Modeling Island Coastal Waves with Spring Systems

Sui Yifan

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Computer Engineering

Prince of Songkla University

2010

Thesis Title	Modeling Island Coastal Waves with Spring Systems
Author	Mr. Sui Yifan
Major Program	Computer Engineering

Major Advisor:	**Examining Committee:**
………….…………………………….	..…….…….……………………….......Chairperson
(Dr. Andrew Davison)	(Dr. Nikom Suvonvorn)
	…………….………………………………….........
Co-advisor:	(Dr. Andrew Davison)
....	…………….………………………………….........
(Asst. Prof. Dr. Pichaya Tandayya)	(Asst. Prof. Dr. Pichaya Tandayya)
	…………….………………………………….........
	(Assoc. Prof. Dr. Wattanapong Kurdthongmee)

The Graduate School, Prince of Songkla University, has approved this thesis as partial fulfillment of the requirements for the Master of Engineering Degree in Computer Engineering.

…………….………………………………..
(Prof. Dr. Amornrat Phongdara)
Dean of Graduate School

Thesis Title	Modeling Island Coastal Waves with Spring Systems
Author	Mr. Sui Yifan
Major Program	Computer Engineering
Academic Year	2010

ABSTRACT

This project is concerned with developing a dynamic 3D model for island coastal waters which balances realism with computational efficiently. The land is generated from a height map, and waves move using a combined profile function which includes a phase function for wave refraction.

The novel features of this work are a series of spring-based systems that manage the interaction of the water and land around the coastline, based on a wave curves data structure. Wave motion is controlled with position and wave springs, and employs collision detection for water/land and waves collisions. A wave affects the waves around it, and rebounds when it hits the land with a suitably changed height and velocity. The movement of waves in the vertical plane is controlled with height springs.

The model's appearance is enhanced using procedural, detail, and material textures. Wave crest shading, water transparency, water spray and breaking waves further improve the model's realism, the later two implemented as particle systems.

The system animates a 128*128 mesh model at about 56 frames/second (FPS) on a PC, illustrating the speed o this approach, and can render 256*256 mesh models at around 22 FPS.

Keywords: Spring systems, wave curves, particle systems.

ACKNOWLEDGEMENT

First of all, I would like to express my gratitude to Dr. Andrew Davison, my supervisor, for his advice, suggestions, teaching, and help with this thesis.

I would also like to thank to Asst. Prof. Dr. Pichaya Tandayya, my co-advisor, for her advice, comments, teaching, and help with this thesis, and my study.

Thirdly, I would like to thank the other members of my committees Dr. Nikom Suvonvorn and Assoc. Prof. Dr, Wattanapong Kurdthongmee, who gave me many useful suggestions for finishing my thesis.

I thank all the lecturers and staff in the Department of Computer Engineering who helped me with my studies. In addition, I thank the Department of Computer Engineering, Faculty of Engineering, Graduate School, and Prince of Songkla University, who offered me help to finish my university work.

I would like to express my heartfelt gratitude to my family who have supported my study in Thailand. I also thank my friends for their help with my study and life.

Sui Yifan

CONTENTS

CONTENTS (CONTINUED)

CONTENTS (CONTINUED)

CONTENTS (CONTINUED)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (CONTINUED)

CHAPTER 1

INTRODUCTION

The rendering of large areas of water is well understood, and has become common in games [1, 2]. Compared to several years ago, the processing power of current graphic cards can render moving water in real-time. For example, a Perlin Noise function can create real-time waves in a large area of water [3]. Since 1986, considerable effort has been devoted to simulating the interaction between fluids and solids, and this thesis contributes to this topic, within the domain of coastal water. Previous work has examined ocean waves' effects, such as refraction and collision with obstacles [2].

1.1 Problem Statement

Fast rendering of water is more and more familiar in games, but there is little physics-based interaction with the shoreline as waves move up and down, and generate spray and foam. There should be a method combines fast rendering and physics for a large body of water.

Peachey [4], Fournier and Reeves [5] render waves approaching and breaking on a sloping beach as shown in Figure 1.1. Particle systems were used to model the foam and the spray generated by wave breaking and collisions with obstacles. But there is no force interaction between the water and land. For example, the wave profile only changes according to wave steepness and water depth in Peachey's model.

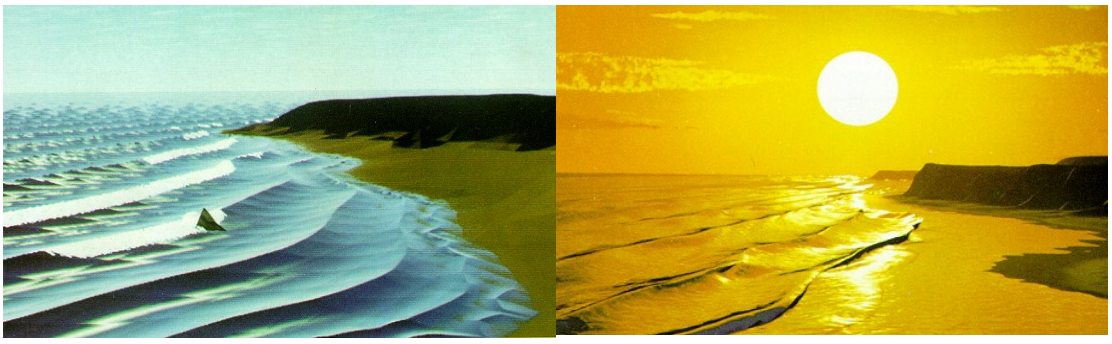


Figure 1.1: Peachey's model (left) [4] and the model of Fournier and Reeves (right) [5].

Foster and Fedkiw [6], and Enright, Marschner and Fedkiw [7] simulate poured liquid and breaking waves by a combination of textures and particles, as shown in Figure 1.2. But the computational cost is approximately several minutes per frame. For example, Foster and Fedkiw's model with 150*75*90 cells running on a Pentium II 500MHz takes four minutes per frame.

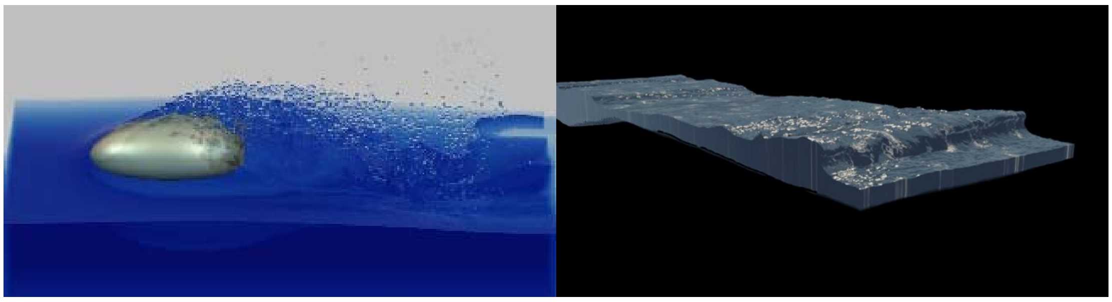


Figure 1.2: Foster and Fedkiw's model (left) [6] and the model of Enright, Marschner and Fedkiw (right) [7].

Maes, Fujimoto and Chiba [8] develop three-dimensional fluid animation suitable for water flowing on irregular terrains, intended for interactive applications as shown in Figure 1.3. Columns are used to render the water, their heights varying due to the flow through pipes between neighboring columns. The flow in these virtual pipes is determined by the physics of hydrostatics. The water columns have variable heights and lie directly on the terrain, so they can not move in the horizontal direction.

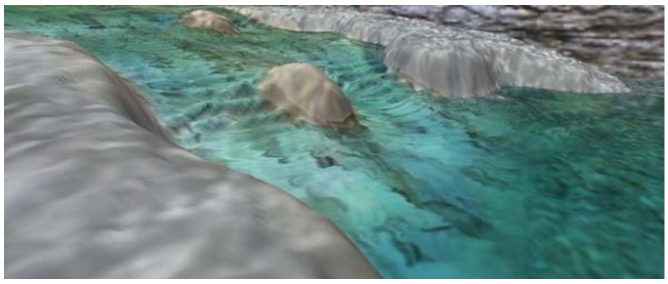


Figure 1.3: Maes, Fujimoto and Chiba's model [8].

1.2 Proposed Idea

A spring system can be used to simulate fluid and waves, with mass-spring systems arguably the simplest and most intuitive of all deformable models [9, 10]. It is feasible, easy, and intuitive to use spring systems to simulate the motion of water near a coastline while still allowing fast rendering with the limited resources of a standard personal computer.

This thesis proposes the original idea of connecting water vertices by springs, to make water move realistically in the vertical and horizontal planes while reducing the complexity of the necessary algorithms. The water surface near the land will move forward and back to represent water/land interaction based on velocities and forces calculated using spring systems following Newton's and Hooke's laws [11].

Some existing technologies will be reused in this system. Wave refraction will be based on Peachey's paper [4], and multitexturing of land and water surfaces will improve realism [12]. Other features include wave crest shading [13] and water transparency [14]. Water spray and breaking waves will be generated using particle systems [15].

To simplify the coding tasks, the implementation is limited to one island, the water and land surfaces are based on meshes. Figure 1.4 shows a screenshot of the final model; the details and the effects will be explained in later chapters.

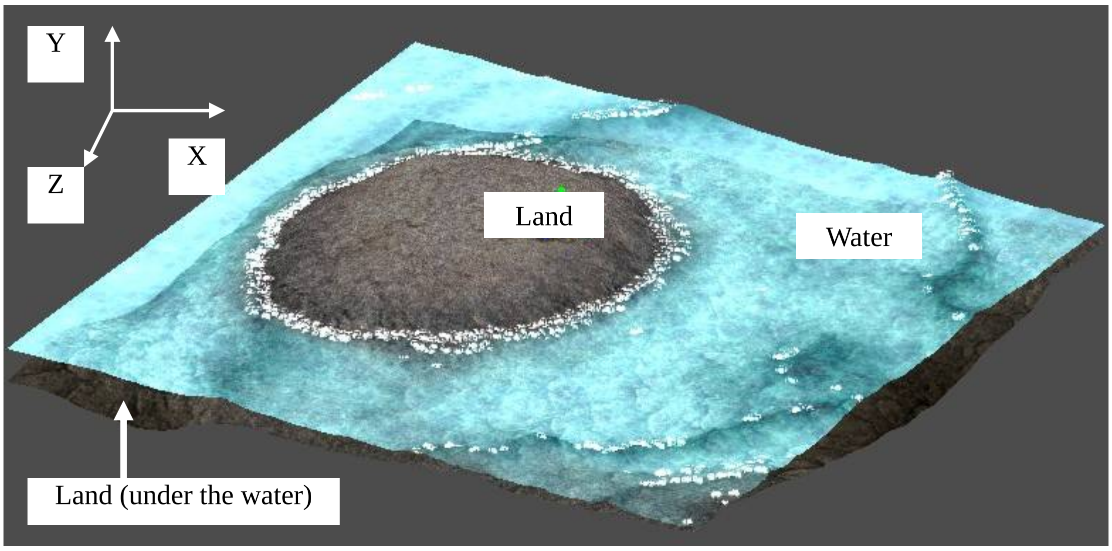


Figure 1.4: The final model.

1.3 Objectives

1. The code utilizes Java, JOGL (a Java binding for OpenGL) and GLSL

(OpenGL Shading Language).

2. 3D rendering elements include:

1) Terrain: the land surface is built use a 2D mesh and height map.

2) Water: height functions animate the water surface with different waves.

3) Terrain and water interaction: phase functions are used to produce wave refraction around the land; wave curves are placed around the land in the water mesh; spring systems adjust the motion of the water vertices; collision detection checks the water-land interaction and the interaction between wave curves.

4) Terrain and water multitexturing improve the appearance of the land and water surface (the water surface also employ shaders).

5) A particle system represents the spray around the coastline; another particle system represents foam on the breaking wave crests.

1.4 Scope

1. Programming utilizes JOGL and GLSL shaders. The use of shaders requires graphics hardware which supports OpenGL 2.0 or later.

2. The terrain mesh is created with a height map, and the water mesh is the same size (typically 128*128). The terrain is assured to consist of one island or harbour, but there is no restriction on the shape of the land.

3. Water waves are created, and wave refraction is implemented based on the depth of the water.

4. Coastline and wave curves are applied to the water vertices. Since collisions occur between the water and the land around the coastline, collision detection is utilized with the help of spring systems.

5. Multitexturing (both procedural and detail texturing) is applied to the land surface, and to the water surface (material and detail texturing).

6. Coastline spray and breaking wave foam are rendered by particle systems.

1.5 Tools

1. Testing is carried out on two Windows XP machines. A notebook with Core T2250 CPU 1.73 GHz, RAM 2GB, and an Intel 950 graphic card which supports OpenGL 1.4. This card means that shaders are not supported, making the water texturing less realistic on the notebook. A PC with a two-Core CPU 1.86 GHz, RAM 1GB, and a Nvidia 9800GT 1GB graphic card which supports OpenGL 3.1 (so shaders are available).

2. The J2SE Development Kit v1.6.0, from http://www.java.com/en/.

3. The JOGL v1.1.1 package, from https://jogl.dev.java.net/.

4. JCreator 4.00 Pro, a Java program editor, from http://www.jcreator.com/.

5. FRAPS 2.99, a screenshot utility, from http://www.fraps.com/.

CHAPTER 2

LITERATURE REVIEW

This chapter presents background information on OpenGL, JOGL, programmable shaders, mesh and height maps, water surfaces and wave refraction, spring systems, collision detection, multitexturing, and particle systems.

2.1 OpenGL

OpenGL is a 3D graphics programming API [16]. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics API, bringing thousands of applications to a wide variety of computer platforms [17].

All OpenGL applications produce consistent visual results on any OpenGL API-compliant hardware, regardless of operating system or windowing system [17]. The API is supported on Windows, Unix, Linux, MacOS, OS/2.

OpenGL is well structured with an intuitive design. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features [17].

Early versions of OpenGL operated using a fixed-function graphics pipeline, which took geometry specified as vertices and pixel data as input, and produced pixels in the framebuffer as shown in Figure 2.1.

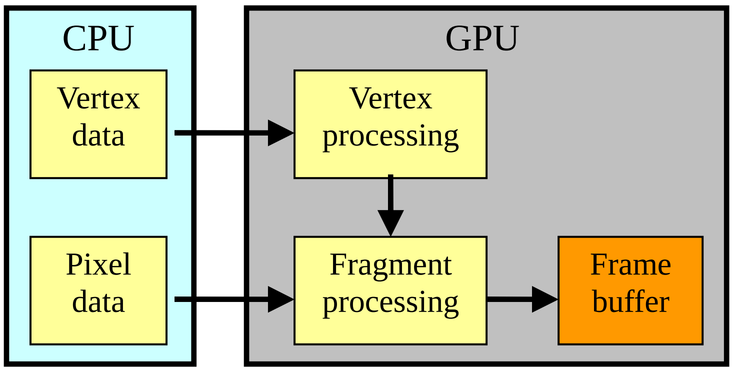


Figure 2.1: A preview of the OpenGL fixed-function graphics pipeline.

As geometric data travels down the pipeline, it goes through a series of conceptual stages that process the data [18].

OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering functions, texture mapping, special effects, and other forms of visualization [17]. OpenGL contains over 200 functions.

A small OpenGL example coded in C is outlined below, based on one by Angel [16]. It displays a white rectangle in a window as shown in Figure 2.2.

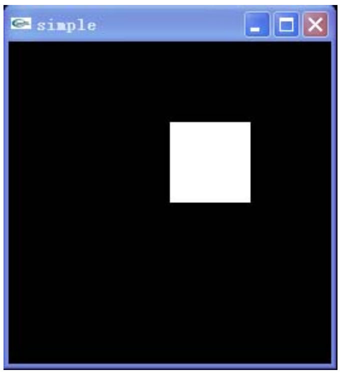


Figure 2.2: Output from OpenGL called from C.

The following is a fragment of the program; the full code is in Appendix A1. `glBegin()` specifies the type of the object (a polygon) and its vertices. `glEnd()` denotes the end of the vertex list.

```
glBegin(GL_POLYGON); //specifies the object type
//set the vertices of the object
   glVertex2f(0, 0);
   glVertex2f(0.5, 0);
   glVertex2f(0.5, 0.5);
   glVertex2f(0, 0.5);
glEnd(); //end of vertices
```

2.2 JOGL

This code in this thesis calls OpenGL via a Java binding called JOGL. JOGL can access most of OpenGL via two packages [19, 20].

1. The “javax.media.opengl” package, contains Java bindings for all the core OpenGL methods through version 2.0, as well as most OpenGL extensions defined at that time. OpenGL extensions incorporated into core OpenGL by version 1.3, are excluded.

2. The “javax.media.opengl.glu” package, contains bindings for the OpenGL Graphics System Utility (GLU) Library version 1.3, with the exception of the NURBS routines [19].

The Java version of the C example is shown in Figure 2.3; it is the same as Figure 2.2.

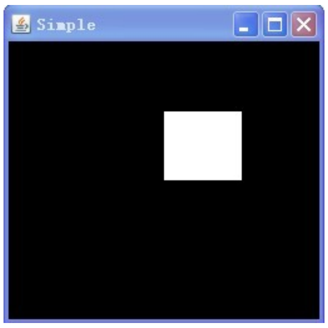


Figure 2.3: Output from OpenGL called from Java.

The following is a fragment of the program, the full code is in Appendix A2. The Java version is very similar to the C version: vertices are defined with `gl.glVertex2f()`, between `gl.glBegin()` and `gl.glEnd()`.

```
gl.glBegin(GL.GL_POLYGON); //specifies the object type
//set the vertices of the object
   gl.glVertex2f(0,0);
   gl.glVertex2f(0.5f, 0);
   gl.glVertex2f(0.5f, 0.5f);
   gl.glVertex2f(0, 0.5f);
gl.glEnd(); //end of vertices
```

2.3 GLSL Shaders

A shader is a program executed on the PC's GPU, a processor attached to the graphics card dedicated to calculating floating point operations for the display [18]. A GPU executes graphics primitive operations much faster than the host CPU.

For years, OpenGL operated using a fixed-function graphics pipeline, as shown in Figure 2.1, which gave the programmer a limited degree of rendering control through state settings. With shaders, the vertex and fragment processing stages become programmable, as shown in Figure 2.4. The programmer can write vertex and fragment shaders that will run at these stages in the pipeline in place of the predefined operations.

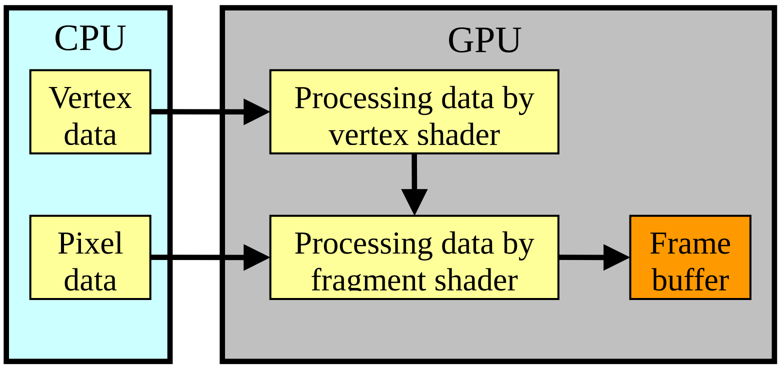

CPU
GPU
Vertex data
Pixel data
Processing data by vertex shader
Processing data by fragment shader
Frame buffer

Figure 2.4: A preview of the new OpenGL graphics pipeline with shaders.

Shaders are can handle special effects such as per-vertex coloring, transparency, and particles, so reducing the work of the CPU. Vertex and fragment shaders are written with the OpenGL Shading Language (GLSL), a high-level graphics programming language similar to C.

2.3.1 Vertex Shader

A vertex shader can replace/enhance the following fixed-function graphics pipeline features:

- Matrix manipulation (modelview, projection, texturing).
- Normal transformation, rescaling, and normalization.
- Texture coordinate generation.

- Per-vertex lighting.
- Point-size distance attenuation.

```
void main(void)
{
   vec4 a = gl_Vertex; //get vertex position

   //scale vertex in x, y, z coordinates
   a.x = a.x * 0.5;
   a.y = a.y * 0.5;
   a.z = a.z * 0.5;

   //calculate the new vertex position
   gl_Position = gl_ModelViewProjectionMatrix * a;
}
```

The vertex shader example below scales each vertex of a model by 0.5. Figure 2.5 shows the model before scaling (left picture) and after (right picture). A vertex position is stored in a 4D variable `a`, then its x, y, z coordinates are multiplied by 0.5. The new vertex position is output as `gl_Position`. The shader is automatically called for every vertex in the teapot shape.

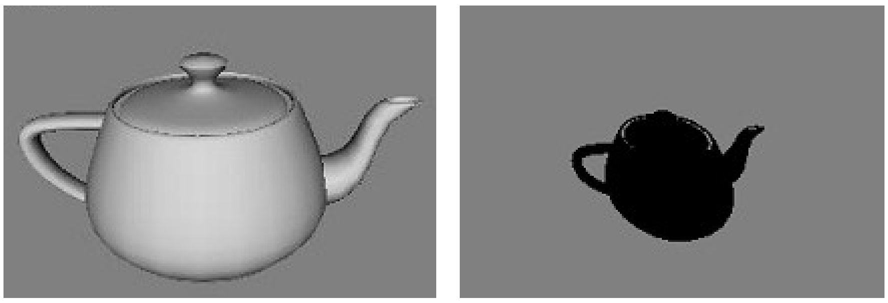


Figure 2.5: A model before (left picture) and after (right picture) scaling by a vertex shader.

2.3.2 Fragment Shader

A fragment shader can replace/enhance the following fixed-function pipeline steps:

- Texture application (including texture environment).
- Fog.
- Color sum.

The fragment shader below changes the color of all the pixels in a model to green. Figure 2.6 shows the model before (left picture) and after (right picture).

```
void main(void)
{
   //define the color to green
   gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

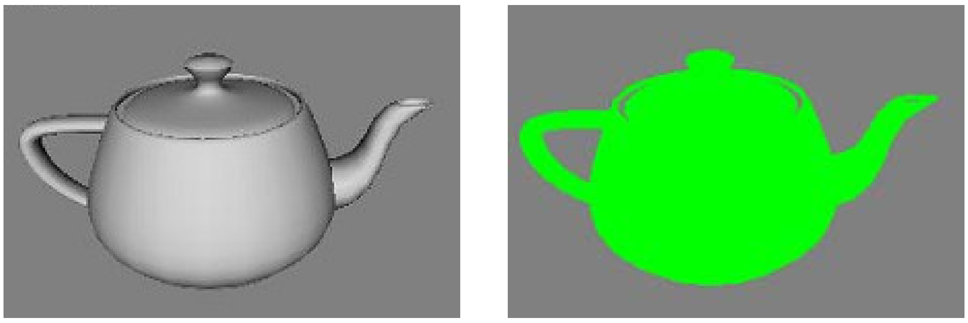


Figure 2.6: A model before (left picture) and after (right picture) being turned green by a fragment shader.

2.3.3 Using Shaders in Java

A Java/OpenGL shader program must read in the source code of the GLSL shaders, have OpenGL compile them, then use them for drawing, as in Figure 2.7.

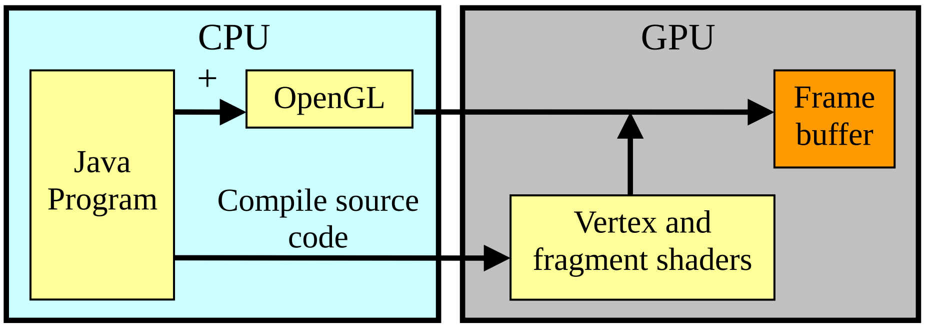


Figure 2.7: Java, OpenGL and GLSL shaders.

The code in this subsection is a fragment of a Java program to render a square with the help of the vertex and fragment shaders from Sections 2.3.1 and 2.3.2. Figure 2.8 shows the output without using shaders – a white polygon. Figure 2.9 shows the output with shaders – the white polygon is scaled by 0.5 and turned green.

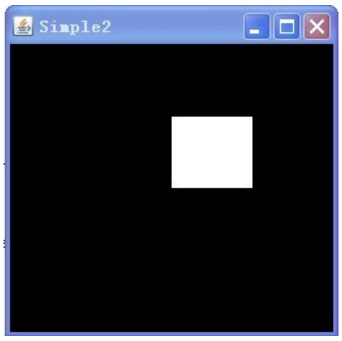


Figure 2.8: Output without shaders.

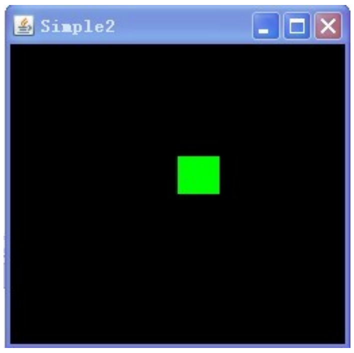


Figure 2.9: Output with shaders.

The following code segment links the shaders to JOGL, the full code is in Appendix A3. Empty vertex and fragment shaders are created first, then an empty shader program.

```
//empty vertex shader
drawVertex = gl.glCreateShader(GL.GL_VERTEX_SHADER);

//empty fragment shader
drawFragment = gl.glCreateShader(GL.GL_FRAGMENT_SHADER);

//empty shader program object
drawShaderProgram = gl.glCreateProgram();
```

The vertex shader source code is loaded as a string, stored in the vertex shader, and compiled. Once the vertex shader is compiled, the same is done to the fragment shader. Then the vertex and fragment shaders are attached to the shader program object.

```
//load vertex shader source code
BufferedReader brv = new BufferedReader(new FileReader("draw.vert"));
String vsrc = "";
String lineV;
while ((lineV = brv.readLine()) != null)
{ vsrc += lineV + "\n"; }
String Vsrc [] = new String [1];
```

```
Vsrc [0] = vsrc;

//put source code in the vertex shader
gl.glShaderSource(drawVertex, 1, Vsrc, null);

//compile vertex shader
gl.glCompileShader(drawVertex);

//loading, compiling of fragment shader

//attach vertex shader to the shader program object
gl.glAttachShader(drawShaderProgram, drawVertex);

//attach fragment shader to the shader program object
gl.glAttachShader(drawShaderProgram, drawFragment);

//link the program object specified by program
gl.glLinkProgram(drawShaderProgram);
```

The shaders are employed for drawing the polygon between `gl.glUseProgram()` and `gl.glUseProgram(0)` calls.

