and relevant specifics are given. Finally, significant results are discussed in section 4.4.

4.1 Modeling

The open seas are typically dominated by wind-driven waves. Statistical methods supported by oceanographic research, see section 2.2, have perhaps provided the most complete solution to the problem of modeling such scenarios effectively and realistically. As noted by Tessendorf, such techniques have proven themselves in the production of several films, among others *Waterworld* and *Titanic* [Tes04]. The first two sections here (4.1.1 and 4.1.2) follow the principles from Tessendorf's course notes, turning to statistical analysis in the modeling of a basic structure for an ocean surface. The next section (4.1.3) discusses the artificial periodicity that may become a concern with this FFT-based method, and possible countermeasures. The last section (4.1.4) presents a GPU-based solver of the wave equation, as an extension to the wind-driven simulation.

4.1.1 Synthesis

Assume that an unbounded ocean surface can be expressed as the sum of 2D sinusoid waves of different amplitudes, direction, phases and wavenumbers. Thus, the surface can be represented as a spectrum of frequency components. In the discrete case, there exists an efficient algorithm, the Fast Fourier Transform, which computes the sum of wave components given by such a spectrum at $O(N^2 \cdot \log N^2)$ time for an N^2 image. Naïvely computing the sum of N^2 wave components, on the other hand, would result in an asymptote of $O(N^4)$ for an equally sized image.

Stating the surface heights at discrete horizontal positions $\mathbf{x}$ as the inverse Fourier transform of a frequency spectrum, the surface has the form:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t)\, e^{i\,\mathbf{k}\cdot\mathbf{x}}\,, \tag{3}$$

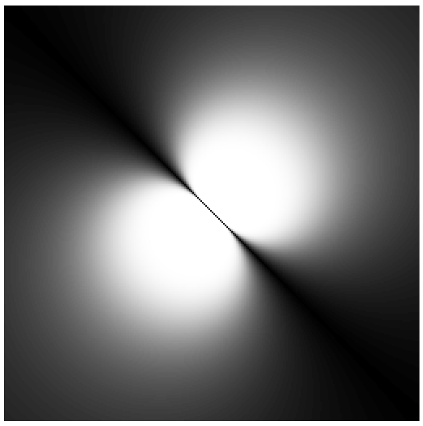
where $\mathbf{k}$ is a *wave vector* specifying a wave by its direction and wavenumber. $\tilde{h}(\mathbf{k}, t)$ denotes a complex entry in the spectrum, whose modulus and argument correspond to the amplitude and phase of wave $\mathbf{k}$, respectively. A time argument t is included for later, since the height map is intended for animation. The slope of the height field can be expressed as:

$$\nabla h(\mathbf{x}, t) = \sum_{\mathbf{k}} i\,\mathbf{k}\,\tilde{h}(\mathbf{k}, t)\, e^{i\,\mathbf{k}\cdot\mathbf{x}}\,. \tag{4}$$

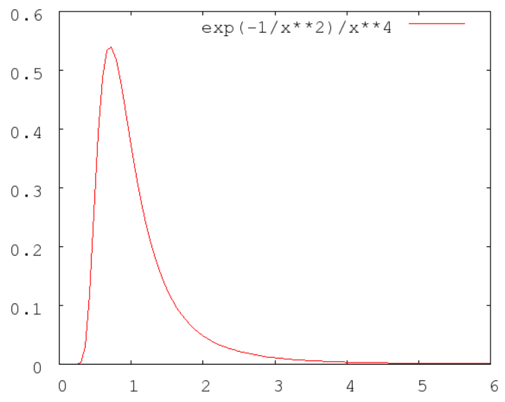
To guide the selection of wave components for a representative ocean surface, the *Phillips spectrum* can be used. This spectrum approximates amplitude variances in empirical ocean spectrums, and is given by:

$$P_h(\mathbf{k}) = A\,\frac{e^{-1/(k\,L)^2}}{k^4}\,|\hat{\mathbf{k}} \cdot \hat{\mathbf{w}}|^2\,. \tag{5}$$

Here, A is a constant used to amplify the wave field, k is the wavenumber of wave $\mathbf{k}$, and $\mathbf{w}$ is a vector denoting the wind over the field. L is the largest possible wave resulting from this wind, and is given by $|\mathbf{w}|^2/g$, where g is the gravitational constant.



(a) frequency spectrum



(b) 1D distribution function

Figure 5: The Phillips spectrum

For the spectrum shown in figure 5 (a), the wind is blowing at a 45° angle. As can be seen in the figure, frequency components with an angle dissimilar to this is suppressed by the cosine factor, $|\hat{\mathbf{k}} \cdot \hat{\mathbf{w}}|^2$, in equation 5. Further, the 1D distribution function, i.e. the factor that depends on k in the equation, acts as a filter that emphasizes lower frequencies. See figure 5 (b). This function can be interpreted as the amplitude variance for a given wavenumber. By equation 5 alone, waves traveling along the wind in both directions are favored. To avoid having waves traveling against the wind, components with $\mathbf{k} \cdot \mathbf{w} < 0$ can be suppressed, e.g. the lower left half of the spectrum in figure 5 (a).

Using this model to generate a scenario with the prescribed statistical properties, the first step is to populate a spectrum with draws from a random number generator. Each frequency component is then assigned values $(\xi_r + i\,\xi_i)$, where ξ_r and ξ_i are independent draws from a probability distribution. As noted by Tessendorf, a gaussian distribution goes well with experimental data, so the standard normal distribution is suitable here. With a standard deviation of 1, the complex random variable can be scaled, i.e. divided by $\sqrt{2}$, so its corresponding amplitude also has a standard deviation of 1. Finally, each component is filtered by the Phillips spectrum, according to its

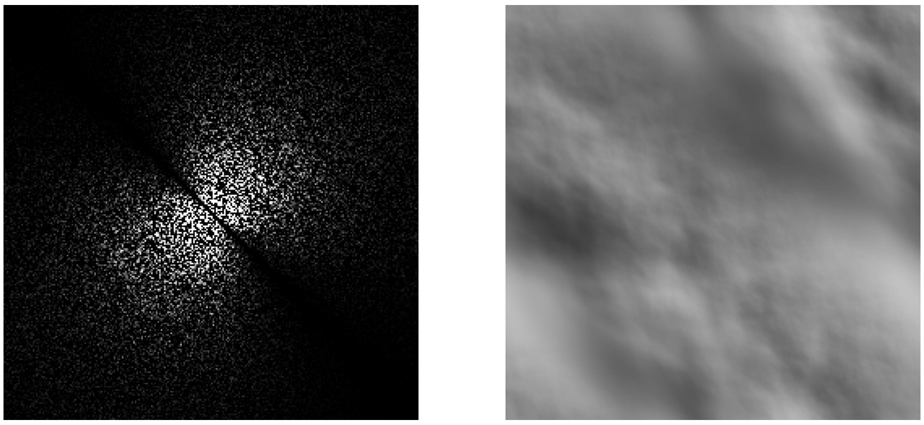

(a) frequency domain (b) spatial domain

Figure 6: Ocean synthesis. (a) shows the magnitudes of a sampled frequency spectrum, and (b) shows the spatial height map synthesized from this spectrum.

wave vector. Thus, the final formula for generating an ocean spectrum is:

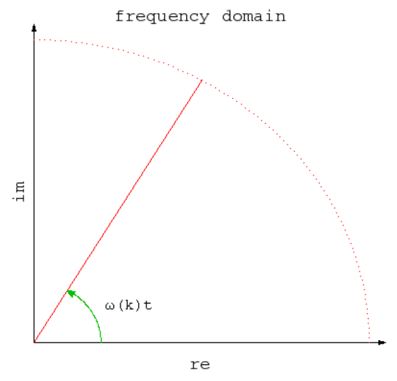
$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}\,(\xi_r + i\,\xi_i)\,\sqrt{P_h(\mathbf{k})}\,. \tag{6}$$

Figure 6 shows an example scenario with a resolution of 256^2, i.e. more than sixty thousand wave components. To summarize, the spectrum (a) is generated using equation 5 and 6, and the corresponding height map (b) is synthesized using equation 3.

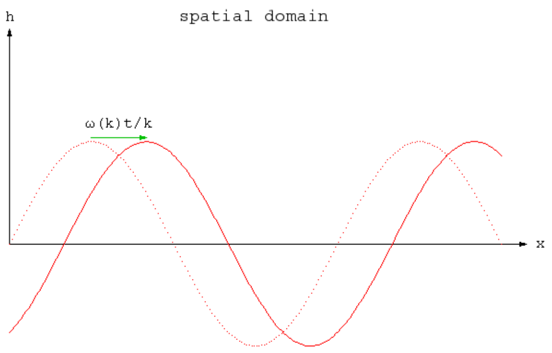
4.1.2 Animation

With a sampled set of wave components, a method for realistically animating these components is needed. A look at how surface waves propagate in water is then in order. Water is a *dispersive medium*, which means that the velocities of waves traveling in water stand in relation to their wavenumbers. Taking a surface wave, in this case a sinusoid, $\sin(\mathbf{k}\cdot\mathbf{x} - \omega\,t + \phi)$, the angular frequency ω for a given wavenumber k must be determined. This is resolved by the *dispersion relation*, which can be stated as:

$$\omega(k) = \sqrt{g\,k\,\tanh(k\,h)} \overset{h\gg 0}{\approx} \sqrt{g\,k}\,, \tag{7}$$



argument change



corresponding phase shift

Figure 7: Animating the phase of a wave **k** in the frequency domain

where g is the gravitational constant and h is the height of the water. For large values of h the hyperbolic tangent factor can be equated with 1, ignoring influence from the sea bottom, and reducing the dispersion relation to $\omega(k) = \sqrt{g\,k}$. The effects of surface tension is also ignored, since the waves considered here are large enough for such effects to diminish.

The relation between velocity c and angular frequency ω in a wave is $c = \omega/k$. Thus, by equation 7, a wave's velocity as a function of its wavenumber is $c = \sqrt{g/k}$, meaning that waves with long wavelengths, or low wavenumbers, travel faster in water than waves with shorter wavelengths.

