Ocean Surface Shader

Kara Lauren Gundersen
Clemson University

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses
Part of the Art and Design Commons

Recommended Citation
Gundersen, Kara Lauren, "Ocean Surface Shader" (2015). All Theses. 2316.
https://tigerprints.clemson.edu/all_theses/2316

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
OCEAN SURFACE SHADER

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

by
Kara Lauren Gundersen
December 2015

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Shaundra Daily
Tony Penna
Abstract

The Renderman Layered Ocean Shader (RLOS) is a RenderMan displacement shader that makes it easier and more efficient to generate realistic oceans. The shader implements a Fast Fourier Transform (FFT) wave algorithm to generate tileable ocean surface displacements. These displacement layers can be added to other displacement layers to increase surface detail. The RLOS can optionally generate multiple layers with editable parameters so that it is possible to create extended ocean surfaces that can be viewed up close. The result is demonstrated with an animated time-lapse.
Table of Contents

Title Page .......................................................... i
Abstract ............................................................. ii
List of Figures ....................................................... iv
1 Introduction ....................................................... 1
2 Background ......................................................... 15
3 Workflow ........................................................ 18
4 Wave Structure ................................................... 24
5 Results ............................................................ 32
6 Conclusion ......................................................... 37
Appendices .......................................................... 39
   A RenderMan Layered Ocean Surface Maya GUI .................... 40
   B RLS Displacement Code ........................................ 41
   C Shell to call C++ code ....................................... 43
Bibliography ....................................................... 46
List of Figures

1.1 (left) Paramount’s Blue Sky Tank as a parking lot as a water tank for filming [10] ......................... 2
1.2 A view of the Life of Pi set, with wave dampening blocks on left side [14] ................................. 2
1.3 A picture from the filming of Jaws [14] ...................................................................................... 3
1.4 A view of the atoll from the filming of Waterworld 1995 [8] .................................................. 4
1.5 Original CG plate(left), composited into film(right), CG water by Cinesite using Render World [20] ................................................................. 4
1.9 Cast Away 2000. Rendered using Render World [22] ................................................................. 7
1.12 Harry Potter and the Sorcerer’s Stone 2005 [3] ................................................................. 8
1.13 Superman Returns 2006 [22] .......................................................... 9
1.14 Happy Feet 2006. Uses the same FFT ocean code as this thesis [22] ........................................ 9
1.15 Surf’s Up 2007 [12] ........................................................................................................ 10
1.16 2012 2009 [1] ................................................................................................................ 10
1.18 Life Of Pi 2012. CG water by Rhythm and Hues using 5 water simulation approaches blended together, plus a custom-built wave tank, FFT ocean simulation uses the same code as this thesis [5] ........................................................................................................ 11
1.22 Peanut Butter Jelly 2014, CG water by Clemson DPA. Uses the same FFT ocean code as RLOS ................................................................................................................... 13
1.23 Peanut Butter Jelly 2014, CG water by Clemson DPA. Uses the same FFT ocean code as RLOS ................................................................................................................... 13
1.24 Peanut Butter Jelly 2014, CG water by Clemson DPA - RLOS ............................................ 14
2.1 RLOS displacement applied to a phong surface shader .......................................................... 16
2.2 RLOS applied using the RLOS surface shader ...................................................................... 17
3.1 (left)wireframe geometry (center)shaded geometry (right)rendered geometry ..................... 18
3.2 Displacement map generated from the Ocean Surface Generator ........................................ 20
3.3 (left)Render out of Maya (right)Final composite ...................................................................... 20
3.4 (left)geometry render (center)texture map render (right)RLOS render .............................. 21
3.5 PBJ shot composite using an ocean surface generated from geometry ................................... 23
3.6 PBJ shot composite using an ocean surface generated from texture maps ............................ 23
3.7 PBJ shot composite using an ocean surface generated from RLOS ..................................... 23
Chapter 1

Introduction

The ocean has always been a source of mystery for humans; exploring and accurately portraying it has always been a difficult task in films. Before the option of computer generated water, there were quite a few approaches to include ocean scenes in films. The most common approaches were to use a large pool and scaled models to film water scenes. Paramount Studios is famous for its Blue Sky Tank. To the untrained eye, it may appear to be a parking lot next to a large blank wall, as seen in Figure 1.1. However, with cars removed, 914,023 gallons of water, and a lot of chemicals and machinery, this parking lot can be converted into the set of The Ten Commandments, TORA TORA TORA, Orange County, and many other films [10]. This approach is expensive and not realistic due to the fact that the tank cannot get deep ocean swells, the reverb created from waves hitting the wall does not generate realistic ocean movement, and the color of the water is very difficult to control. In order for the waves generated in these tanks to look larger than they were, scaled down models had to be used to get the appropriate size ratio. The movement of the smaller waves is not the same as deep ocean swells however, and can break the audience’s suspension of disbelief. Furthermore, putting people into these large pools filled with chemicals is extremely dangerous.