```
gl.glUseProgram(drawShaderProgram); //use shaders
gl.glBegin(GL.GL_POLYGON); //draw the polygon
   gl.glVertex2f(0,0);
   gl.glVertex2f(0.5f, 0);
   gl.glVertex2f(0.5f, 0.5f);
   gl.glVertex2f(0, 0.5f);
gl.glEnd();
gl.glUseProgram(0); //stop using shaders
```

2.4 Mesh and Height Map

The most straightforward way of representing a surface is to draw it as a rectangular regular mesh [21]. In the 2D plane, the vertices are aligned in a rectangle, a definite distance from each other as in a tessellation scheme. The vertices are connected by triangles or rectangles, as shown in Figure 2.10.

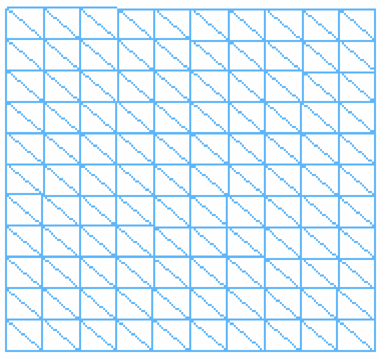


Figure 2.10: A quadratic mesh.

2.4.1 Height Map

A height map is a gray-scale image, whose pixels vary from 0 to 255: 0 is black, 255 is white [22]. Each pixel determines a height value using black as the minimum height and white as the maximum height. A mesh uses each pixel to determine the height for each of its vertices; the terrain is then displaced from 2D into 3D space.

Figure 2.11 shows a screenshot of the NeHe height map tutorial called "Beautiful Landscapes by Means of Height Mapping" written by Humphrey in 2001 [22]. The height map is shown in Figure 2.12.

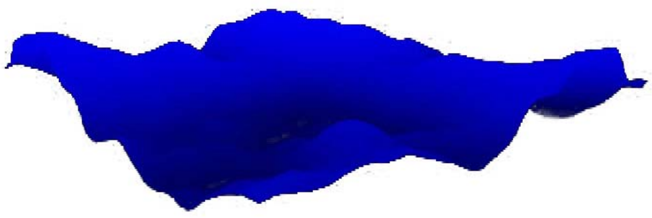


Figure 2.11: A terrain [22].

Figure 2.12: A height map [22].

Similar examples can be found in books by Hawkins and Astle [23, 24]; Figure 2.13 shows screenshots.

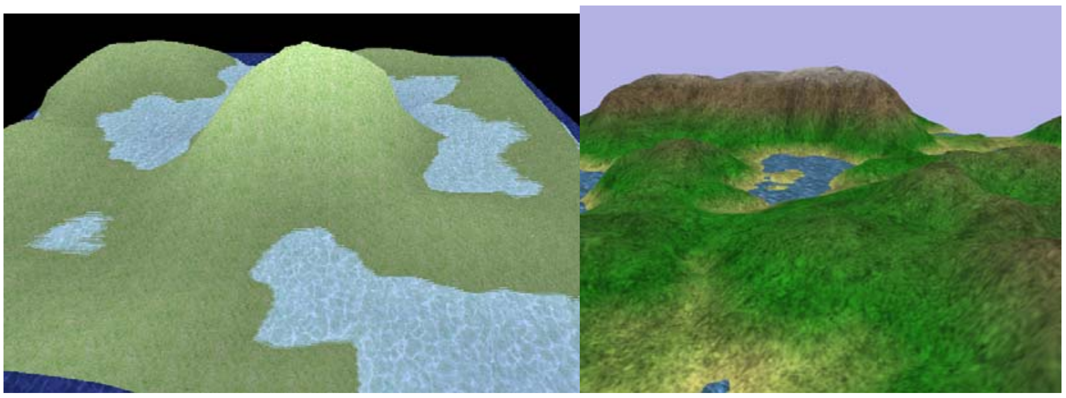


Figure 2.13: Hawkins and Astle’s terrains [23, 24].

2.5 Water Surface and Wave Refraction

The water surface must be rendered as a constantly changing mix of large, medium-size, and small waves in real time.

Water can be represented as a volume, but this is unnecessary in this project which is only concern with wave shape. Instead, the water can be represented as a height function of points in two-dimensional space (e.g. height = f(x, z)). This height function will displace the 2D mesh into 3D space.

2.5.1 Water Surface

A sine function is a familiar and easy way to compute height values as illustrated by Astle's example called "Flag" [18]. The sine equation below can be applied to a water surface as shown in Figure 2.14.

$$height = float(\sin((((x/5)*40)/180)*\pi))$$

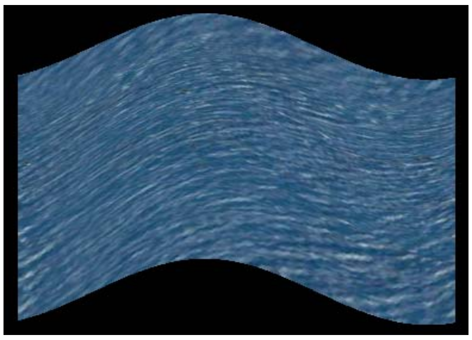


Figure 2.14: Sine function applied to a water surface [18].

Perlin Noise is another popular way to render water surfaces [3]. Guertault's example combines a sine function and a Perlin Noise function to produce waves as shown in Figure 2.15 [14]. Perlin Noise adds various frequencies and amplitudes to the smooth sine function to produce large and small waves. The code is in Appendix A4.

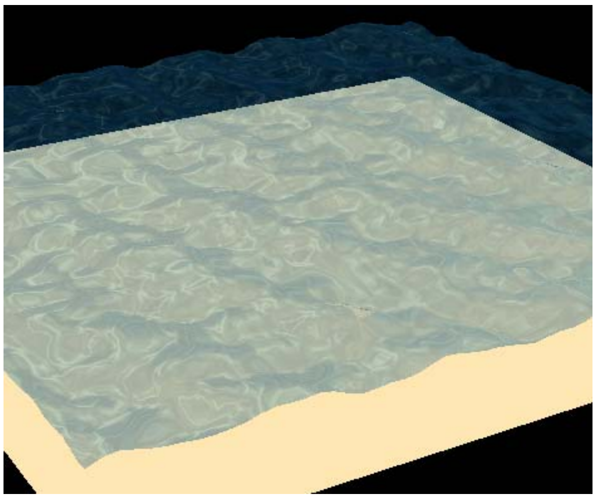


Figure 2.15: Guertault's water surface [14].

2.5.2 Wave Refraction

As waves approach the shore from deep water, the crests tend to become parallel to the shoreline regardless of their initial orientation, as a result of wave refraction [4]. Peachey rendered ocean waves by following the basic equation of liquid dynamics and separating it into phase function and wave profile [4].

The water is considered in three parts: deep-water, intermediate water, and shallow water as shown in Figure 2.16. The depth values, d=L/20 and d=L/2, are from Sverdrup's *Fundamentals of Oceanography* [25].

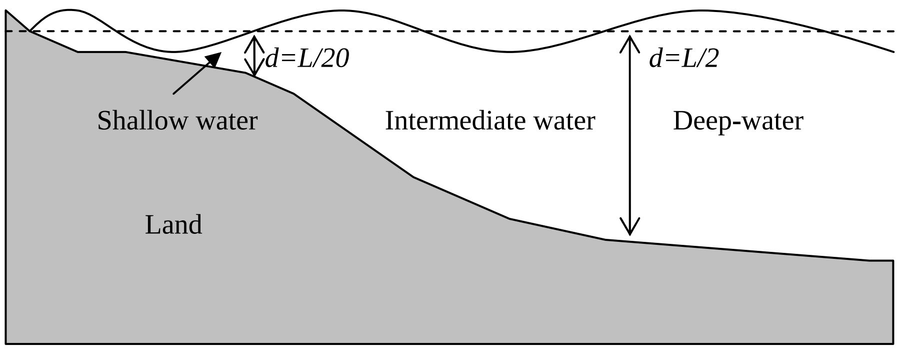


Figure 2.16: Deep, intermediate, and shallow water.

In deep water, the water must be deeper than one-half the wave's length. The propagation speed of a wave can be shown to be:

$$L = \frac{gT^2}{2\pi} \tag{2.1}$$

where L is the wavelength, g is gravity, T is the period [25].

When the wave enters shallow water, the water has a depth of less than one-twentieth of the wavelength, and the propagation of a wave is determined by:

$$L = T\sqrt{gd} \tag{2.2}$$

where d is the depth of the water [25].

Our model simplifies this three-part division into two, as shown in Figure 2.17. Shallow water begins at one-half the wave length.

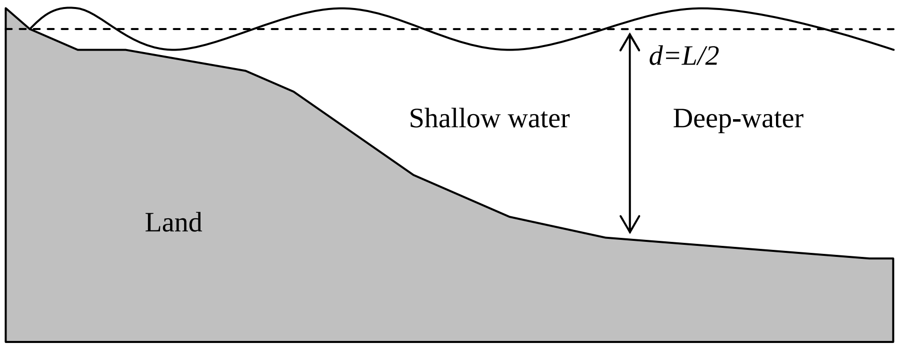


Figure 2.17: Deep and shallow water.

Phase Function

The implication of the dependence of the wavelength and speed on depth is that the phase function depends on the cumulative effects of the depth of the water between the wave origin and the point of interest, as shown in Figure 2.18. The phase value has the same period as the wave in the range [0, 1). In Figure 2.18, Phase 1 is at the 1/4 period point with the value 0.25; Phase 2 is at the 1/2 period point with the value 0.5; Phase 3 is at the 5/4 period point with the same value as the 1/4 period point (0.25).

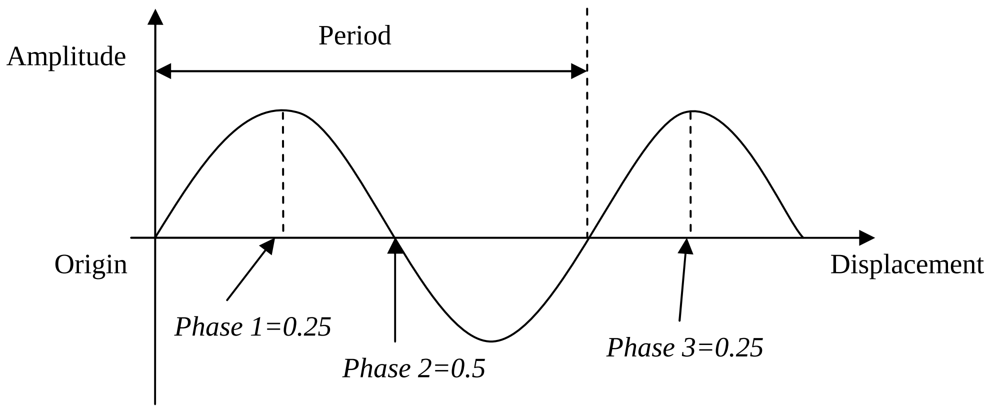


Figure 2.18: Phase value.

The phase function can be defined as:

$$\theta_i(x_i, z_i) = \frac{x_i}{L_i} \tag{2.3}$$

where x and z are the coordinates at the current vertex i, *L* is the wavelength at i.

Since the wavelength varies in water of varying depth, the phase function is a convenient way to calculate the height value. To calculate the phase value in 2D, the x and z coordinates must be considered at the same time, and the displacement offset from the wave origin is $\sqrt{x_i^2 + z_i^2}$. The phase function becomes:

$$\theta_i(x_i, z_i) = \frac{\sqrt{x_i^2 + z_i^2}}{L_i} \tag{2.4}$$

The phase function has a very simple dependency on the time t. Just as each wave component has the same constant period T at all points in space, it is also true that the wave component has the same constant rate of phase change at all points in space, namely the frequency. Thus, the time phase is added into the phase function:

$$\theta(x, z, t) = \theta(x, z, t_0) - \frac{t - t_0}{T} \tag{2.5}$$

The negative sign is necessary to make the waves propagate in the direction of increasing phase values. t is the current time, T the period. t_0 the start time is usually 0, so the phase function can be simplified to:

$$\theta(x, z, t) = \theta(x, z) - \frac{t}{T} \tag{2.6}$$

Wave Profile

The wave profile function is a single-valued periodic function of one parameter with a value between 0 and 1. For greater realism, the function is changed according to the wave steepness *S=H/L*, where H is double the amplitude for the simple sine wave, and *L* is the wavelength. When the steepness is small, a cosine function can be employed as the wave profile; when the steepness is large, a sharp-crested quadratic function can be substituted [4]:

$$w_i(\theta(x, z, t)) = 8\,|\,\theta(x, z, t) - 1/2\,|^2 - 1 \tag{2.7}$$

where $\theta(x, z, t)$ is the phase value from Equation 2.6. Normally, the amplitude is a constant, the steepness changes with wavelength, and the wavelength changes with the water depth (Equation 2.2). So the steepness is determined by the water depth, which can be simplified to a fixed depth for different wave profile functions.

The quadratic funtion has the effect of steepening the front of the wave crest and stretching out the back of the crest [4]. Figure 2.19 shows the difference between the cosine and quadratic functions for different phase values.

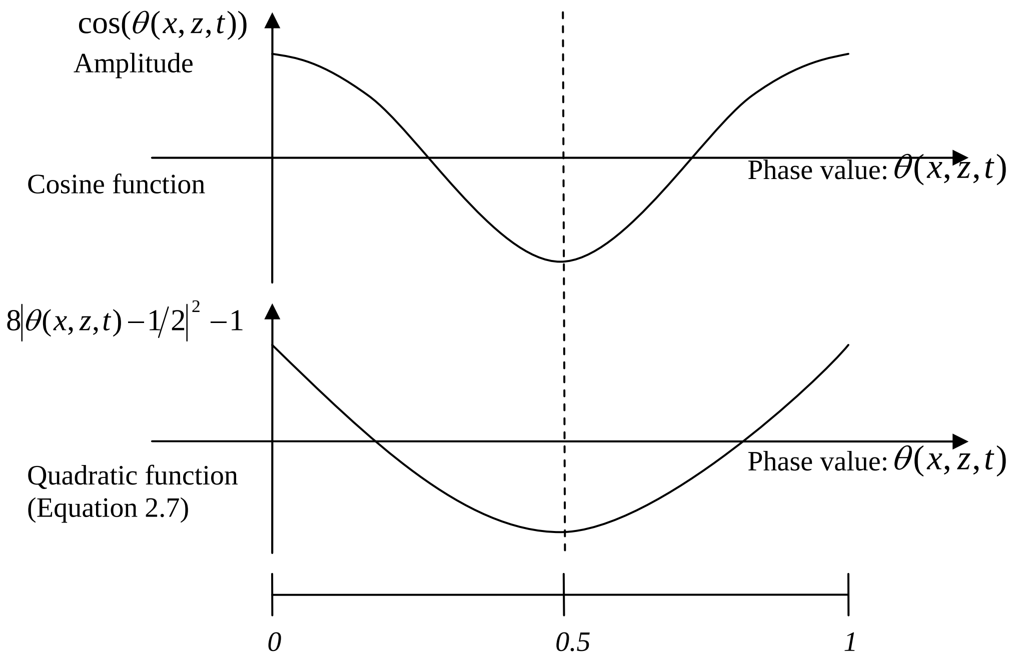


Figure 2.19: Cosine and quadratic functions.

2.6 Spring Systems

Spring systems will handle the water and land interactions, and collision detection will be used to control the interactions between the spring systems.

Springs that are not stretched or compressed beyond their elastic limit obey Hooke's law, which states that the force with which a spring pushes back is linearly proportional to the distance from its equilibrium length [11]:

$$F = -kx \tag{2.8}$$

x is the distance the spring is elongated or compressed, F is the restoring force exerted by the spring, and k is the spring constant or force constant of the spring.

Figure 2.20 is an illustration of Hooke’s law from Chapter 7 of *Fundamentals of Physics* [11]. The force exerted by a spring on a block varies with the block’s displacement x from the equilibrium position x = 0. (a) When x is positive (a stretched spring), the spring force is directed to the left. (b) When x is zero (the natural length for the spring), the spring force is zero. (c) When x is negative (a compressed spring), the spring force is directed to the right.

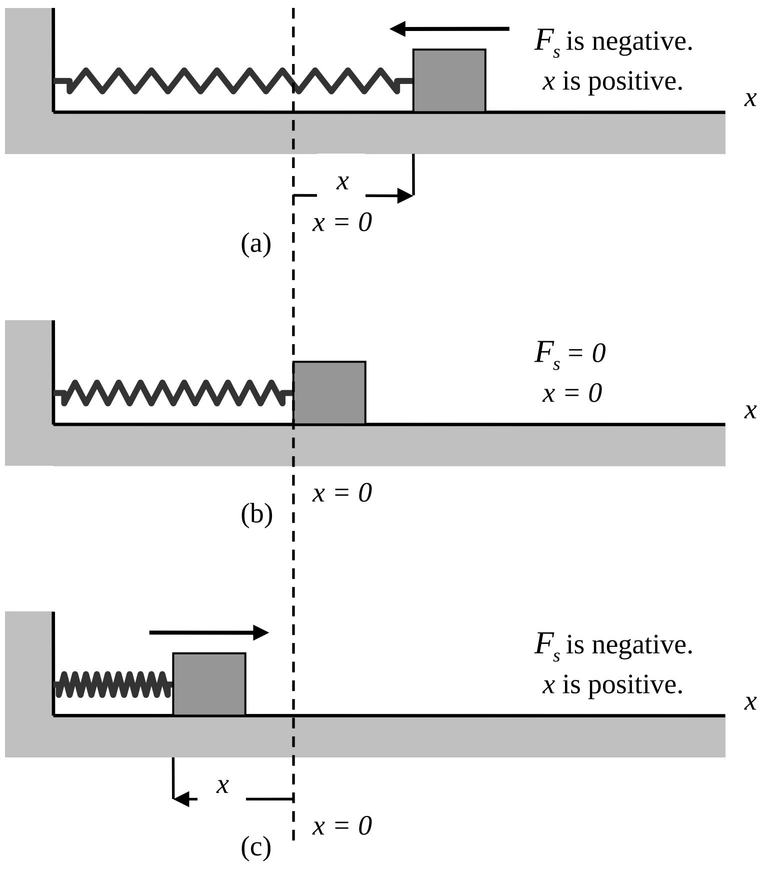


Figure 2.20: Hooke's law.

A spring system in our model consists of a network of water mesh nodes, connected by springs. Each spring has a resting length. When the distance between two nodes linked by a spring is equal to the resting length, the spring does not affect the nodes. When the distance between the nodes is greater than the resting length, the spring will apply a force to move them closer, and vice versa [26]. Figure 2.21 shows a simple spring system: a network of nine nodes connected by 12 springs.

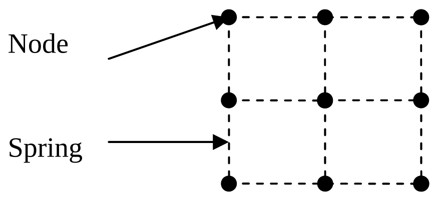


Figure 2.21: A spring system.

Let $p_i(t)$, $v_i(t)$, $a_i(t)$, where i=1,…, n, be respectively the positions, velocities, and accelerations of the mass points at time t. The system is governed by Newton's law $f_i = ma_i$, where m is the mass of each point and f_i is the sum of all forces applied at point p_i as shown in Figure 2.22 [27].

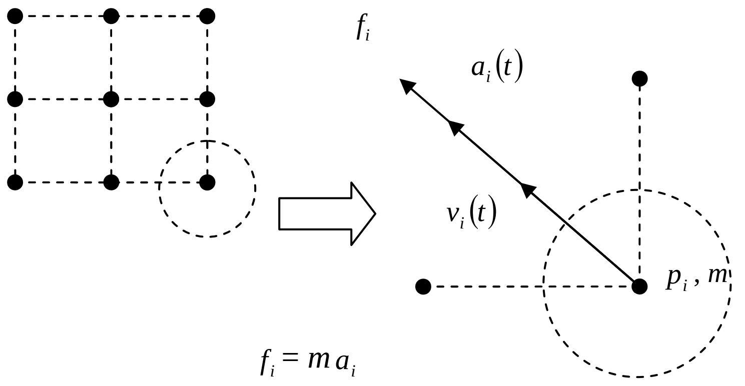


Figure 2.22: Spring system and Newton's law.

2.7 Collision Detection

Collision detection, together with the spring system, helps to control the interaction between the water and the land. In this section, the collision detection equations used in this model are introduced.

Collision between a Point and a Circle

In Section 5.3.1, water/land collision is represented by a point and a circle. The equation of a circle is:

$$(x-a)^2 + (y-b)^2 = r^2 \qquad (2.9)$$

where a and b are the coordinate of the centre of the circle, r is the radius of the circle, x and y are the coordinate of a point on the circumference. So, if the point (x, y) hits or enters the circle, which denotes a collision, then the equation will be:

$$(x-a)^2 + (y-b)^2 \leq r^2 \qquad (2.10)$$

Collision between a Point and a Line Segment

In Section 5.3.2, the collision between waves is viewed as a point touching a line segment. However, the path of the point over a time interval is a line segment, so line segments intersection can be checked instead of point/line touching. Equations 2.11 and 2.23 are the equations of two line segments:

$$(x - x1)/(x2 - x1) = (y - y1)/(y2 - y1) \tag{2.11}$$

$$(x - x3)/(x4 - x3) = (y - y3)/(y4 - y3) \tag{2.12}$$

where (x, y) are the coordinate of the collision point, and (x1, y1), (x2, y2), (x3, y3), (x4, y4) are the four end points of the two line segments. If the two line segments intersect (i.e. collide), then the two equations can be solved for (x, y).

2.8 Texture Mapping

Texture mapping allows us to attach images to polygons to provide more realistic graphics. In this section, procedural texture generation, detail texturing, and multitexturing are introduced [28].

2.8.1 Use a Texture Map

Once a texture image has been loaded into memory, the texture coordinates determine how to map the texture onto a polygon. In OpenGL, the lower-left corner of a texture is the coordinate (0, 0), while the upper-right corner is (1, 1) [24]. For a 2D texture, the coordinates are assigned the notation (s, t), where s and t can vary from 0 to 1. Figure 2.23 shows a 512*512 texture in (s, t) space and as the original image [16].

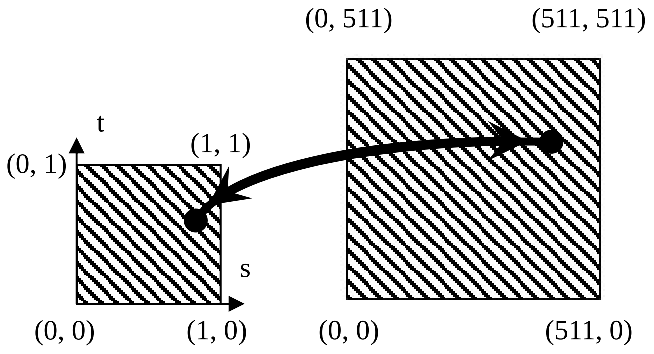


Figure 2.23: Texture map in (s, t) space and as the original image.

2.8.2 Procedural Texture Generation

For greater realism, a landscape should utilize more than one texture. Franke introduced such a technique for adding snowy mountains and sandy beaches to his terrain [29]. The left picture of Figure 2.24 shows the height map for the terrain, with black denoting a 0 height and white 256. The right-hand picture of Figure 2.24 is a procedural texture based on the height map with the textures changing with the terrain's height. Figure 2.25 shows procedural texturing applied over the terrain – the highest region is snow, then rocks, grass, and the lowest is sand.

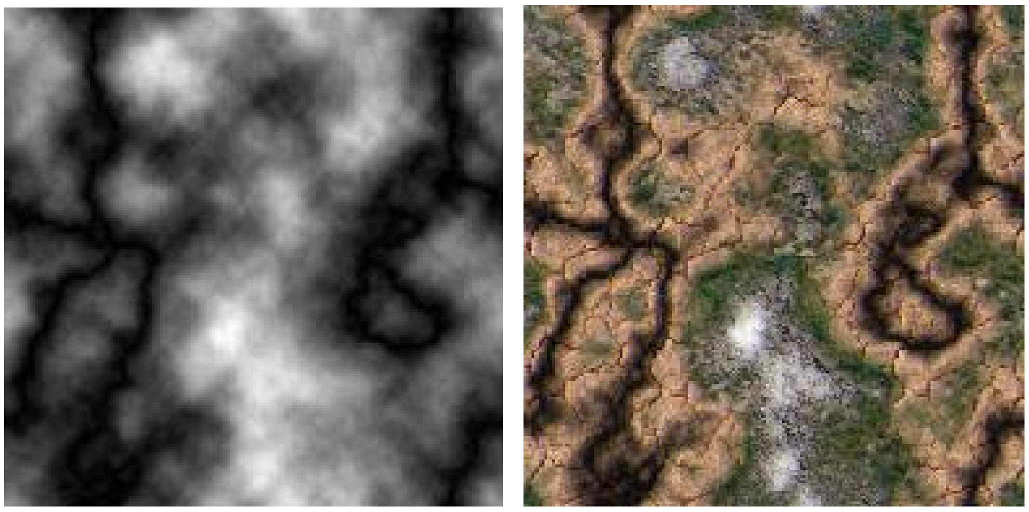


Figure 2.24: The height map of the terrain (left) and procedural texturing based on the terrain's height (right) [29].

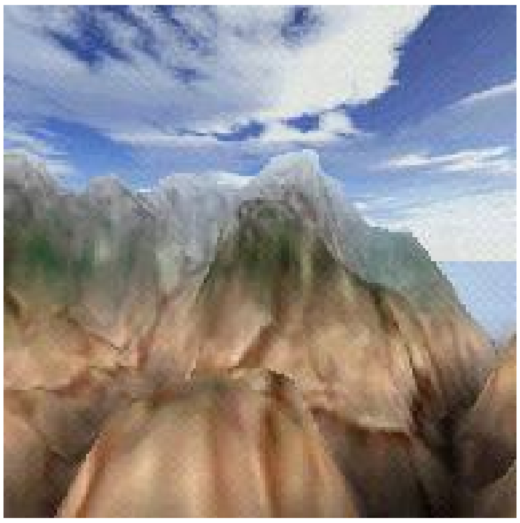


Figure 2.25: Procedural texturing applied over the terrain [29].

Franke divided the terrain into four regions based on height values (the lowest point is 0, the highest 256):

Region 1 (Snow): 192-256

Region 2 (Rock): 128-192

Region 3 (Grass): 64-128

Region 4 (Sand): 0-60

Snow and rocks textures appear on the mountain tops, grass on the plains, and sand on the beaches. The four texture maps (Snow, Rock, Grass, Sand) are combined to get a single new texture.

The combination can utilize the RGB color values of each pixel, by deciding how many percent of the color values should be visible for each component texture. Franke calculates four texture maps percentages according to the height of the terrain at a pixel, as shown in Figure 2.26.

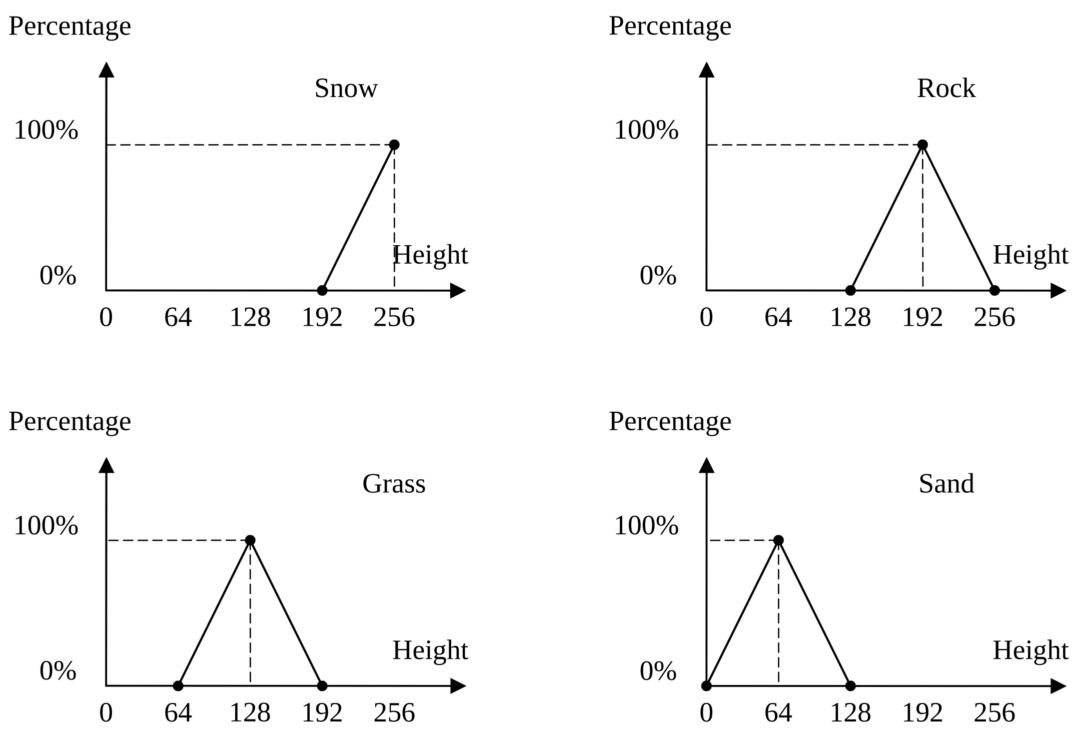


Figure 2.26: The percentage of four textures combined at different heights.

For example, at a height of 200, the resulting texture is a mix of snow and rock. The percentage of snow is 12.5% ($(64 - abs(256 - 200))/64 = 12.5\%$) and the percentage of rock is 87.5% (1-12.5%).

2.8.3 Detail Texture

When a terrain is large, a normally-sized texture is stretched so much that it loses resolution. A larger texture may solve the problem, but impacts performance, and its size may be limited by the graphics card. Detail texturing adds high levels of detail, such as cracks, bumps, and rocks, to any size of terrain, and is simple to use.

A detail map is a grayscale texture, like the one in Figure 2.27, that is repeated many times over a landscape [28].

Figure 2.27: Detail texture map [28].

Normally, detail texturing is applied as a second (or even third) texture to a terrain, so multitexturing must be utilized [12].

2.8.4 Multitexturing

Multitexturing combines several textures into one. Guertault's example, called "Simple Water Rendering", applies multitexturing to water [14]. This is sometimes called *mix-texturing* because two images are employed – one is a color image for the texture, the other a gray image for alpha values. Figure 2.28 shows the two images used in Guertault's example.

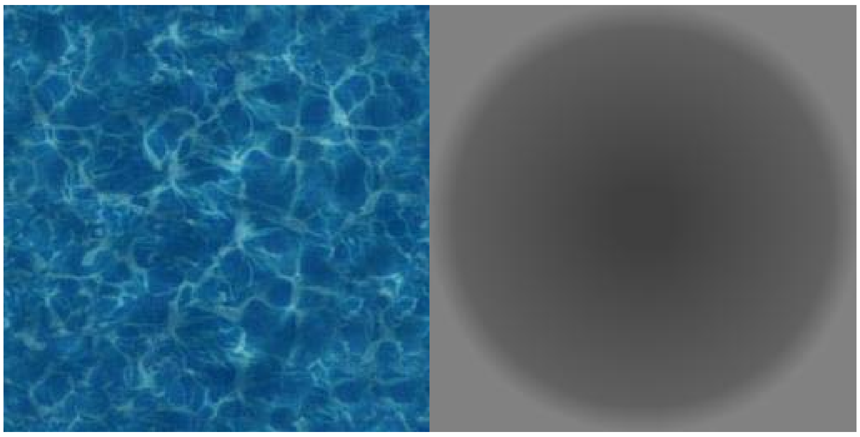


Figure 2.28: Color image (left) and gray image (right) for mix-texturing [14].

The color component of a vertex is obtained from the color image, while its alpha value (i.e. its transparency level) is read from the gray image.

2.9 Particle Systems

A particle system can model fuzzy objects such as fire, clouds, and water [15].

Figure 2.29 shows green and blue fireworks (left) and multicolored fireworks (right) created by particle systems in Reeves's paper [15]. Particle systems model an object as a cloud of particles that define its volume. Over a period of time, new particles are added to the system, move and change, and finally disappear. Particle-based special effects will be employed in our model to represent water spray and breaking wave foam.

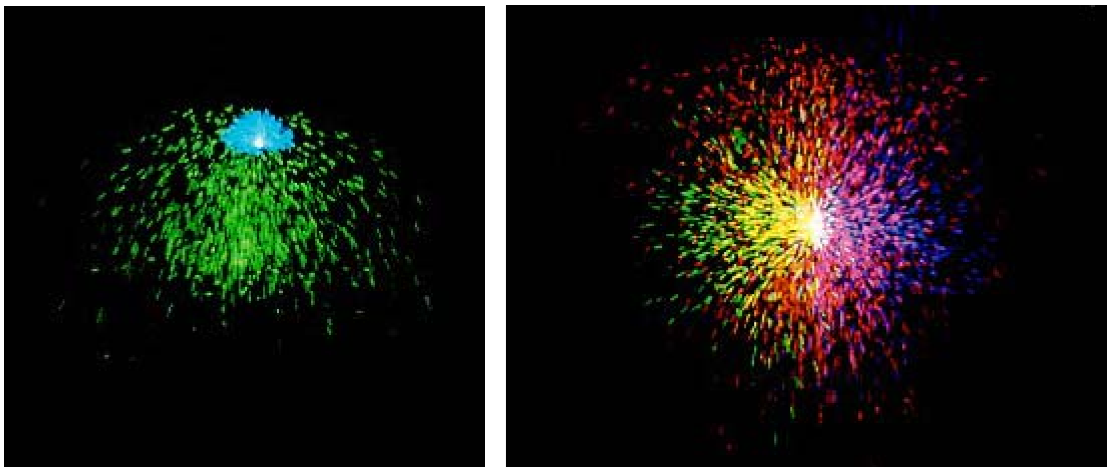


Figure 2.29: Green and blue fireworks (left) and multicolored fireworks (right) made by particle systems [15].

A particle system is a collection of individual particles. Each particle has individual attributes, such as velocity, color, and life, and does not interact with other particles. Particles share some attributes, such as mass, and force, which allow them to exhibit common behavior [18]. Figure 2.30 shows particles with different positions and velocities, being affected by gravity and other forces.

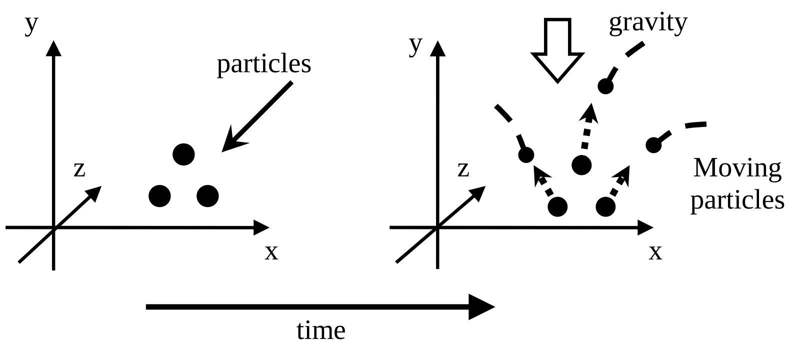


Figure 2.30: Particles with different positions, velocities and forces.

Normally, particles have the following attributes:

- Position: the coordinate of a particle in 3D space, which is affected by the particle's velocity.
- Velocity: a particle stores its velocity as a vector representing both speed and direction in order to update its position.
- Life: particles are created, age, die, and may be reborn.
- Size: different particles may vary in size.
- Mass: a particle's mass determines how it will be affected by external forces.
- Force: the forces acting on a particle add to its realism.
- Rendering: a particle may be rendered as a point sprite, very short line, or texture-mapped quadrilateral.

2.10 Water Effects

The model in this thesis will employ two particle-based water effects – water spray and breaking wave foam.

Water spray particles have random velocities, various sizes, and keep appearing and disappearing. Figure 2.31 shows the water spray around a piece of real coastline.

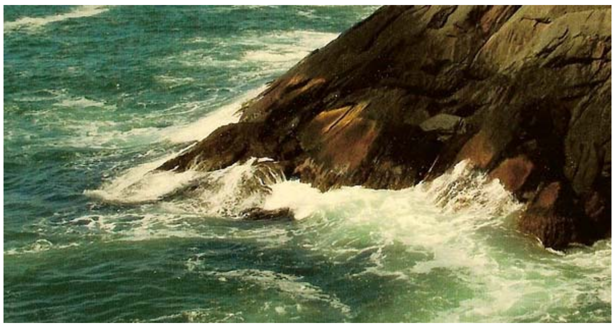


Figure 2.31: Water spray.

Breaking waves are accompanied by foam on top of the waves as shown in Figure 2.32.

Figure 2.32: A breaking wave and foam.

There is a maximum possible wave height for any given wavelength, determined by the ratio of the wave's height to its length (steepness). When the steepness *H/L* exceeds 1:7, the wave becomes too steep and breaks. This ratio can be approximated by the angle formed at the wave crest, which is about $120°$, as shown in Figure 2.33 [25].

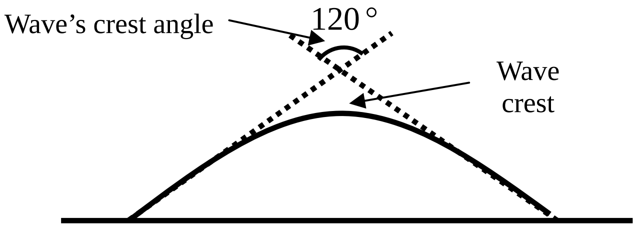


Figure 2.33: The wave's crest angle approaches $120°$ and the wave breaks.

2.11 Summary

This chapter introduced OpenGL, JOGL, and programmable shaders. Mesh and height maps will be used to build the land and water surfaces in Chapter 3; realistic water surfaces with waves are examined in Chapter 4; spring systems and collision detection are employed for the interaction between water and land in Chapter 5; texture mapping for more realistic land and water surfaces is the topic of Chapter 6; particle system are used to create water effects in Chapter 7.

CHAPTER 3

MESH CREATION

The land and the water surfaces in this project are created from meshes, whose construction is explained in this chapter.

3.1 Create the Land Mesh

A 128*128 land mesh is created in the 2D plane (x-z plane) and then displaced into 3D space with the height map shown in Figure 3.1 [22].

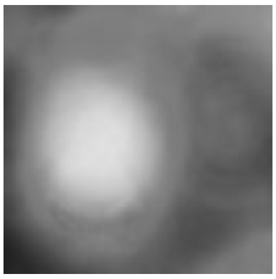


Figure 3.1: The height map for the land.

The Loadpixels class reads pixels information from the height map and converts it into land heights. Figure 3.2 shows its class diagram, and the full code is in Appendix A5.

Loadpixels (in Appendix 5)

`+ Loadpixels(String filename)`
`- getImage(String filename)`
`- getImagePixels(Image image, int pixels[])`

Figure 3.2: Loadpixels class diagram.

The first stage of Loadpixels (shown in Figure 3.3) is to load the height map into an image called `bufferImage`, and extract its dimensional information.

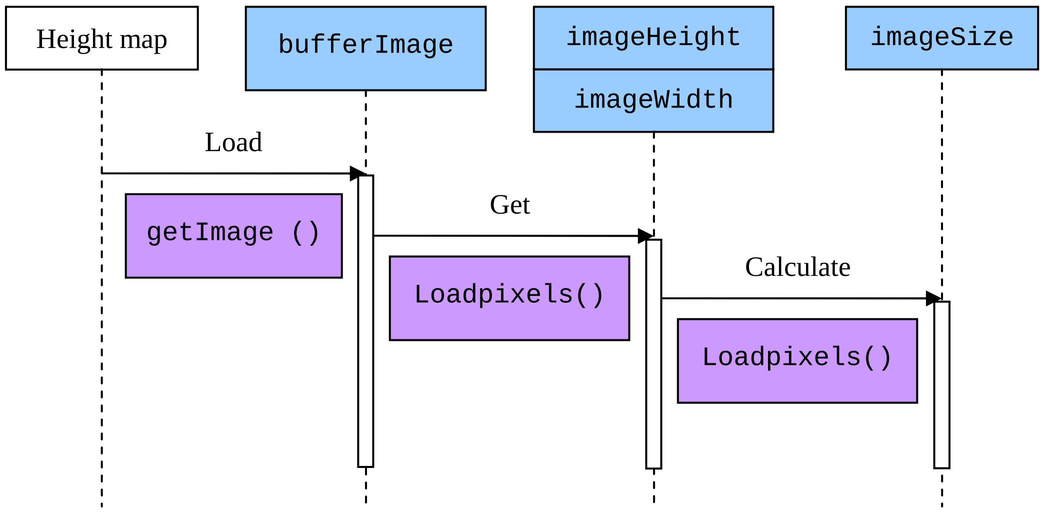


Figure 3.3: Loading the height map.

The second stage converts the image into land height data (see Figure 3.4).

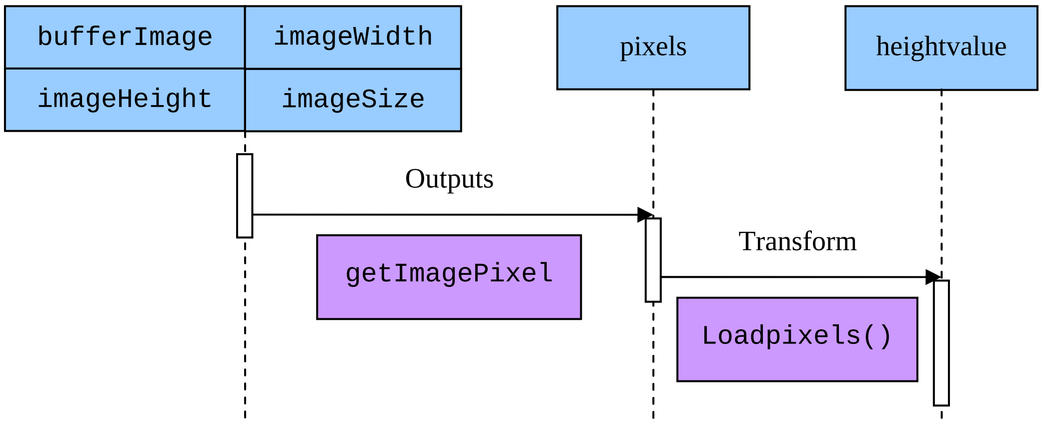


Figure 3.4: Calculating the height data.

The y values for the land mesh are calculated from the height data. The land mesh is drawn in 3D space by using triangle strips, as shown in Figure 3.5. Figure 3.6 shows a detail of part of the land mesh.

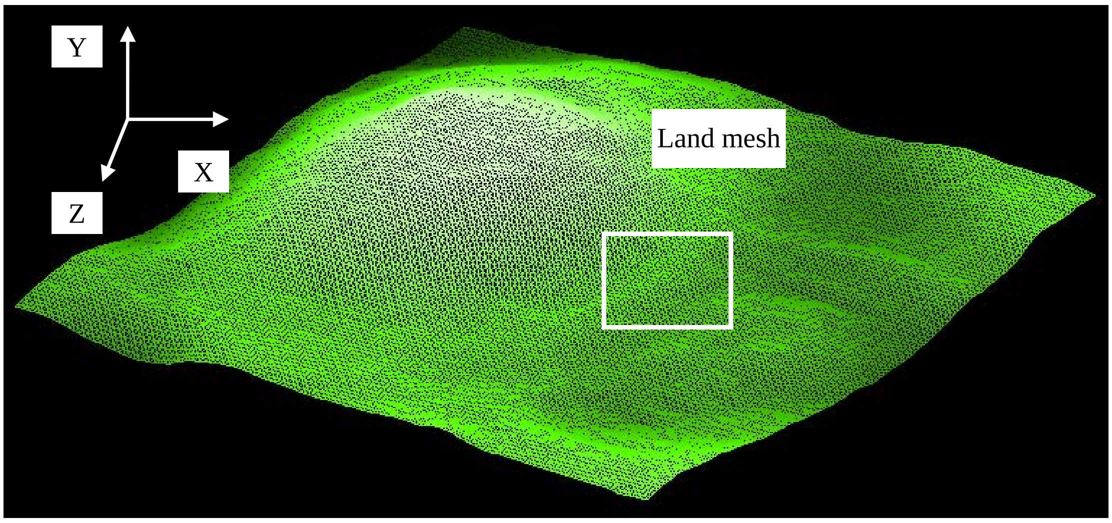


Figure 3.5: The land mesh.

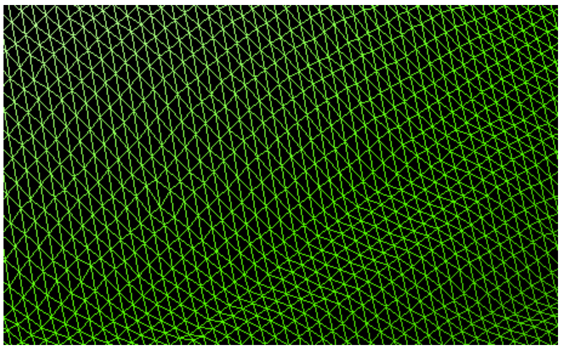


Figure 3.6: A part of the land mesh.

The following pseudocode shows how to adjust the x, y, z values of the land vertices (the full code is in Appendix A6). The land's y value is divided by 8 to make it less steep, and the x and z coordinates are offset so the land is centered in the middle of the screen.