Using the dispersion relation to animate wave phases, a spectrum generated by equation 6, $\tilde{h}_0$, serves as the initial state of the ocean. Manipulating the argument of a component in $\tilde{h}_0$, without modifying the modulus, corresponds to a phase shift in the spatial domain. See figure 7. Multiplying $\tilde{h}_0(\mathbf{k})$ by $e^{i\,\omega(k)\,t}$ shifts the initial phase of wave $\mathbf{k}$ by $\omega(k)\,t/k$, propagating the wave in accordance with the dispersion relation. The expression

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k})\, e^{i\,\omega(k)\,t} + \tilde{h}_0^*(-\mathbf{k})\, e^{-i\,\omega(k)\,t} \tag{8}$$

additionally preserves the complex conjugation property $\tilde{h}^*(\mathbf{k}, t) = \tilde{h}(-\mathbf{k}, t)$, which guarantees a real-valued image in the spatial domain. Using equation 8 and 3, expressions for an animated height map $h(\mathbf{k}, t)$ that are only dependent on the initial spectrum $\tilde{h}_0(\mathbf{k})$ are now obtained. Not being dependent on

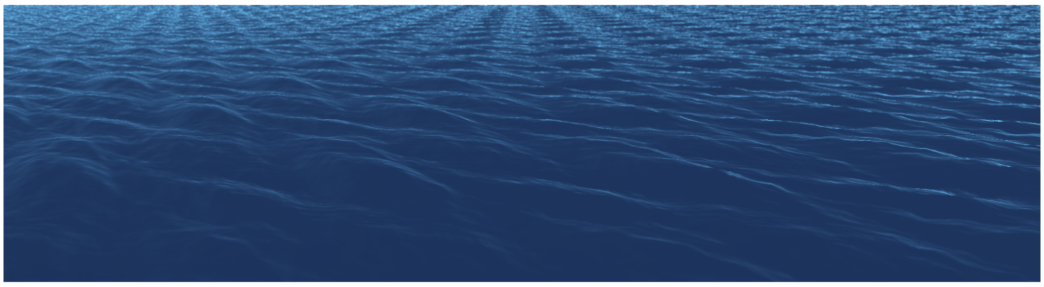


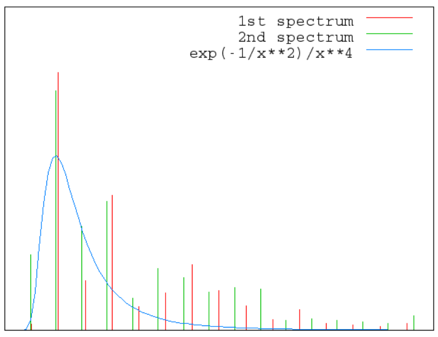
Figure 8: Realism disrupted by repeating patterns

previous time steps, the time resolution of the model can be controlled freely during simulation.

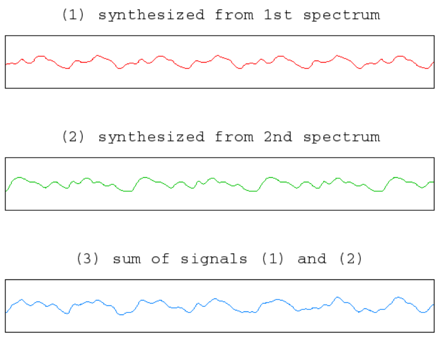
4.1.3 Overcoming artificial periodicity

The height maps acquired by the above methods, such as the spatial map in figure 6, are ultimately intended for displacement mapping of a 3D surface. Since maps synthesized by equation 3 are spatially periodic, these maps tile seamlessly, enabling an unbounded surface. This periodicity is apparent when large regions of a tiled height field is visible, however, posing an unnatural look on the surface. Figure 8 illustrates the artificial periodicity introduced when several tiles are concurrently visible over a rendered ocean surface.

According to Tessendorf, map resolutions of up to 2048^2 were used in the production of *Waterworld* and *Titanic*. Since an inverse Fourier transform is performed at every time step of the animation, however, desired frame rates can only be expected to emerge at lower resolutions in real-time, say at 256^2 or below. The scene in figure 8, for example, was rendered at real-time rates using a resolution of 128^2. The higher the map resolution, the larger surface regions could the tiles be scaled across, without losing too much of higher frequency details on the surface. Thus, artificial periodicity is particularly problematic in real-time situations, where small maps are necessary. By scaling up the tiles, the surface assumes a less detailed look, while by keeping a small scale, repeating patterns are more easily noticed.



(a) interleaved spectra



(b) synthesized signals

Figure 9: Interleaved spectra. (a) shows two spectra (magnitudes shown here) following the same continuous distribution, both with 16 frequency components. (b) shows resulting spatial signals: the upper two signals are synthesized from the spectra in (a). Both have more than three spatial periods visible in the shown range. The bottom signal, which is the sum of the above signals, has a much longer period.

One possible solution to this problem is to decompose the ocean spectrum into two, or more, interleaved spectra. See figure 9. The three signals shown in figure (b) have similar characteristics, as guided by the underlying statistical model, but the bottom signal has a much longer spatial period. By deploying multiple discrete ocean spectra whose wave components are sampled at different intervals, with a relatively large least common multiple, a linear combination of the resulting spatial maps will have a much longer period. The scene in figure 10 was rendered using two interleaved spectra with resolutions of 128^2. Compare to figure 8. Using this technique, the inherent periodicity in the wave field is much less obvious, without excessively adding to the computational complexity.

To correctly account for wave dispersion, the spectra must be animated separately, in accordance with the dispersive relation. Simply modifying the horizontal scales of the wave fields would lead to incorrect dispersion between waves from the different maps in a combined wave field. Moreover, the statistical properties of the combined field would no longer follow the

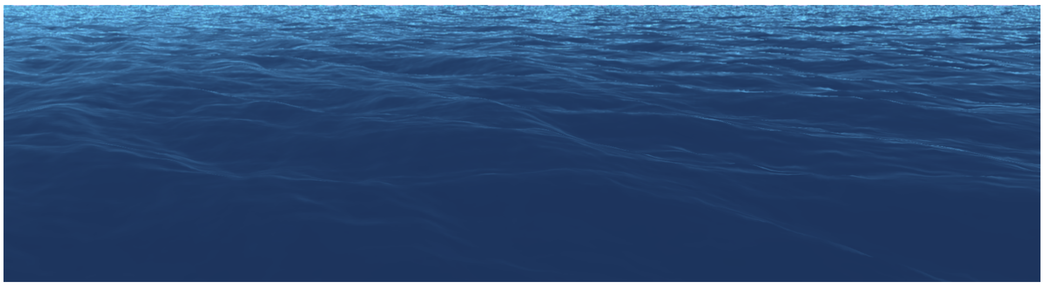


Figure 10: Using two combined wave fields to avoid artificial periodicity

desired distribution. By separately sampling and animating the spectra, accounting for the scaled wavenumbers, the resulting spatial maps can be scaled and combined with correct relative wave speeds, following the governing statistical model.

See section 4.3.2 for implementation details.

4.1.4 A GPU-based water solver

The model so far is able to reproduce wind-driven oceans quite realistically, when nothing else is interfering with the water's development. Also interesting are interactions with water, e.g. by user-supplied forces. As Mitchell suggests, a synthesized wave field could be composited with arbitrary waveforms, opening opportunities for shore interactions, boat wakes, etc [Mit05]. If physical accuracy has lower priority than computational cost and interactive frame rates, a fast water solver could run in parallel with the FFT-based simulation, and used to add complexity to the underlying structure in regions of interest.

One equation that can be used to simulate a water surface quite efficiently, is the 2D *wave equation*, which has the form:

$$\frac{\partial^2 h}{\partial t^2} = c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) . \tag{9}$$

Here, h is the surface height at location (x, y), and c is the wave speed. Assuming constant depth, this equation coincides with the linearized shallow

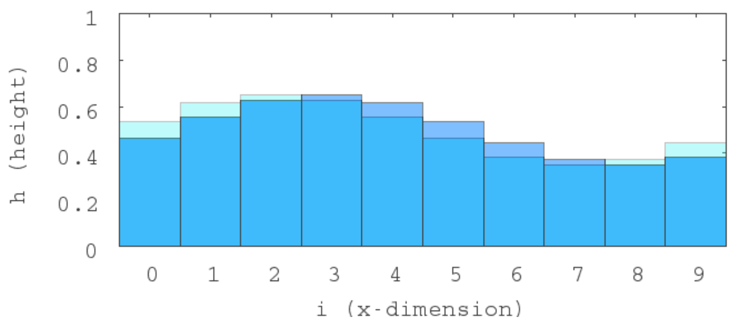


Figure 11: The wave equation approximated on a finite difference grid. A one-dimensional slice of water heights is shown here, with faint areas showing differences from the previous state.

water equations, as addressed by Kass and Miller [KM90]. See equation 2, section 2.3. To plausibly model water movement, the waves need to undergo some form of damping as they propagate. This can be achieved by introducing a dissipative term to equation 9, bringing the wave equation to the form:

$$\frac{\partial^2 h}{\partial t^2} - k\frac{\partial h}{\partial t} = c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) , \tag{10}$$

where k is a positive damping constant.

In the following, a discrete analogue of equation 10 is used to model water surfaces subjected to deflection. The applied formulation is described in the work of Nishidate and Nikishkov, who solve the wave equation with damping on the CPU [NN05]. In this implementation, the GPU is employed as the computational kernel in the time integration, using textures as render targets, as Krüger and Westermann [KW05]. Further, hardware accelerated rasterization is exploited to efficiently supply the algorithm with input, i.e. arbitrary forces and obstacles. With anti-aliasing conveniently available, this can be used to provide smooth deflections and connected paths, e.g. to produce boat wakes. Figure 12 shows two real-time scenarios, where smooth wake-like waves are achieved by letting the user paint deflective paths into the framebuffer using anti-aliased lines.