Filming has come a long way since The Ten Commandments was filmed back in 1956. In 2011, the world’s largest self-generating wave tank was created for the filming of the Life of Pi. The tank measured 98 feet long, 246 feet wide, ten feet deep, and held 1.86 million gallons of water [6]. They were able to create a rhythm and control the shape and size through computer programming and 12 wave machines. The left side of Figure 1.2 shows special suppressors that were used to reduce...
reverb and allow for the waves to be predictable and more realistic. This was a great alternative to shooting on the ocean because it was a controlled environment, and they were able to recreate predictable waves. Then artists used these realistic movements and composited computer generated (CG) water to extend the size of the pool to the horizon.

The history of filming ocean scenes in order to get to the realistic quality shown in Life of Pi involves a few approaches. Figure 1.3 is a shot from the iconic 1975 movie, Jaws. The ocean surfaces throughout the movie were extremely realistic because Steven Spielberg made the decision to move his film set to the open ocean. This can be felt by the fact that the cameras are moving with the swell and are never stationary when on the ocean. One of the reasons Spielberg did this was to make the audience feel as if they were about to fall in. This was not the ideal solution because some of the crewmembers accidentally fell in, and many suffered from seasickness. Consistency with the weather
and sky was another issue since the ocean scenes were filmed over 155 days on the ocean [14].

Figure 1.4 shows the set up the filmmakers of Waterworld used to film scenes that took place on an atoll. The atoll was a floating set piece attached to a dock, which could rotate when needed. Through strategic camera angles and the rotation of the set, the cameras only ever saw real ocean water in the background. This turned out to be expensive and time consuming, because in order to get the correct background without any ships in it, the whole set was rotated to get the correct reaction shots. The cast was also large, and moving them on and off the set was labor-intensive. Seasickness was also an issue with this type of filming. This was, however, the price of getting a realistic ocean in a film at the time [22].

Waterworld did have a few other approaches to producing ocean scenes. Waterworld is actually the first film to contain photo-realistic computer generated water effects that implemented a Fast Fourier Transform (FFT) wave algorithm that is based on a statistical model. Figure 1.5 shows the computer generated water before compositing on the left, and the final composited shot on the right [20].

From that point on, CG ocean surfaces developed quickly. Two years later, Titanic came out which used CG water throughout the movie, as well as a long list of other ocean-based films. Figure 1.6- 1.18 are stills from films that have implemented a variation of the same FFT ocean
Figure 1.4: A view of the atoll from the filming of Waterworld 1995 [8]

Figure 1.5: Original CG plate(left), composited into film(right), CG water by Cinesite using Render World [20]
algorithm used in Waterworld. Figure 1.19- 1.21 are examples of game engines that have also implemented a variation of the same FFT ocean algorithm.

The Clemson Digital Production Arts short animation film, Peanut Butter Jelly, also generated ocean surfaces using the FFT wave algorithm that were comparable in quality to the film industry, as seen in Figure 1.22. When viewed from at least a few meters away, the FFT wave algorithm produces realistic ocean surfaces. Figure 1.23 shows a still from Peanut Butter Jelly where the camera is four feet from the surface. When viewed up close, the lack of detail is palpable and can prevent the audience from suspending their disbelief. In order to improve the quality of close up ocean surface shots, the RenderMan Layered Ocean Surface (RLOS) was created. Through the use of RLOS, artists can incorporate CG oceans into real life plates quickly and easily without causing the audience’s suspension of disbelief to be compromised. The RLOS is an easy way for artists to generate realistic ocean surfaces with many layers of detail that allows fast iterations. The RLOS implements a FFT wave algorithm for displacement and allows the user to dynamically add more detail by adding new displacement layers. Figure 1.24 shows the same shot as Figure 1.23, but with the RLOS implemented. The RLOS improves the way ocean surfaces are generated by allowing the artists to easily edit the parameters, choose the amount of detail, and quickly see the results.
Figure 1.7: Prince Of Egypt 1998 [11]