```
for (z less than landsize)
   for (x less than landsize)
      land'x value = x-(landsize/2);
      land'y value = heightvalue/8;
      land'z value = -z+(landsize/2);
```

3.2 Create the Water Mesh

To make it easier to calculate the coastline and wave curves in Chapter 4, the water mesh is created using the same x and z values as the land. The initial water height is 19, but will be varied in Chapter 4.

Figure 3.7 shows the water and the land meshes drawn with triangle strips, and Figure 3.8 is a top view of the water and land meshes showing that they are structured identically.

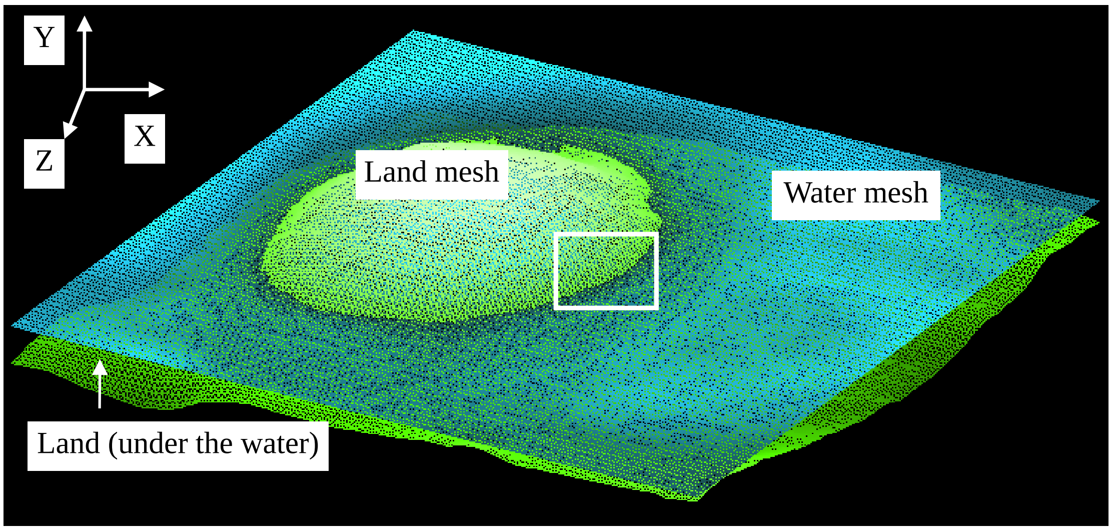


Figure 3.7: The water and the land meshes.

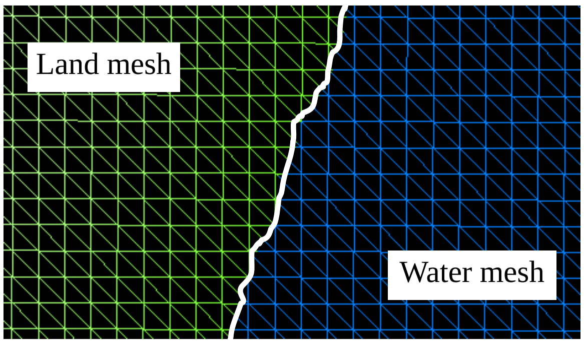


Figure 3.8: A top view of part of the water and land meshes.

The following pseudocode shows how the water mesh is centered on the screen and its height fixed at 19 (the full code is in Appendix A6).

```
for (z less than watersize)
   for (z less than watersize)
      water’s x value = x-(watersize/2);
      water’s z value = -z+(watersize/2);
      water’s y value = 19;
```

3.3 Summary

This chapter explained how the 3D land mesh employs a height map, and how a static water mesh is combined with it. The water will start to move in the next chapter.

CHAPTER 4

THE WATER SURFACE

In this chapter, phase functions are employed to create a moving water surface with wave refraction. Novel water surface data structures are also introduced in this chapter – the coastline and four wave curves – which will help us to efficiently model water interaction in Chapter 5.

4.1 Wave Height

Water vertices need to move up and down to create waves, and thereby produce a realistic animated water surface. This is achieved by combining Equations 2.4 and 2.6 of Section 2.5.2 to get the phase function:

$$\theta_i(x_i, z_i, t) = \frac{\sqrt{{x_i}^2 + {z_i}^2}}{L_i} - \frac{t}{T} \tag{4.1}$$

x and *z* are the coordinates at the current vertex *i*, *t* is current time, *T* is the period, and L_i is the wavelength at *i*. Since almost all the water is shallow in our model, it is possible to use Equation 2.2 in Section 2.5.2 calculate the wavelength.

The following pseudocode calculates the phase value using Equation 4.1 (the full code is in Appendix A7). The water depth is used to decide on the choice of wave length (i.e. based on Equation 2.1 or 2.2).

```
calculate deep-water wavelength;
calculate water depth;
if(water depth large than a half deep-water wavelength)
   use deep-water wavelength;
else
   calculate and use shallow-water wavelength;
calculate phase function;
```

The sharp-crested quadratic function (Equation 2.7 in Section 2.5.2) is used as the wave profile function. The phase function (Equation 4.1 above) should be restricted to the range [0, 1) in the wave profile function, which results in the final profile function:

$$Y_i = 8*(\frac{\sqrt{x_i^{\ 2} + z_i^{\ 2}}}{L_i} - \frac{t}{T} - \frac{1}{2})^2 - 1 \qquad (4.2)$$

i is the index of a vertex, *Y* is the wave height of the vertex, *x* and *z* are the (X, Z) vertex coordinate, *t* is the time which increases by 0.1 in each frame. *T* is the period of the function, which is set to 80 frames to look realistic, and *L* is the wavelength at the vertex position. The function is shown in Figure 4.1.

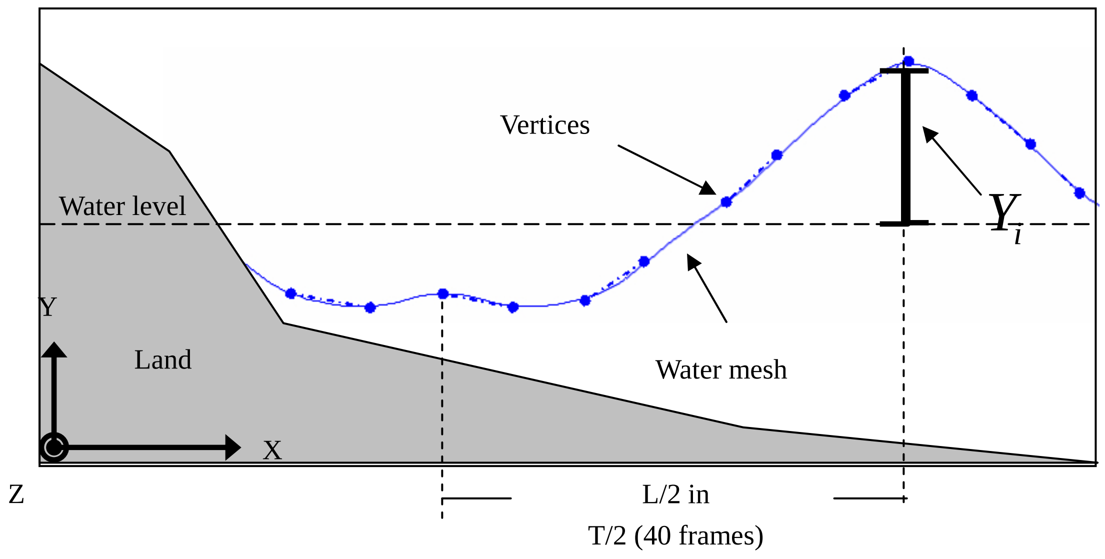


Figure 4.1: Wave height derived from the final profile function.

Four versions of this profile function, with different periods and amplitudes, are combined. The resulting height function creates pleasingly varied waves and wave refraction. T_1 is the period of the first version of profile function, and A_1 is its amplitude. T_2, T_3, T_4 and A_2, A_3, A_4 are the periods and amplitudes of the other versions of the profile function.

$$T_1 = 80\, frames, \quad A_1 = 0.8;$$

$$T_2 = \frac{1}{2}T_1, \quad A_2 = \frac{1}{4}A_1;$$

$$T_3 = \frac{1}{4}T_1, \quad A_3 = \frac{1}{16}A_1;$$

$$T_4 = \frac{1}{8}T_1, \quad A_4 = \frac{1}{64}A_1$$

Figure 4.2 shows the shapes of the four basic profile functions as blue curves. The shape of the combined profile function (labeled as Y) is drawn as by a black dotted curve.

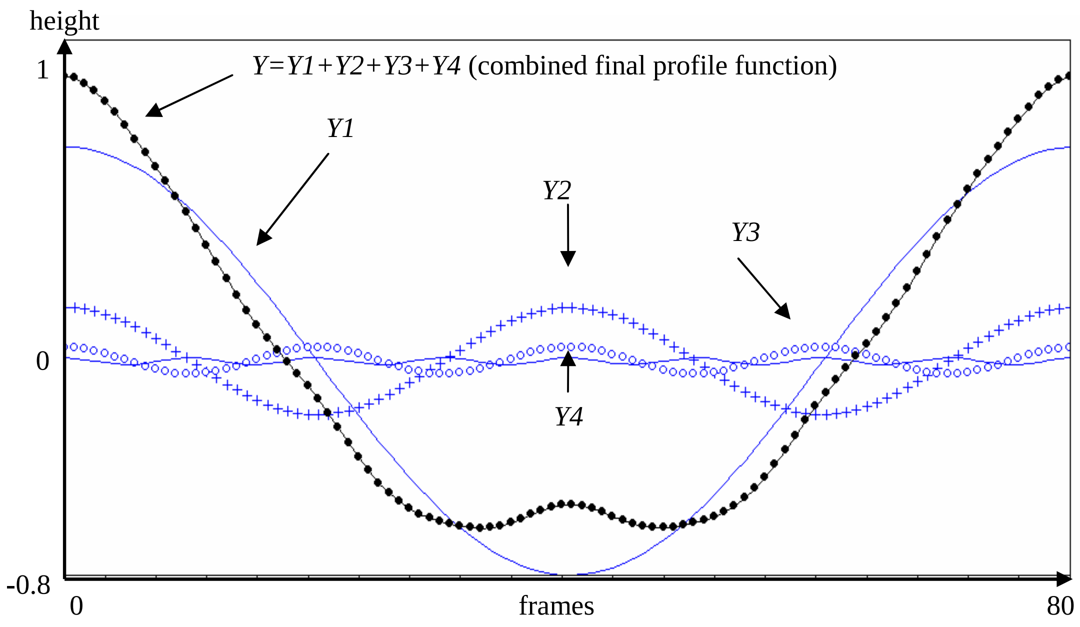


Figure 4.2: The shapes of the four versions of the profile function and their combination.

Figure 4.3 shows that the combined profile function produces a water surface with waves. Figure 4.4 is a close-up view showing the wave crests parallel to the land, demonstrating wave refraction as defined by Peachey [4].

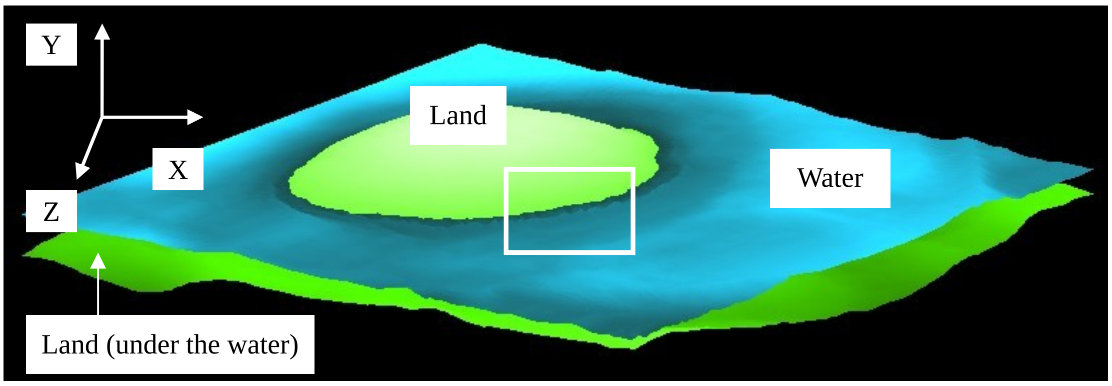


Figure 4.3: Water surface with waves.

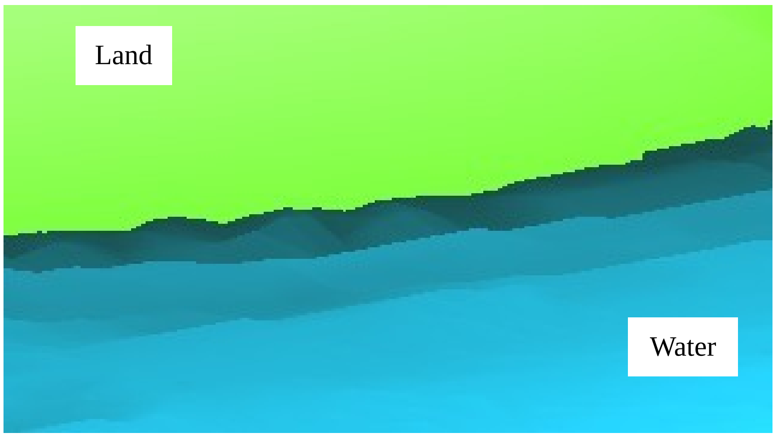


Figure 4.4: The crests of the waves parallel to the land.

4.2 Coastline and Wave Curves

One of the novel aspects of this work is the utilization of coastline and wave curve data structures to represent the interaction between the water and the land. They will be utilized in the spring systems and collision detection of Chapter 5, but the creation of these data structures will be described in this chapter.

Figure 4.5 shows that the coastline is the series of water vertices closest to the land. They represent the places where water/land interaction will occur.

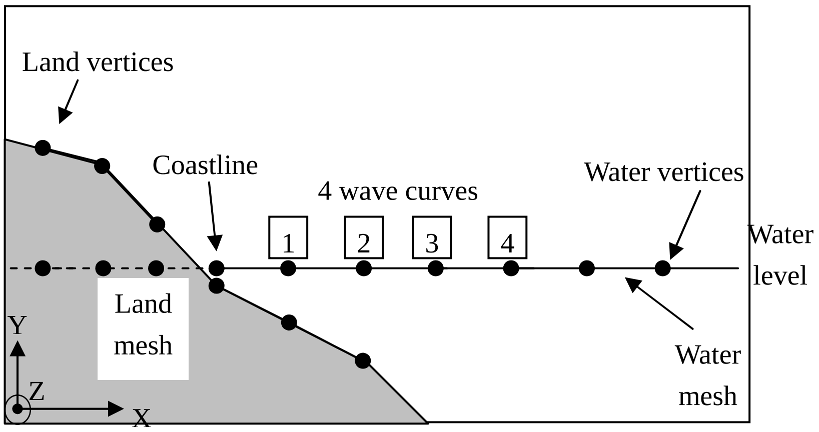


Figure 4.5: The coastline and wave curves at rest position.

Wave curves 1 and 2 will be linked by height springs to improve the realism of vertical wave movement. Wave curves 3 and 4 will be linked by position and wave springs for modeling water/land interaction over the horizontal X-Z plane. It is possible to use more wave curves in the model, but four were chosen as a balance between interaction realism and computational efficiency.

Wave curve 1 is the line of vertices one mesh interval away from the coastline, wave curve 2 is the line of vertices one mesh interval away from wave curve 1, and so on to wave curves 3 and 4.

4.2.1 Building the Coastline

As water vertices move up and down, they cut the land mesh along a series of line segments. These segments are collected, as the first approximation for the coastline. The cut height is made a little higher than the default water level (19). A cut is noted when an edge has one end point above the cut height, and one below.

The aim is to produce a series of edges which form a closed curve, as in Figure 4.6. Unfortunately, the initial collection of edges looks more like Figure 4.7, which includes unnecessary line segments and vertices, and segments that are out of order.

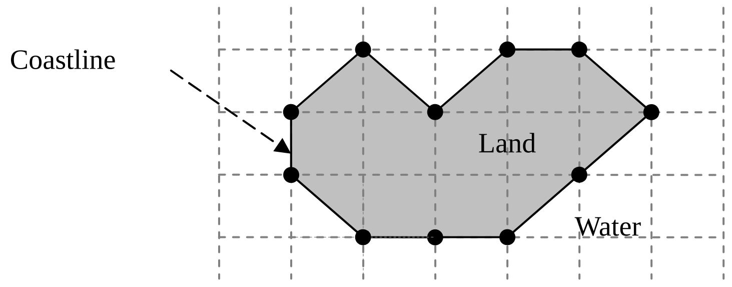


Figure 4.6: What the final coastline should look like.

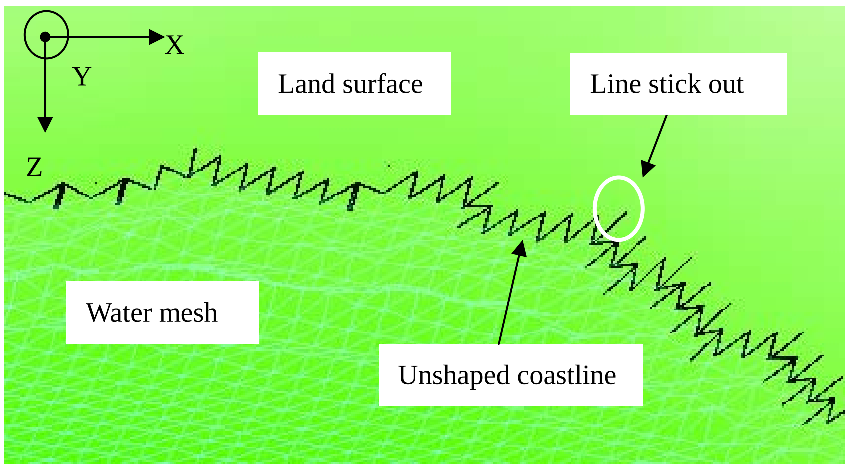


Figure 4.7: Unshaped coastline.

The sequence diagram in Figure 4.8 shows the steps in ‘tidying up’ the coastline – deleting useless line segments, ordering line segments, and deleting useless vertices.

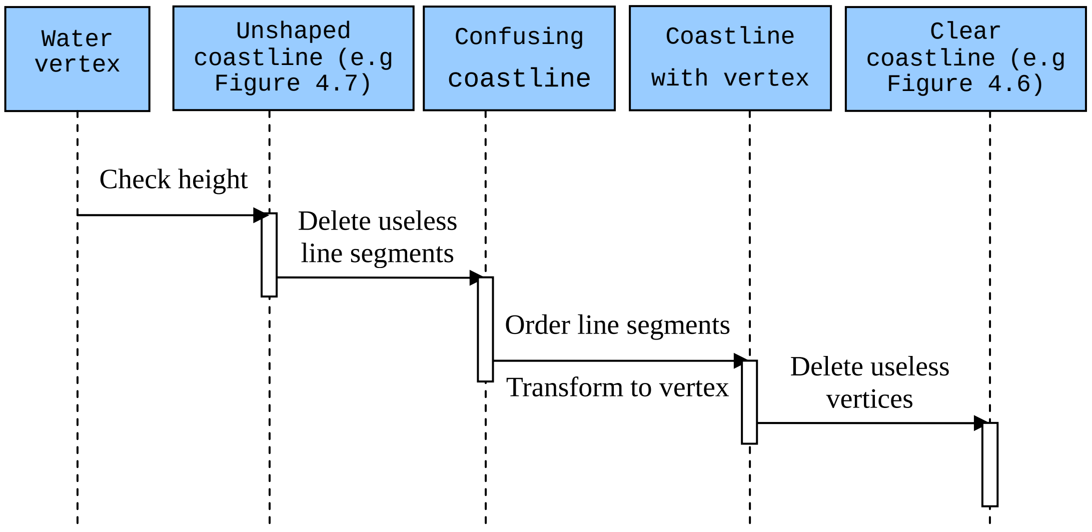


Figure 4.8: The steps of get the coastline.

Deleting Useless Line Segments

A useless line segment is one that ‘sticks out’ from the coastline. In Figure 4.9, line segment 3 ends with a vertex which is not used by any other line segments, and so line segment 3 can be tagged as ‘sticking out’, and so be deleted.

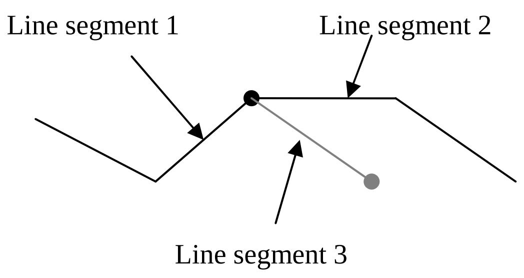


Figure 4.9: Useless line segment.

Figure 4.10 shows the coastline after useless line segments have been deleted.

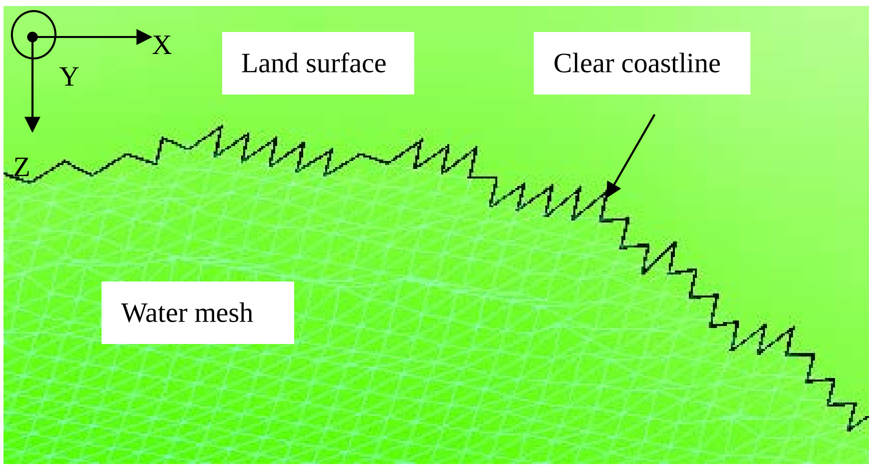


Figure 4.10: Clear coastline.

This seems satisfactory until the coastline is stored as an OpenGL line strip (an optimized version of the original lines shape which is a smaller size and will render more quickly). When the strip is drawn, the result is shown in Figure 4.11.

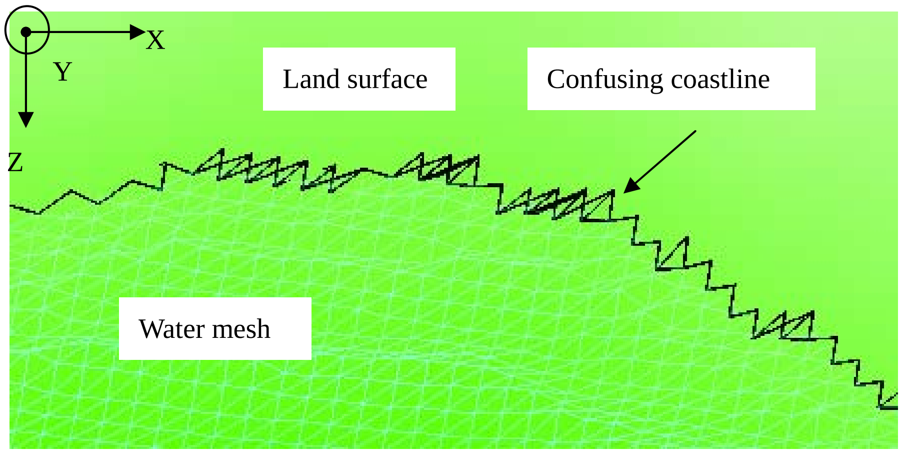


Figure 4.11: Confusing coastline.

Ordering Line Segments

When a series of edges are represented as distinct line segments, as in Figure 4.12, edges can be drawn between any vertices in any order. In this case five edges are drawn 1-2, 3-4, 5-6, 7-8, 9-10.

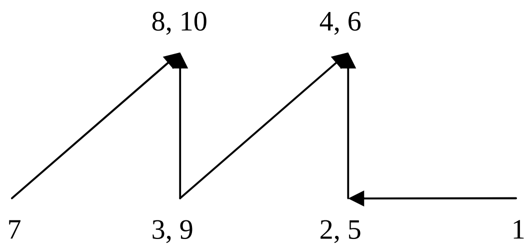


Figure 4.12: Drawing line segments.

However when the same shape is encoded as a line strip, additional lines will be drawn between adjacent vertices (between line segments 2-3, 4-5, 6-7, and 8-9). The result is shown in Figure 4.13.

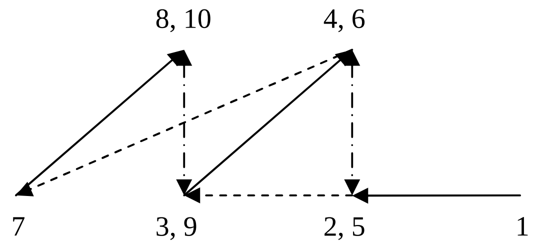


Figure 4.13: Drawing a line strip.

The start and end vertices for each line segment must be adjusted to avoid this problem. A line segment is chosen with a given direction, and another line segment found which can connect to it. An end point of a line segment must connect to the start point of another line segment, except for the two endpoints of the coastline. In Figure 4.14, the gray line is the initial line segment with a direction. Lines 1, 2 and 3 need to be adjusted. Line segment 1 can be connected to the end point of the gray line segment, but needs to be upended (i.e. the start and end points must be exchanged). Line segment 2 can be connected to the start point of the gray line segment without upending. Line segment 3 can be connected to the start point of line segment 2 after upending.

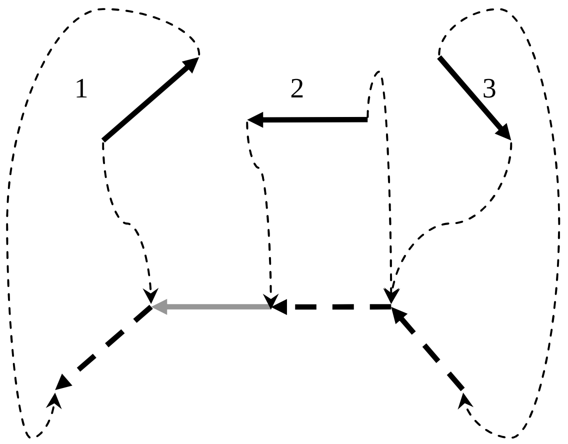


Figure 4.14: Adjusting the line segments.

After the order of the line segments has been adjusted, the drawn line strip looks like Figure 4.10.

Unfortunately, this is still not satisfactory, which can be seen by looking at the vertices making up the coastline in Figure 4.15. There are too many vertices for a simple coastline shape.

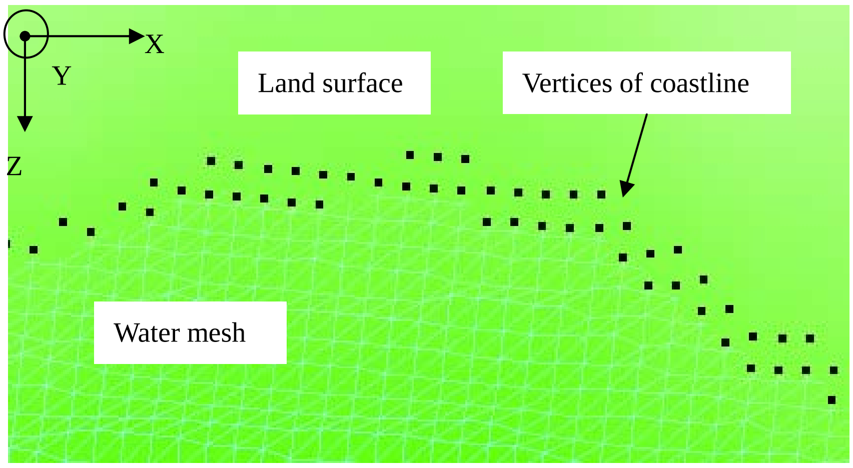


Figure 4.15: Coastline made up of vertices.

Deleting Useless Vertices

Useless vertices are identified by counting how many vertices are used in each mesh box. Two vertices are enough to draw a line through a box, so excess vertices are deleted. For example, Figure 4.16 is a mesh box with three vertices used by the coastline, and so vertex 2 is deleted. Figure 4.17 shows the resulting coastline.

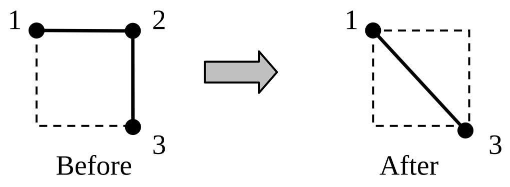


Figure 4.16: Vertex deletion in a mesh box.

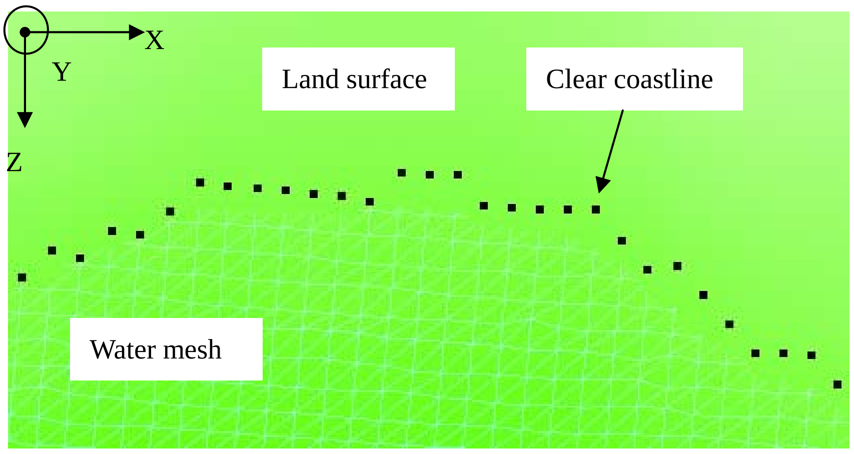


Figure 4.17: Simplified coastline.

The algorithm is a little more complicated than outlined since a coastline vertex may be shared between at most four mesh boxes (see Figure 4.18). A vertex should only be deleted if it will not reduce the number of coastline vertices below two in any of the boxes.

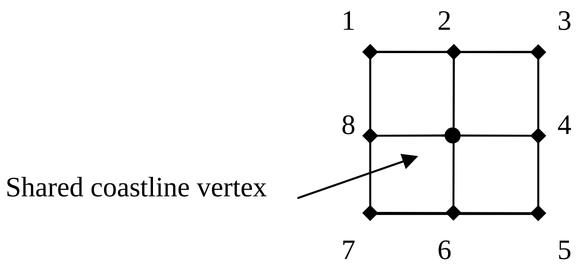


Figure 4.18: Shared coastline vertex between four mesh boxes.

Often a coastline can be simplified even farther by running the delete useless vertices algorithm twice, as in Figure 4.19.

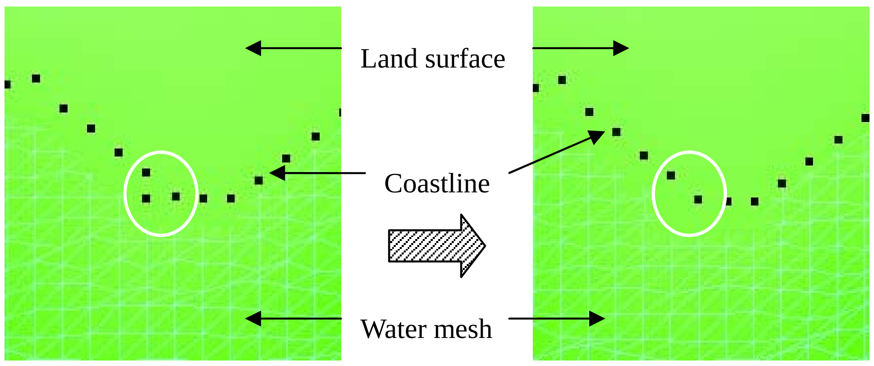


Figure 4.19: The coastline before and after running delete useless vertices again.

Figure 4.20 shows a top view of the entire simplified coastline drawn with points. Figure 4.21 shows an enlarged view of part of the coastline.

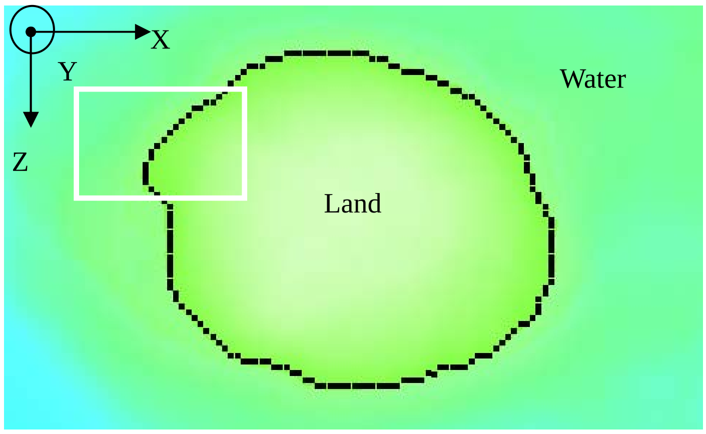


Figure 4.20: Entire coastline.

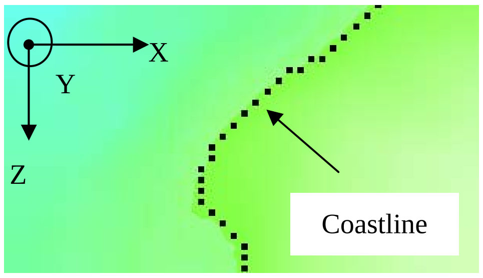


Figure 4.21: A part of the coastline.

4.2.2 Building Wave Curve 1

Wave curve 1 is the series of water vertices one mesh interval away from the coastline, and at a lower height, as shown in Figure 4.22.

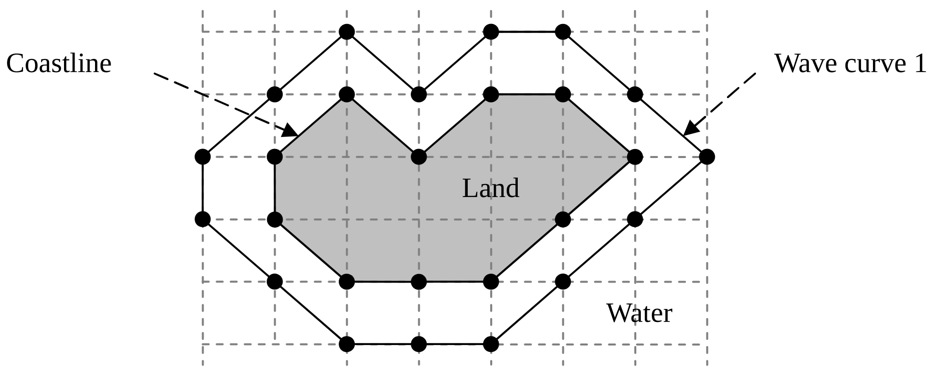


Figure 4.22: The coastline and wave curve 1.

A coastline vertex V has at most four neighbor vertices, as shown in Figure 4.23. However vertices which are higher than V can be excluded, as well as vertices already in the coastline or wave curve 1. This leaves two lower vertices in Figure 4.23

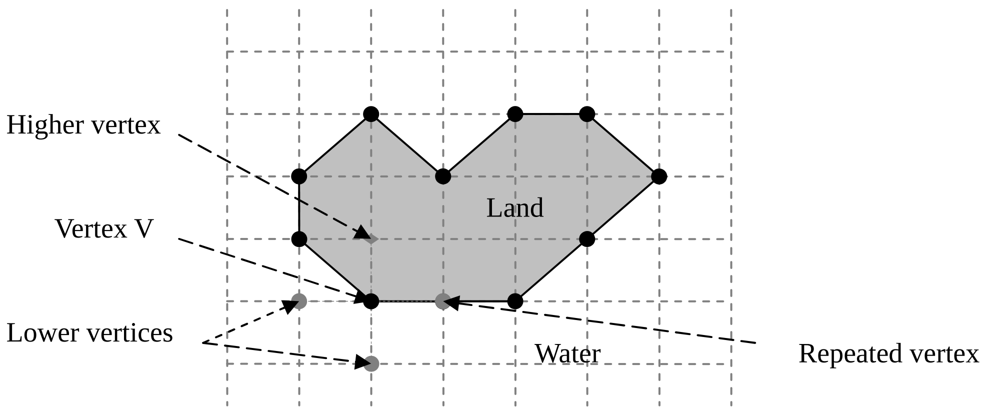


Figure 4.23: Neighbors of vertex V.

The sequence diagram in Figure 4.24 gives the main steps in calculating wave curve 1.

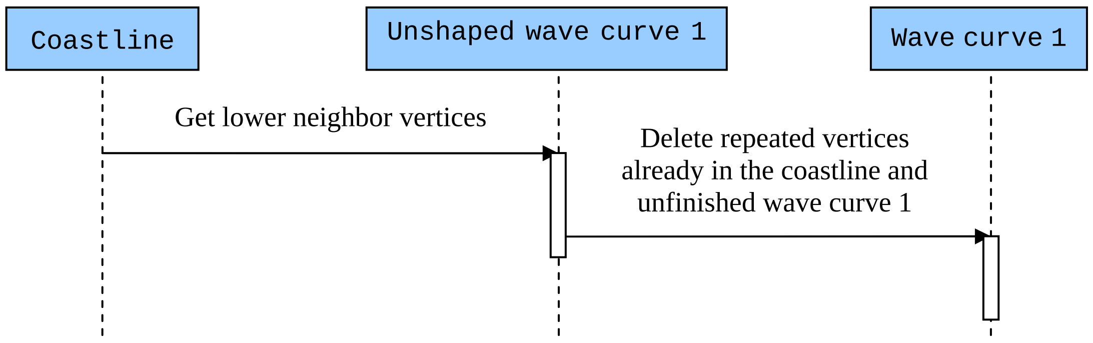

Coastline
Unshaped wave curve 1
Wave curve 1
Get lower neighbor vertices
Delete repeated vertices already in the coastline and unfinished wave curve 1

Figure 4.24: Calculating wave curve 1.

Figure 4.25 shows the coastline drawn as a black curve and wave curve 1 drawn as gray vertices. Figure 4.26 shows an enlarged part of Figure 4.25.

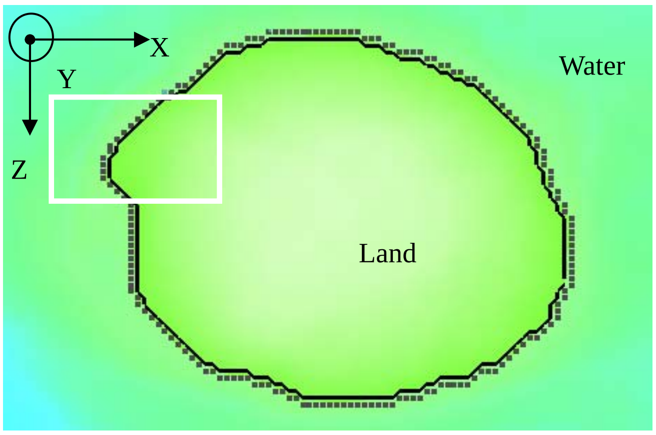


Figure 4.25: The coastline and wave curve 1.

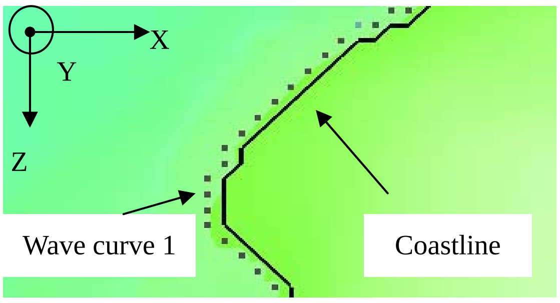


Figure 4.26: A part of the coastline and wave curve 1.

4.2.3 Building the Other Wave Curves

The other wave curves (2, 3 and 4) are calculated in a similar way to wave curve 1. For example, wave curve 2 is shown in Figure 4.27.

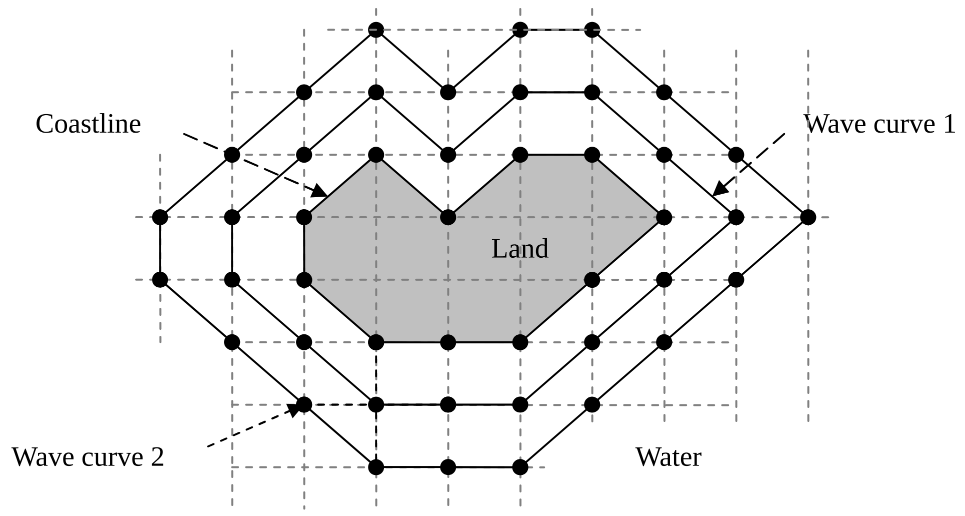


Figure 4.27: A diagram of the final wave curve 2 should look like.

Vertex W in Figure 4.28 has four neighbors, but vertices already in the coastline, wave curve 1, or wave curve 2 can be excluded. That leaves just two points to add to wave curve 2.

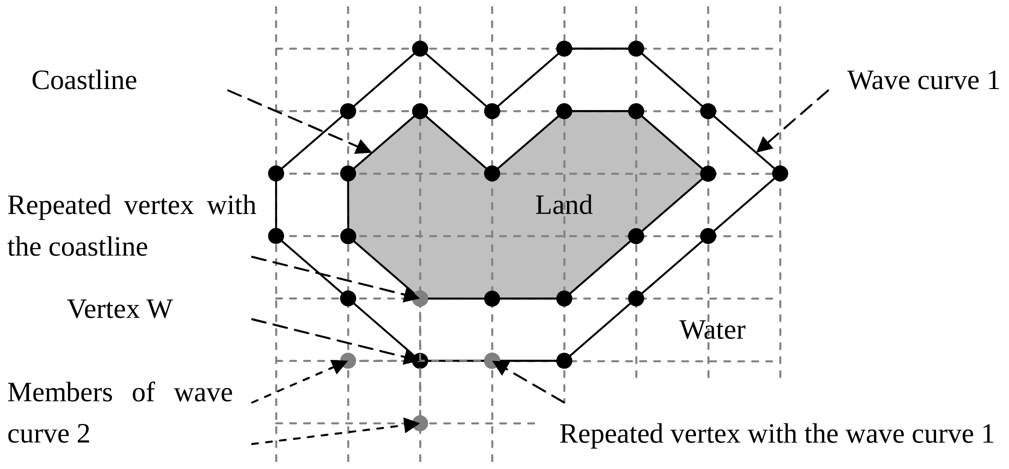

Coastline
Wave curve 1
Land
Repeated vertex with the coastline
Vertex W
Water
Members of wave curve 2
Repeated vertex with the wave curve 1

Figure 4.28: Neighbors of vertex W.

The sequence diagram in Figure 4.29 shows the main steps in calculating wave curve 2.

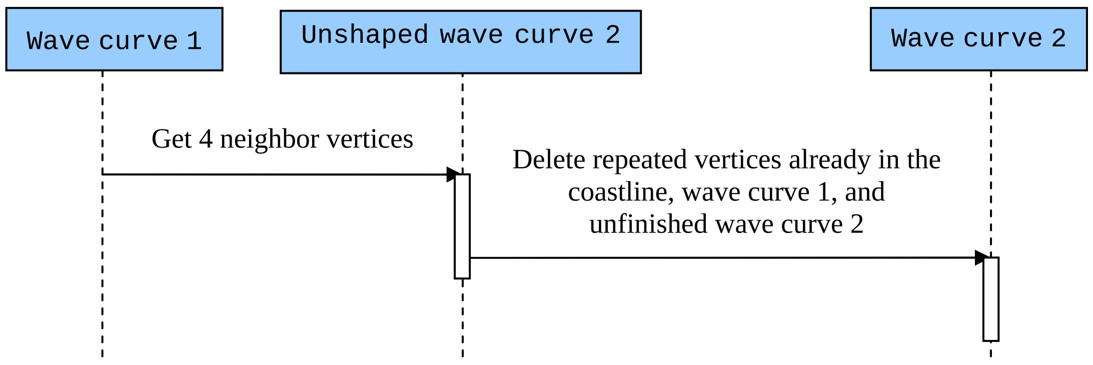

Wave curve 1
Unshaped wave curve 2
Wave curve 2
Get 4 neighbor vertices
Delete repeated vertices already in the coastline, wave curve 1, and unfinished wave curve 2

Figure 4.29: Calculating wave curve 2.

Figure 4.30 shows wave curve 2 drawn with gray vertices, wave curve 1 as white vertices, and the coastline as a black curve. Figure 4.31 shows an enlarged view of the model.

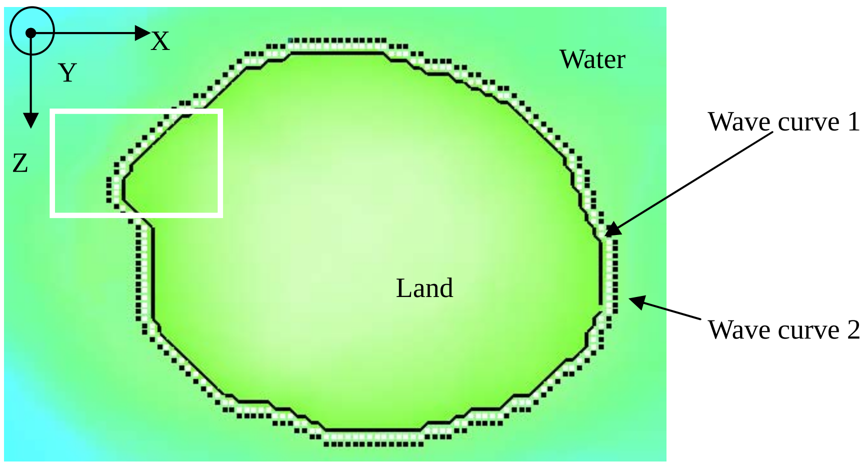


Figure 4.30: Coastline, wave curves 1 and 2.

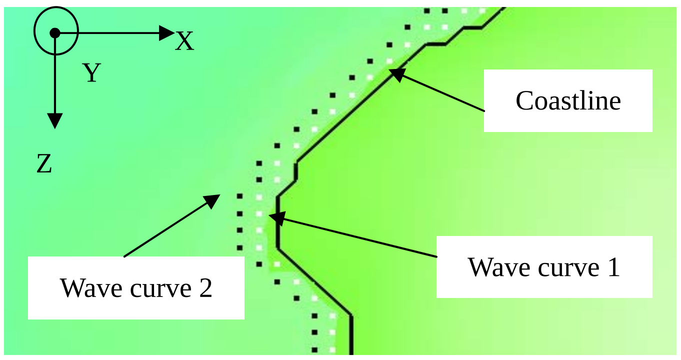


Figure 4.31: A part of the coastline, wave curves 1 and 2.

4.2.4 Wave Curves as Line Strips

When a wave curve is drawn as a line strip, some line segments appear out of order as shown in Figure 4.32.

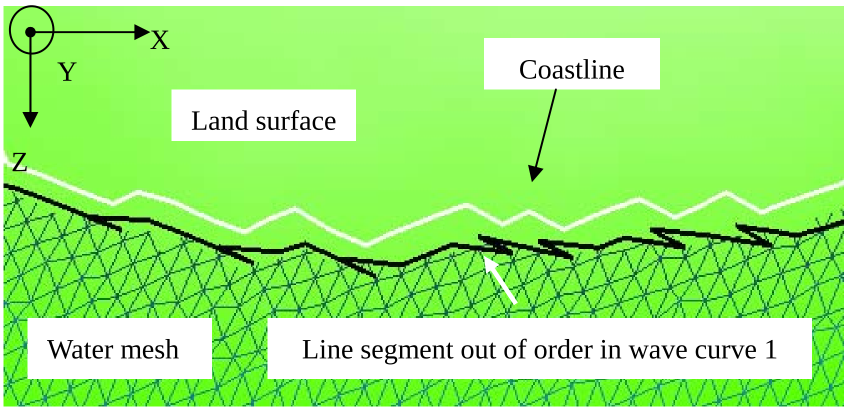


Figure 4.32: Some line segments out of order.

The reason is that the wave curve vertices are not ordered, as shown in Figure 4.33. The solution is to sort the vertices before storing them as a strip. The result is Figure 4.34.

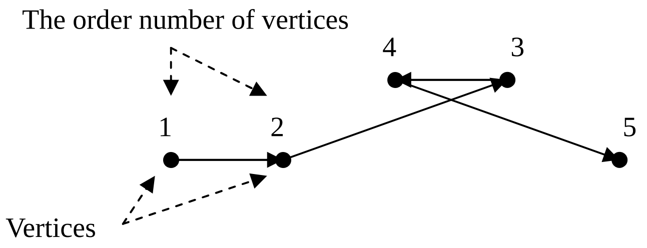


Figure 4.33: Line segment 3-4 out of order.

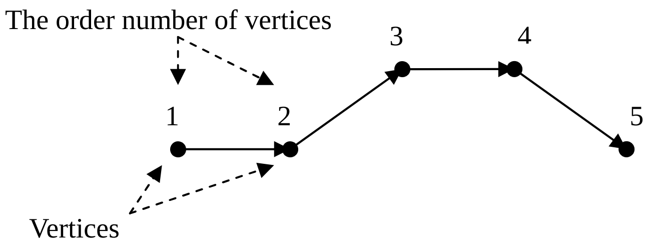


Figure 4.34: The order of line segment 3-4 has been changed.

Figure 4.35 shows a screenshot of the wave curves as line strips after they have been ordered. The coastline is drawn as a white strip.

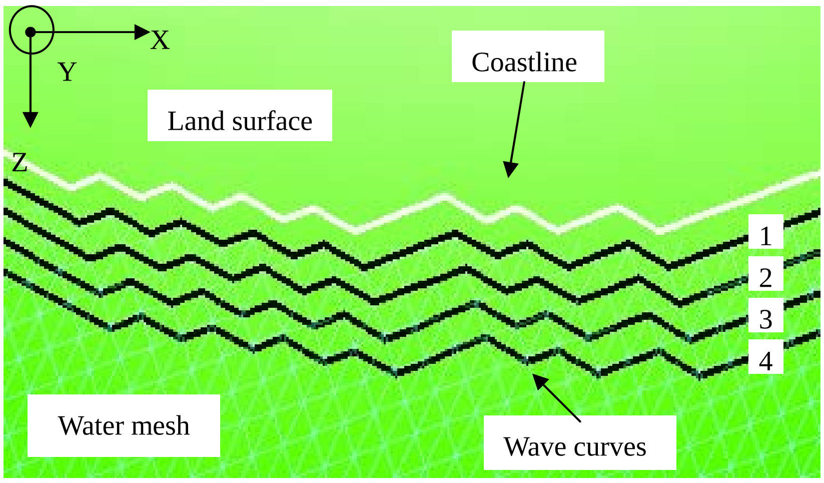


Figure 4.35: Wave curves after adjusting the vertex order.

4.3 Implementation Details

The WaveCurves class is responsible for creating the coastline and wave curves; its class diagram is shown in Figure 4.36.

WaveCurves (Appendix A8)

`+ getLine(int x1, int z1, int x2, int z2)`
`+ getCoastline(List<Points> p1)`
`+ getWaveCurve(List<Points> p1, List<Points> p2, List<Points> p3)`

Figure 4.36: WaveCurves class.

The main actions of the methods in the WaveCurves class are listed in Table 4.1. These methods will be explained with pseudocode in the rest of this section; the full code is in Appendix A8.

Method	Tasks
`getLine()`	Build a basic coastline or wave curve.
`getCoastline()`	Delete useless line segments.
	Adjust the vertex order.
	Transform line segments to vertices.
	Delete useless vertices.
`getWaveCurve()`	Build and order wave curves 1, 2, 3 and 4.

Table 4.1: Methods and their tasks in the WaveCurves class.

Build the Coastline

All the land mesh triangles are examined, as explained in Section 4.2.1. If a mesh edge cuts the water level+1 (a height of 20 units), then it is added to a line segments list.