Discretized in time and space, the wave equation can be approximated using a finite difference scheme. See figure 11. On a uniform 2D grid, equally spaced in both dimensions ($\Delta x = \Delta y$), the discrete analogue of equation 10 is:

$$0 = \frac{h_{i,j}^{t+\Delta t} + h_{i,j}^{t-\Delta t} - 2\,h_{i,j}^{t}}{\Delta t^2} - k\,\frac{h_{i,j}^{t-\Delta t} - h_{i,j}^{t}}{\Delta t} - \\ c^2\,\frac{4\,h_{i,j}^{t} - h_{i+1,j}^{t} - h_{i-1,j}^{t} - h_{i,j+1}^{t} - h_{i,j-1}^{t}}{\Delta x^2}\,,$$

where $h_{i,j}^{t}$ is the water height in grid cell (i,j) at time t. The unknown term here is $h_{i,j}^{t+\Delta t}$, i.e. the water heights at the next time step of the simulation. A regrouping yields:

$$h_{i,j}^{t+\Delta t} = h_{i,j}^{t} + (1 - k\,\Delta t)\,(h_{i,j}^{t} - h_{i,j}^{t-\Delta t}) + \\ \frac{\Delta t^2\,c^2}{\Delta x^2}\,(4\,h_{i,j}^{t} - h_{i+1,j}^{t} - h_{i-1,j}^{t} - h_{i,j+1}^{t} - h_{i,j-1}^{t})\,. \tag{11}$$

The height map at time $t + \Delta t$, as given by equation 11, relies on the two previous states of the simulation, h^t and $h^{t-\Delta t}$. While simple, this explicit form is only stable for sufficiently small values of Δt, as noted by Nishidate and Nikishkov. Specifically, no wave should travel more than one cell in a single time step. Thus, there is not too much freedom in controlling the time resolution. The implicit scheme used by Krüger and Westermann is unconditionally stable, allowing for longer time steps, though not as simple.

Disregarding the term containing $h_{i,j}^{t-\Delta t}$ in equation 11, the form is similar to a spatial smoothing filter. If four additional neighboring cells $h_{i\pm1,j\pm1}^{t}$ are considered in the calculation of $h_{i,j}^{t+\Delta t}$, the height map will tend to have a smoother look, typically making it more suitable for displacement mapping.

See section 4.3.4 for implementation details.

4.2 Tessellation

The representation of the ocean surface is so far in the form of elevation maps, i.e. discrete height fields, from the methods of section 4.1. To render the surface in three dimensions, this representation should be translated into a

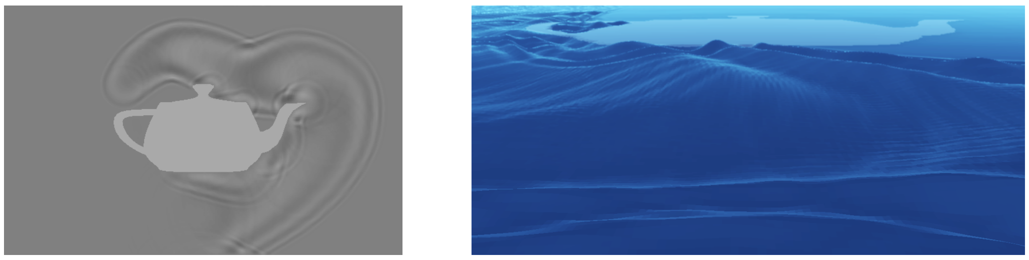

(a) height map (b) 3D surface

Figure 12: 2D water solver. (a) A height map resulting from the simulation, with the Utah teapot rendered into the map as an obstacle. In this example, a map resolution of 640×400 was used. (b) A similar height map used to displace a 3D mesh.

set of rendering primitives that can be efficiently rasterized. Modern graphics hardware is primarily optimized for polygon rendering, and triangular meshes are very common 3D representations. Preparing such a representation, the task is to find good configurations of triangles that preserve relevant details, yet avoid wasteful use of geometry.

A common problem with static polygonal representations is situations where lots of polygons end up contributing very little to the final image. Complex objects viewed from afar may reduce to a few pixels, and geometry that is occluded by some other part of the scene, facing away from the camera, or simply outside the field of view, ends up with no contribution but a reduction in performance. Level of detail (LOD) and hidden surface removal schemes address these problems, using view dependent mesh resolutions and culling heuristics. Much research has focused on designing such techniques specifically for height fields, e.g. elevation data from terrains. Thus, since the water surfaces under consideration here are regarded as height fields, there exists several techniques that could be used for these purposes.

Apart from the height field trait, there are further properties of ocean surfaces that could be considered in the search for a suitable tessellation scheme. Compared to typical terrain scenarios, the sea has rather monotonic height variations. Sea surfaces are rather flat, while landscapes tend to exhibit more

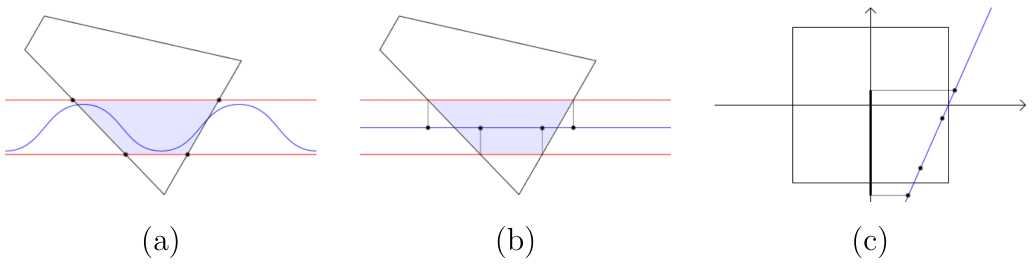


Figure 13: Computing a screen-aligned bounding rectangle, whose projection onto the ocean plane is guaranteed to cover the visible ocean region, even after displacements are applied. 2D simplification: (a) intersection between the viewing frustum and the bounding volume of the ocean surface, in world space. (b) resulting vertices projected onto the ocean plane. (c) after a transformation into normalized device coordinates a bounding rectangle is obtained, discarding the depth of the intersections (horizontal axis).

irregular features. Hand in hand with this comes the fact that very large parts of ocean scenes are often concurrently visible, and yet another side of it is that only small regions are typically occluded at a time, making it harder to take advantage of self-occlusion. A final characteristic that should be kept in mind is the periodic nature of the underlying FFT-based height maps.

In this section, the concept of projecting a uniform grid from the view plane and onto the ocean plane is explored. The purpose is to achieve a continuous level of detail, providing coarser tessellation with increasing distance from the camera, and at the same time minimize geometry outside the field of view. This technique was introduced by Johanson, and implemented on the CPU [Joh04]. It is desirable to do this on the GPU, however, to avoid a continuous transfer of vertices to the GPU, and to refrain from transformations on a per vertex basis on the CPU. Chiu and Chang implemented such a scheme on the GPU, but few implementation details and results are given [CC06].

Consider the ocean plane and the two parallel planes that touch the highest and lowest possible surface deflections, respectively. Between the upper and lower planes lies an unbounded volume containing the ocean surface. The first step in the tessellation scheme is to identify the intersection between the viewing frustum and this volume. This can be done by transforming the eight corners of the frustum from clip space to world space, and intersect the twelve corresponding edges with both the upper and lower planes. See figure 13 (a). Denoting the xy–plane in world space as the ocean plane, the upper and lower planes are defined by the normal vector $(0, 0, 1)$ and their distance from the ocean plane $\pm D$, where D is the largest possible displacement. The corners of the viewing frustum in world space are $(\mathbf{P}\cdot\mathbf{V})^{-1}\cdot(\pm1, \pm1, \pm1, 1)^T$, where $\mathbf{V}$ is the view transformation and $\mathbf{P}$ is the projection transformation.

Now, by projecting the intersection onto the ocean plane, a minimal region of interest is defined. This region is just large enough to ensure that no gaps are introduced between the ocean surface and the viewing volume after displacement mapping. The next step is thus to project the vertices resulting from the intersection onto the ocean plane. See figure 13 (b). If less than three vertices in both the upper and lower planes were found, however, the ocean is outside the view and needs not be rendered. By further transforming the points to normalized device coordinates, a screen-aligned bounding rectangle for the visible ocean region can be obtained, simply by identifying the minimum and maximum x– and y–coordinates. See figure 13 (c).

The final step of the scheme is to project a uniformly spaced rectangular grid from the view plane and onto the ocean plane. See figure 14. This can be done by projecting the corners of the bounding rectangle, given by the previous steps, onto the ocean plane, then perform bilinear interpolation between the resulting points, using homogeneous coordinates. The corners are given in normalized device coordinates, so the projected points can be found by transforming the line defined by $(x, y, \pm1, 1)$ for each corner (x, y) to world space, then intersect these lines with the xy–plane, also using homogeneous coordinates.

As Johanson points out, there are cases when the direction of the camera leads to instability when computing the projected grid, e.g. when the camera points away from the ocean plane or is positioned inside the bounding volume. Johanson suggests computing the projection with a second camera,

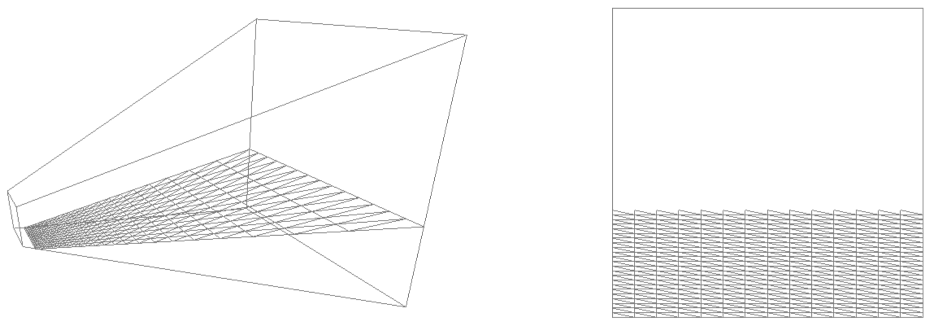

(a) seen from outside the frustum (b) seen from the projecting camera

Figure 14: A projected grid. (a) tessellation seen from a second camera. Notice the increased vertex spacing in the far end of the viewing frustum. (b) tessellation seen from the projecting camera. Notice the uniform spacing of vertices.

a projector, which can be aimed slightly differently than the viewing camera, using simple heuristics to avoid such cases. More can be read about this in Johanson's work.

Computing the four projected corner points is performed once per frame, and is a suitable task for the CPU. Interpolating between them to position the grid, however, is performed on a per vertex basis, and should be handled by a vertex processor. By feeding the GPU with the corner positions, the bilinear interpolation could be implemented in a vertex shader, where subsequent displacement mapping and transformations are also applied. This not only frees up the CPU, but avoids the continuous transfer of grid positions to the GPU, since a cached static grid can now be used. Assuming grid points in the range $[0..1, 0..1]$, vertex positions can be used as interpolants in the vertex shader.