Figure 1.8: Deep Blue Sea 1999. Rendered using Render World [9]
Figure 1.9: Cast Away 2000. Rendered using Render World [22]

Figure 1.10: Pearl Harbor 2001. Rendered using Render World [22]
Figure 1.11: Peter Pan 2003. Rendered using Render World [13]

Figure 1.12: Harry Potter and the Sorcerer’s Stone 2005  [3]
Figure 1.13: Superman Returns 2006 [22]

Figure 1.14: Happy Feet 2006. Uses the same FFT ocean code as this thesis [22]
Figure 1.15: Surf’s Up 2007 [12]

Figure 1.16: 2012 2009 [1]

Figure 1.17: Battleship 2012 [2]
Figure 1.18: Life Of Pi 2012. CG water by Rhythm and Hues using 5 water simulation approaches blended together, plus a custom-built wave tank, FFT ocean simulation uses the same code as this thesis [5]

Figure 1.19: Real-time Ocean Surface in video game implemented by Cryengine [15]
Figure 1.20: Myst III: Exile 2001. Rendered using Render World [22]
Figure 1.21: Real-time Ocean Surface in video game implemented by osgOcean [7]

Figure 1.22: Peanut Butter Jelly 2014, CG water by Clemson DPA. Uses the same FFT ocean code as RLOS
Figure 1.23: Peanut Butter Jelly 2014, CG water by Clemson DPA. Uses the same FFT ocean code as RLOS

Figure 1.24: Peanut Butter Jelly 2014, CG water by Clemson DPA - RLOS
Chapter 2

Background

2.0.1 RenderMan and the RLOS

RenderMan is commercial photorealistic, three-dimensional, rendering software that was originally released by Pixar in 1988. For Peanut Butter Jelly, RenderMan 18 was the rendering engine chosen because of its rendering efficiency and the ability to write custom shaders using the RenderMan Shading Language (RSL). The RLOS is a RenderMan shader incorporating pre-existing C++ code for the calculation of FFT-based displacements, and slopes, normals, and texture coordinates. The RSL, according to Cortes [16], is a C-like, higher-level programming language designed specifically for surface materials, light, and phenomena description. Although RSL is a C-based shading language, it restricts and limits a lot of the C-like functionality. In order to create a shader that incorporates pre-existing C++ code, a RSL plug-in should be used. A RLS plug-in is a way to call external C/C++ functions in a shader by passing information to the C/C++ code through RSL parameters. The RLOS uses the RSL plug-in workflow to incorporate pre-existing C++ code to generate the displacement layers.

RSL gives the artist the ability to write custom shaders in categories: surfaces, lights, volumes, imager, and displacement. The two basic types of shaders used to create the RLOS are a surface shader and a displacement shader.

Displacement shaders are calculated first because the displacement function manipulates the surface of the geometry, which will change global variables that are used by the other shading categories. Using displacement allows fine details to be rendered without complex geometry. In the
RLOS, the displacement shader takes advantage of RSL’s plug-in capabilities, and each dynamic layer makes a function call to C++ code that generates height maps. When rendering, each point that is tested will call the C++ ocean layer function for each layer. Each ocean layer function call passes in fifteen different parameters specific to each layer. The displacement data returned from each layer is added together to generate the ocean displacement. In order to get the correct displacement, the tangent, normal, and binormal vectors of the base surface are each multiplied by the corresponding displacements in the X-axis, Y-axis, and Z-axis respectively. RenderMan translates the geometry and recalculates the surface normal for use in the other surface categories.

Surface shaders output the final color of a point based on the incoming light and the physical properties of the shader. Different surface shaders calculate different surface types like metal, plastic, or wood. The RLOS is set up primarily to be used as a displacement shader that can work with any existing surface shader. For example, Figure 2.1 is an image of the RLOS displacement applied to a phong shader placed on a simple plane. The RLOS also has a surface shader (illustrated in Figure 2.2) which calculates Fresnel reflections and refractions using the index of refraction of water which is 1.333. Figure 2.2 was generated from underneath the water surface. To generate this image, the RLOS surface shader was used in conjunction with a simple environment sphere, mimicking a particular shot in Peanut Butter Jelly.
Figure 2.2: RLOS applied using the RLOS surface shader
Chapter 3

Workflow

In Peanut Butter Jelly the ocean surface is visible in fifteen of the nineteen shots, and figuring out the best way to create the ocean surface turned out to be more difficult than anticipated. DPA has a few tools that implement the FFT wave algorithm. The only tool that could be utilized at the start for Peanut Butter Jelly generated a geometric representation of the ocean surface as polygons and is illustrated in Figure 3.1. The tool gives the user the ability to choose the size of the plane and resolution of the wave, which is then used to determine the divisions of the plane. The user also has the ability to edit other aspects of the wave, such as wave size and height.