```
if((start point of the line segment lower than 20
    && end point of the line segment higher than 20) ||
   (start point of the line segment higher than 20
    && end point of the line segment lower than 20))
  put the line segment in the list;
```

Delete Useless Line Segments

Each line segment is examined. If both points appear in other line segments, then this line segment is saved, otherwise it is deleted.

```
for(k less than list size)
   for(i less than list size)
      is start point of k in i?
   for(j less than list size)
      is end point of k in i?
   if(k’s start point is in i and k’s end point is in j)
      put k in a new list
```

Order Line Segments

The pseudocode determines if a line segment in list l1 has to have its points switched in order to connect to the previous line segment. It is then stored in list l2.

```
for(i less than l1’s size)
   if(the last point of list l2 is the same as a start point of l1’s
      line segment i)
      put i into the end of the list l2;
   if(the last point of list l2 is the same as a last point of l1’s
      line segment i)
      Turn around i then put it into the end of the list l2;
```

Translate Line Segments to Vertices

The start point of each line segment, and the end point of the last line segment, are saved in a points list.

```
for(i less than list l’s size)
   put the start point of i into point list p
put the end point of the last line segment of l into p;
```

Delete Useless Vertices

The pseudocode compares the first and the third points, by iterating through the points list. If they are in the same mesh box (as shown in Figure 4.18), then delete the second point.

```
n=p’s size;
for(i less than n)
   if(i+2 large than n-1)
       break;
   if(points i and i+2 in a mesh box)
      delete point i+1;
      n=n-1;
```

Build Wave Curve 1

Figure 4.37 shows the main elements in building wave curve 1: deleting repeated vertices in the coastline, and deleting the same vertices in unfinished wave curve 1.

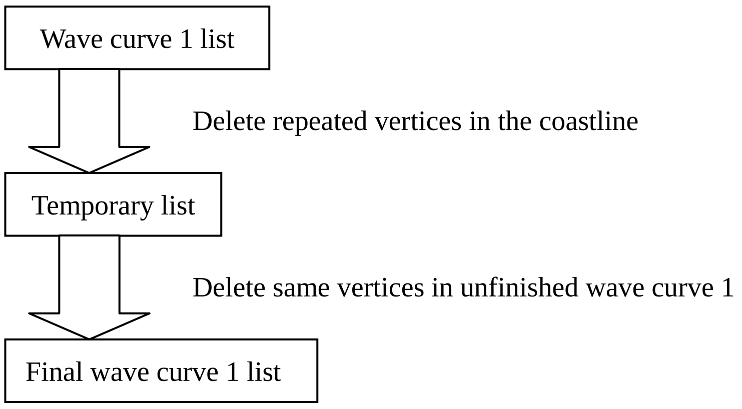

Wave curve 1 list
Delete repeated vertices in the coastline
Temporary list
Delete same vertices in unfinished wave curve 1
Final wave curve 1 list

Figure 4.37: Calculating wave curve 1.

The pseudocode checks for repeated vertices, and keeps the last:

```
for(is less than point list p1’s size)
   RepeatedPoints=false;
   for(j from i to p1’s size-1)
      if(points i and j are the same)
            RepeatedPoints=true;
            break;
      if(RepeatedPoints==false)
         put point i in a new point list;
```

The pseudocode removes vertices from wave curve 1 if they are also in the coastline:

```
for(m less than point list points’ size)
   SamePoints=false;
   for(n less than point list p2’s size)
      if(points m and n are the same)
         SamePoints=true;
         break;
   if(SamePoints==false)
      put point m to a new point list;
```

Build the Other Wave Curves

Wave curves 2, 3, and 4 are calculated in much the same way as wave curve 1. Figure 4.38 shows the main steps: delete the first repeated vertices, delete the second repeated vertices, and delete the same vertices in unfinished wave curve. The code is very similar to Section 4.3.2.

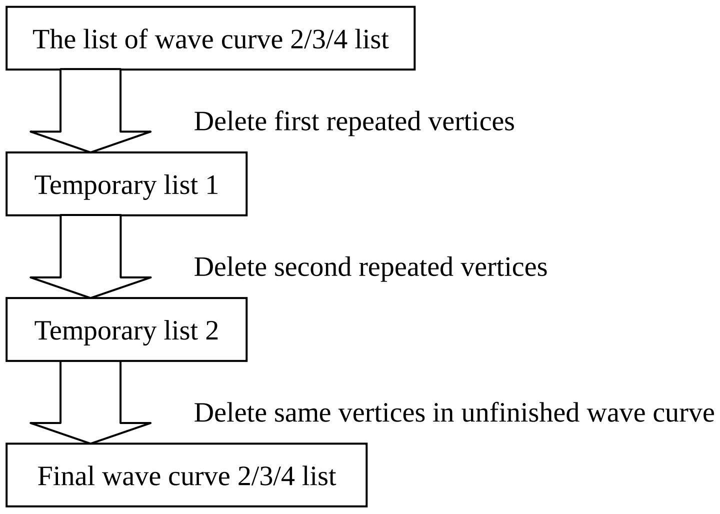

The list of wave curve 2/3/4 list
Delete first repeated vertices
Temporary list 1
Delete second repeated vertices
Temporary list 2
Delete same vertices in unfinished wave curve
Final wave curve 2/3/4 list

Figure 4.38: Calculating wave curves 2/3/4.

Ordering the Wave Curves

The following pseudocode orders a wave curve. Each connectable point is copied to its correct position in a new list, and removed from the original list.