After placing a vertex in the ocean plane, its position may be used to look up a displacement from a texture map. The FFT-based displacement maps describe tiled height fields in world space, and so, any position may be used to address a displacement, with texture coordinate wrapping enabled. If the height field is decomposed into two or more different scales, as suggested in section 4.1.3, multiple displacements are looked up with differently scaled texture coordinates. The result is a seamless unbounded surface, and a tesselation that follows the visible ocean region. Moreover, the resolution is close to uniform in post-perspective space, meaning that regions near the camera automatically receive a high resolution in world space, and conversely for farther regions.

See section 4.3.3 for implementation details.

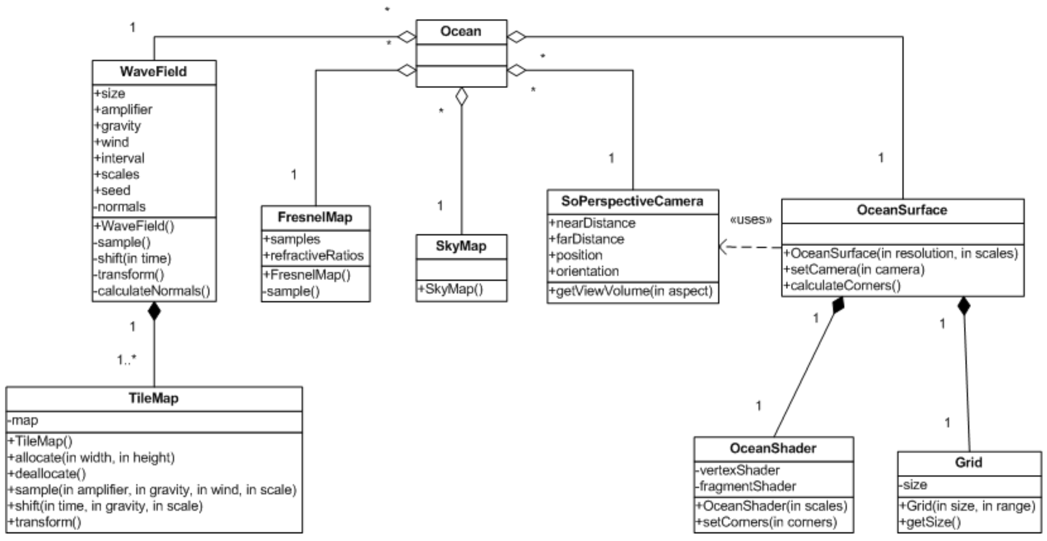


Figure 15: Class diagram for the implemented Coin3D nodes.

4.3 Implementation details

The techniques described in section 4.1 and 4.2 were implemented in *C++*, and abstracted as scene-graph nodes, using *Coin3D*. Coin3D is a high-level scene-graph library build around *OpenGL*, whose API is compatible with *TGS*'s *Open Inventor*. More information about the Coin3D-library is found in appendix B. Additionally, OpenGL was used for more low-level operations, and the *OpenGL Shading Language* was used to program vertex and fragment shaders.

4.3.1 Overview

Figure 15 shows the most important classes that have a part in the ocean surface simulation and visualization. The top class, *Ocean*, is a simple container node for the ocean related classes, used for grouping the nodes as an entity in a scene-graph. The classes *WaveField* and *TileMap* handle the sampling, animation and synthesis of wind-driven waves field maps, which are described in sections 4.1.1 to 4.1.3. The *FresnelMap* class generates a texture containing Fresnel reflectance approximations as functions of refractive

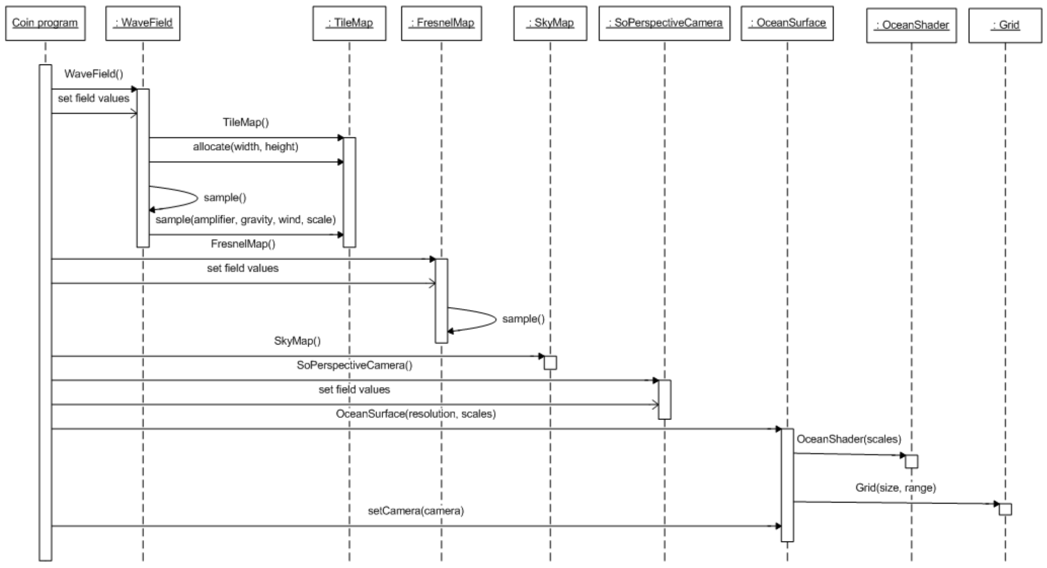
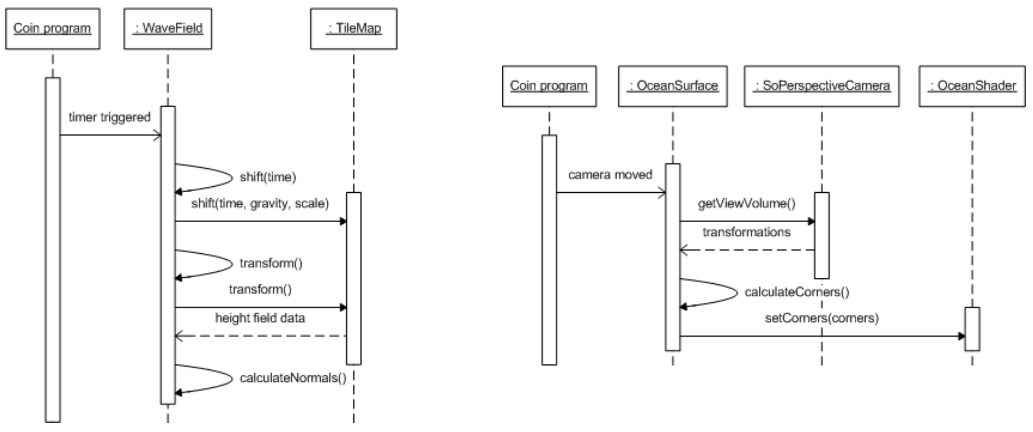


Figure 16: Sequence diagram for the setup stage of an ocean scene.

indices and the incidence angle. See appendix C. The *SkyMap* class loads images of a natural environment, and sets up a texture cube map for environment mapping. See appendix D. The *SoPerspectiveCamera*, a part of the Coin library, is important to the *OceanSurface* class, which uses information about the view volume to determine how the currently viewed region should be tessellated. The *OceanShader* class is a container for the shader program, which handles the final positioning of the grid and the surface color shading on the GPU. The *Grid* class is responsible for generating a triangle strip set, forming a uniform rectangular mesh that can be used in the tessellation scheme.

Figure 16 shows in sequence how an ocean scene is set up before the main program loop. It should be mentioned here that *fields* are the main mechanism for manipulating nodes in Coin. Changes to a node field are automatically detected by the Coin system, which uses this to determine what OpenGL instructions need to be updated and cached before continuing with rendering, and to notify nodes that have a registered interest in another node's state. The ocean setup stage consists of instantiating and configuring the needed nodes, and adding them to a container. First, a WaveField node is



(a) triggered by a timer (b) triggered by camera motion

Figure 17: Both diagrams describe sequences that are part of the main loop in an ocean scene. (a) Height field animation is controlled by a timer that fires at specified intervals. (b) Tessellation updates are triggered by changes in the camera's parameters.

set up. This node allocates the desired number of TileMap instances, and initial spectra are sampled using the given parameters. FresnelMap, SkyMap and camera nodes are configured and added to the scene, and finally, an OceanSurface instance is set up with a given grid resolution and TileMap scales. This object creates an OceanShader node, which is informed about the height map scales, and a grid with the specified resolution and vertices in the range $[0..1, 0..1]$.

After the setup stage is completed, the data generated by the FresnelMap, SkyMap and Grid instances are cached. Since the data does not change, the system does not need to update this during the course of the simulation.

Figure 17 shows sequences that control animation and teseelation, which are part of the program's main loop. Animation is shown in figure (a), where the height field from the FFT-based simulation is updated. The WaveField node is notified about changes in a timer node, which is set to fire at certain intervals. This triggers phase shift, inverse Fourier transform, and normal map calculation. In figure (b), the OceanSurface node is notified about changes in the camera node. The OceanSurface node then obtains information about the view volume from the camera, calculates the corners of the projected grid in world space. The corners, which are used to position the grid vertices during vertex shading, are finally passed to the OceanShader. See section 4.2.

After these sequences are completed, Coin will have detected changes in the WaveField and the OceanShader nodes. Upon rendering, all modified nodes are invoked. The WaveField node then updates the texture maps that contain height field data, and the OceanShader node updates the vertex shader with the new grid corners.

4.3.2 WaveField and TileMap

The *WaveField* node is responsible for generating animated height maps, as described in section 4.1.1 and 4.1.2, and corresponding normal maps. With the technique described in section 4.1.3 in mind, this node is designed to operate with multiple spectra, where each spectrum is handled by a TileMap instance. The fields of this node, seen in figure 15, are: *size*, which determines the resolution of the height maps; amplifier, gravity and wind, which are parameters for the statistical wave field model; *interval*, which specifies the interval at which wave field updates are triggered; and *seed*, which is the initial value handed to the random number generator prior to the sampling of frequency components.

With two spectra, which was mostly used during testing, two TileMap instances are created, generating two separate sets of data. The WaveField node packs this data into two 2D textures, for displacement mapping, and calculates two corresponding normal maps, which are stored as separate slices in a 3D texture for normal mapping. The displacement maps are stored as 32-bit precision floating point textures with nearest neighbor filtering, since this is typically the only type supported by hardware for vertex texture fetches. The normal maps are only needed in the fragment shader, and are set up with linear filtering.