In order to achieve a higher level of detail with the geometric model, the polygon count must be increased enormously, taxing hardware and Maya. To get a resolution of one vertex at

Figure 3.1: (left)wireframe geometry (center)shaded geometry (right)rendered geometry
every inch on a 400-foot by 400-foot plane, the resolution would have to be 4,800 polygons by 4,800 polygons. The polygon count for that plane would be 23,040,000, and that surface did not have as much detail as the director wanted. Maya does not handle large polygon counts and crashes when the geometry resolution was anything larger than 2048x2048. Since Maya is inept at dealing with high polygon count, the 23,040,000 polygon surface was never attempted, and the use of geometry was quickly abandoned.

Another approach was to use a displacement map generated by the ocean surface software on a flat simple surface. Using the ocean surface generator, texture images, like the one shown in Figure 3.2, were created, converted to a Photorealistic RenderMan texture file, and then applied as a displacement map. Since the type of displacement is in all three axes, using EXR images to store the information was a simple solution. The R-value in the image holds displacement in the X-axis, the G value of the image holds the Y-axis displacement, and the B value holds the displacement amount for the Z-axis. The RGB values of the displacement map are not 0-1 as normal images, but are in fact floats and can be as large as needed in both the positive and negative direction. In practice, these numbers ranged from -10 to 10 depending on wave height and cuspiness. The displacement maps used in Peanut Butter Jelly had a resolution of 2048x2048 and were applied to a 400-foot by 400-foot plane. The finest level of detail that could be achieved is that one texture map element covered a three-inch by three-inch area. Although the displacement maps eliminated the high polygon count, the resolution was still an issue. For far away shots it was possible to blur the displacement map to remove some of the pixilation, but there were still issues with shots like the one shown in Figure 1.23.

Figure 3.3 shows the ocean surface and final composite for a shot from Peanut Butter Jelly. The ocean surface layer had to be manipulated and blurred four times in order to reduce the pixilation, but it was never fully gone.

To increase the detail, smaller, tillable, patches were generated at the same resolution and added to the displacement data. One of the reasons the FFT algorithm is so powerful is its ability to be layered to reduce the pixilation. Textures can be tiled at different frequencies and then added together to create enough variation so that the repetition and pixilation are masked. This solution worked for Peanut Butter Jelly, but since all the images had to be generated and then converted to Photorealistic RenderMan texture files, it was difficult to test different parameters and quickly see results. This is where the idea for the RLOS originated. Figures 3.4 shows how each ocean surface
Figure 3.2: Displacement map generated from the Ocean Surface Generator

Figure 3.3: (left) Render out of Maya (right) Final composite
rendered out, and Figures 3.5 - 3.7, show the final composite using the three different approaches to ocean surfaces.

The original idea of the RLOS was developed using texture images to get height information, but iterations were slow due to texture generation and conversion. The way RLOS generated its height information was then revamped to calculate the height information at render time by calling the FFT wave algorithm, allowing faster iterations. This allows the user to test different variables and quickly see results. Previously, in order to change one variable it would require an hour and a half to render out a new sequence of displacement maps, another 15 minutes to convert them to Photorealistic RenderMan texture files, and then however long it took for that particular scene to render, to be able to see results. With the implementation of RLOS, the variables can be changed in Maya, and it only takes the render time to see results.

The pre-existing C++ code was incorporated into the RLOS by using a RSL plug-in. As shown in Appendix C, the RSL connection code is encapsulated in extern C so that the C++ code has C linkage. There is also a table that defines the RSL function calls to the C++ implementation at the end of the connecting code. For the RLOS, there is only one RLS function that is defined, but it passes in fifteen parameters. These parameters are then used to generate the ocean surface.

There were issues that had to be considered when incorporating this particular code into RSL. The first major issue that presented itself, was in the form of rectangular image artifacts due to multi-threading. Since the computer has multiple threads running for a single render, initiating more than one would cause the first thread to generate a result of zero, or no displacement. To fix
this issue, a mutex was created around the initializing functions so only one thread could initialize
the functions. The next issue was the waves were not consistent from frame to frame in a sequence.
This was a simple fix by seeding the random number generator so that the wave was getting the
same seed instead of a different random number every time. The seed is now an option the artist
can change in the Maya GUI.