```
for(m less than point list points3’s size)
   for(n from m+1 to points3’s size)
      if(points at m and at n are connectable)
         put point n to the final wave curve list;
         copy the point at n next m;
         remove the point be copied from old place;
```

4.4 Summary

This chapter looked at how to build a moving water surface with wave refraction. Also, the coastline and four wave curves data structures were constructed, and will be used in the next chapter to implement land/water interaction.

CHAPTER 5

ADDING INTERACTION

This chapter introduces spring systems and collision detection for modeling land/water interaction with moveable wave curves. The model is limited to four wave curves, as a balance between interaction realism and computational efficiency. Figure 5.1 shows the position of the coastline and wave curves at their rest position (at time 0).

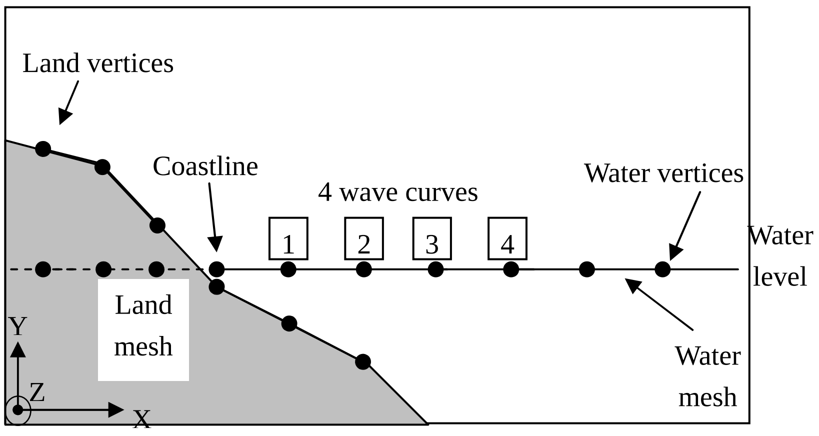


Figure 5.1: The rest position of the coastline and wave curves.

Wave curves 3 and 4 move in the X-Z plane, towards and away from the coastline to simulate tidal activity. Each vertex in the curves has a movement direction pointing from its original position toward the nearest coastline vertex. The vertices of wave curve 3 can move up to the coastline, while the vertices of wave curve 4 can move up to wave curve 1 (see Figure 5.2). Wave curve 4 can not easily pass through wave curve 3.

Wave curves 1 and 2 are fixed in the X-Z plane because of their proximity to the land, which would preclude them from moving in and out by very much in any case. Instead, their main purpose is to rise and fall in the vertical plane to simulate wave collision with the land.

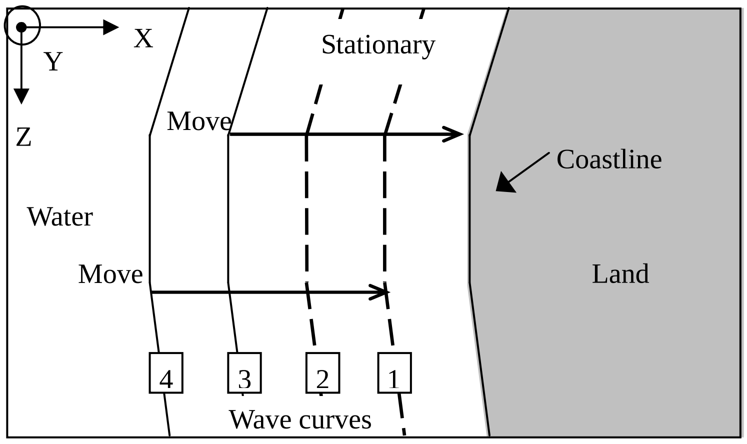


Figure 5.2: The movement of wave curves.

The tidal effects of wave curves 3 and 4, and the impact simulation of wave curves 1 and 2 are intended to produce realistic coastal wave behavior, and be easily implemented using a novel mix of spring systems and collision detection, as detailed in the following sections.

The interaction between the water and land uses *position springs* and *wave springs* to modify the X- and Z- velocities of the vertices in wave curves 3 and 4. Collision detection is used to check the collisions between wave curve 3 and the coastline, and between wave curves 3 and 4. Other novel features explained in this chapter include how the Y- velocities of wave curves 1-4 are modified to make them more realistic, and how *height springs* (a third spring system) control the movement of wave curves 1 and 2 up and down.

5.1 Position Springs

Position springs are used to constrain the horizontal movement of wave curves 3 and 4 towards and away from the coastline. Every vertex in wave curve 3 and 4 has its own position spring, which ensures that it is pulled back to its rest position after moving towards the land.

Figure 5.3 shows a vertex *s*. At time 0, it is at its rest position, labeled as $N_{s,0}$. At time t, it has moved to be at position $N_{s,t}$. The position spring P extends from the $N_{s,0}$ rest position and will pull N_s back from its $N_{s,t}$ position.

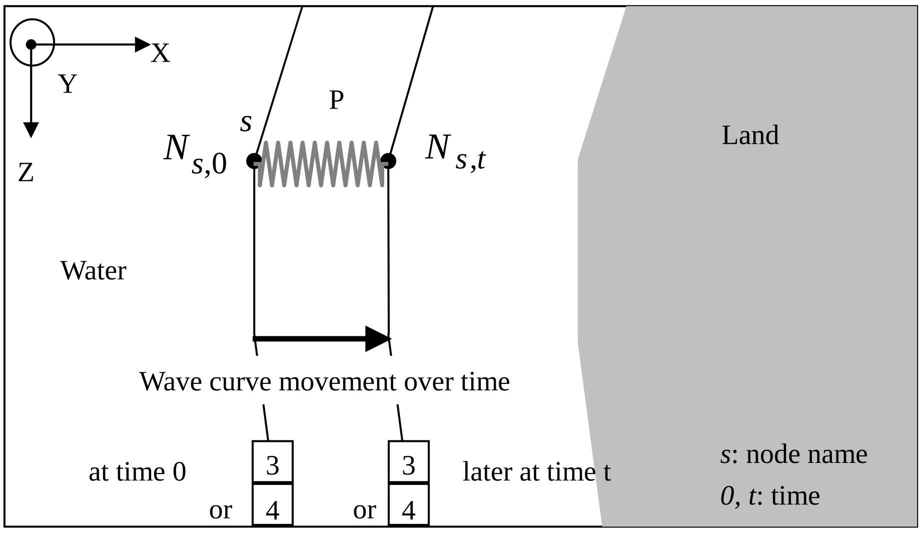


Figure 5.3: A position spring P for vertex *s*.

Figure 5.4 shows a position spring connected to a vertex *s* moving from time *t1* to *t2*.

$N_{s,0}$ is the rest position of vertex *s*.

At time *t1*, *s* is at the position $N_{s,t1}$, P is at P_{t1}, its length is $l_{P,t1}$.

At time *t2*, *s* is at the position $N_{s,t2}$, P is at P_{t2}, its length is $l_{P,t2}$.

l_P is the change in length of P when elongated or compressed over time (see the detail in Figure 5.5), so equation $l_P = l_{P,t2} - l_{P,t1}$ can be derived.

By applying Hooke's law and Newton's law [11], the velocity of the spring P is:

$V_P = k_p * l_P * (t2 - t1)$

where k_p is the elasticity coefficient of P.

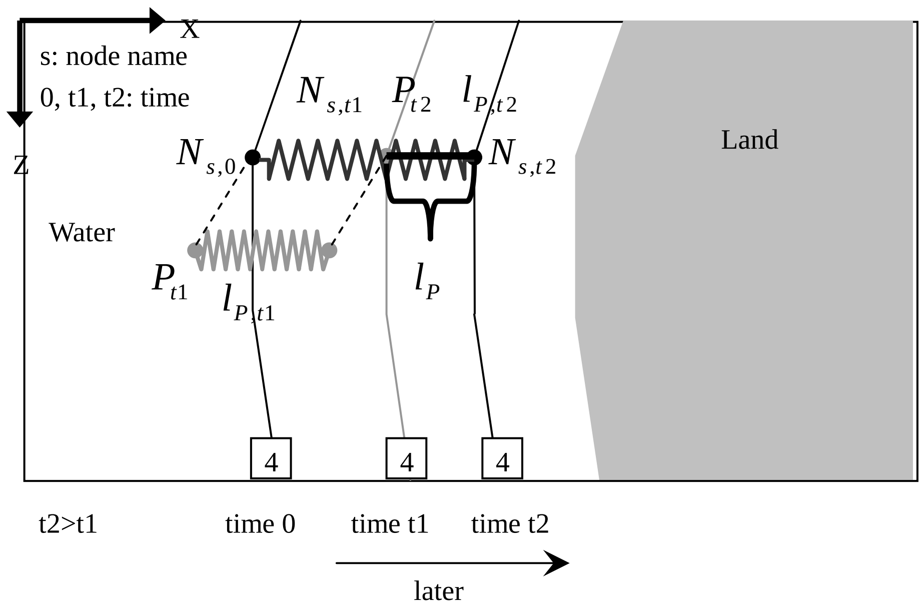


Figure 5.4: A position spring P for vertex *s* over time.

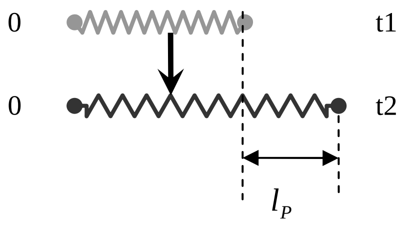


Figure 5.5: The length change of the position spring P.

5.2 Wave Springs

Wave springs are used to constrain wave curves 3 and 4 from passing through each other, a behavior rarely seen in the real world. Wave springs slow down the vertices of wave curve 4 as they approach wave curve 3.

Every neighboring pair of vertices in wave curves 3 and 4 are linked by a wave spring. For example, Figure 5.6 shows a wave spring W linking the vertices *s* and *r* of wave curves 3 and 4.

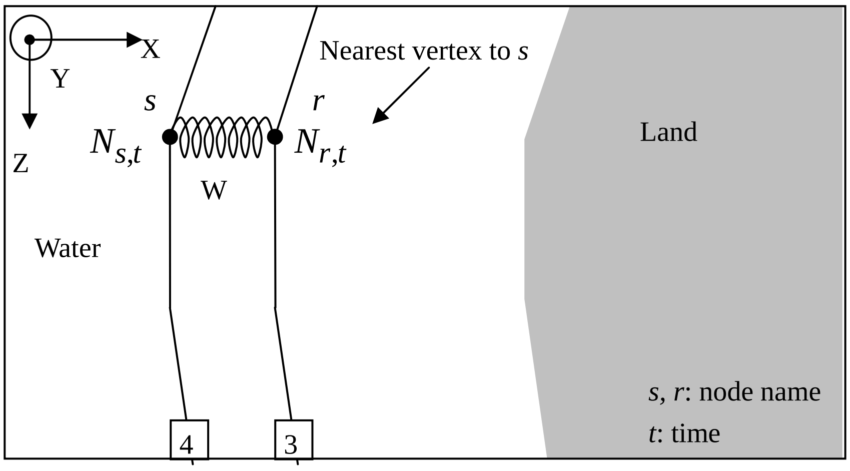


Figure 5.6: A Wave spring W between vertices *s* and *r* in wave curves 3 and 4 at time t.

Figure 5.7 shows a wave spring W connected to the vertices *s and r* moving from time *t1* to *t2*.

At time *t1*, *s* is at the position $N_{s,t1}$, r is at the position $N_{r,t1}$, W is at W_{t1}, its length is $l_{W,t1}$.

At time *t2*, *s* is at the position $N_{s,t2}$, r is at the position $N_{r,t2}$, W is at W_{t2}, its length is $l_{W,t2}$.

l_W is the change in length of W when elongated or compressed over time (see the detail in Figure 5.8), so the equation $l_W = l_{W,t2} - l_{W,t1}$ is derived.

By applying Hooke's law and Newton's law, the velocity of the spring W is:
$V_W = k_W * l_W * (t2 - t1)$
where k_W is the elasticity coefficient of W.

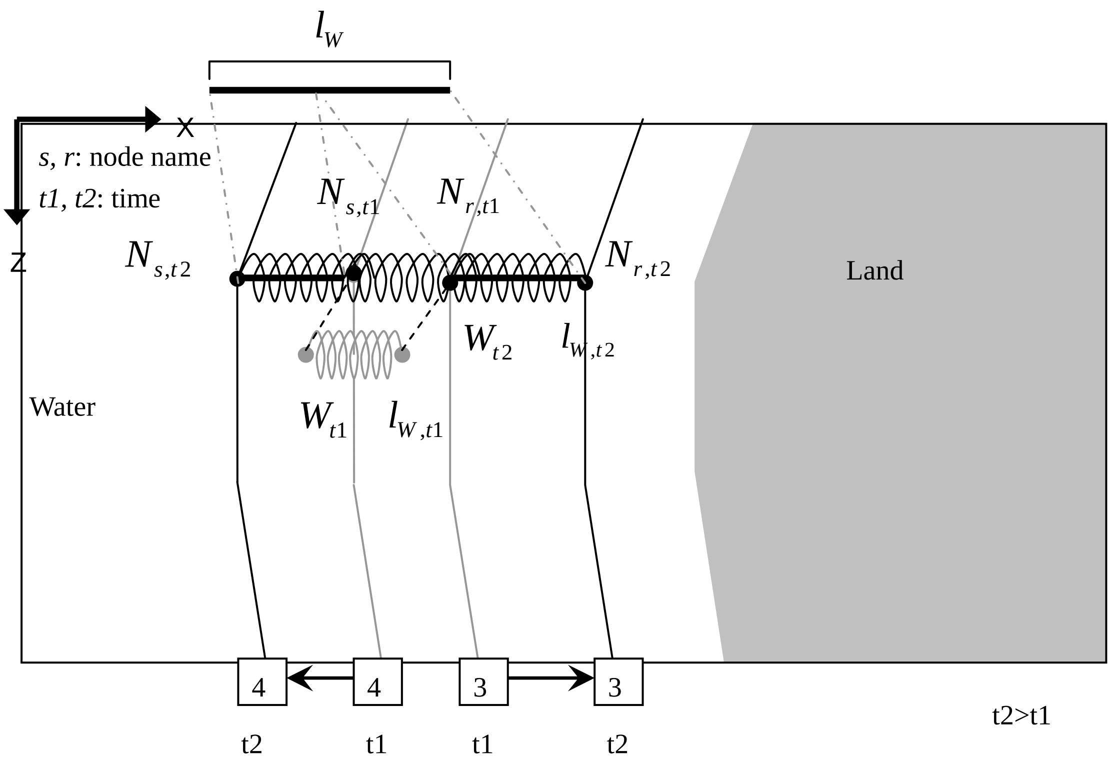


Figure 5.7: A wave spring W for vertex *s* and r over time.

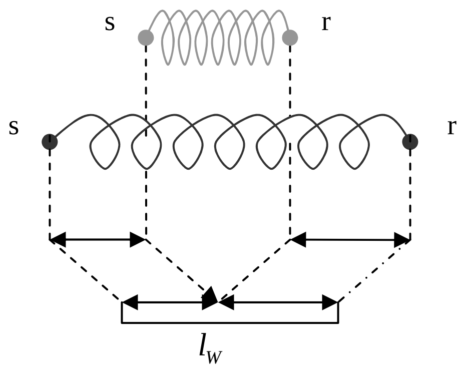


Figure 5.8: The length change of the wave spring W.

5.3 Collision Detection

Collision detection helps the spring system to control the motion of wave curves 3 and 4, in two distinct situations:

1) when the water collides with the land, as represented by the collision of wave curve 3 with the coastline;

2) when waves collide, as represented by wave curves 3 and 4 hitting each other.

Water/Land Collision

The problem of detecting when water hits the land is greatly simplified by restricting the problem to the intersection of wave curve 3 with the coastline.

Each coastline vertex is surrounded by a bounding sphere, whose diameter is equal to the initial inter-mesh spacing. If a wave curve 3 vertex moves inside the bounding sphere of a coastline vertex, a collision is registered using Equation 2.10 in Section 2.7.1. As a result, the velocity of the offending wave curve 3 vertex is reversed, to make it head back towards its rest position.

Figure 5.9 shows a vertex N_s in wave curve 3. At time 0, it is at position $N_{s,0}$, then moves towards the coastline and 'hits' the coastline vertex C_p at time t. The velocity of N_s, $V_{s,t}$, is reversed to be a little less than $-V_{s,t+1}$ at the next time interval *t+1.* A scaling factor reduces the velocity to take account of the way a wave loses energy when rebounding.

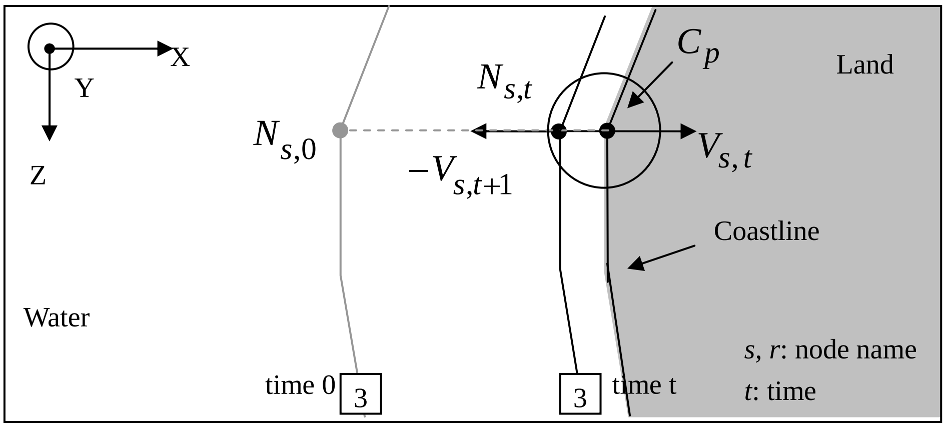


Figure 5.9: Water and land collision at time t.

Waves Collision

Wave springs inhibit wave curve 4 from passing through wave curve 3 as explained in Section 5.2, but if the velocity of wave curve 4 is much higher than wave curve 3 then crossover may still occur. This is prevented by collision detection between the vertices of wave curves 3 and 4. When a vertex in wave curve 4 hits wave curve 3, their velocities are equalized, so the two wave curve segments will move together. This is implemented by updating the velocity of the vertex in wave curve 4 and its nearest neighbor in wave curve 3.

Figure 5.10 shows the case when vertex N_s is about to hit the wave curve segment *V1-V2*. A collision is detected between N_s and the segment, and the velocities of N_s, *V1* and *V2* are modified.

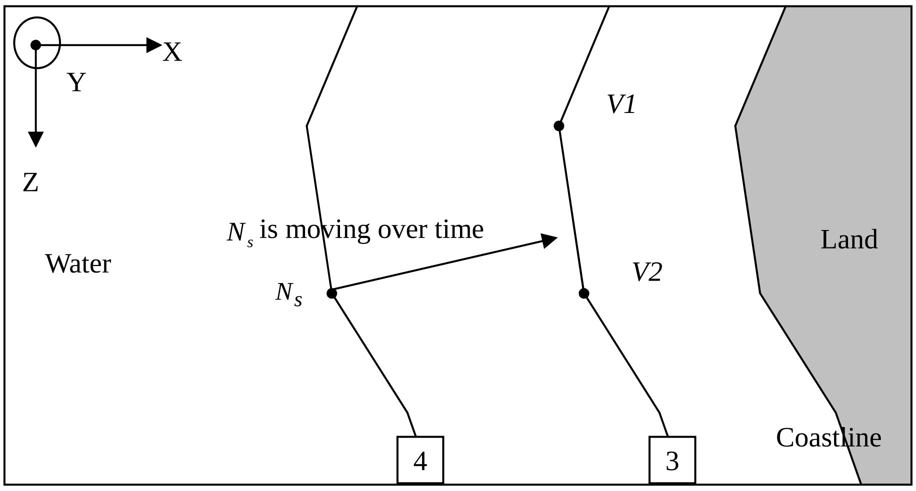


Figure 5.10: Waves Collision.

The overall behavior of N_s will be more complicated than this (and more realistic) by also being affected by a wave spring linking it to *V2* (its nearest neighbor in wave curve 3), which is not shown in Figure 5.10.

5.4 More Realistic Wave Curves 3 and 4

The position and wave springs used by wave curves 3 and 4 concentrate on modeling horizontal wave movement. However, their interaction with the land is still unrealistic, as shown in Figure 5.11. As wave curves 3 and 4 move toward the land, their heights are lower than land, so it looks like the water is moving into the land.

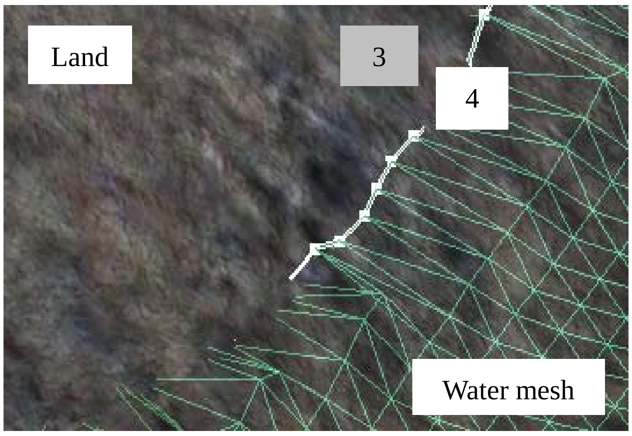


Figure 5.11: Wave curves 3 and 4 movement under the land.

In Figure 5.11, wave curve 3 is completely under the land, and half of wave curve 4 is hidden.

Figure 5.12 illustrates the problem in a more abstract fashion – wave curves 3 and 4 can move horizontally through the land mesh; instead, they should flow over it.

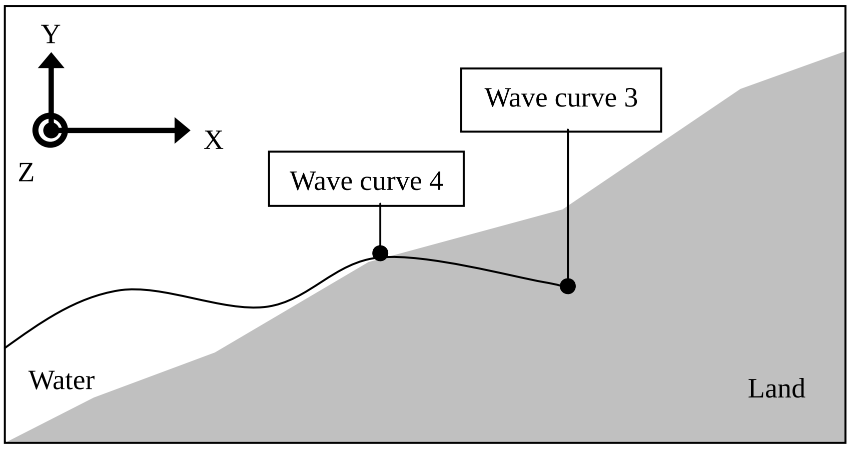


Figure 5.12: Wave curves 3 and 4 movement into the land.

This problem is avoided by raising wave curves 3 and 4 as they move closer to the land, making them appear to flow over the land.

The first solution to this problem is shown in Figure 5.13, which has wave curves 3 and 4 climb the land along the gradient of the land, where s is a water vertex, N_c is the land vertex c closest to s which has the same (x, z) coordinate as the coastline.

At time 0, s is at the rest position $N_{s,0}$, the horizontal distance between s and c is D_0, the vertical distance between s and c is H_0.

At time t, s is at the position $N_{s,t}$, the horizontal distance between s and c is D_t, and the vertical distance between s and c is H_t. According to the theory of similar triangles, $H_t = D_t * H_0 / D_0$.

Since the height of vertex c is always Y_c due to the coastline not moving, then the height of s at time t is: $Y_{s,t} = Y_c - H_t$.

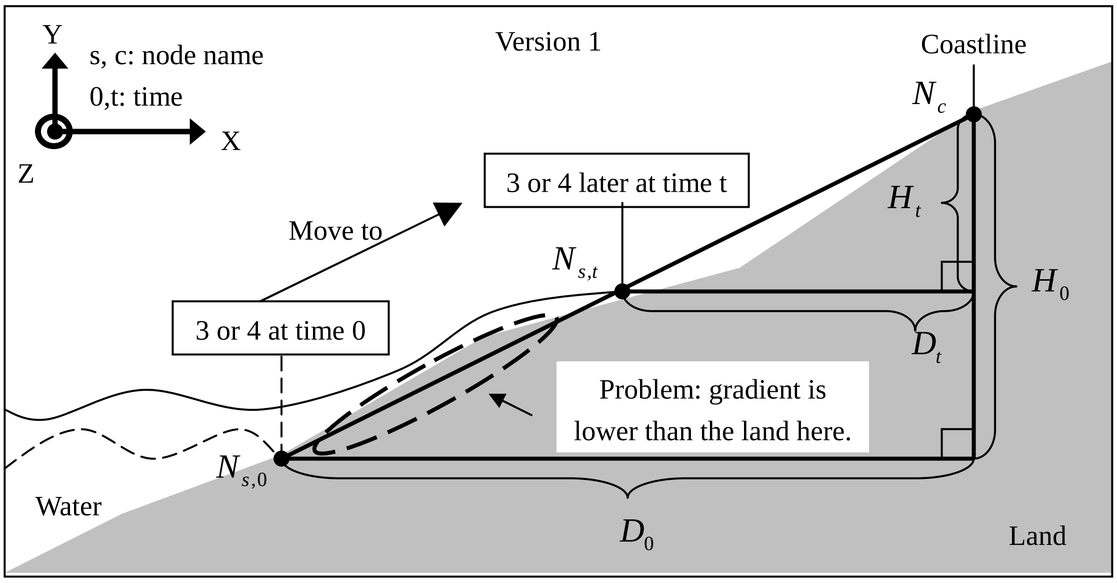


Figure 5.13: Wave curves 3 and 4 along the gradient of the land.

The problem with this form of gradient following is illustrated in Figure 5.13 – sometimes the gradient is lower than the land, so the wave curve will still disappear below the land.

This drawback is fixed in the second version, where height offsets are added to wave curve 3 and 4 to keep them a little higher than the land (see Figure 5.14). The offset of wave curve 3 is a little smaller than wave curve 4 to enhance the effect of collision detection.

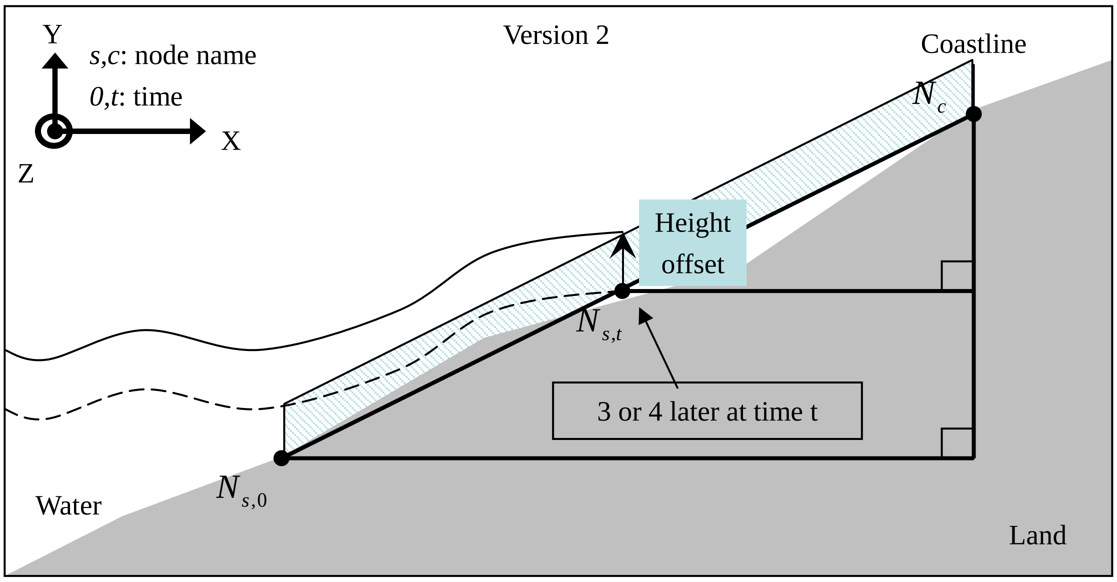


Figure 5.14: Add height offsets to wave curves 3 and 4.

5.5 Height Springs

Height springs are used to control the vertical movement of wave curves 1 and 2. These curves do not move horizontally because of their proximity to the land.

Figure 5.15 shows a height spring connected between a vertex *s* in wave curve 1 and its rest position. The same approach is also used with vertices in wave curve 2.

At time 0, *s* is at the rest position $N_{s,0}$, it will get a velocity $V_{s,0}$.

At time t, *s* arrives at the position $N_{s,t}$.

From time 0 to time t, the velocity V_s will be damped in each frame by the height spring. The velocity is reset after 1 period (80 frames) because wave curves 1 and 2 should lose power as they move. The same method as in Section 5.1 (derived from Hooke's law and Newton's law) is used to calculate the spring's velocity, which is also reduced a fraction by being multiplied by 0.97 in each frame.

The coastline does not require height springs since it does not move, remaining fixed at the default water level, which means it stays hidden beneath the land's surface.

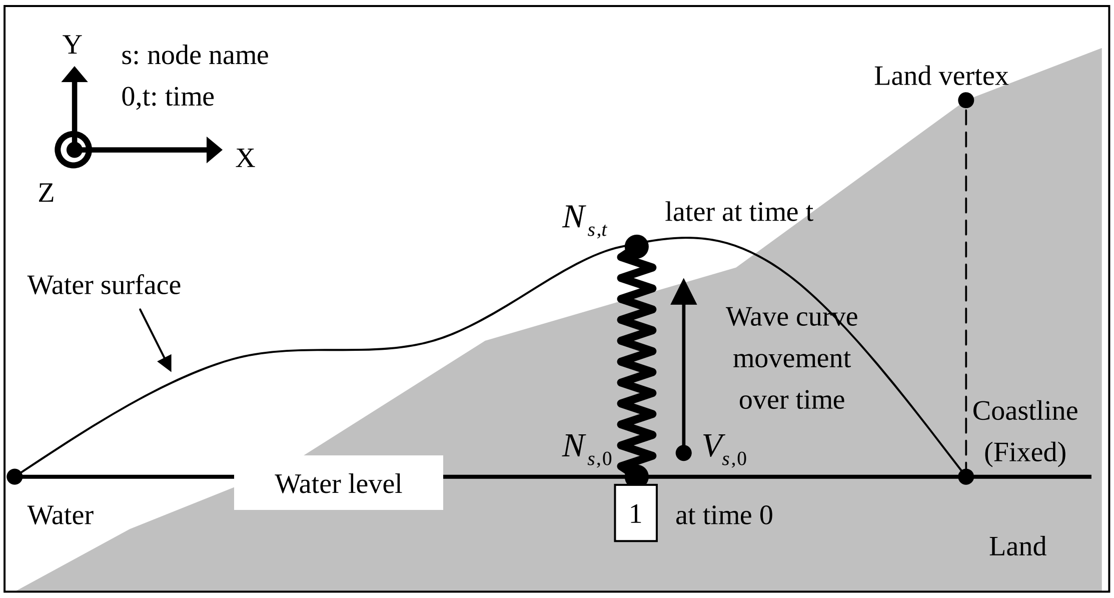


Figure 5.15: A height spring for wave curve 1 vertex *s*.

5.6 Implementation Details

Interaction is implemented across three classes whose class diagrams are shown in Figures 5.16-5.18. The Velocities class utilizes position wave springs to control the velocities of wave curves 3 and 4. Collision detection is managed by the Collision class. The HeightOfWaveCurve class controls the heights of wave curves 3 and 4 as they pass over the land, and applies height springs to wave curves 1 and 2.

Velocities (Appendix A9)

`+ getVelocityForHittingWaveCurve(...)`
`+ getVelocityForOtherWaveCurves(...)`
`+ getPlusVelocity(...)`
`+ getOriginalVelocity(...)`
`+ getNearestOutPoints(...)`
`+ getNearestInsidePoints(...)`

Figure 5.16: Class Velocities.

Collision (Appendix A10)
`+ getVelocityForWaveCurves (...)`

Figure 5.17: Class Collision.

HeightOfWaveCurve (Appendix A11)

`+ adjustHeightOfWaveCurve (...)`
`+ adjustHeightOfW12 (...)`
`+ getSlope(...)`

Figure 5.18: Class HeightOfWaveCurve.

The sequence diagram in Figure 5.19 shows how position springs, wave springs, and collisions detection control the motion of wave curves 3 and 4. The wave spring force is added to the position spring force, and used to calculate the x-z velocity of wave curves 3 and 4. This velocity will be modified if water/land or waves collisions occur. The final position of the wave curve is calculated from this final velocity.

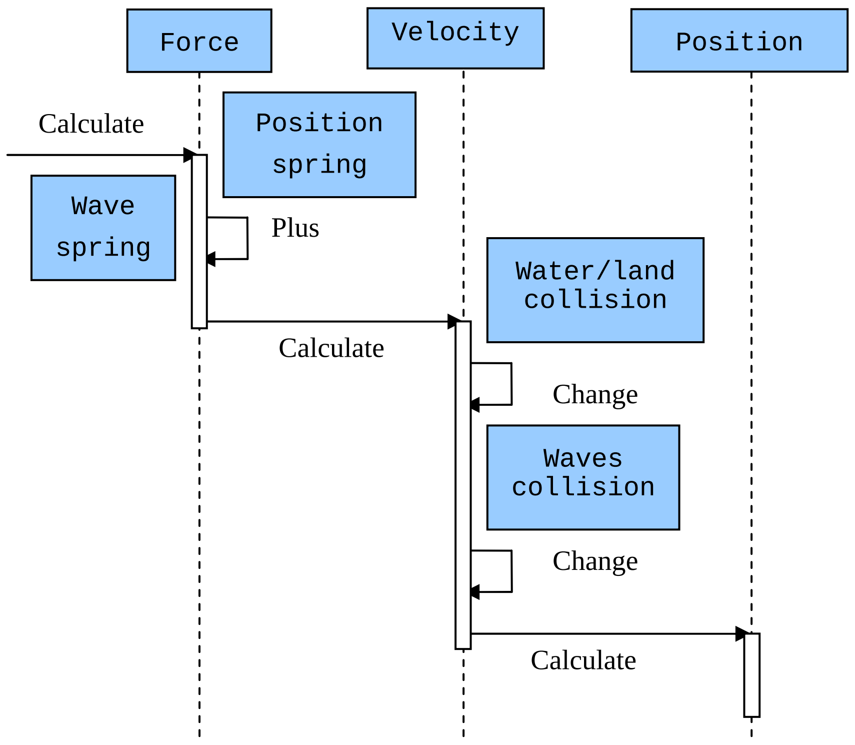


Figure 5.19: Springs and collisions control the motion of wave curves 3 and 4.

5.6.1 Position and Wave Springs

`getVelocityForHittingWaveCurve()` is used to calculate the velocity of wave curve 3, and `getVelocityForOtherWaveCurves()` is used to calculate the velocity of wave curve 4. The following pseudocode is the same part of these methods, and the full code of the class is in Appendix A9.

```
for(i less than wave curve size)
   calculate wave spring force in x axis for vertex i;
   calculate wave spring force in z axis for vertex i;

   plus position spring force in x axis;
   plus position spring force in z axis;

   x velocity = x force effect + previous x velocity;
   z velocity = z force effect + previous z velocity;
```

The wave spring force is separated into X and Z components, and added to the position spring force. Vertex velocity is calculated by adding the velocity due to the springs to the previous velocity for the vertex.

5.6.2 Collision Detection

`getVelocityForWaveCurves()` handles water/land collisions and collisions between waves. It is explained with the following pseudocode, and the full code is in Appendix A10.

```
for(i less than wave curve 3 size)
   //water/land collision
   if(i hit the land)
      calculate x velocity for i;
      calculate z velocity for i;
      multiply the x and z velocity 0.2;
   ...
   //collision between waves
   if(i move through line segment bc of wave curve 4)
      x velocity is the average x velocities of i, b, c;
      z velocity is the average z velocities of i, b, c;
```

Water/land collision is processed first using the equation

$(x-a)^2+(y-b)^2 \le 0.5^2$ from Section 2.7.1. The wave curve vertex (x, y) is compared to the nearest coastline vertex (a, b). If their distance apart is equal to or less than 0.5 (the mesh box size is 1*1), then a collision has occurred. The new vertex velocity is calculated by using the distance and direction between its current position and its rest position. A scaling factor of 0.2 reduces the velocity to represents the energy lost when rebounding.

Wave collision uses the vertex and a line segment technique, outlined in Sections 5.3.2 and 2.7.2. The vertex's old position is (x1, y1), its next position is (x2, y2), and the end points of the segment are (x3, y3) and (x4, y4). Collision occurs if the following equations can be solved for (x, y):

$$(x-x1)/(x2-x1)=(y-y1)/(y2-y1)$$
$$(x-x3)/(x4-x3)=(y-y3)/(y4-y3)$$

The new velocities of the three vertices is calculated by averaging their current velocities.

5.6.3 Height of Wave Curves

The height of wave curves 3 and 4 as they pass over the land is based on the theory in Section 5.4. In the following pseudocode, the `height` argument refers to a vertex in wave curve 3 or 4, as illustrated by Figure 5.13.

```
for(i less than the size of wave curve 3 or 4)
   height of i = yc - Slope * distance + plus;
```

yc is the height of vertex c, distance is D_t, Slope is H_0 / D_0, Slope*distance means H_t , and `plus` is the height offset in Figure 5.14. The full code is in `adjustHeightofWaveCurve()` in HeightWaveCurve in Appendix A11.

Height springs control the height of wave curves 1 and 2 by using the following pseudocode:

```
for(i less than the size of wave curve 1 or 2)
   calculate the force of height spring;
   update the next height use current velocity;
   update the next velocity use current force;
```

In the for-loop, the current spring length is used to calculate its force. The height of the wave curve is updated by adding the velocity effect, plus the spring effect modified by a dampening factor (0.97). The full code is in `adjustHeightOfW12()` of class HeightOfWaveCurve in Appendix A11.

5.7 Testing

Figure 5.20 is a cross-sectional view of the model showing water moving towards the land. This figure is redrawn in Figure 5.21, from an overhead viewpoint, emphasizing the coastline, and wave curves 3 and 4. The wave curves 3 and 4 are moving towards the coastline without crossing over each other, which is controlled by a mix of position springs, wave springs, and collision detection.

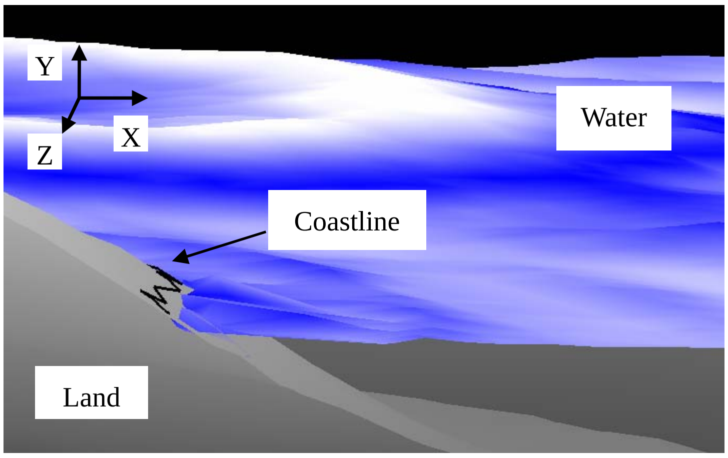


Figure 5.20: Water moving towards the land.

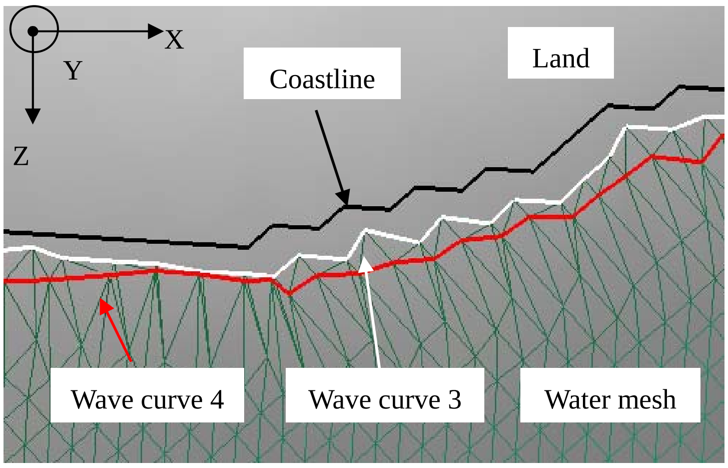


Figure 5.21: Wave curves moving towards the land.

Figure 5.22 shows the scene later after the water has rebounded from the land. This figure is redrawn in Figure 5.23 from an overhead viewpoint to emphasize the coastline and wave curves. Again the interaction is managed by a mix of springs and collision detection.

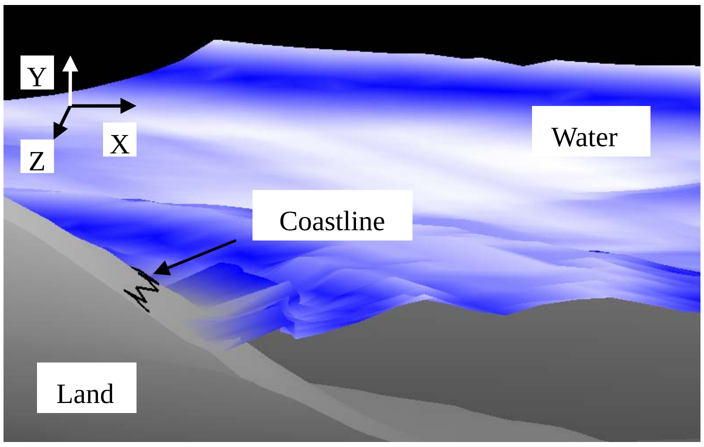


Figure 5.22: Water retreating from the land.

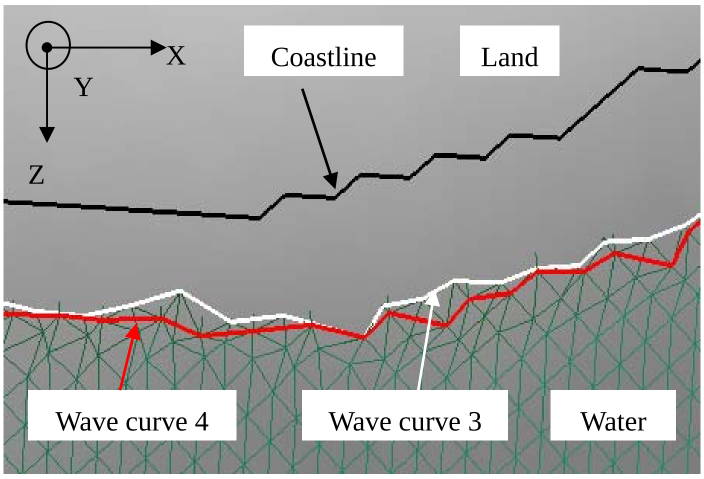


Figure 5.23: Wave curves rebounding from the land.

Figure 5.23 illustrates that crossover still occurs, as it does in real waves, but is a rare event, and is soon followed by the waves either moving in unison or pulling apart.

5.8 Summary

Position and wave springs control the horizontal motion of wave curves 3 and 4, helped by land/water and waves collision detection. Height springs control the vertical movement of wave curves 1 and 2.

CHAPTER 6

MULTITEXTURING AND SHADING

In this chapter the appearance of the land and water surface is improved by employing multitexturing. The land combines procedural and detail texturing, while the water surface uses material and detail texturing. Two other improvements to the realism of the water surface are also described: wave crest shading and water transparency.

6.1 Texturing the Land Surface

Two kinds of land texturing are used: procedural texture generation which combines textures based on simple height calculations, and detail texturing which tiles a small texture many times over the land mesh.

6.1.1 Procedural Texturing

At different heights the land surface should have different features, which is achieved by combining four texture maps (representing soil, rock, silt, and dirt). The combination is based on land height, using Franke's method from Section 2.8.2 [29]. The terrain is divided into five regions based on height values, with at most two textures assigned to each region:

Region 1 (Soil) : height 210-255

Region 2 (Rock and Soil) : height 160-210

Region 3 (Silt and Rock) : height 110-160

Region 4 (Dirt and Silt): height 60-110

Region 5 (Dirt) : height 0-60

The percentage of each texture combined into the final land texture is calculated using four texture graphs (Soil, Rock, Silt, and Dirt) shown in Figure 6.1.

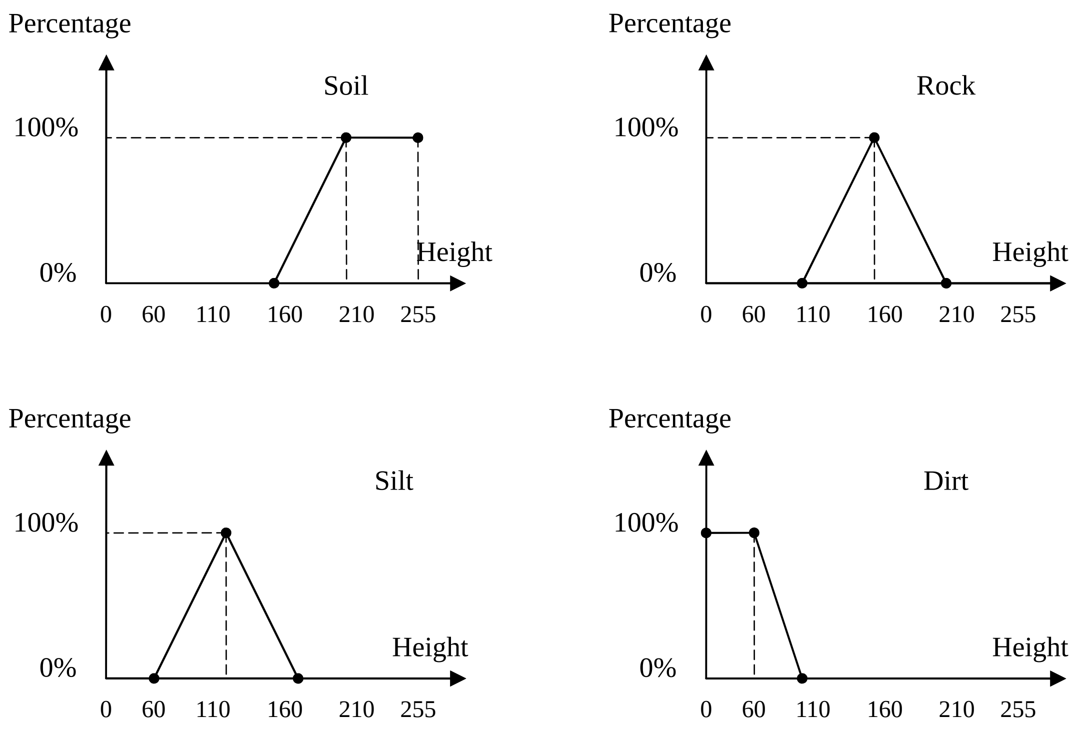


Figure 6.1: The percentages of four textures used in the model for a given height.

For example, anywhere in region 3, the percentages of the Silt and Rock textures are:

Percentage (Rock) = (height-110)/(160-110)

Percentage (Silt) = 1-Percentage (Rock)

Therefore if the region 3 height is 150, the percentages of (rock, silt) become (80%, 20%). These are used to calculate the RGB values for the land texture as:

RGB(rock)*80% + RGB(silt)*20%

The textures are shown in the Figure 6.2. The generated land texture and the resulting terrain are shown in Figure 6.3.

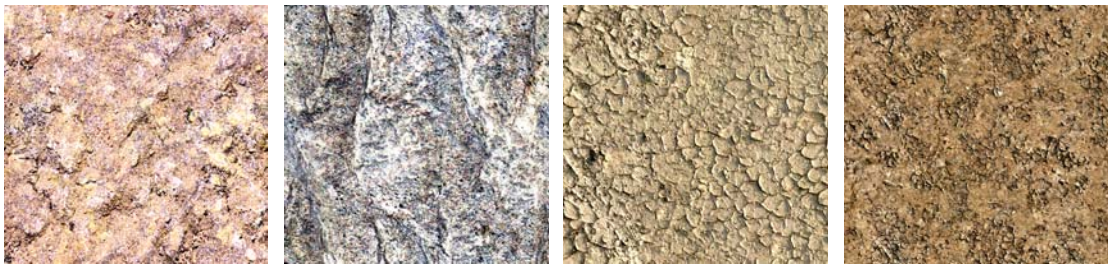


Figure 6.2: The four textures used for procedural texturing.

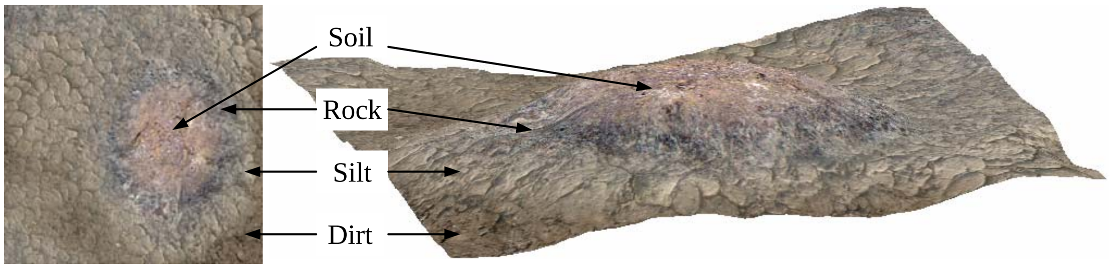


Figure 6.3: The generated texture and terrain.

6.1.2 Detail Texturing

Even after procedural texturing, the resolution of the resulting texture is still poor when viewed up close (e.g. see the left hand side of Figure 6.6). A detail texture, such as the one in Figure 6.4 can solve this problem.

Figure 6.4: Detail texture map.

The detail texture is tiled 36 times over the terrain resulting in Figure 6.5. The right hand side of Figure 6.6 shows an enlarged view of part of the model which is a clear improvement over the left hand picture.

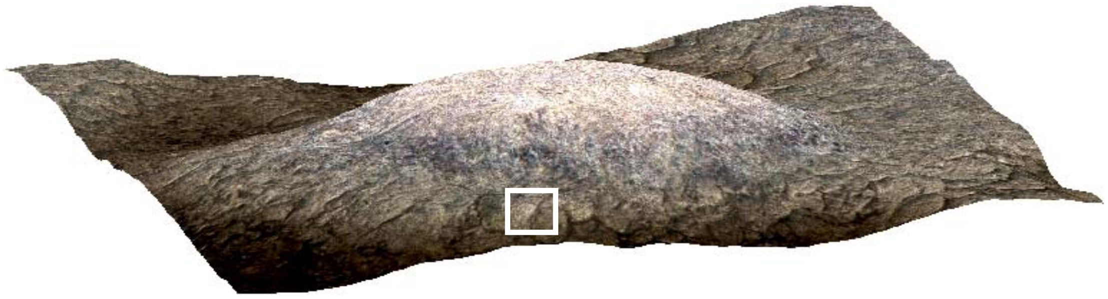
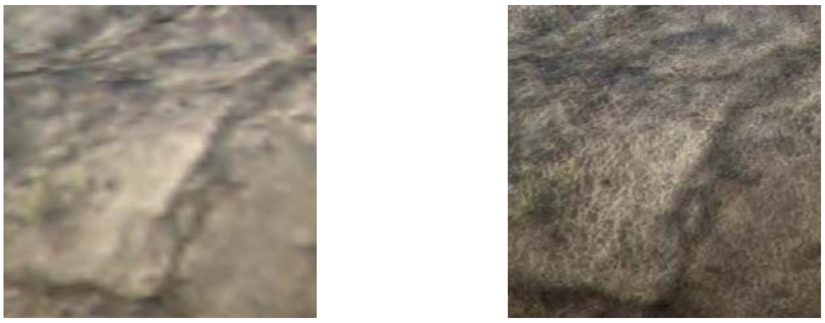


Figure 6.5: The terrain with procedural and detail texturing.


Without detail texturing. With detail texturing
Figure 6.6: Without/with Detail Texturing.

6.1.3 Implementation Details

Pseudocode for the multitexturing carried out by the Mesh class is shown below; the full code is in Appendix A6.

```
active No.0 texture;
enable 2D texture;
bind generated texture to No.0 texture;

set mode for combine multitexture;
active No.1 texture;
enable 2D texture;

choose No.0 texture;
enable texture coordinate array;
set generated texture coordinate array;

choose No.1 texture;
enable texture coordinate array;
set detail texture coordinate array;
```

Two texture units must be activated one after the other, the first for procedural texturing, the second for detail texturing.

6.2 Texturing the Water Surface

The drawback of using a single color for the water is illustrated in Figure 6.7. Multitexturing improves the image quality by combining:

1) a single mesh-wide texture representing the water material, and

2) a tiled texture representing water detail.

In addition, wave crest shading and water transparency are employed to further enhance the realism, as detailed in later sub-sections.

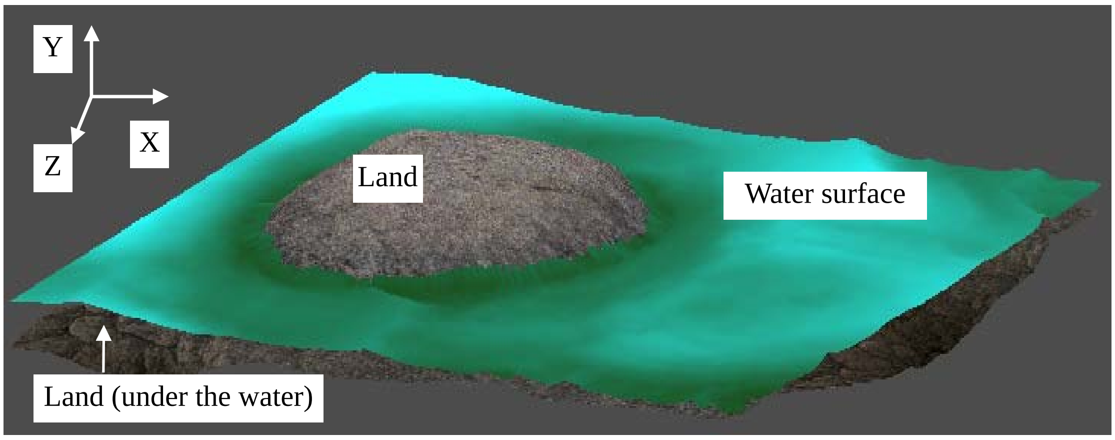


Figure 6.7: One color water surface.

6.2.1 Material Texturing

The water material texture is a texture covering all the water mesh, as shown in Figure 6.8.

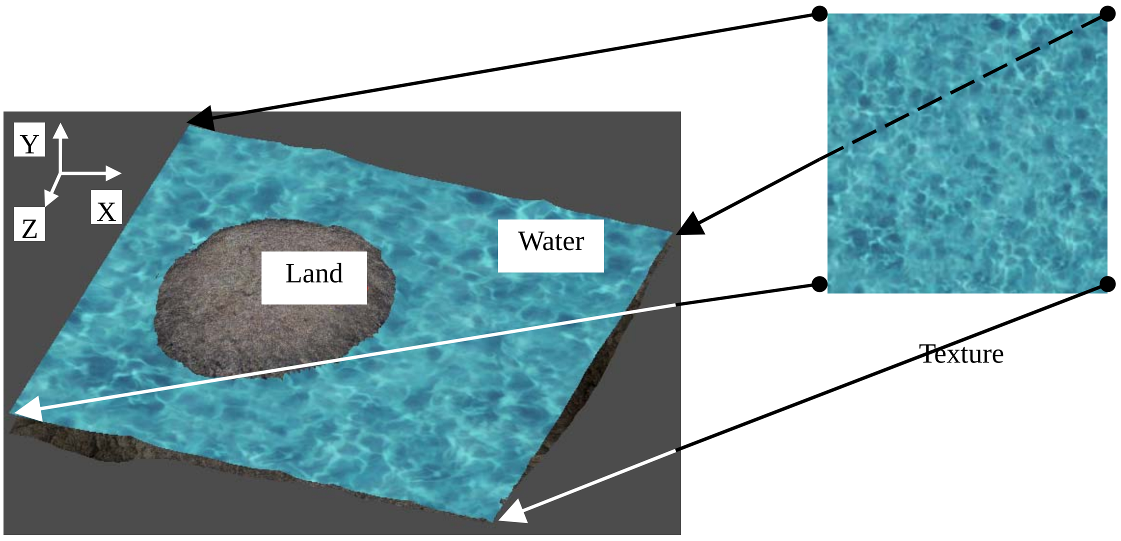


Figure 6.8: Water Material texturing.

6.2.2 Detail Texturing

As with the land, a single texture lacks resolution when viewed up close, and so a water detail texture is added. It is tiled sixteen times over the water mesh, as shown in Figure 6.9.

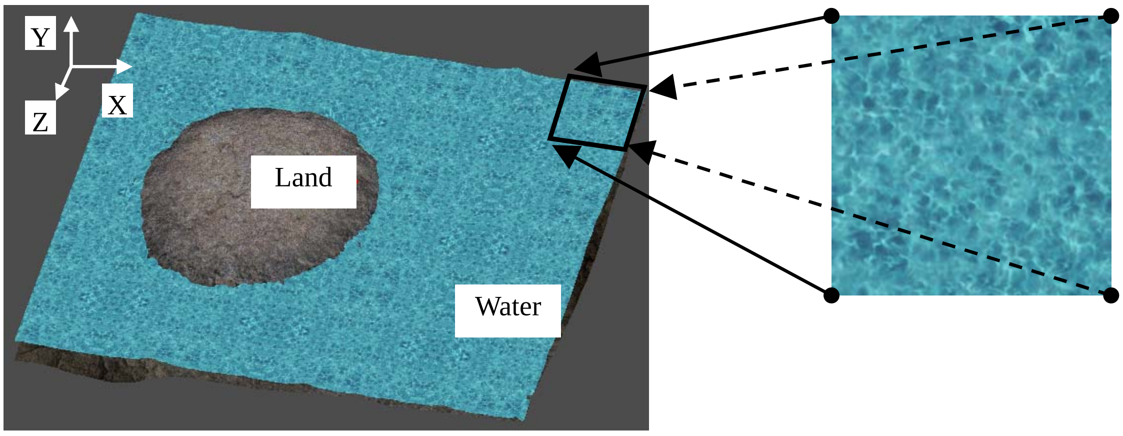


Figure 6.9: Water Detail Texturing.

The material and detail textures are combined using multitexturing in a similar way to Section 6.1.2. Figure 6.10 shows the result.

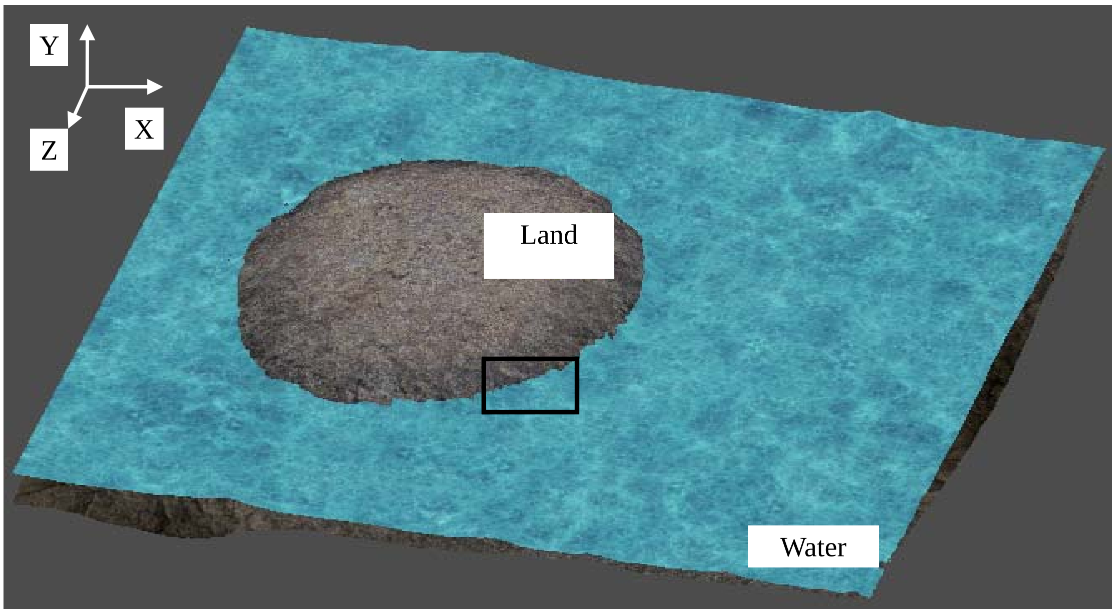


Figure 6.10: Combined material and detail texturing.

Figure 6.11 shows an enlarged part of the water surface with only material texturing, only detail texturing, and their combination with multitexturing. The latter displays various shades of blue and ripple shapes.

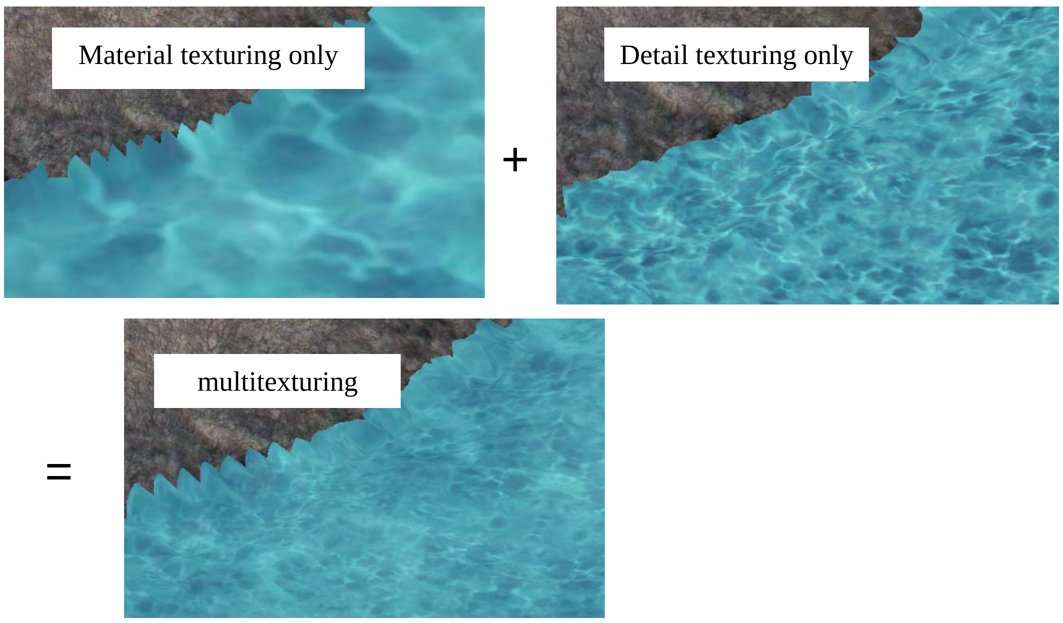


Figure 6.11: A part of the model with material texturing only, detail texturing only, and multitexturing.

6.2.3 Wave Crest Shading

A wave crest is normally darker than other parts of the water surface, as shown in Figure 6.12. Without this darkening, it is hard to see wave behavior in the model, as shown in Figure 6.13.

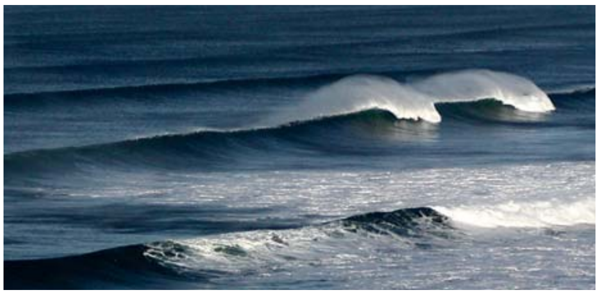


Figure 6.12: Water around a crest is darker than other place.

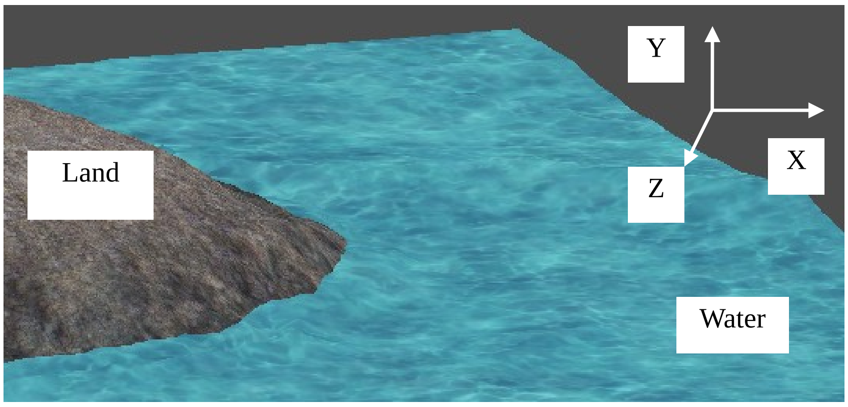


Figure 6.13: Model without wave crest shading.

This problem can be corrected by making the water darker as it rises above the default water level, as shown in Figure 6.14.

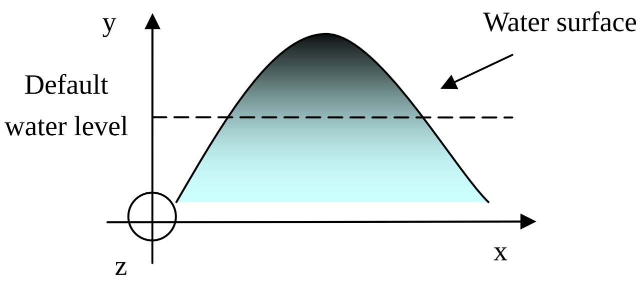


Figure 6.14: Higher water becomes darker.

The y- value is transformed into a color in the range of (0, 1) by adding a scale factor of 0.25 and multiplying by -1. Extremes of brightness and darkness are avoided by subtracting 0.1 from the RGB pixel.

Figure 6.15 shows that wave crest shading makes the waves more visible.

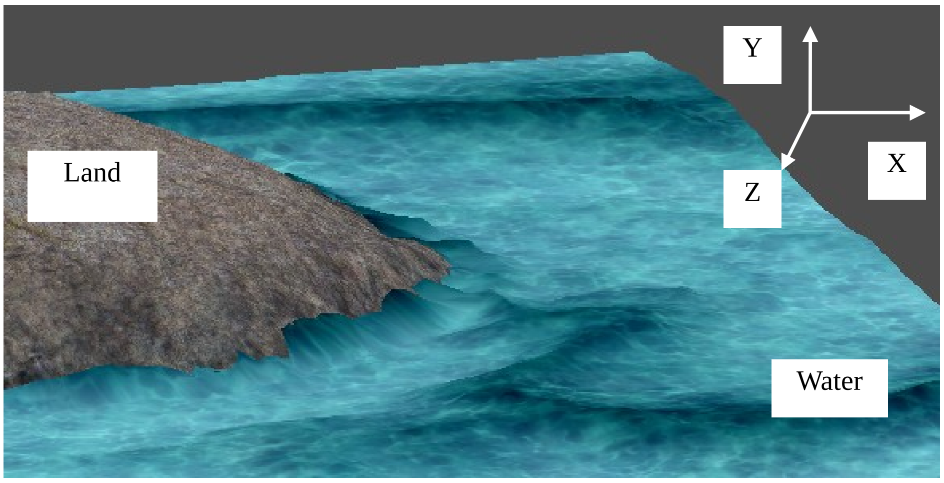


Figure 6.15: Wave crest shading.

6.2.4 Water Transparency

Of course, real water is partially transparent, but up to now the model's water surface has been opaque, as in Figure 6.16.

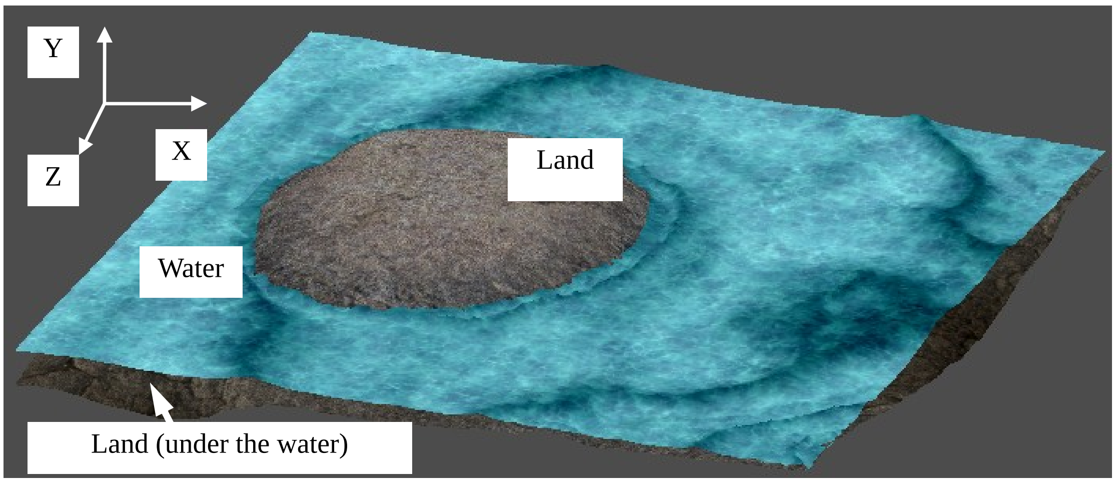


Figure 6.16: Water surface without transparency.

The added transparency ranges over (1, 0.01] for land heights between (15, 19]. In other words, the water is not transparent for heights lower than 15 but gradually becomes more transparent between heights 15 to 19, as shown in Figure 6.17.

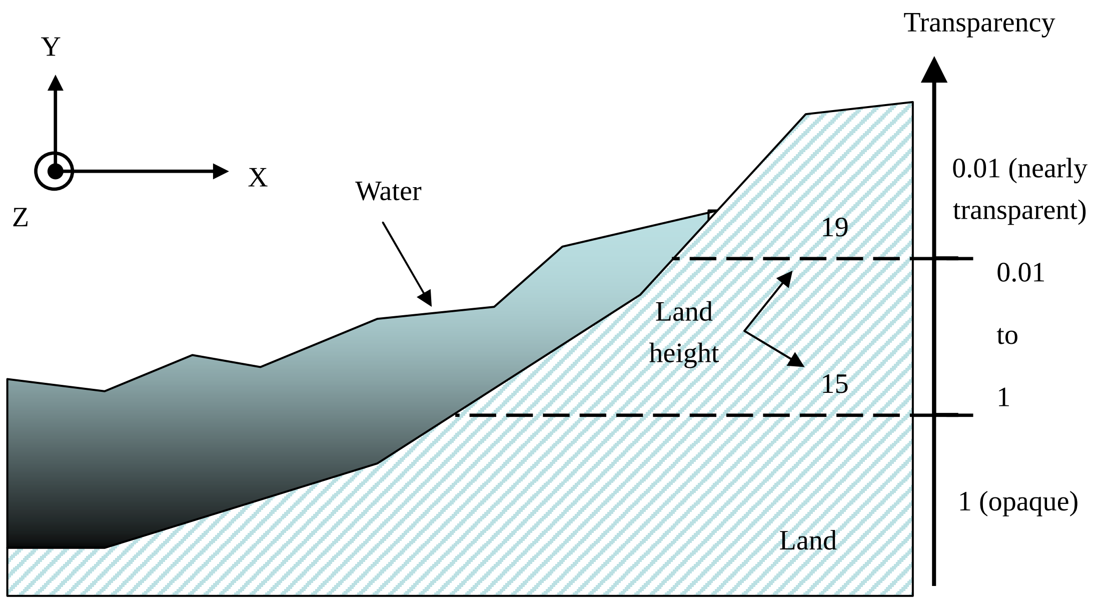


Figure 6.17: Water transparency.

After adding transparency, the water surface looks more realistic since the transparency varies with land height as shown in Figure 6.18.

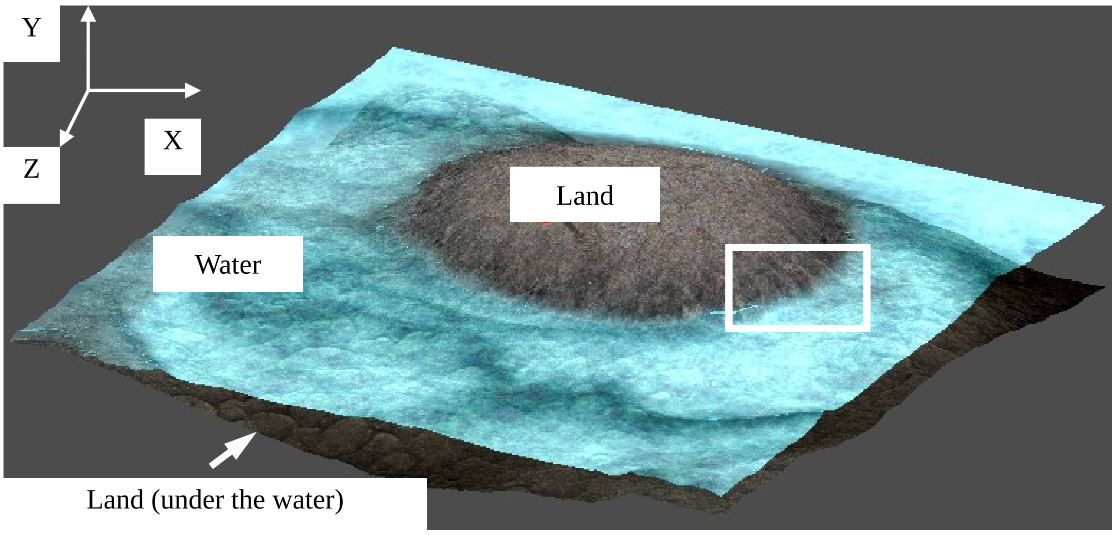


Figure 6.18: Water with transparency.

The invisible nature of transparency makes it hard to observe, as illustrated by

Figure 6.19.

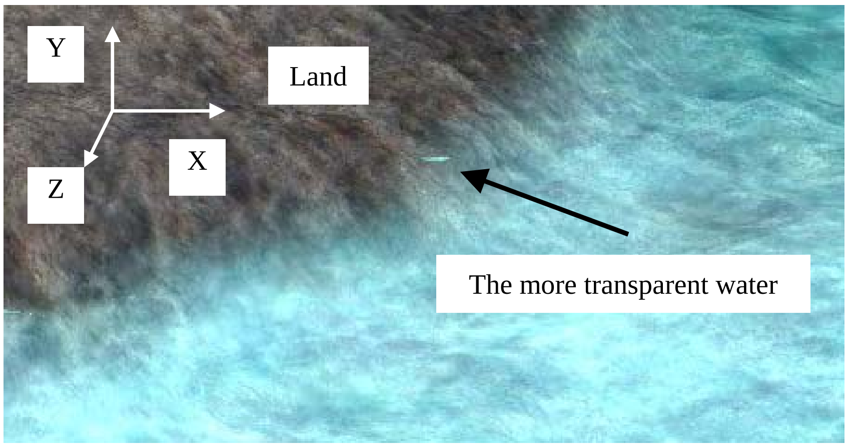


Figure 6.19: The shallower water is more transparent.

To make the varying transparency easier to check, a blue box was added to the water near the coastline, as shown in Figure 6.20.

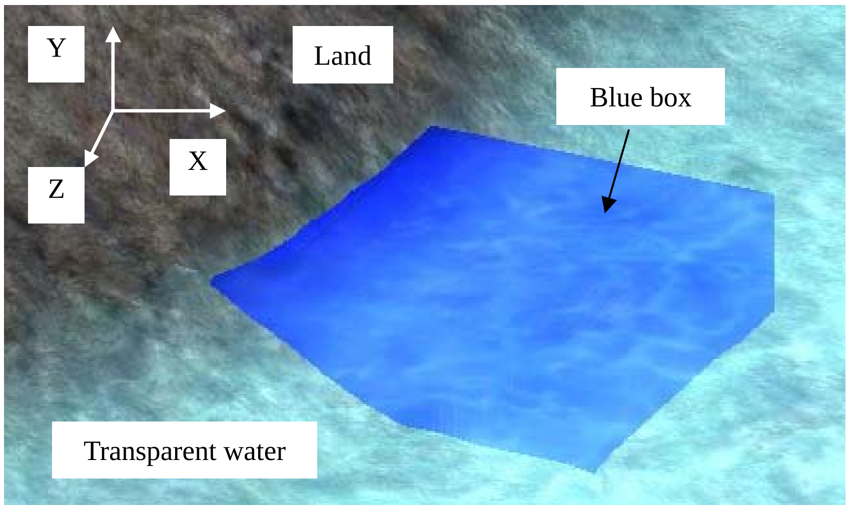


Figure 6.20: Transparent water with a blue box.

The left hand picture of Figure 6.21 shows the land model without water. The right hand picture adds the water, whose varying transparency is evident by the way that the box is easier to see nearer the water's surface.

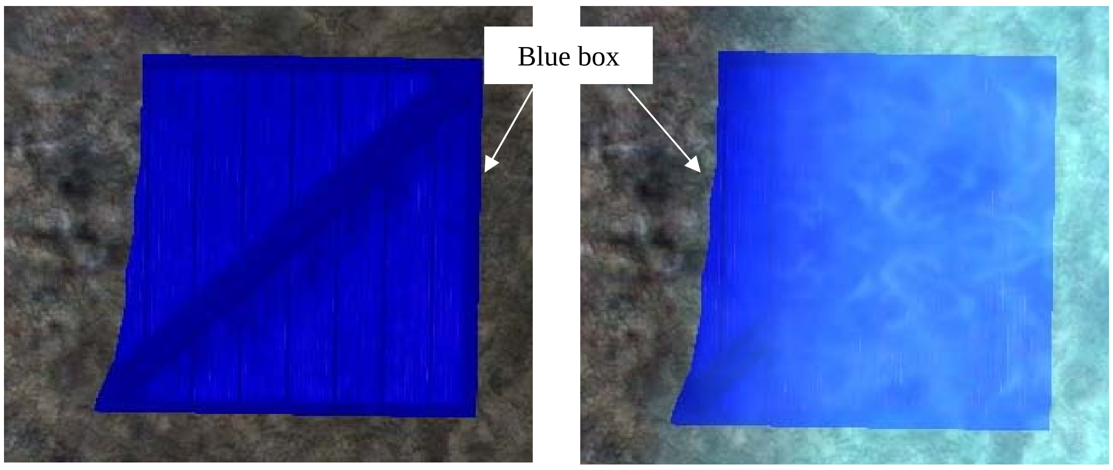


Figure 6.21: A blue box without water (left) and with water (right).

6.2.5 Implementation Details Using Shaders

A vertex shader and a fragment shader implement water surface multitexturing, wave crest shading, and transparency (see Appendix A12). Figure 6.22 shows the main steps: each pixel of the water surface is assigned a mix of the water material and detail texture, and then crest shading and transparency are added.

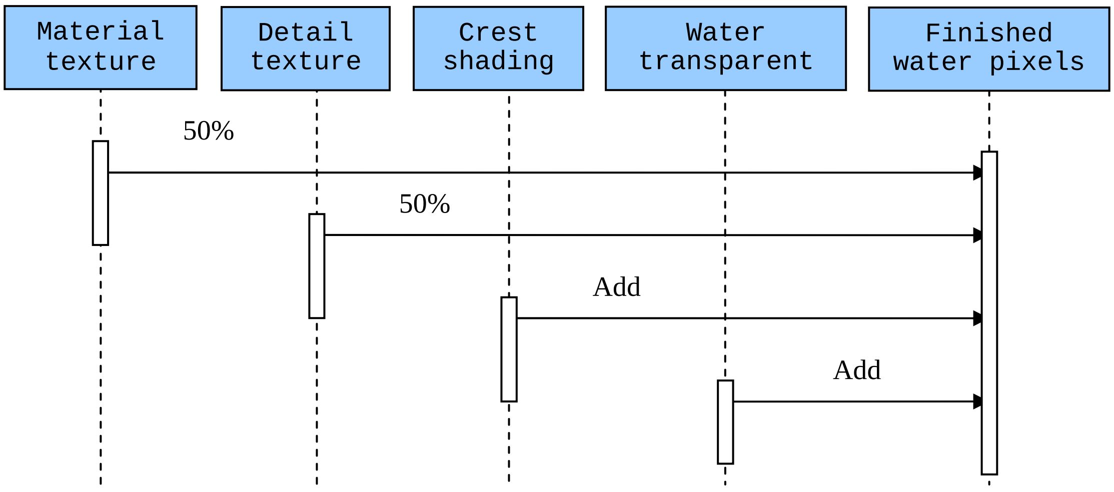


Figure 6.22: Steps of processing the water surface.

The pseudocode for the vertex shader reads in the texture data, the water coordinates, and the land coordinates which will be used by the fragment shader. The shader is called separately for each vertex of the model.