The *FFTW* library was used to perform inverse FFT on the CPU. This could be replaced by any other FFT-library, or, as Mitchell suggests, a GPU-based implementation of the FFT [Mit05].

4.3.3 OceanSurface and OceanShader

The OceanSurface instance calculates the intersection between the view volume and the ocean volume each time the camera node changes. See section 4.2. If no intersection is found, the rendering of the ocean surface is deferred. If an intersection volume is found, four appropriate corners for the projected grid are calculated and passed to the OceanShader node.

The vertex shader operates on the static grid generated by the Grid class, and assumes that incoming vertices are in the range $[0..1, 0..1]$. The vertices are positioned in the ocean plane by interpolating between the specified corner vertices. Thus, the world space position of a vertex prior to displacement is given by:

```
gl_Position = c[0] * (1.0 - x) * (1.0 - y) + c[2] * x * y +
              c[1] * x * (1.0 - y) + c[3] * (1.0 - x) * y;
gl_Position /= gl_Position.w;
```

where $c[0..3]$ are homogeneous coordinates specifying the corners of the viewed region, and (x, y) is the position of the incoming vertex.

The vertex position is further manipulated by displacing its z–coordinate. Displacements are looked up from the available displacement maps, using the new world space position, (x, y), multiplied with the scale specified for each height map. Since graphics cards commonly support only nearest neighbor filtering for floating point textures, a function *linear_lookup* is implemented, which looks up the four nearest texels for a given texture coordinate, and returns the interpolated value. Thus, for each available displacement map $displacements[i]$, the instructions:

```
gl_Position.z += linear_lookup(displacements[i], scale[i] * gl_Position.xy);
```

are called. The vertex is finally transformed to clip space.

In the fragment shader, the first step is to look up a normal vector from each slice of the 3D normal map. As in the vertex shader, the texture coordinate is scaled to match the specified scales of the corresponding spectra. It is assumed that no model transformation is needed for the ocean surface, since wind direction and scale can be controlled by setting the perameters of the WaveField node. Thus, the average of the looked up normals is used directly in the lighting calculations.

Next, the view vector is calculated, and reflectance is looked up from the map containing the Fresnel function, using $\mathbf{n} \cdot \mathbf{v}$ as the texture coordinate, where $\mathbf{n}$ is the normal vector and $\mathbf{v}$ is the view vector towards the camera. Reflections from the global environment are looked up from a cube map, using the reflected view vector as the lookup argument. Optionally, Phong shaded sun light is added to the reflective color, before the color is mixed with a refractive color, using the reflectance coefficient to control the blending. A static bluish reflection is used here, but, as mentioned earlier, the cube map could also be used to store refractive colors. Finally, the calculated color is written to the output register.

4.3.4 Solver

The solver examples were set up using the libraries *GLEW* and *GLUT*, as well as a class, *FramebufferObject*, for handling framebuffer objects. See appendix B for information about these resources.

The sequence diagram in figure 18 shows how a water solver simulation is initialized, how it receives input, and how it updates the animated height map from time step to time step. During the initalization stage, a Solver object and a Shader object is instantiated. The Solver object, which runs on the CPU, acts as an interface for the shader, which runs on the GPU. At the end of the initalization stage, the simulation is set up with the desired parameters, and the simulation buffer is filled with inital data, usually monotone data

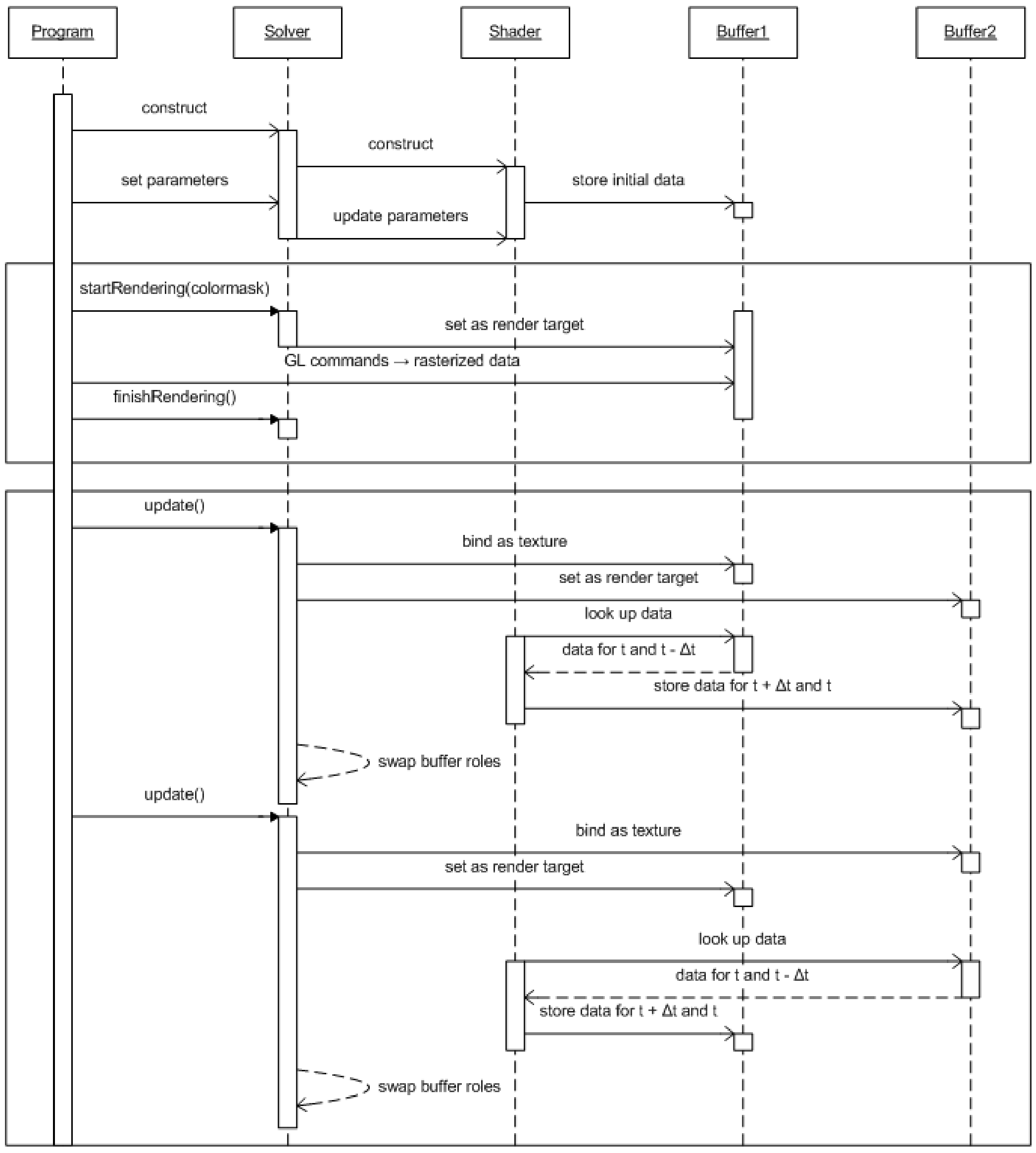


Figure 18: Sequence diagram for a program using the water solver. The diagram shows the initialization stage, and input and update cycles, which are outlined by the upper and lower frames, respectively.

describing a water surface at rest.

The data are held in two buffers in texture memory. One buffer holds the data from the previous state of the simulation, a *source* buffer, and the other buffer is filled information about the new state as updated values are computed, a *destination* buffer. The buffers swap roles at the end of each time step. The data are organized in 32-bit 4-channel textures as follows (with reference to equation 11):

Channel 0 - heights from the previous state, i.e. $h_{i,j}^t$

Channel 1 - the difference in heights from the two previous time steps, i.e. $h_{i,j}^t - h_{i,j}^{t-\Delta t}$

Channel 2 - quantities describing user-supplied forces

Channel 3 - information about the environment

Channel 1 can be interpreted as the vertical speed in each cell at a given time. Channel 2 lets the user specify accelerations, which will change the vertical speed in corresponding cells over a few iterations of the simulation. This seems to produce better results than simply applying deflections directly and abruptly. In this implementation, channel 3 is interpreted as a boolean value, specifying whether the cell is occupied by a surface obstacle or not. If the mathematical model was extended to account for depth, it would be natural to store depth data in this channel.

When the main loop starts, the input and update cycles run as separate processes. The input cycle is invoked when the Solver objects receives a *startRendering* call from the program. This lets the user modify the active source buffer using OpenGL calls, e.g. glLine or glPoint, before new values are computed. An optional argument, *colormask*, lets the user specify which channel(s) will receive the input.

During the update cycle, the new state of the simulation is computed. The fragment shader program is the workhorse here. The iteration is executed by first adjusting the viewport to ensure on-to-one pixel to texel mapping, and then draw a quadrilateral which covers the viewport exactly. For each fragment of the rasterized quadrilateral, the shader looks up the corresponding texel and its neighboring texels, updates the heights, velocities and accelerations, and writes the new values to the destination buffer. After the output cycle is completed, this buffer can be bound as a texture and be used in displacement mapping. In fragment shading, normals are calculated by looking up the finite difference between neighboring cells.

4.4 Results

The proposed techniqes were tested both isolated and in combonation, with a focus on experimenting with parameters that are critical to performance, e.g. frequency resolution and grid resolution. The first section here examines how the tiled wave fields perform, focusing on the technique for avoiding artificial periodicity over large regions, described in section 4.1.3. The next section tests the wave equation solver, described in section 4.1.4, and how this might perform as a part of a larger oceanic scene. The last section examines the visual quality and performance provided by the tesselation scheme described in section 4.2.