The Maya GUI also allows the artist to change any of the fifteen parameters used to generate
the wave. Appendix A shows a screen shot of the Maya GUI that is available to the artist. The
artist can dynamically choose to have one to three levels of detail generated through the use of check
boxes. Through this API, the artist can easily isolate a single layer, edit the parameters, see the
local changes, and quickly add the other layers back to it to see the final result.
Figure 3.5: PBJ shot composite using an ocean surface generated from geometry

Figure 3.6: PBJ shot composite using an ocean surface generated from texture maps

Figure 3.7: PBJ shot composite using an ocean surface generated from RLOS
Chapter 4

Wave Structure

The displacement generated by the RLOS uses a statistical model of Fast Fourier Transform (FFT) waves. In order to get variation in the waves, a seed value for the random number generator is chosen, as well as selected ranges of parameters that control the wave anatomy. The terminology in Figure 4.1 will be used throughout this chapter.

The ocean has many different types of waves that should be accounted for, which is why layering FFT waves generates better results than just having one layer. Figure 4.2 is a visualization of an approach that might be taken if the artist was trying to generate an ocean surface for a 146-meter square plane. The black square represents the 146-meter plane and will calculate the deep ocean swell waves. Deep ocean swell waves travel long distances and are generated by distant storms so they move differently than smaller waves and should be calculated as an individual layer. The

Figure 4.1: Anatomy of a Wave
orange squares have a length of 81 meters on each side and generate medium sized waves. These are the waves that create the choppy texture as shown in Figure 4.6. Medium sized waves are generated by local wind moving across the surface of the water and express the winds of the current location. The blue squares represent an ocean surface layer that is 11 meters square and account for the fine detail on the surface, which is also caused by local wind. In most situations, three layers are all that is needed to account for the different types of waves that can be seen at one time. There are practical limits for how many ocean layers can be used. For instance, if an artist were to add 100 layers together, it would lose the individual structure of deep, medium, and fine detail waves and just look like noise.

In order to create realistic ocean surfaces, each layer needs to have the appropriate structure in order to be believable. The vertical distance between the crest and trough is called the wave height. The wavelength is the distance between two consecutive crests or troughs. The ratio of wave height to wavelength is the main factor determining the size of the wave that is being generated, and how stable the wave is. As the ratio of wave height to wavelength gets closer to zero, the ocean is considered calmer. Increasing the ratio away from zero move towards steeper, unstable waves. Figure 4.3 shows a wedge of three frames with a height of 1.2 meters. The first frame has wavelengths in the range of 3.2 meters to 20 meters and would be considered deep swell waves, the second frame has wavelengths in the range of 0.7 meter and 0.1 meter and is medium sized waves, and the third has a wavelength of 0.02 meter, which generates the detail waves. Figure 4.4 uses the same layers as Figure 4.3, but the layers are added to each other from left to right instead of being viewed individually.
Figure 4.3: (left) deeper ocean waves known as swells (center) medium sized waves (right) detail waves for close up shots

Figure 4.4: (left) deeper ocean wave (center) medium sized waves added to deep ocean wave (right) detail waves added to the deep ocean waves and medium sized waves
Figure 4.5: Waves at Bells Beach, Australia that were referred to as corduroy [18]

The randomly generated length variation gives the waves more realism and less of an organized effect. By changing the biggest and smallest wave parameters to closer to the same value in the GUI shown in Appendix A, the more consistent and organized the waves will be. The more organized a set of waves are in the surfing community, the more likely they will be referred to as corduroy, which is in reference to the fabric corduroy. The alignment of the waves also plays into how organized the waves are. Figure 4.5 is an example of corduroy because they are aligned, parallel, and consistent. If they are not aligned and moving in all directions, as seen in Figure 4.6, then the ocean surface can be referred to as choppy.

Another determining factor for the roughness of the ocean is the cuspiness of the wave. This is how round or pointed the top of the wave is. The cuspier the waves are the rougher the ocean is. Figure 4.7 is a wedge that changes the cuspiness parameter from one to three to ten in the RLOS.