```
Vertex shader
{
   read water coordinate;
   read land coordinate;
   read the coordinate of water material texture;
   read the coordinate of water detail texture;
}
```

The pseudocode for the textures fragment shader textures the water surface, adjusts its crest color, and adds transparency. The shader is called separately for each fragment in the model (a fragment is roughly equivalent to a pixel).

```
Fragment shader
{
   H = water height - 19;
   color value = -H*0.25 – 0.1;

   tex1 = water material texture pixel;
   tex2 = water detail texture pixel;

   if(land height in the range 15, 19)
      water transparency = 1.01 – (land height - 15)*0.25;
   else if(land height less than or equal to 15)
      water transparency = 1;
   else if(land height more than 19)
      water transparency = 0.01;

   water pixel = ((tex1 R value + tex2 R value)*0.5 + color value,
                  (tex1 G value + tex2 G value)*0.5 + color value,
                  (tex1 B value + tex2 B value)*0.5 + color value,
                  water transparency);
}
```

The color of a water pixel is calculated using:

```
color value = -H*0.25 – 0.1;
```

`H*0.25` limits the color to a range smaller than (0, 1), and, the negative sign makes larger values darker. The reduction by 0.1 keeps the pixel from being too bright or dark.

6.3 Summary

Procedural and detail texturing were added to the land surface to make it have different features at different heights and improve the detail. Material and detail texturing applied to the water surface improved its look in a similar way. Wave crest shading and transparency were incorporated make the water more realistic.

CHAPTER 7

PARTICLE SYSTEMS

This chapter describes the use of particle systems to render two kinds of water special effects: water spray and breaking waves.

7.1 Water Spray

Water spray around the coastline (as in Figure 7.1) can be viewed as a particle system.

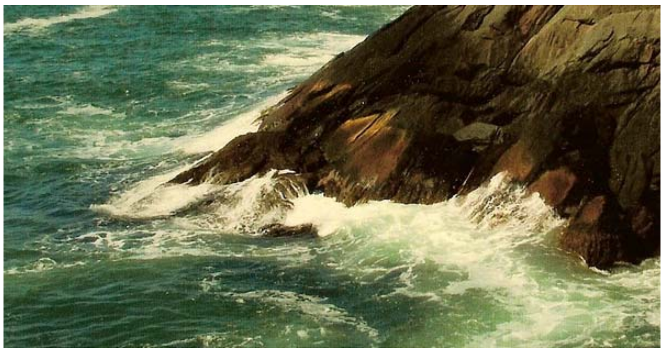


Figure 7.1: Water spray.

Aside from improving the realism of the model, the particle system will also hide some of the jagged edges of wave curves 3 and 4 as they move over the land (as seen in Figure 7.2).

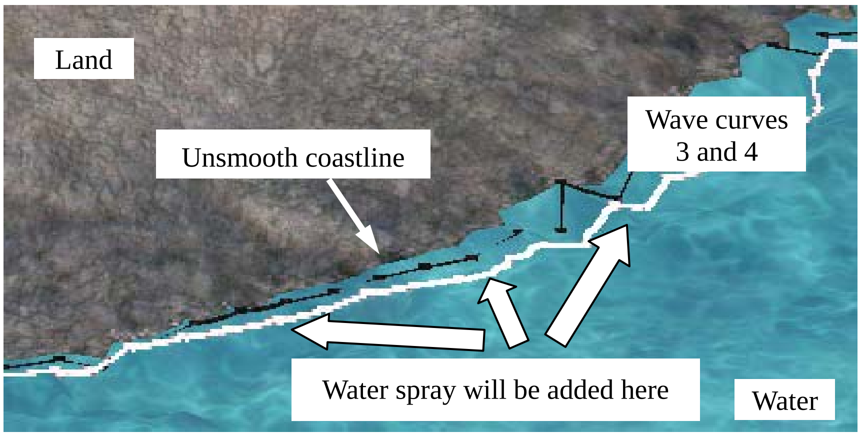


Figure 7.2: Coastline and wave curves without spray.

The spray particles will originate from the vertices of wave curves 3 and 4 as shown in Figure 7.3. The particle projectiles can be summarized as:

Position: there are 25 particles assigned to each vertex of the wave curves, randomly offset in the x-z plane to avoid excessive uniformity.

Velocity: a particle's vertical velocity is random, while it's horizontal velocity will be inherited from its wave curve vertex. The velocities will be assigned a random offset to avoid excessive uniformity.

Life time: A particle disappears when it drops below the water surface, and is recreated in the next frame.

Force: the only force is gravity.

Rendering: a particle is rendered using a texture (see Figure 7.4). Each particle is a point sprite rendered using a fragment shader, with its size adjusted by a vertex shader.

Time interval: the time interval is 0.1 in each frame.

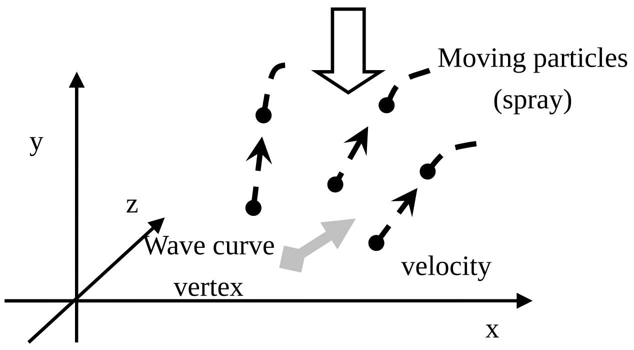


Figure 7.3: Water spray particles around a wave curve vertex.

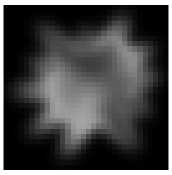


Figure 7.4: Texture for a particle.

Figure 7.5 shows the model after spray particles have been added to wave curves 3 and 4.

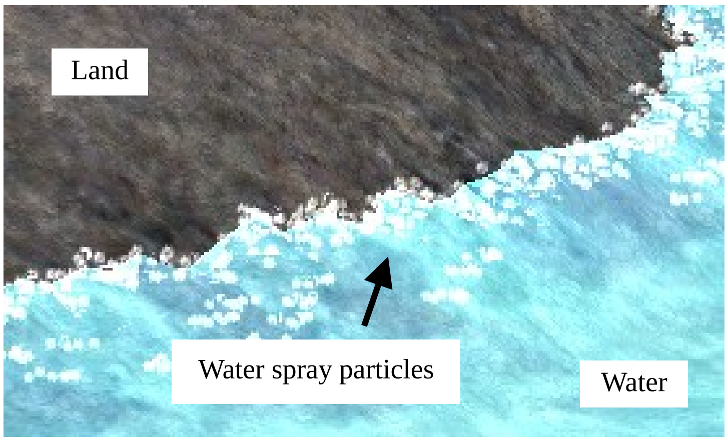


Figure 7.5: Water spray particles.

7.1.1 Implementation

The water spray particle system (and the breaking wave particle system described in Section 7.2) are managed by the Particle class (see Figure 7.6) and two shaders listed in Appendices A13 and A14 respectively.

Particle (Appendix A13)

`+ buildWaterSprayParData (...)`
`+ buildWaterSprayParticle(...)`
`+ buildBreakingWaveParticle(...)`
`+ buildBreakingWaveParData(...)`

Figure 7.6: Class Particle.

The following two subsections explain the Particle class and shaders using pseudocode.

Water Spray Particle Code

Data must be initialized in a particle when it is created or reused. Its position must be assigned the current wave curve vertex position plus a random offset. Similarly, a particle's horizontal velocity must be assigned that of the wave curve vertex, plus a random offset.

```
for(i less than wave curve 3 or 4 size)
   for(j less than 25 (particles size))
      particle’s x = x value of wave curve i + random (-1, 1);
      particle’s y = y value of wave curve i + random (-0.5, 0.5);
      particle’s z = z value of wave curve i + random (-1, 1);

      particle’s x velocity = x velocity of wave curve 1 + (-0.5, 0.5);
      particle’s y velocity = (-0.5, 0.5);
      particle’s z velocity = z velocity of wave curve 1 + (-0.5, 0.5);
```

In each rendering frame, a particle's data must be updated. It's position will be updated by its velocity multiplied by the time interval (`0.1`). It's y velocity particle needs to be updated due to the affects of gravity, and, a particle disappears when it drops below the water surface.

```
for(i less than wave curve 3 or 4 size)
   for(j less than 25)
      particle’s x += 0.1*particle’s x velocity;
      particle’s y += 0.1*particle’s y velocity;
      particle’s z += 0.1*particle’s z velocity;

      particle’s y velocity -= 9.8 * 0.1;

      if(particle’s y < 19) //default water surface level
         particle disappears;
```

Particle System Shaders

The same vertex and fragment shaders are used by both the water spray and breaking waves particle systems. The vertex shader gathers vertex and texture data, and adjusts the particle size in the range (2, 6).

```
Vertex shader
{
    particle position = vertex position;
    read particle texture coordinate;
    particle size = 8 - (particle y value - 19)*3;
    if(particle size less than 2)
       particle size = 2;
    else if (particle size more than 6)
       particle size = 6;
}
```

The particle size change is intended to reflect the idea that spray gets smaller as it travels upwards, away from the water (see Figure 7.7). The equation is:

```
    particle size = 8 - (particle y value - 19)*3;
```

The height of particle is offset by the default water level (`19`), then multiplied by `3` to increase its size difference. Subtraction from `8` makes higher particles smaller.

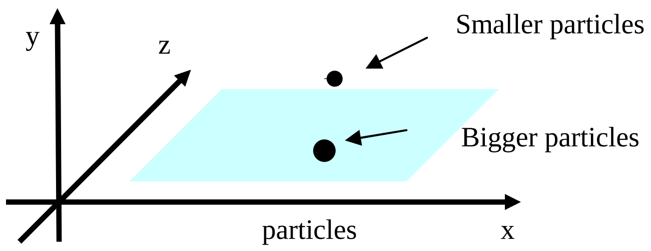


Figure 7.7: The particle gets smaller at increased heights.

The fragment shader applies a texture to the particle, using the texture in Figure 7.4.

```
Fragment shader
{
   tex = particle texture pixel
   particle pixel = (tex R, tex G, tex B, 1);
}
```

7.2 Breaking Waves

A breaking wave is visible because of the foam which appears along its top edge as illustrated by Figure 7.8.

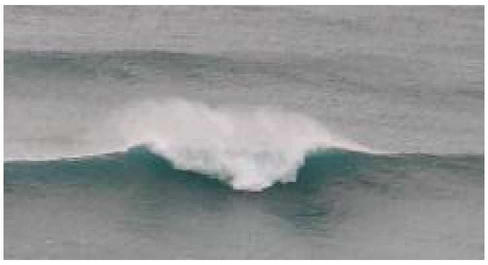


Figure 7.8: Foam on top of a breaking wave.

Figure 7.9 shows wave crests in the model without any breaking waves. The crests are quite hard to see even with crest shading (explained in Chapter 6). Adding foam will increase their realism. The foam will be created with a particle system whose particles originate on top of the wave crests as shown in Figure 7.10.

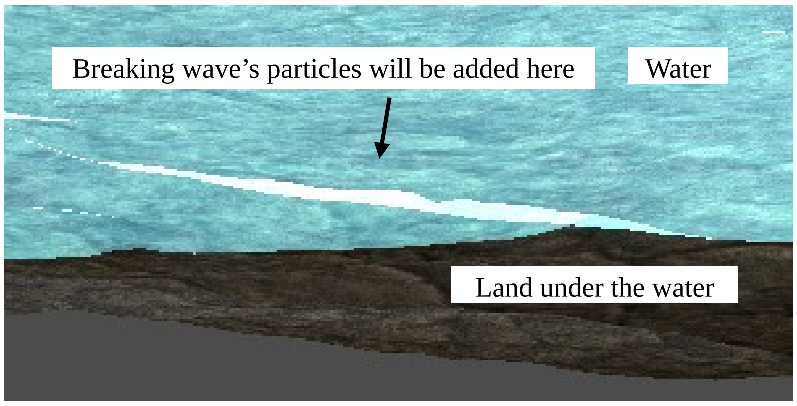


Figure 7.9: The model without breaking waves.

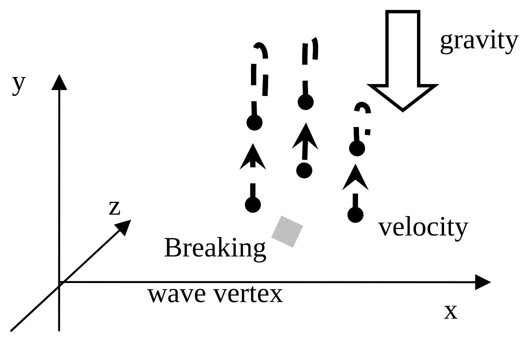


Figure 7.10: Breaking wave particles around the breaking wave's vertex.

The particle properties can be summarized as:

Position: 15 particles are assigned to each wave curve vertex, randomly offset to avoid excessive uniformity.

Velocity: the particles do not move in the horizontal plane but have a random vertical velocity.

Life time: particles disappear when they drop below the water, but are reused when the wave breaks again.

Force: the only force is gravity.

Rendering: each particle is rendered as a textured point sprite (the texture is shown in Figure 7.4). Rendering employs the same shaders as the water spray particle system.

Time interval: the time interval is 0.1 in each frame.

Figure 7.11 shows the model with particles added to the crests of the breaking waves.

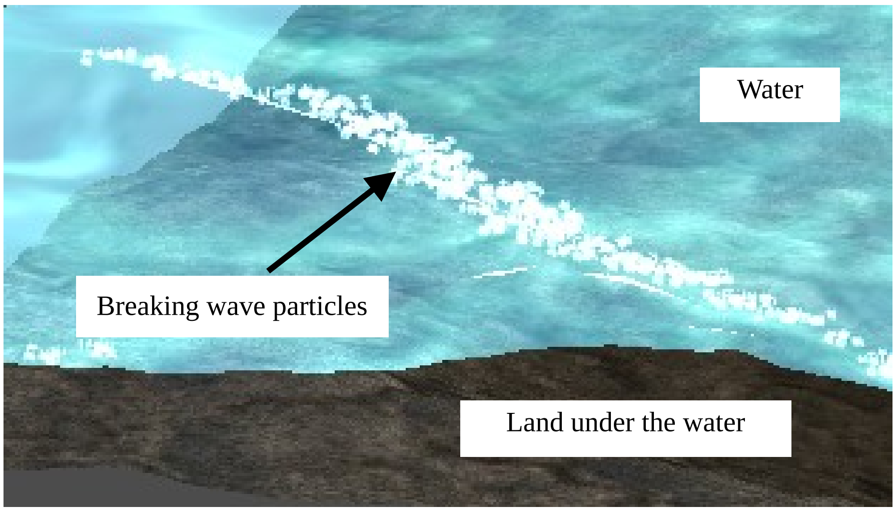


Figure 7.11: The model with breaking wave particles.

7.2.1 Implementation

As mentioned earlier, the Particle class in Appendix A13 generates both the water spray particle system and the breaking wave particle system. Also the same shaders are used for both particle systems (see Appendix A14). As a consequence, the pseudocode in this section only describes the parts of the Particle class used by the breaking wave system; the shaders will not be explained again.

Breaking Wave Particle Code

Data must be initialized in a particle when it is created or reused. It's position must be assigned the current wave curve vertex position plus a random offset. Since a breaking wave particle has no horizontal velocity, only it's vertical velocity is assigned a random, small value.

```
for(z less than mesh size -1)
    for(x less than mesh size -1)
        for(i less than 15 (particle size))
            particle’s x =  x value of vertex (x, z) + random(-0.5, 0.5);
            particle’s y =  y value of vertex (x, z) + random(-0.5, 0.5);
            particle’s z =  z value of vertex (x, z) + random(-0.5, 0.5);
            particle’s y velocity = (-0.5, 0.5);
```

In each rendering frame, a particle’s data must be updated. Its vertical position is updated by its velocity multiplied by the time interval (`0.1`), and the vertical velocity is affected by gravity. A particle disappears when it drops below the wave curve vertex.

```
for(z less than mesh size -1)
   for(x less than mesh size -1)
      for(i less than 15 (particle size))
         particle’s y += 0.1f * particle’s y velocity;

         particle’s y velocity += 9.8 * 0.1;

         if(particle’s y less than vertex’s y value - 0.5)
            particle disappears;
```

7.3 Summary

Water spray and breaking waves were added to the model in this chapter, implemented using particle systems and shaders. Water spray particles represent spray around the coastline, and improve the realism of the water surface. The breaking waves particle system increases the realism of the water crests by generating foam on the top of the crests.

CHAPTER 8

TESTING

This chapter presents data on the computational speed in frames/sec (FPS) of our model, by examining at four versions of it:

- the *original* model – a 128*128 island mesh with four wave curves
- the same island model, but with *five* wave curves
- a *larger* 256*256 mesh island, with four wave curves
- land in the form of a *harbour* rather than an island, using a 128*128 mesh and four wave curves

The five wave curves and harbour versions of the model allow us to briefly outline how to extend and change the model.

Each version will be timed with different rendering aspects enabled or disabled to see how they affect overall FPS speed:

- *all rendering* enabled: spring systems, collision detection, multitexturing, shaders, and particle systems
- rendering *without particle systems* (so water spray and breaking wave foam will not be available)
- rendering *without shaders*, so allowing the hardware to function without a GPU and/or a less advanced OpenGL driver (wave crest shading, water transparent and particles will not be available)

Each scenario will be tested on a PC and a less powerful notebook. The PC has a two-core 1.86 GHz 1GB DDRII-533 RAM, a Nvidia 9800GT 1GB graphics card. The notebook is a Core T2250 CPU 1.73 GHz device, with 2GB of RAM, and an Intel 950 graphic card which does not support shaders. Both machines are running Windows XP SP3 Professional. The FPS information is obtained using a screenshot utility called FRAPS, version 2.99 [30] which can also display FPS values for applications. The FPS for a particular run can be seen in the screenshots in this chapter, displayed in the top left corner (e.g. see Figure 8.1).

8.1 Original Model

The original model is a 128*128 size mesh containing a single island, and four wave curves. Figure 8.1 shows the "all rendering" case utilizing multitextured land and water, crest shading, water transparency, spray and breaking waves. The model executes at about 54 FPS (the FPS numbers supplied in this chapter are averages, calculating over 10 or more runs of a model, and rounded to the nearest integer).

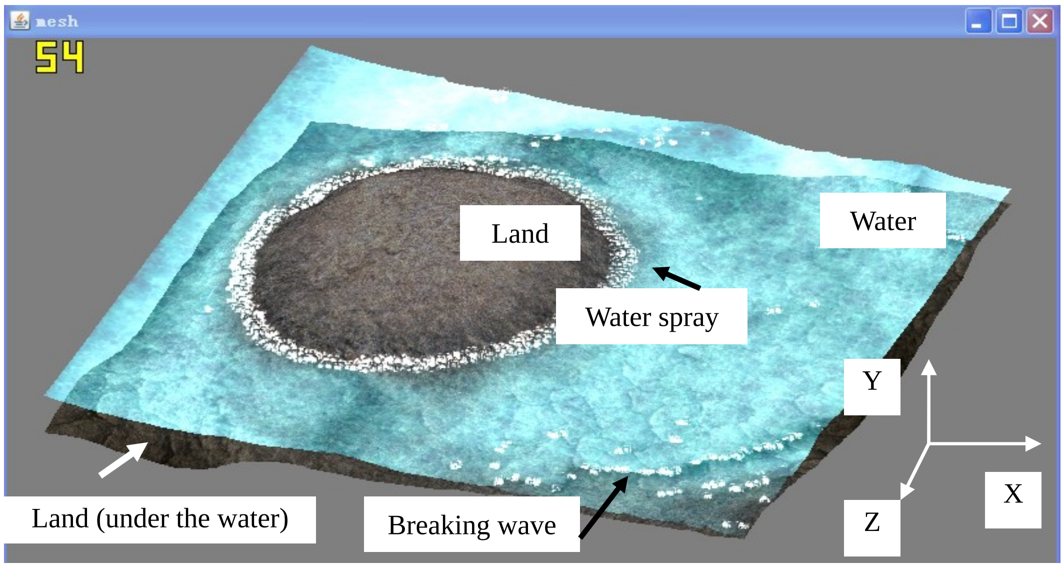


Figure 8.1: Original model/full rendering.

Figure 8.2 shows the model without particle systems for the spray and breaking waves. The rendering FPS value is 135, a much improved value, which indicates the resource intensive nature of the particle systems.

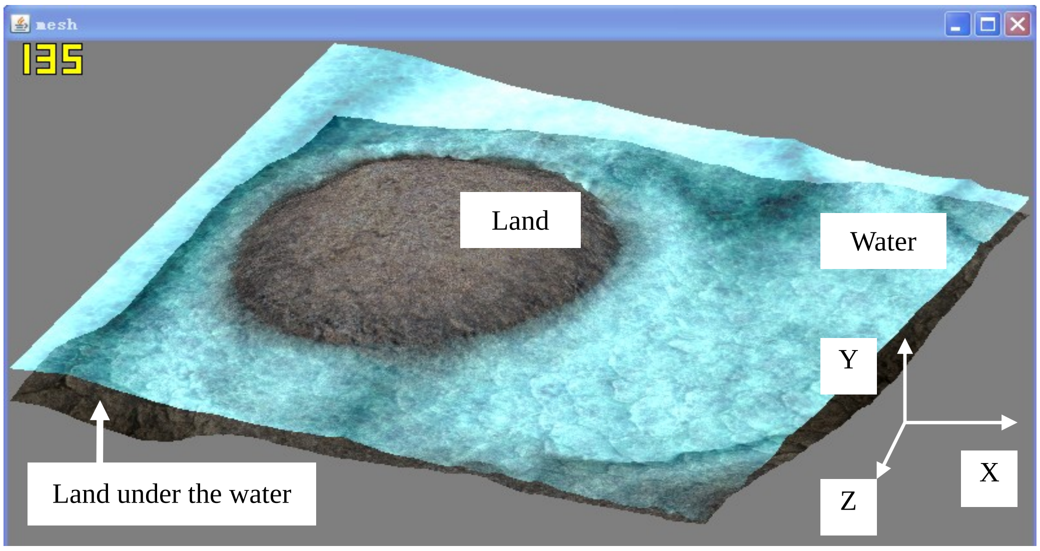


Figure 8.2: Original model/no particle systems.

Figure 8.3 displays the model without shaders (shaders support water multitexturing, wave crest shading, water transparency, and particles). The much less realistic model executes at about 138 FPS, which is almost unchanged from the “no particles” version. This suggests that shaders have little impact on the frame rate, probably because they are processed by the GPU; the real bottleneck is the CPU. Shaders are used to carry out multitexturing, but this is so important for the model realism, that it has been reimplemented to use fixed pipeline operations in the models with no shaders.

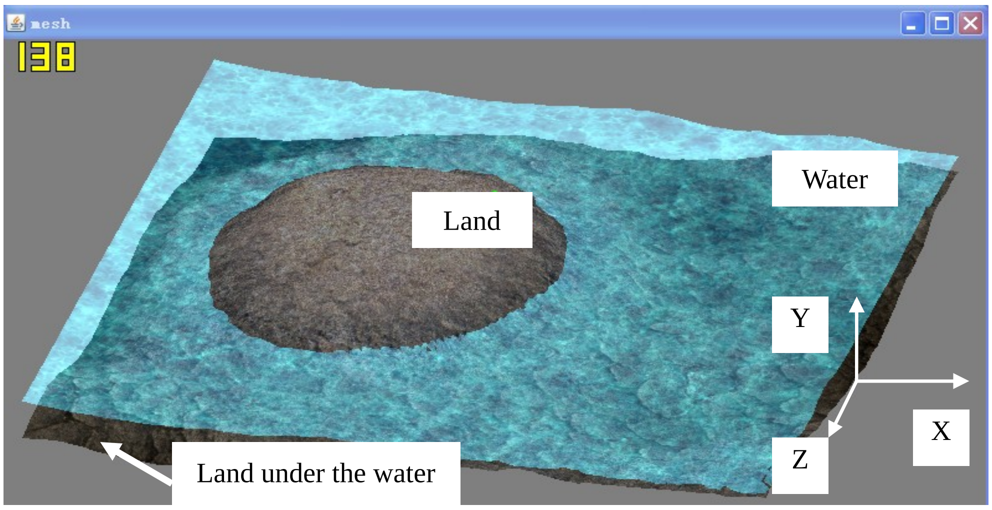


Figure 8.3: Original model/no shaders.

If this model version (no shaders) is run on the notebook, then the FPS drops drastically to 83. Its not possible to run the full or no particles version of the model on the notebook since it does not support shaders. Figure 8.5 uses the same model as in Figure 8.4 but moved off-screen. This relieves the machine of almost all its rendering tasks, and the FPS goes up to 111. This same scenario on a PC gives a FPS of 138, which indicates that the notebook is about 27 frames slower than the PC or (111/138)% as fast.

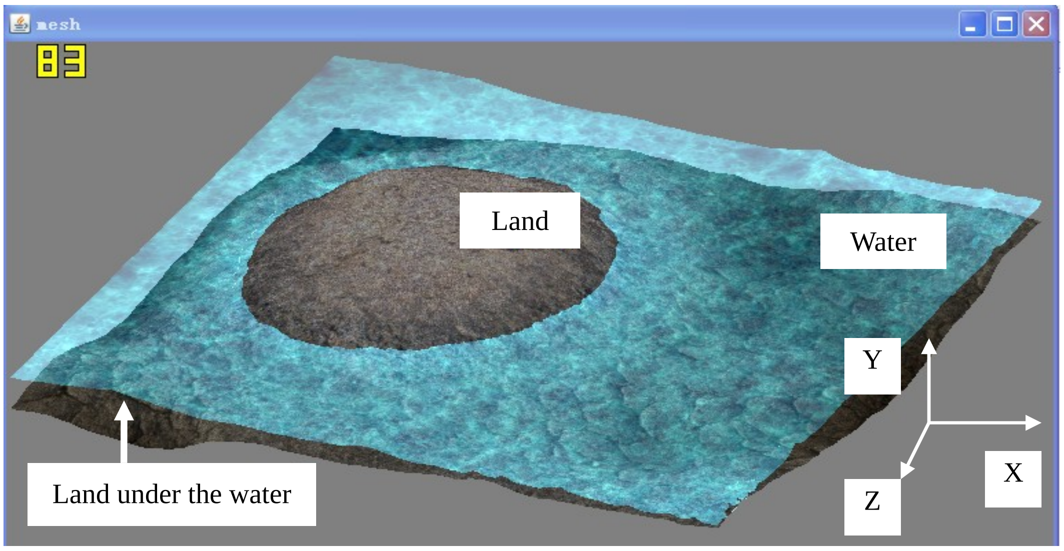


Figure 8.4: Original model/no shaders/on a notebook.

Figure 8.5: Off-screen model on a notebook.

8.2 Five Wave Curves

This version of the model is the same as in Section 8.1: a 128*128 mesh, but with *five* wave curves (see Figure 8.6).

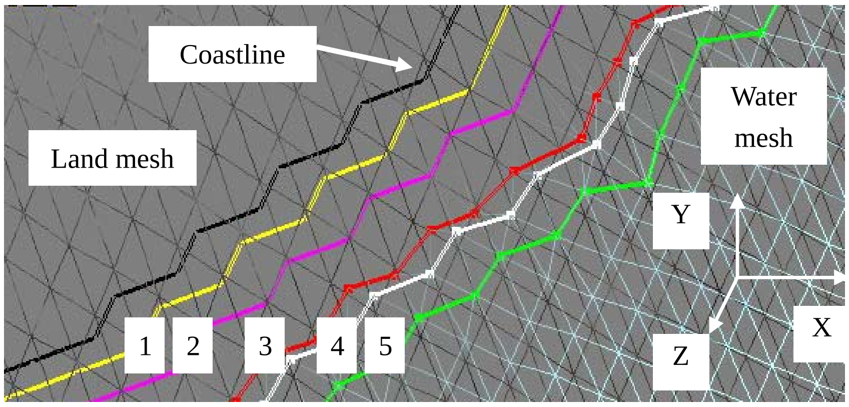


Figure 8.6: Five wave curves of the meshed model.

The extra curve has its own springs, waves collisions, and water spray particles. Wave curve 5 is similar to wave curves 3 and 4, in that it uses position springs and wave springs to connect it to inner curve (wave curve 4). Waves collision is monitored between wave curves 4 and 5, and water spray particles are utilized as in wave curves 3 and 4.

Wave curve 5 is built in a similar way to wave curves 2, 3 and 4, as explained in Section 4.2.3; the sequence diagram in Figure 8.7 shows the main steps: `getWaveCurve()`in the WaveCurves class (Appendix A8) is used to build wave curve 5.

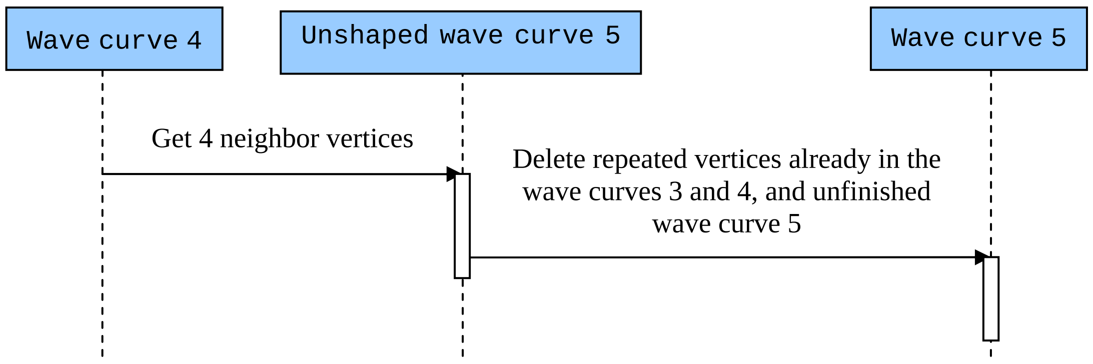

Wave curve 4
Unshaped wave curve 5
Wave curve 5
Get 4 neighbor vertices
Delete repeated vertices already in the wave curves 3 and 4, and unfinished wave curve 5

Figure 8.7: Calculating wave curve 5.

The sequence diagram in Figure 8.8 shows how position springs, wave springs, and collisions detection control the motion of wave curve 5. The wave spring force is added to the position spring force, to calculate the x-z velocity of wave curve 5. This velocity will be modified if waves collisions occur. The final position of the wave curve is calculated from this final velocity.

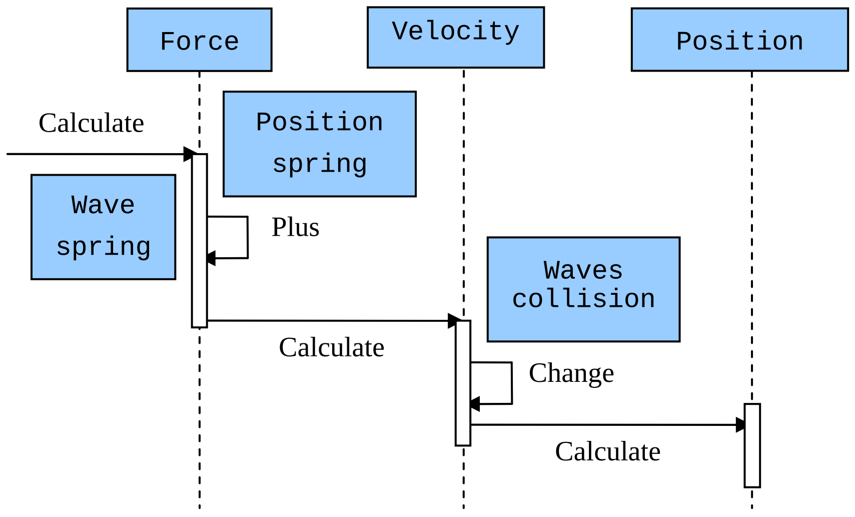


Figure 8.8: Springs and collisions control the motion of wave curve 5.

`getVelocityForOtherWaveCurves()` in the Velocity class (Appendix A9) is used to calculate the velocity of wave curve 5.

The `getVelocityForWaveCurves()` method in the Collision class (Appendix A10) controls the velocity of wave curve 5. Its water/land collision features are not utilized, because that type of collision can not occur with wave curve 5. The following pseudocode shows `getVelocityForWaveCurves()`for wave curve 5.