The tests were executed on a laptop PC, with the following relevant specifications:

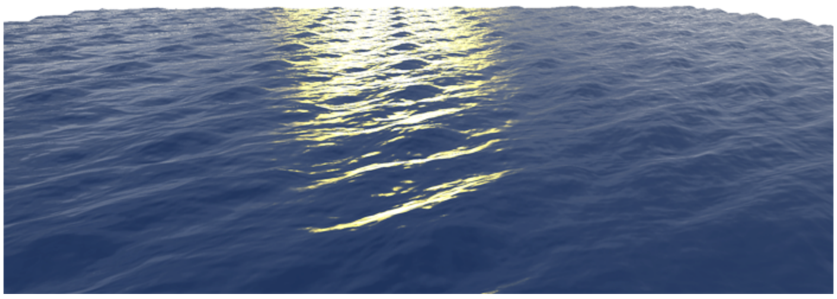
CPU - Dual core, 2 GHz, with 2046 MB of RAM

GPU - NVIDIA GeForce Go 7900 GS, with 256 MB memory

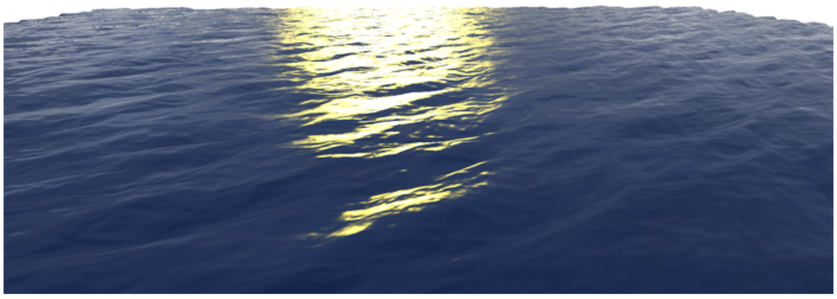
4.4.1 Wave field tiling

Figure 19 shows the visual difference, and the difference in performance, for simulations running with one and two spectra, respectively. Two spectra means more CPU-processing, lead to a doubling of data transfer, and require twice the number of texture fetches, compared a sole spectrum. The increase in CPU usage, and cut in frame rate does not seem discouraging, however, considering that the two tiles are sufficient to describe the structure of an ubounded ocean surface.

The tests shown in figure 19 uses frequency spectra of 128^2 components, which might be excessive if performance has a higher priority than visual detail. Figure 20 shows the visual results for simulations with lower frequency resolutions. The corresponding performance measures are:

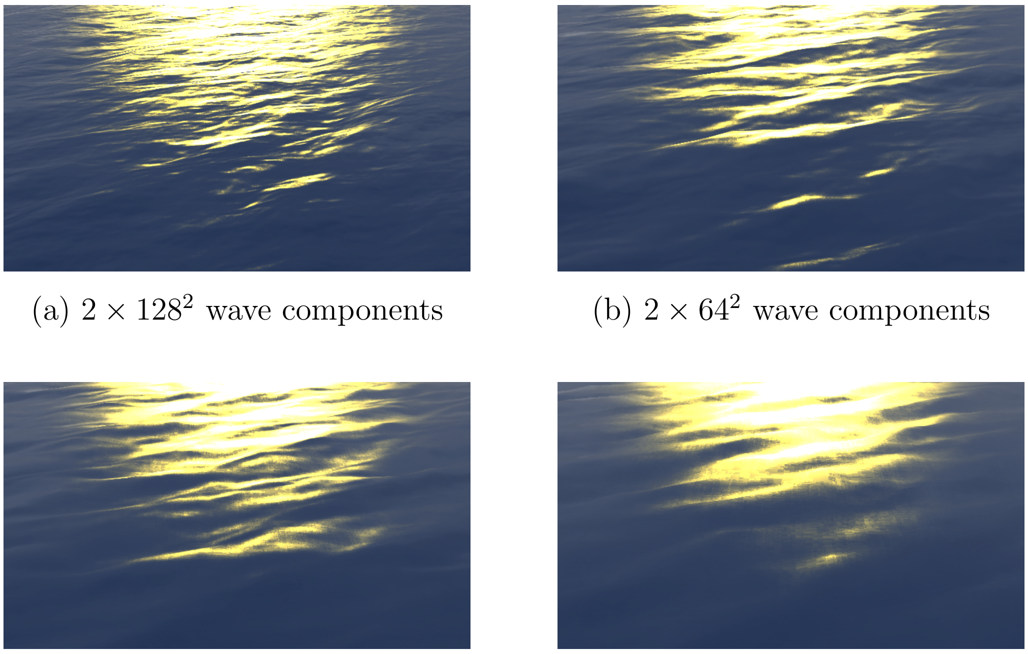


(a) single spectrum, 60 fps, 30% CPU



(b) two spectra, 45 fps, 35% CPU

Figure 19: One versus two spectra. Both scenes were rendered from the same camera location, with a uniform grid of 10.000 vertices. (a) Using one spectrum of 128^2 components. The animated height field tiles 10 times across the grid in each direction. (b) Using two spectra with a total of 2×128^2 components. The two height field decompositions repeats 10 and 6.77 times across the grid, respectively.



(a) 2×128^2 wave components

(b) 2×64^2 wave components

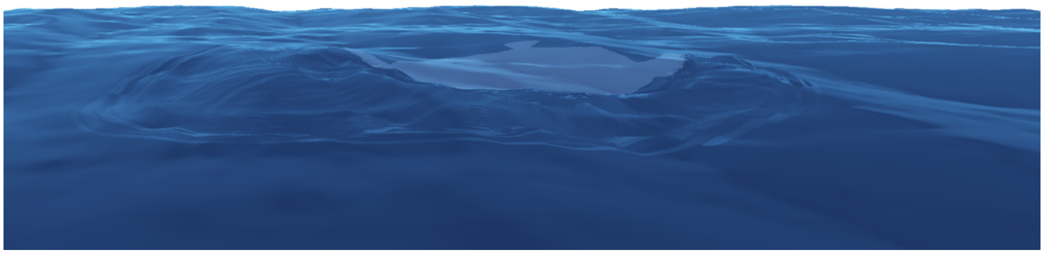
(c) 2×32^2 wave components

(d) 2×16^2 wave components

Figure 20: Four scenes with different frequency resolutions, all using two spectra.

# components	FPS	CPU
2×16^2	105	20%
2×32^2	100	25%
2×64^2	80	30%
2×128^2	45	35%

The frame rate drops dramatically above 2×64^2 components. Though the scene in figure 20 (d) is rendered at a high frame rate, it seems a bit lacking in detail. The scene in (b) looks much more visually pleasing, and performs quite well, with almost double the frame rate of scene (a). Figure 19 showed a cut in frame rate of about 25% with two spectra, compared to one. Going down to spectra of 64^2 components, however, the cut in framerate is less than 10% with two spectra.


(a) filled polygons

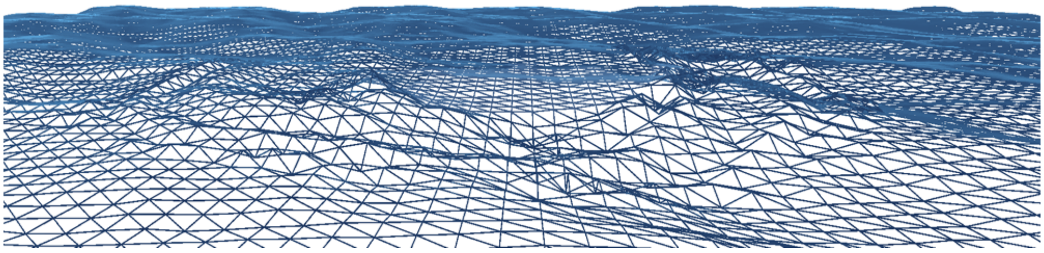

(b) wireframe

Figure 21: The wave equation solver in combination with the FFT-based simulation. The scene was rendered at 130 fps, using a uniform 128×128 geometric grid, 64×64 wind-driven wave components, and a resolution of 512×512 for the wave equation solver.

4.4.2 Wakes and obstacles

The GPU-based 2D wave equation solver was tested against a CPU-based equivalent, to examine the advantages of a GPU-accelerated implementation. The following performance was measured when animating height maps of the given resolutions:

resolution	CPU	GPU
256^2	215 fps	600 fps
512^2	55 fps	220 fps
1024^2	14 fps	60 fps

The CPU usage lies at about 25%, in all cases, for the GPU-based solver, while the corresponding percentage is 100% for the CPU-based solver, as could be expected.

The scene in figure 21 shows the GPU-based solver in combination with a wind-driven wave field. The height maps were combined by simple superposition, with the solver texture positioned in the center of the grid, and the wind-driven wave field tile repeating across the entire grid. The simulatoin was not tested on a very large scale, but the results seem to indicate that interesting complexity and interactions can be added in confined regions of a scene, without taking too large a bite out of performance. This could serve as a nice option if the viewer is examining a particular area closely, and if available application resources are detected.

4.4.3 Tessellation

The tessellation scheme described in section 4.2 was first tested with no simulation running, to compare the GPU-based implementation with a solely CPU-based implementation. The following differences in performance were measured at given mesh resolutions:

# vertices	CPU	GPU
2^{10}	245 fps	710 fps
2^{11}	130 fps	460 fps
2^{12}	70 fps	280 fps
2^{13}	10 fps	160 fps

Rendering was constantly triggered in these examples, and the CPU (one of the CPUs in the dual core) pushes close to a 100% with both implementations. Since the GPU handles the vertex positioning, and the grid remains cached, i.e. no data transfer, much higher frame rates can be achieved with the GPU-based implementation, as indicated by the results.

Further testing showed that the mesh resolution must be quite high to avoid swimming artifacts as the camera moves over the ocean surface. These are seen as slight flickering in the mesh geometry, due to the fact that the vertices

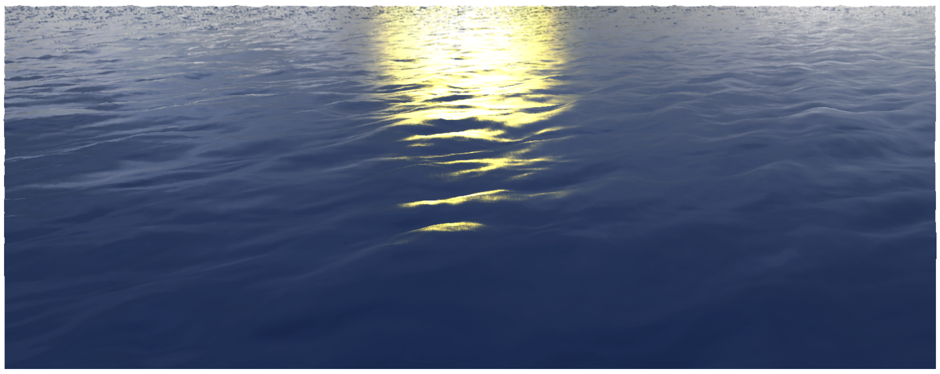


Figure 22: A tessellated surface with 129^2 vertices, or 2^{15} triangles. The height field is composed of two frequency spectra, both of 64^2 components. The scene was rendered at 42 frames per second.

constantly change positions in world space, which the height field does not, revealing that there is more information in the height data than resolved by the mesh. This is particularly noticeable if the waves have large amplitudes or features that are not as smooth, and if the screen resolution is high, requiring very fine tessellations. Thus, optimized vertex processing is crucial for the useability of this technique.