These concepts are all parameters that can be controlled in RLOS. Since the RLOS waves are computer generated, however, there are some artifacts that should be noted. The resolution of the layer determines the amount of detail that can be seen. Figure 4.9 has a resolution of 256x256, and the grid pattern is very evident in this image. There are a couple of ways the visible pixilation can be reduced. The first way is to increase the resolution, which will allow the surface to be smoother, however, there will not be finer details. The other option is to add smaller area layers at the same resolution seen in Figure 4.4. By adding a layer that covers a smaller area, the
Figure 4.6: Waves described as choppy due to their lack of organization [19]

Figure 4.7: Cuspiness Wedge
grid artifact is reduced. When another layer of fine detail is added the grid artifacts are no longer noticeable. Figure 4.8 is a very basic visualization of how adding different frequency waves can break up regularity. If tiles repeat at the same points, however, tiling artifacts will occur. Figure 4.10 is a 400x400 meter plane with a RLOS that has implemented a 50 meter wave layer, and a 10 meter wave layer. Without a larger wave to break up the repetitive features, the tiling can be very obvious. By adding a larger wave, the repetition is not as noticeable. All of these settings can be adjusted in the Maya interface, which allows for the artist to easily change parameters and quickly see results.
Figure 4.9: Resolution Artifacts
Figure 4.10: Tiling Artifacts
Chapter 5

Results

Figure 5.1 has an extremely simple background of parallel lines to demonstrate the amount of detail and movement that is generated by each layer. The distance from the camera to the horizon is 400 meters. The left side is a layer of deep swell, and the individual color lines are still distinct because the displacement on the surface is on a larger scale. The middle section is the deep swell and a layer of medium sized waves displacement. This causes more break-up of the lines, and the pattern becomes noisier, but are more realistic. There are, however, over ten distinct color lines half way to the horizon. The right side has all three layers implemented and is the most realistic of all. It has roughly five color lines half way between the camera and horizon and is the best layering option for close up shots.

Figure 5.2 has the same layer parameters as Figure 5.1, but the environment is different. The different environment allows for the ocean wave structures to be more recognizable and less like noise in a still image. The loss of structure is less of an issue when viewing a movie of the ocean surfaces instead of a still image.

Figure 5.3 is a still image from the final shot that inspired the RLOS. The camera is roughly a meter from the surface. The scene has all three types of ocean surface waves: deep swell, medium sized waves, and detail waves. RLOS gave the shot more of a definite shallow depth instead of a blurred out hazy surface.

Figure 5.4 is the first of a three part image sequence that shows how the RLOS reflects a time-lapse sunrise video taken by Brett Junvik [17]. The scene has only one ocean layer because of the tranquility desired with this particular shot. This shot was inspired after watching Life of Pi.
Figure 5.1: Vert Line Reflection (left) 1 layer (center) 2 layers (right) 3 layers

Figure 5.2: Image Reflection (left) 1 layer (center) 2 layers (right) 3 layers
In this still image, the ocean surface looks a little more choppy than it really is due to the low angle of the sun; however, in the movie the motion of the water is so slow it does not read as choppy.

Figure 5.5 is a frame from the middle of the time-lapse, the sun has just fully peaked over the horizon but the color of the light is still very saturated. The waves don’t have as much contrast, which allows them to look calmer.

Figure 5.6 is one of the last images in the time-lapse video. This is the one that I believe looks the most like the Life of Pi scene. The surface looks mirror-like, and due to the higher angle of the light, the waves have almost no contrast and an extremely peaceful ambience.
Figure 5.4: 1 layer RLOS, low intensity light. Timelapse by Brett Junvik [17]

Figure 5.5: 1 layer RLOS, medium intensity light. Timelapse by Brett Junvik [17]
Figure 5.6: 1 layer RLOS, high intensity light. Timelapse by Brett Junvik [17]
Chapter 6

Conclusion

The RLOS is a RenderMan shader that can be used on its own or can be applied as a displacement shader. With the ability to turn layers on and off, the artist can create the level of detail that is desired. The GUI also allows the artist to change the features of the wave layers, generating anything from a calm ocean to a choppy, stormy ocean. The RLOS could be further developed by including algorithms for white caps and caustics. The RLOS could also be used in conjunction with e-wave data to integrate interactive waves with the ocean surface. Furthermore, the artist could possibly paint current maps so that ocean currents could be visible as seen in Figure 6.1. Another possibility that currently does not simply work, is to incorporate a way to keyframe the parameters of each ocean layer. This way an artist could start off with a calm ocean and slowly generate a stormy ocean.
Figure 6.1: A picture of a lake surface with two distinct surface textures [4]
Appendices
Appendix A  RenderMan Layered Ocean Surface Maya GUI

This is a screenshot of the Maya GUI for the RenderMan Layered Ocean Shader. Each ocean layer has individual controls, including a control that turns the layer on and off. Patch size, biggest wave, smallest wave, and typical height are all measured in meters. Patch size is in reference to the size of the ocean tile displacement being sampled. Biggest wave and smallest wave is in reference to the wavelength, or distance between crests. The wave height is how high the wave is from trough to crest. CuspyX and CuspyY is a multiplier which determines how round the top of the wave is. As you increase the number above one, the wave tops become more pointed. Each ocean layer is composed of many layers, and the direction, alignment, and travel control the organization of the ocean layer’s sub-layer. The resolution is how much detail is in each ocean layer, and should be in base-two and does not give as successful results above 2048x2048.
Appendix B  RLS Displacement Code

Appendix B is the section of the RLS code that calculates the displacement. It first creates variables that will be used throughout the function, changes over to the Gram-Schmidt coordinate system, pulls the frame number, and then multiplies the UV coordinates, which range from 0-1, by the size of the geometry in order to get the appropriate location in meters. Then it calls the C++ function with 15 input parameters for each layer. The C++ code returns a 3D vector, which then multiplies the X and Z vectors by the appropriate cuspiness. Once the three layers are added together into vector nD, the X coordinate of nD is then multiplied by the tangent to the face normal, the Y coordinate of nD is multiplied by the normalized face normal, and the Z-coordinate is multiplied by the bimormal to the face normal. This 3D vector is then added to the original surface location as the displacement.

```cpp
public void displacement(output point P; output normal N){
    //------ Displacement vector ------
    color nD = 0;
    //------ Ocean Layer vectors ------
    color oceanDisp1 = 0;
    color oceanDisp2 = 0;
    color oceanDisp3 = 0;
    //------ Layer Number ------
    uniform float layer1 = 1.0;
    uniform float layer2 = 2.0;
    uniform float layer3 = 3.0;
    //------ UV coords between 0–1 ------
    float ss = mod(s,1);
    float tt = 1.0 - mod(t, 1);
    //------ Normal, tangent, binormal ------
    nX = normalize(dPdu);
    nY = normalize(N);
    nZ = normalize(dPdv);

    //***************OCEAN SURFACE***************
    //------ Gram–Schmidt–coord–system------
    vector nXp = nX;
    vector nZp = nZ;
    nZp = normalize(nZp);

    //------ pull frameNum from RIB file ------
    float frameNumber = 0;
    option("Frame", frameNumber);

    //------ Multiply UVs by geo size ------
    float xWidth = ss * geoSize;
    float yWidth = tt * geoSize;

    //------ call–ocean–plugin–function ------
    if (ocean1 != 0)
        oceanDisp1 = choppywave(patchsizex1,patchsizey1,biggestwave1,
```
smallestwave1, typicalheight1, direction1, alignment1, travel1, nbx1, nby1, xWidth, yWidth, frameNumber, seed1, layer1);

if (ocean2 != 0)
oceanDisp2 = choppywave(patchsizex2, patchsizey2, biggestwave2, smallestwave2, typicalheight2, direction2, alignment2, travel2, nbx2, nby2, xWidth, yWidth, frameNumber, seed2, layer2);

if (ocean3 != 0)
oceanDisp3 = choppywave(patchsizex3, patchsizey3, biggestwave3, smallestwave3, typicalheight3, direction3, alignment3, travel3, nbx3, nby3, xWidth, yWidth, frameNumber, seed3, layer3);

oceanDisp1[0] = oceanDisp1[0] * cusp1;
oceanDisp2[0] = oceanDisp2[0] * cusp2;
oceanDisp3[0] = oceanDisp3[0] * cusp3;


//-------------- add the oceanDisp-------------
nD += oceanDisp1;
nD += oceanDisp2;
nD += oceanDisp3;

//-------------- nX is the tangent to the face normal-----------------
P[0] += nX[0] * nD[0];
P[1] += nX[1] * nD[0];

//-------------- nY is the normalized face normal-----------------
P[0] += nY[0] * nD[1];

//-------------- nX is the binormal to the face normal-------------
P[0] += nZ[0] * nD[2];
Appendix C  Shell to call C++ code