Figure 22 shows a scene rendered with $2^(15)$ triangles, and the corresponding performance. With such a grid resolution, the geometric artifacts may still be noticeable towards the horizon, depending on the position, orientation and movement of the camera. Since there is more information in the far regions of the scene, it might be better not to aim for an as close to uniform grid resolution in post-perspective camera space as possible, but to allow for somewhat higher vertex counts towards the horizon of the viewed region. By experimentally pushing vertices towards the far plane, less swimming artifacts were observed. This was done by exponentiating the y–interpolant in the vertex shader before the vertex position is calculated. A resulting tessellation can be seen in figure 23, which has less superfluous triangles close to the camera than a projected uniform grid, and more well resolved height data towards the horizon of the scene.

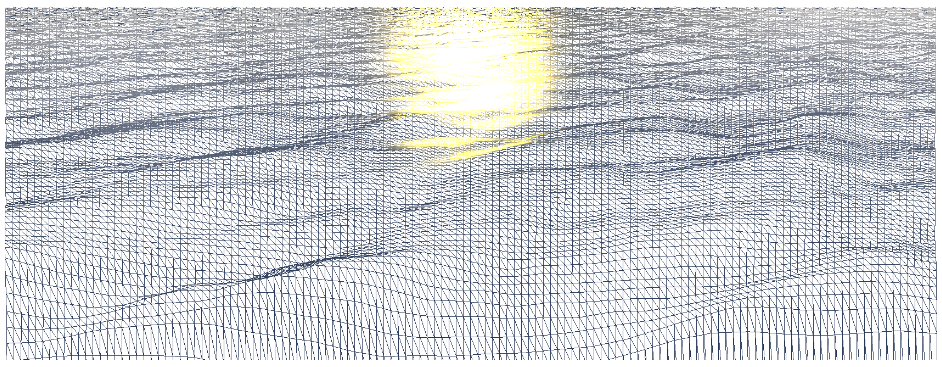


Figure 23: Surface tessellation wireframe with 129^2 vertices, or 2^{15} triangles. The height field is composed of two spectra, both of 32^2 components. The scene was rendered at 48 frames per second.

5 Conclusions

Concluding the document, some of the inferences made preparing the presented work are set forth. A summary of contributions is presented, as well as suggestions for further work, both related to the specific Coin3D-implementation and to future research.

5.1 Inferences

The implemented FFT-based statistical method is capable of generating highly realistic wind-driven wave fields by tuning a few intuitive parameters. The periodic properties of the wave fields allows for the construction of seamlessly tiled surfaces. It is problematic to directly exploit this for large ocean regions, however, due to apparent repeating structures across the surface, becoming disruptive of realism when viewed from arbitrary locations. Using discrete spectra with real-time friendly resolutions, this artificial periodicity can be avoided if the wave field is decomposed into two or more separately animated spectra. By sampling these at different scales, with a large least common multiple, a statistically faithful wave field with correct dispersive properties can be synthesized, without exhibiting the strict periodicity of a single discrete spectrum.

A fast GPU-based 2D water solver is included in the implementation. This simulation can be switched on and off, and positioned in regions of interest. The efficient interface allows quickly applying smooth and connected deflections, which seems highly useful in this case. Although this part of the ocean simulation is not rigorous, a point of importance in many cases seems to be simply to capture the notion of a responsive surface aware of its environment, even if it is in a theoretically superficial manner. The goal here has been to woo the user, and not to produce scientific data. Moreover, the particular physical model used could easily be extended, e.g. by accounting for varying depth, or replaced by a more sophisticated model.

The adopted tessellation scheme seems suited for the type of surface in question. Due to small and monotonic height variations, relative to the extent of the surface, the polygonal resolution across the visible region is close to uniform in post-perspective camera space. Moving performance critical parts of the algorithm to the GPU proved efficient, enabling high frame rates for unbounded surfaces while preserving relevant geometric details.

Periodic elevation maps seem effective for modeling large ocean regions, as long as obvious repeating structures are avoided. Combined with a suitable tessellation scheme, such as the one discussed in section 4.2, the amount of data that need to be updated and uploaded to the graphics unit is very small compared to the extent of the regions. The only transfer needed here is of the limited elevation data, which are packed into textures. If the animation and synthesis steps are moved to the GPU, which is a proven possibility, the bandwidth required for the simulation is almost eliminated. Elevation maps cannot directly represent all shapes and details characteristic of very rough seas, but this can be improved by calculating horizontal displacements, applying foam textures, and simulating spray, as noted by other authors.

5.2 Summary of contributions

A method of modeling large oceanic regions that avoids frequently occurring patterns has been described. The proposed solution overcomes problems associated with similar previous techniques, without impeding performance. The resulting model is suitable for real-time simulation, and capable of providing large and relatively detailed wave fields that follow natural wave distributions under given wind conditions.

A 2D wave equation solver that utilizes GPU-based fragment processing has been implemented. This coincides with previous work, both specifically related to water surface modeling, and to general purpose GPU-computing. To the author's knowledge, however, a 2D wave equation solver has not been demonstrated in combination with FFT-based statistical ocean models. Further, the provided interface exploits hardware rasterization to efficiently feed arbitrary forces and obstacles to the algorithm.

A GPU-based tessellation scheme that handles unbounded height fields has been implemented and demonstrated. This is compared to a previous implementation with purely CPU-based computation. The scheme seems suitable for large ocean surfaces, with close to uniform geometric resolution in post-perspective camera space, and little geometry ending up outside the view volume.

5.3 Further work

The implementation could benefit from several extensions that have not been in focus in this thesis. More importantly, perhaps, is allowing for choppier waves with foam, to reproduce rougher seas. Foam can be simulated cheaply using alpha-blended foam textures, as suggested e.g. by Jensen and Goliás, looking at the surface slope to determine how much foam should be blended in [JG01]. Tessendorf suggests an easily implemented extension to the FFT-based ocean model, where horizontal displacements are calculated, again based on the surface slope, to provide choppy waves [Tes04]. It would be interesting to see how this could be fruitfully combined with the suggested tessellation scheme. It is suspected that sharp crests will pose difficulties if the tessellation scheme is used straightforwardly, due to the high geometric resolutions needed to reproduce coherent sharp features. A suggestion for overcoming this is to dynamically increase the geometric resolution around such features, exploiting the recent support for geometry shaders on graphics hardware.

Further effects that could be included are local reflections and realistic water color. Real-time friendly approximations for both local reflections and water color are described by Jensen and Goliás [JG01]. To improve the quality of bright reflections, high dynamic range rendering could also be considered.

Using the implemented tessellation algorithm, there are cases where the results are unstable, e.g. when the camera is positioned very close to the surface. If the camera is allowed to move without restrictions, a separate programmatically controlled view transformation, a “projector” transformation, should be used in the calculations. Thus, a heuristic for aiming and positioning the projector is needed. Johanson suggests some simple heuristics, but also notes that improvements could be identified [Joh04].

A possible measure to increase performance is to move the animation and synthesis steps for the FFT-based model to the GPU, as demonstrated by Mitchell [Mit05]. Another possibility, if memory allows it, is to store height maps from a full prerendered period in a 3D texture, then look up along the time dimension as needed.

With the 2D water solver in mind, it would be interesting to allow for the covered region to move continuously during simulation, e.g. with the motion of a vessel. Similar functionality is achieved by Thürey et al., who move a simulated 3D region within a larger 2D region by copying and moving values in the grid structure [TRS06].

References

[CC06] Yung-Feng Chiu and Chun-Fa Chang. GPU-based Ocean Rendering. In *IEEE International Conference on Multimedia and Expo*, pages 2125–2128, July 2006.

[DM97] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 369–378. ACM Press, 1997.

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30. ACM Press, 2001.

[Fin04] Mark Finch. Effective Water Simulation from Physical Models. In *GPU Gems*, pages 5–29. NVIDIA Corporation, 2004.

[FR86] Alain Fournier and William T. Reeves. A simple model of ocean waves. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84. ACM Press, 1986.

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 171–178. ACM Press, 1999.

[HVT$^+$06] Yaohua Hu, Luiz Velho, Xin Tong, Baining Guo, and Harry Shum. Realistic, real-time rendering of ocean waves: Research Articles. *Computer Animation and Virtual Worlds*, 17(1):59–67, 2006.

[IDN03] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. A volume rendering approach for sea surfaces taking into account second order scattering using scattering maps. In *VG '03: Proceedings of* *the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 129–136. ACM Press, 2003.

[IVB02] John R. Isidoro, Alex Vlachos, and Chris Brennan. Rendering Ocean Water. In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Wordware, 2002.

[JG01] Lasse S. Jensen and Robert Golias. Deep-Water Animation and Rendering. *Gamasutra*, September 2001.

[Joh04] Claes Johanson. Real-time water rendering – Introducing the projected grid concept. Master's thesis, Lund University, March 2004.

[KM90] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57. ACM Press, 1990.

[Kry05] Yuri Kryachko. Using Vertex Texture Displacement for Realistic Water Rendering. In *GPU Gems 2*, pages 283–294. NVIDIA Corporation, 2005.

[KW05] Jens Kruger and Rudiger Westermann. A GPU Framework for Solving Systems of Linear Equations. In *GPU Gems 2*, pages 703–718. NVIDIA Corporation, 2005.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

[Max81] Nelson L. Max. Vectorized procedural models for natural terrain: Waves and islands in the sunset. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 317–324. ACM Press, 1981.

[MGM06] Gary McTaggart, Chris Green, and Jason Mitchell. High dynamic range rendering in valve's source engine. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*. ACM Press, 2006.

[Mit05] Jason L. Mitchell. Real-Time Synthesis and Rendering of Ocean Water. In *ATI Research Technical Report*. April 2005.

[Mus93] Forest Kenton Musgrave. *Methods for realistic landscape imaging*. PhD thesis, 1993.

[MWM87] Gary A. Mastin, Peter A. Watterberg, and John F. Mareda. Fourier synthesis of ocean scenes. *IEEE Comput. Graph. Appl.*, 7(3):16–23, 1987.

[NN05] Youhei Nishidate and Gennadiy P. Nikishkov. Fast Water Animation Using the Wave Equation with Damping. In *International Conference on Computational Science (2)*, pages 232–239, 2005.

[PA01] Simon Premoze and Michael Ashikhmin. Rendering Natural Waters. *Computer Graphics Forum*, 20(4):189–199, 2001.

[Pea86] Darwyn R. Peachey. Modeling waves and surf. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 65–74. ACM Press, 1986.

[Per85] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[PTB$^+$03] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross T. Whitaker. Particle-Based Simulation of Fluids. *Eurographics*, 22(3), 2003.

[Sta99] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press, 1999.

[TB87] Pauline Y. Ts'o and Brian A. Barsky. Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Trans. Graph.*, 6(3):191–214, 1987.

[TDG00] Sebastien Thon, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Ocean Waves Synthesis Using a Spectrum-Based Turbulence Function. In *CGI '00: Proceedings of the International Conference on Computer Graphics*, pages 65–73. IEEE Computer Society, 2000.

[Tes04] Jerry Tessendorf. Simulating Ocean Water. In *The Elements of Nature: Interactive and Realistic Techniques*. ACM Press, 2004.

[TRS06] Nils Thurey, Ulrich Rude, and Marc Stamminger. Animation of open water phenomena with coupled shallow water and free surface simulations. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 157–164. Eurographics Association, 2006.

A Contents of the digital disk

The accompanying digital disk contains the images found in this document, in the path *png/*, the document itself, in *pdf/*, and the source code from the implemented techniques, in the path *src/*.

The source code is arranged in a Microsoft Visual Studio solution, which contains the following projects (with references to corresponding test results):

OceanFlyover tests the tessellation scheme in combination with the FFT-based simulation (section 4.4.3).

SolverCPU contains the CPU-based implementation of the 2D wave equation solver (section 4.4.2).

SolverGPU contains the GPU-based implementation of the 2D wave equation solver (section 4.4.2).

TessellationCPUvsGPU tests the tessellation scheme isolatedly, both the CPU-based implementation and the GPU-based implementation (section 4.4.3).

TileSolverCombo tests the GPU-based wave equation solver in combination with the FFT-based simulation (section 4.4.2).

WaveField tests the performance of FFT-based simulation with two combined spectra, compared to a single spectrum (section 4.4.1).

Within these projects, the names of the *C++* source files correspond to the classes described in section 4.3. Fragment and vertex shader sources are given the file extension *glsl*.

B Software libraries

The accompanying source code relies on the following external resources, all available on several platforms:

Coin3D - available at `http://www.coin3d.org/`, licensed under GNU GPL.
Documentation is found at `http://doc.coin3d.org/Coin-dev/`.

FFTW - available at `http://www.fftw.org/`.

FramebufferObject - available at `http://www.gpgpu.org/developer/`.

GLEW - available at `http://glew.sourceforge.net/`.

GLUT - available at `http://www.opengl.org/resources/libraries/`.

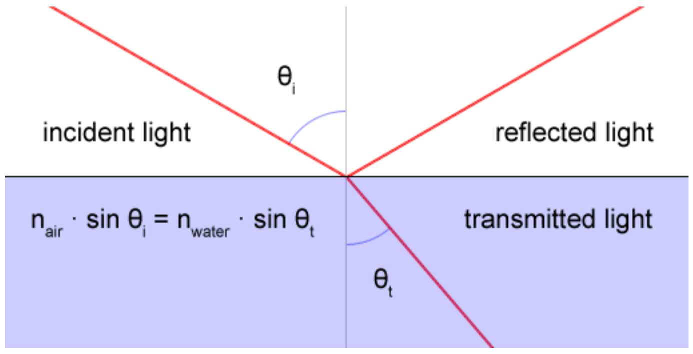


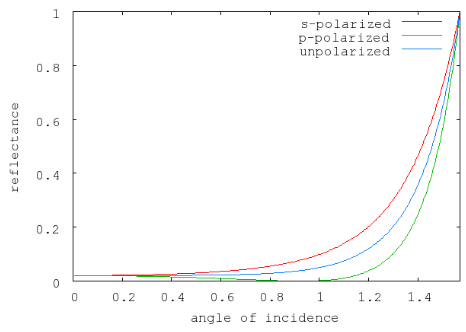
Figure 24: An illustration of Snell's law. Here, the upper medium is air, and the lower medium (shaded) is water. The interface between the two media is the center horizontal line. The angle of incidence is θ_i, and the angle of refraction is θ_t.

C Reflection and refraction

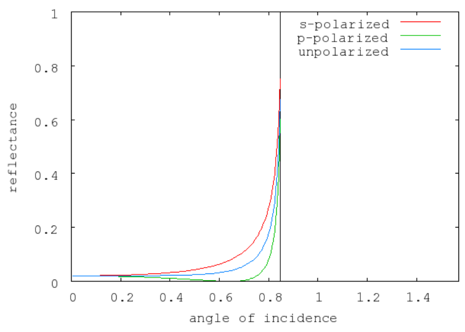
To believably capture the appearance of ocean surfaces, an understanding of light's interactions with water is needed. Particularly interesting here are the events that occur when photons reach the air-water boundary. Water is both a transmissive medium, and an excellent specular reflector. Depending on the angle of which light hits the interface between two transmissive media, such as water and air, one fraction of the photons is reflected back into the first medium, and another fraction is transmitted through the next medium. For reflected photons, the direction of travel is simply reflected about the surface tangent, assuming perfect specular reflection. Photons that are transmitted through a new medium undergo a change in speed, causing refraction, i.e. a slight change in direction. See figure 24. The angle of a refracted ray is given by Snell's law:

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \,, \tag{12}$$

where θ_1 is the angle of the incident light and θ_2 is the angle of the refracted light, or conversely. n_1 and n_2 are the *refractive indices* of the two media. For light, these measure the slowdown of photons in the given media compared to the light speed in vacuum. In this work, the refractive indices of air and



(a) air to water



(b) water to air

Figure 25: Fresnel reflectance at the air-water houndary. (a) and (b) show the reflectance for light transmitted by air and water, respectively. The vertical line in (b) marks the critical angle, beyond which total internal reflection occurs. The middle curve, in both diagrams, is the average of the reflectance for s–polarized and p–polarized radiation.

water are assumed to be 1 and 1.333, respectively. It is also assumed that their values are equal for all wavelengths of light. This is not the case, as evident in e.g. rainbows, but an adequate approximation here.

Another important relation is the probability for a photon to reflect at an interface given its angle of incidence, or in other words, the fraction of the total light intensity that is diverted into a reflected ray. This coefficient is given by the Fresnel equations, which state the reflectance for s–polarized and p–polarized radiation at the interface of two media. Assuming unpolarized light, containing an equal mix of the two polarizations, the reflection coefficient is:

$$R = \frac{1}{2}\left[\frac{\sin(\theta_t - \theta_i)}{\sin(\theta_t + \theta_i)}\right]^2 + \frac{1}{2}\left[\frac{\tan(\theta_t - \theta_i)}{\tan(\theta_t + \theta_i)}\right]^2 , \tag{13}$$

where θ_t is obtained using equation 12. The transmission coefficient T is simply the remaining fraction of the light's intensity, i.e. $T = 1 - R$.

Figure 25 shows the Fresnel reflectance for light hitting ocean water from above the surface (a), and from below the surface (b). Since the reflectance varies rapidly over a rippled surface, the intention here is to evaluate this coefficient at every pixel of the rendered surface. Simply evaluating equation 13 in a fragment shader, however, would be poor design with regards to performance. A much cheaper solution is to approximate the Fresnel coefficient using a precomputed texture map. With linear interpolation, such an approximation has relatively small errors, even at low resolutions, e.g. 32 texels. When precomputing a 1D texture, it is more convenient to substitute θ_i with $\cos^{-1} t$ in equation 12 and 13, using $t = \cos\theta_i$ as the texture coordinate. Thus, the reflectance can be looked up by passing $t = \mathbf{n} \cdot \mathbf{v}$, where $\mathbf{n}$ is the surface normal and $\mathbf{v}$ is the view vector, rather than computing θ_i.

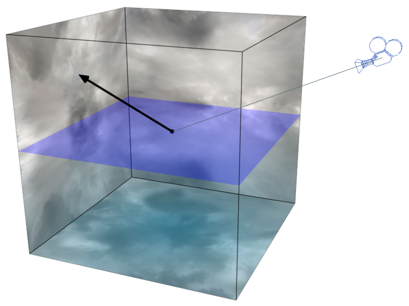


Figure 26: Rendering an ocean surface with a cube mapped environment.

D Environment mapping

Having determined the direction of an incoming light ray at a point on the ocean surface, the next step is to determine the origin of the ray, i.e. what should be reflected, and what should be refracted. *Cube mapping* is a convenient way to look up contributions from far away objects, e.g. the sky. Cube maps are natively supported by current graphics hardware, and integrated into high-level shading languages. With a cube mapped environment, texels are simply addressed using the reflection vector, or refraction vector, at a given surface fragment.

In figure 26, the fragment under evaluation is the center dot. The reflective contribution at this fragment is looked up from a cube map using the reflection vector, then multiplied with the air-to-water reflection coefficient given by equation 13. Assuming deep waters, the cube map could also contain refractions. As Jensen and Goliás suggest, realistic water color could be precalculated into the cube map, and looked up using the refraction vector [JG01]. This refractive contribution is finally multiplied by the water-to-air transmission coefficient, and added to the fragment color.

A cube mapped environment is only suitable for global reflections, from distantly located objects. Reflections from objects closer to the camera, however, are not as easily captured, since the surface will be undulating and the camera will be moving about, making the position of the objects significant.

Only taking first-order rays into account, one common approximation is to treat the ocean surface as a flat horizontal mirror, and render a mirror image of the scene into a projective texture in an additional rendering pass. When reflections are looked up from this texture, the map coordinates can be perturbed, consulting the normal map of the ocean surface, to achieve a rippled mirror image. Local reflections have not been in focus here, but more can be read about this, e.g. in the work of Jensen and Goliás [JG01].