Appendix C is the shell used to call the pre-existing C++ code. It reads in the RSL variable arguments as iterators in order to allow for multithreading. It then searches for the ocean layer, if it hasn’t been created; it locks the mutex, generates the new ocean layer, and then unlocks the mutex. Once the ocean layer has been created, it creates a 3D RSL iterator and populates it with the X, Y, and Z displacements, which are returned to the shader. The table at the bottom defines the functions that are called in the C++ program and only holds one function in the program.

```c
#include <stdlib.h>
#include <stdio.h>
#include <map>
#include <RixInterfaces.h>
#include "WaveSurface.h"
#include "WaveSurfer.h"
#include "RslPlugin.h"

extern "C" { 

//---private-instances---
WaveSurfer *manager;
std::map<int, WaveSurface *> surface;
RixMutex *s_stringLock = NULL;

//---Init-called-at-startup---
void init(RixContext *ctx) {

    //create mutex
    RixThreadUtils *lockFactory = (RixThreadUtils *)
        ctx->GetRixInterface(k_RixThreadUtils);
    s_stringLock = lockFactory->NewMutex();

    //set the manager
    manager = new WaveSurfer();
}

//---Cleanup-called-at-shutdown---
void cleanup(RixContext *ctx) {
    if( manager != NULL ) {

        std::map<int, WaveSurface *>::iterator sfc = surface.begin();

        while( sfc != surface.end() ){
            manager->DeleteWaveSurface( sfc->second );
            surface.erase(sfc++);
        }
    }
}
```
}
surface.clear();
delete manager;
if (s_stringLock) delete s_stringLock;
}

// Update Waves
RSLEXPORT int
updateWaves(RslContext* rslContext,
    int argc, const RslArg* argv[])
{
    // argv[0] is always reserved for outputting a formal return value. Can also
    // declare other argv[#] as outputs results. It is not just a pointer to a
    // single triplet (color), but an array of them.
    RslColorIter result(argv[0]);
    // input paramaters
    RslFloatIter patchsizex(argv[1]);
    RslFloatIter patchsizey(argv[2]);
    RslFloatIter biggestwave(argv[3]);
    RslFloatIter smallestwave(argv[4]);
    RslFloatIter typicalheight(argv[5]);
    RslFloatIter direction(argv[6]);
    RslFloatIter alignment(argv[7]);
    RslFloatIter travel(argv[8]);
    RslFloatIter nbx(argv[9]);
    RslFloatIter nby(argv[10]);
    RslFloatIter xVal(argv[11]);
    RslFloatIter yVal(argv[12]);
    RslFloatIter frame(argv[13]);
    RslFloatIter s(argv[14]);
    RslFloatIter layerIndex(argv[15]);

    // Dereference
    float time = frame/24.0;
    int layer = *layerIndex;
    float seed = *s;
    // Search for layer i
    std::map<int, WaveSurface*>::iterator sfc = surface.find(layer);

    if (sfc == surface.end())
    {
        // if it can't find layer
        lock mutex
        s_stringLock->Lock();

        srand48(seed);
        surface[layer] = manager->NewChoppyOceanWave();

        surface[layer] -> SetPatchSizeX(*patchsizex);
        surface[layer] -> SetPatchSizeY(*patchsizey);
        surface[layer] -> SetBiggestWave(*biggestwave);
        surface[layer] -> SetSmallestWave(*smallestwave);
    }
surface [layer] -> SetWaveDirection(*direction);
surface [layer] -> SetTypicalWaveHeight(*typicalheight);
surface [layer] -> SetTravel(*travel);
surface [layer] -> SetAlignment(*alignment);
surface [layer] -> UseHeight();
surface [layer] -> UseSlope();
surface [layer] -> UseCuspies();
surface [layer] -> Init((int)*nbx, (int)*nby);
surface [layer] -> SetCurrentTime(time);
surface [layer] -> UpdateWaves();

s_stringLock->Unlock();

int numVals = argv[0] -> NumValues();

for(int i = 0; i < numVals; ++i){
    result[0] = surface[layer] -> WaveCuspDisplacementX(*xVal,*yVal);
    result[1] = surface[layer] -> WaveHeight(*xVal,*yVal);
    result[2] = surface[layer] -> WaveCuspDisplacementY(*xVal,*yVal);
    result++; xVal++; yVal++;}
return 0;

static RslFunction myFunctions[] = {
    "color choppywave(uniform float, uniform float, uniform float, uniform float, uniform float, uniform float, uniform float, uniform float, uniform float, varying float, varying float, varying float, varying float, uniform float)", updateWaves, init, cleanup
};

RSLEXPORT RslFunctionTable RslPublicFunctions = myFunctions;
/* extern "C" */
Bibliography