```
for(i less than wave curve 4 size)
   //collision between waves
   if(i move through line segment bc of wave curve 5)
      x velocity (i, b, c) is the average x velocities of i, b, c;
      z velocity (i, b, c) is the average z velocities of i, b, c;
```

The height of wave curve 5 is adjusted by `adjustHeightOfWaveCurve()` in the HeightOfWaveCurve class (Appendix A11), and the water spray particles are created using the unchanged `buildWaterSprayParData()` and `buildWaterSprayParticle()` in the Particle class (Appendix 13) and in the shaders code in Appendix 14.

On a PC, the FPS for the model is about 42 (Figure 8.9) which is less than the version with four wave curves (5 frames/sec). This reduction is to *be expected* because of the increased calculations required for the position and wave springs, waves collisions, and water spray particles. If no particle systems are employed, then the model runs much faster at 123 FPS, as shown in Figure 8.10.

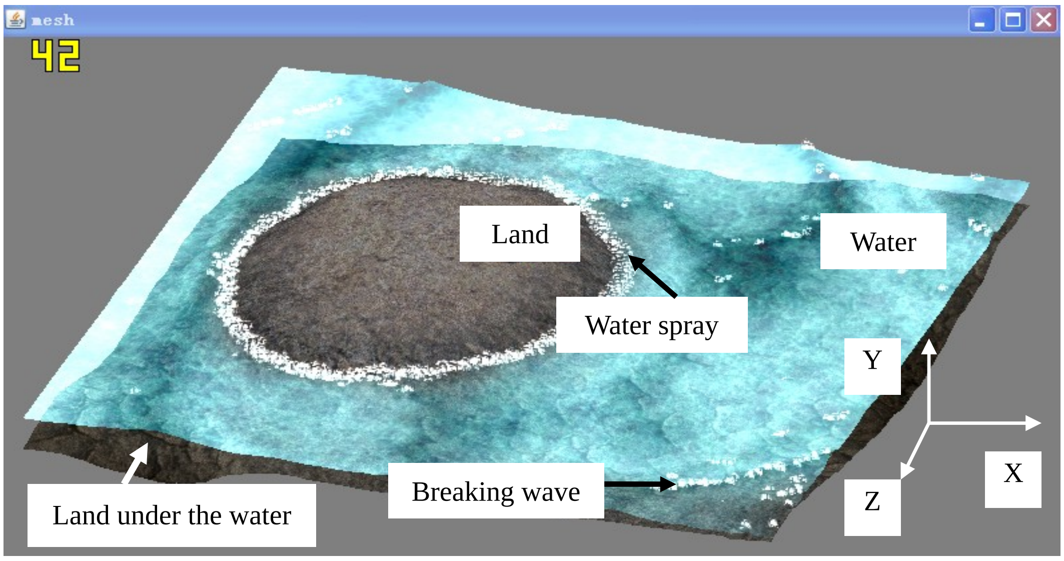


Figure 8.9: Five wave curves model/full rendering.

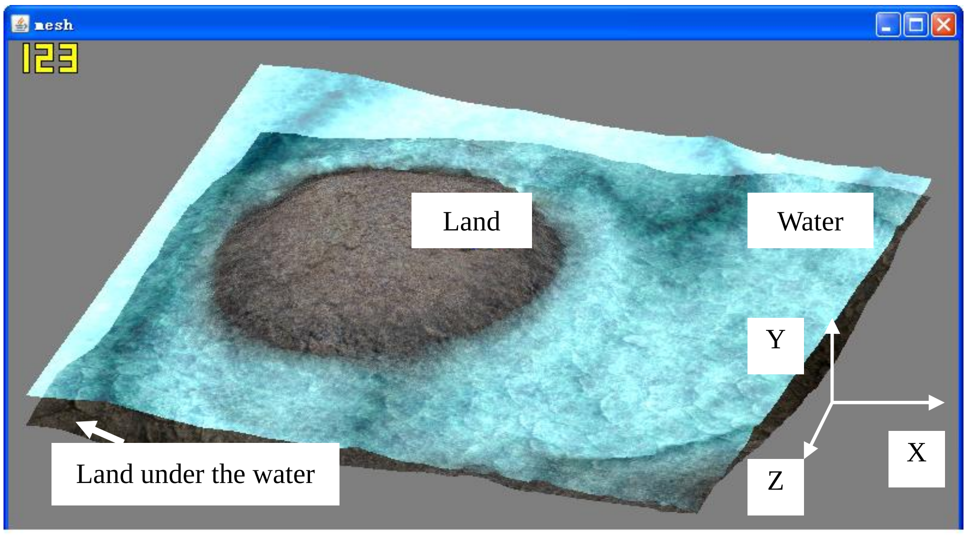


Figure 8.10: Five wave curves model/no particle systems.

Figure 8.11 shows the model running without shaders (126 frame/sec) which suggests that the bottleneck is CPU rendering, not shader support.

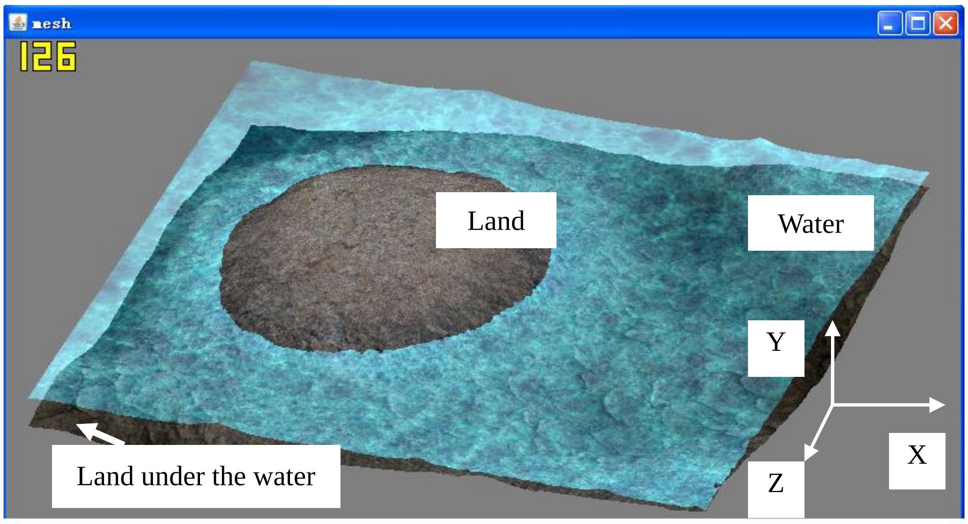


Figure 8.11: Five wave curves model/no shaders.

When this model version is run on a notebook, the frame rate drops to 84 FPS (Figure 8.12) compared to 99 FPS when the model is completely off-screen. This suggests that the graphics card of the notebook is limiting the FPS rate.

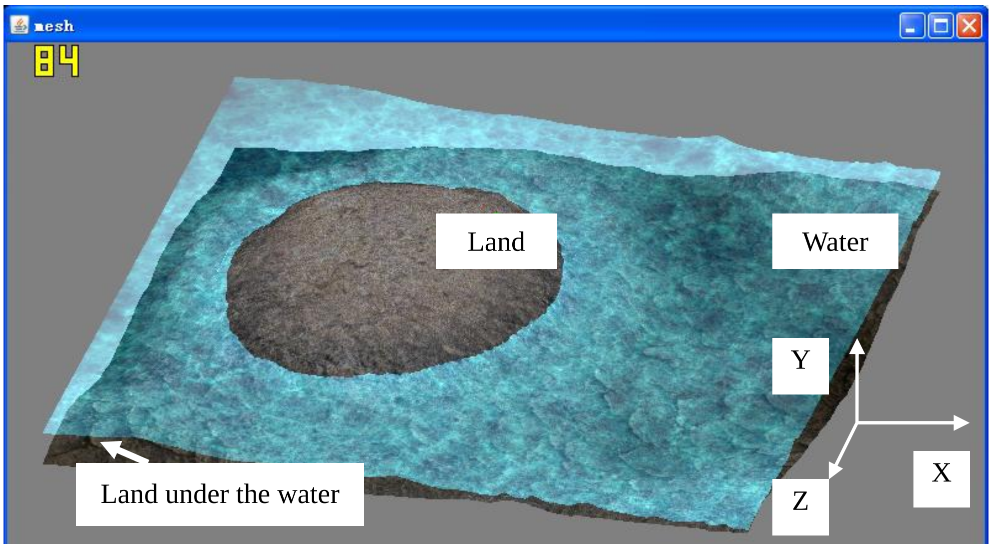


Figure 8.12: Five wave curves/no shaders/on a notebook.

8.3 A 256*256 Island

In this section, the land and water mesh is increased in size from 128*128 to

256*256. The increase requires a 256*256 height map, shown in Figure 8.13.

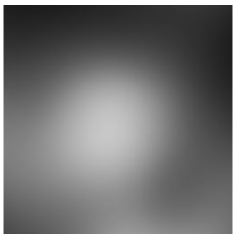


Figure 8.13: The height map of the 256*256 island.

In addition, the landsize and watersize parameters in the Mesh class (Appendix 6) must be changed to 256; no other changes are necessary.

The increased size of the model mesh (four times bigger) has a drastic effect on the FPS, since rendering speeds depend closely on the number of vertices in the model. Figure 8.14 shows that the application runs at about 18 FPS (the original model uses 54 frames/sec). The increased size affects all aspects of the model including the springs, collision detections, and the particle systems.

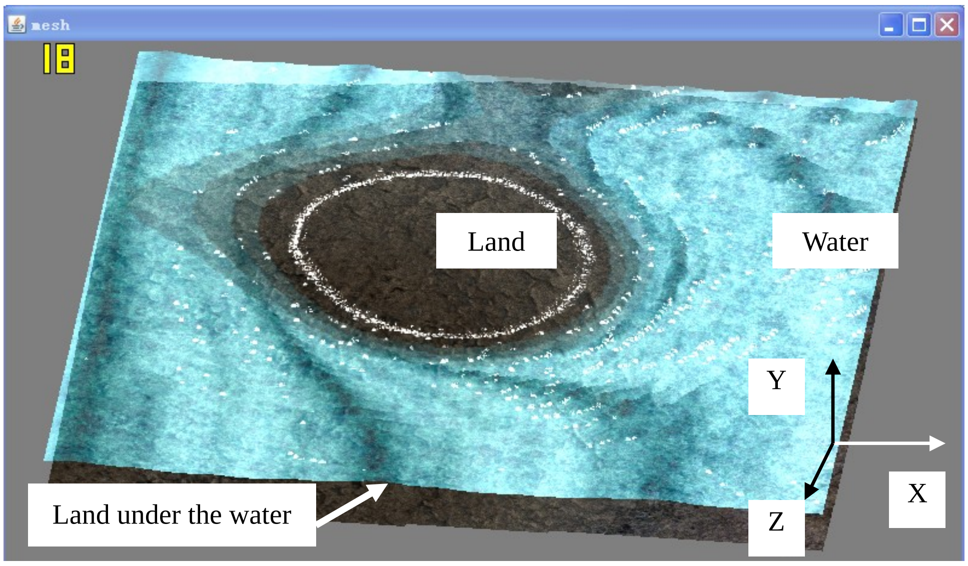


Figure 8.14: 256*256 mesh/full rendering.

Figures 8.15 and 8.16 shows that this model runs faster without particle systems and shaders.

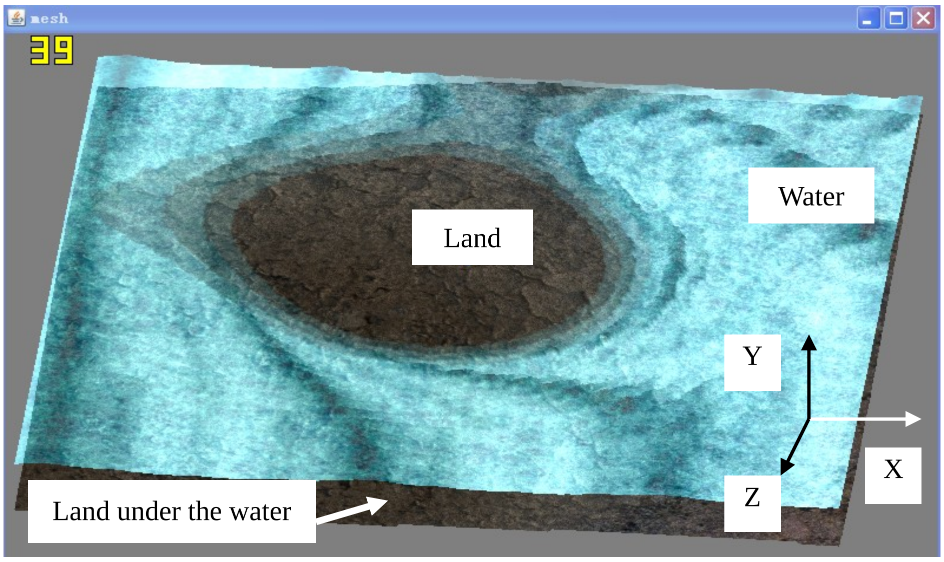


Figure 8.15: 256*256 mesh/no particle systems.

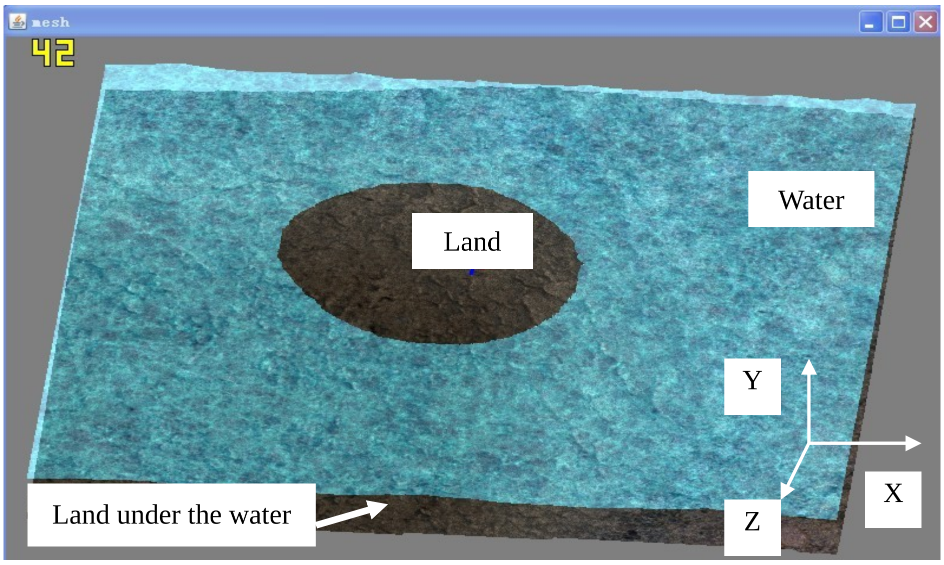


Figure 8.16: 256*256 mesh/no shaders.

The difference between the “no particle systems” and “no shaders” versions is about 3 FPS (42-39), which is the same as the difference for the original model (138-135=3). This again shows that shaders have little impact on the frame rate. The 256*256 mesh without shaders can be run on the notebook, giving a frame rate at 24 (see Figure 8.17).

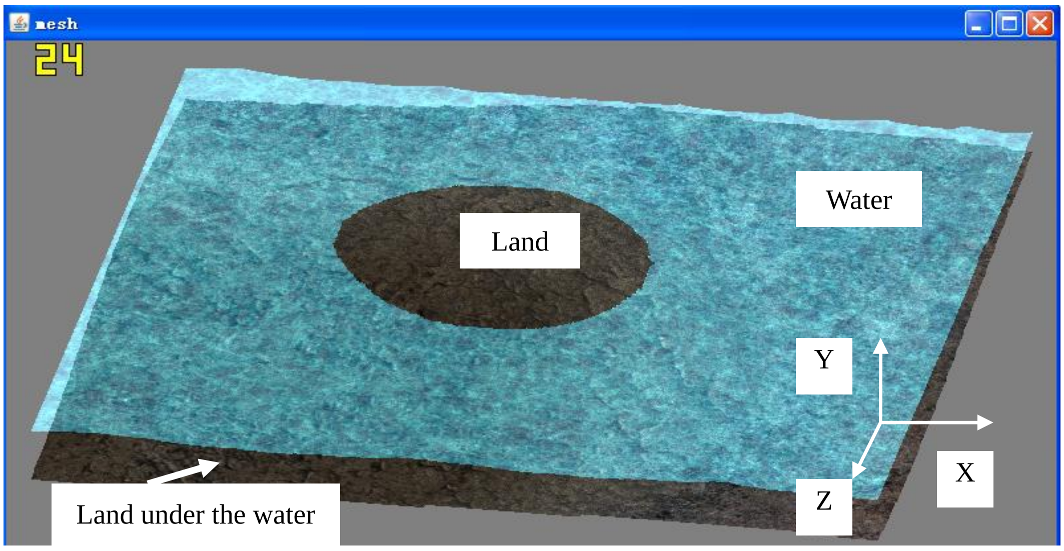


Figure 8.17: 256*256 mesh/no shaders/on a notebook.

Figure 8.17 again shows that the increased mesh size is costly in resources since the original model on the notebook runs at 83 frame/sec (in Figure 8.4).

8.4 Testing with a Harbour

The code developed in this thesis makes the assumption that the model only contains one land mass, which can be any shape. In this section the land is changed to be a 'harbour' shown in Figure 8.19. This requires a new height map, shown in Figure 8.18. Its animation frame rate is 93, as shown in Figure 8.19. It is faster than the same-size island (see Figure 8.14), because the island has a large area of water which require more calculations of the water profile functions and breaking wave particles. The calculations for the spring systems and collisions detection depend on the coastline or wave curves' lengths, which are similar in both models.

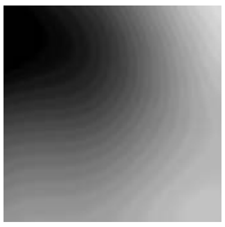


Figure 8.18: The harbour height map.

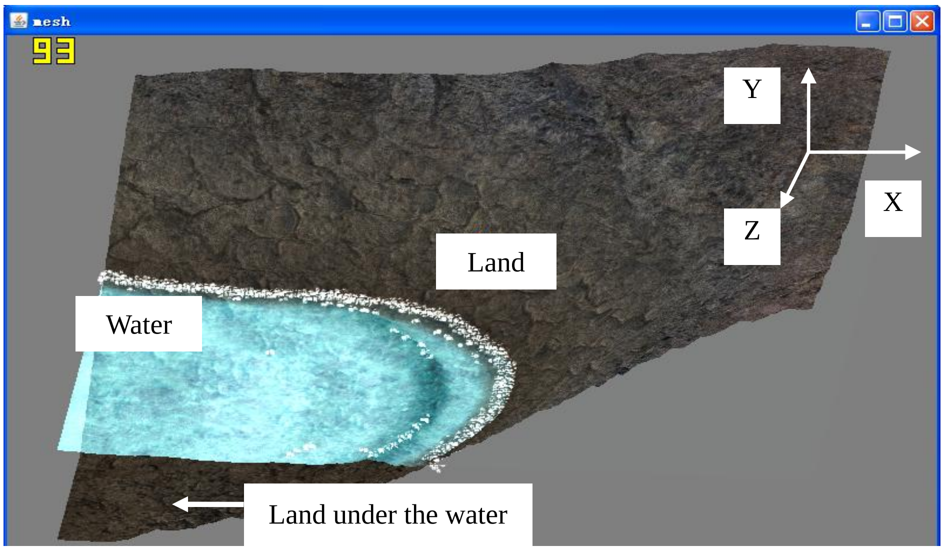


Figure 8.19: Harbour model/full rendering.

Figure 8.20 shows that the same model without particles runs at about 312 FPS, which is much faster than original model without particles (135 FPS in Figure 8.2). It shows that the small area of water greatly reduces the number of calculations required for the water profile functions, water spray, breaking wave particles, spring systems, collisions detection, which all depend on the number of water vertices.

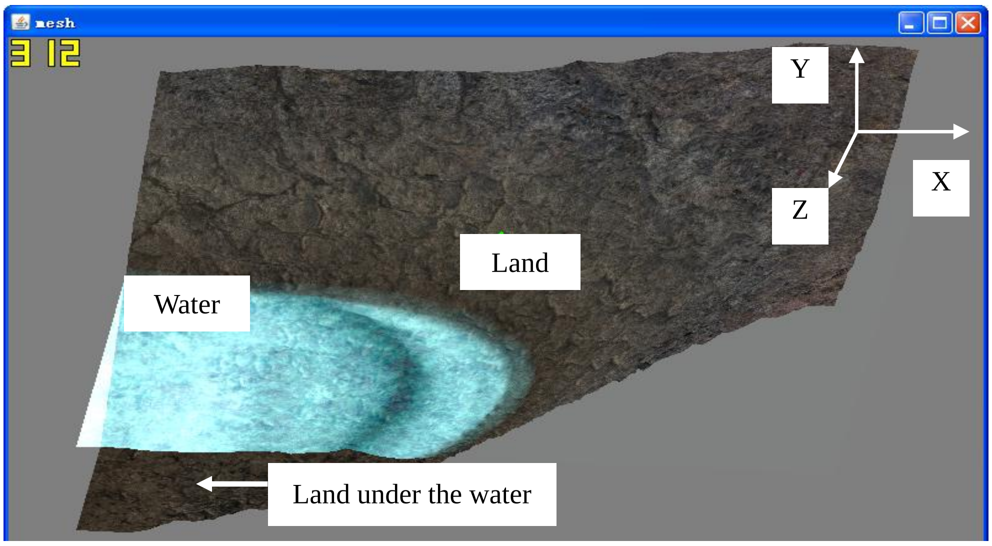


Figure 8.20: Harbour model/no particle systems.

Figure 8.21 shows that the model without shaders executes at 315 FPS, which is also much faster than the original model without shaders (138 FPS in Figure 8.3). It again shows that the water area has a direct relation to rendering speed.

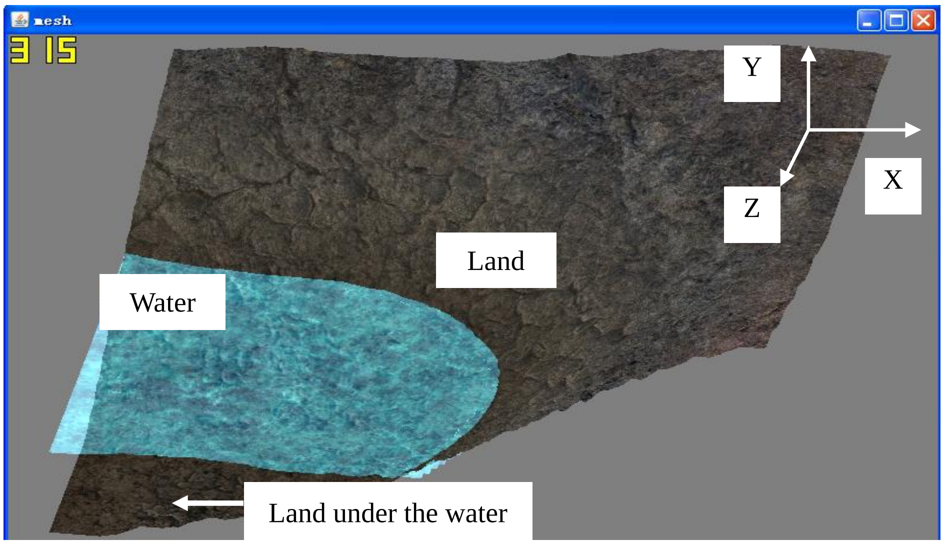


Figure 8.21: Harbour model/no shaders.

Figure 8.22 shows the model running on a notebook at 114 FPS. The corresponding island on the notebook (Figure 8.4) runs at 83 FPS. This shows that the water area greatly affects the rendering speed.

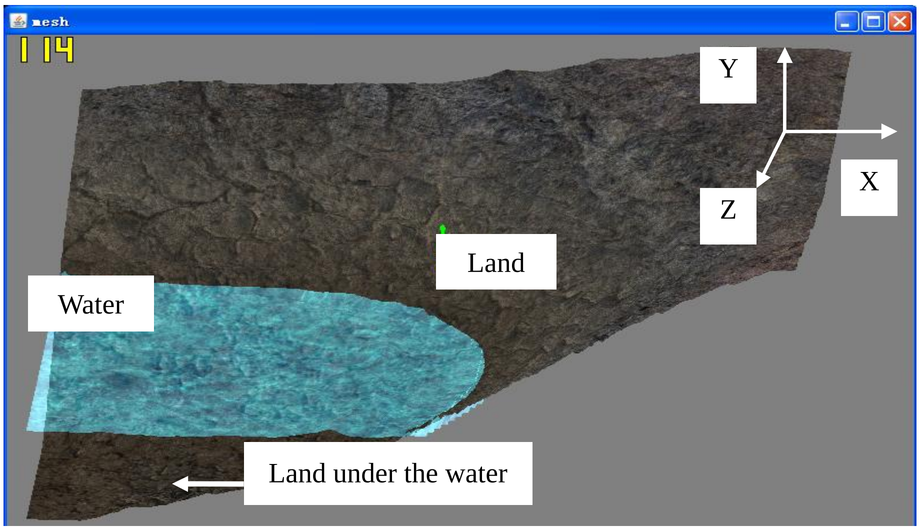


Figure 8.22: Harbour model/no shaders/on a notebook.

If the model requires more than one island or harbour, then each will need to be modeled independently with their own coastline, wave curves, spring systems, collision detections and water spray particles. In other words, the required model will need to be separated into component islands/harbours, and then the results combined. This combination will require non-trivial changes to the model.

8.5 Data Analysis

Table 8.1 shows the average frame rate (FPS) of the different versions of the models running on a PC and a notebook. The notebook does not support shaders, and so it was not possible to test the full versions or particle-free versions of the models on that hardware. Only the versions of the models without shaders were tested on the notebook. Figure 8.23 shows the table data in graph form. The average FPSs were obtained by running each model at least 10 times.

Model version		FPS on PC	FPS on notebook
Original island with four wave curves (128*128)	Full version	54	–
	Without particles	135	–
	Without shaders	138	83
Original island (five wave curves)	Full version	42	–
	Without particles	123	–
	Without shaders	126	84
A 256*256 island with four wave curves	Full version	18	–
	Without particles	39	–
	Without shaders	42	24
A harbour with four wave curves	Full version	93	–
	Without particles	312	–
	Without shaders	315	114

Table 8.1: Average FPS of different models.

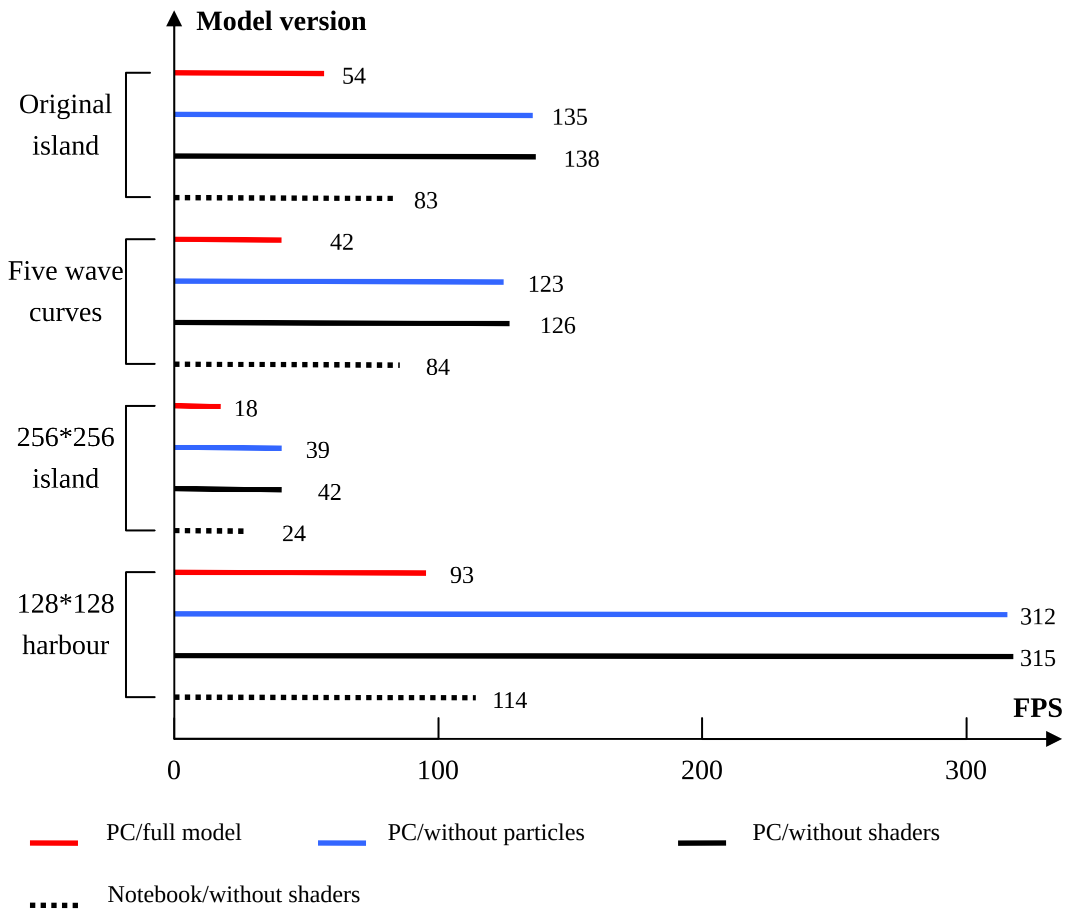


Figure 8.23: Average FPS of different models on different hardware.

The particle systems use a large amount of resources, as can be seen by the increase in FPS when particles are not used. Also GPU-based shaders are very efficient, hardly affecting the FPS. This suggests that more effects should be moved into the shaders, such as particle dynamics (at present only particle size and texturing are done by the shaders).

Model mesh size has a significant effect on frame rate, since it impacts the calculations of the water profile functions and breaking wave particles. One possible optimization is to use a single profile function to calculate the height of the water vertices in deep water, since there is no need to model wave refraction except in shallow water [25].

Increasing the number of wave curves reduces the rendering speed only slightly, which is somewhat surprising since wave curves affect the spring systems and collision detection. This is a good result since adding more wave curves will make the model more realistic.

The notebook used in these tests is typical of current low-end devices in that it does not support shaders. For improved realism and speed, it is better to choose a device that has a GPU.

8.6 Summary

This chapter tested the system with four models – the original model, the model with five wave curves, a 256*256 island, and a harbour. Each model was timed with various rendering aspects turned on or off, including without particle systems and without shaders. The models were tested on a typical PC and notebook.

The rendering speeds were compared, speed factors identified, and suggestions made about how to speed up the system. This chapter also explained how to change the mesh size of the model, and change the shape of the island into a harhour.

CHAPTER 9

DISCUSSION AND CONCLUSIONS

The system described in this thesis models the interaction of water and coastal land using a novel combination of three types of springs (position, wave, and height springs), and two forms of collision detection. The simulation exhibits realistic behavior between waves and the coastline, and between the waves themselves, while rendering at good speeds. The spring system is relatively simple to understand and fine-tune, and is based on the physical characteristics of real waves.

The land and water surfaces are created from mesh of the same size and box structure, land height is obtained from a height map (Chapter 3).

Water waves, including wave refraction, are constructed from a combination of profile functions, obtained from a phase function (Chapter 4). Although our use of profile functions comes from Peachey [4], the combination of several functions to produce more natural looking waves is our own work.

Novel coastline and wave curves data structures are employed to represent the interaction between the water and the land at different depths (Chapter 4). Wave curves 1 and 2 model the vertical movement of waves as they impact the land, while wave curve 3 and 4 represent the water's tidal behavior, including wave/wave impact and crossover. Our use of coastline and wave curves to represent water/land interaction is original. Fournier and Reeves [5] do not address this issue while Maes, Fujimoto and Chiba's model [8] can not move in the horizontal plane.

The most novel aspects of this thesis are discussed in Chapter 5. In our system, three spring systems are employed together to control the water motion in the vertical and horizontal planes. The tidal behavior of wave curve 3 and 4 are defined using position and wave spring systems and collision detection, while wave curves 1 and 2 utilize a height spring system.

Multitexturing is introduced in Chapter 6 to improve the realism of the land and water surfaces. The land uses procedural texturing to combine different textures at different land heights, and detail texturing for increased resolution. The water utilizes a material texture, and its own detail texturing. Wave crests shading and water transparency are also included by the use of shaders.

Chapter 7 utilizes particle systems to represent water spray and foam on breaking waves.

Chapter 8 investigates the speed of our approach by comparing the average frames/second (FPS) across four models running on a PC and a notebook. The 128*128 original model runs at about 54 FPS on a PC; with no shaders it can run at 135 FPS, and at about 83 FPS on a notebook. A 256*256 mesh model runs at 18, 42 and 24 FPS in similar situations. Chapter 8 also illustrates that it is straight-forward to change the shape and size of the landmass, but the system is restricted to a single landmass. Adding more islands or harbours would require some classes in the implementation to be redesigned.

Compared with the models of Peachey [4], Fournier and Reeves [5], this system can render real force interactions between water and land because of its spring systems. Unlike Maes, Fujimoto and Chiba's model [8], water vertices around the coastline in this model really move in the horizontal direction instead of as variable height water columns. This model can render a large water surface much faster than in Foster and Fedkiw [6] or Enright, Marschner and Fedkiw's simulations [7].

My long term goal is to use this approach to model tsunami-land interaction. The spring system will need to be modified to deal with large waves (over 30m in height) moving at very high speeds (more than 800 km/h) [31]. The coastline interaction will need to be more complicated to deal with the way a tsunami can wash over a large body of land. Perhaps, more than eight wave curves will be needed, four of them close to the coastline moving up and down as wave curves 1 and 2 do in the existing model. The others waves will need to move over long horizontal distances from their rest positions towards the coastline.

Model Features Illustrated

Figure 9.1 shows water moving towards the land, and water spray particles around the coastline. The coastline is hidden under the land normally, but is made visible in the figures of this chapter to show its relationship to the wave curves.

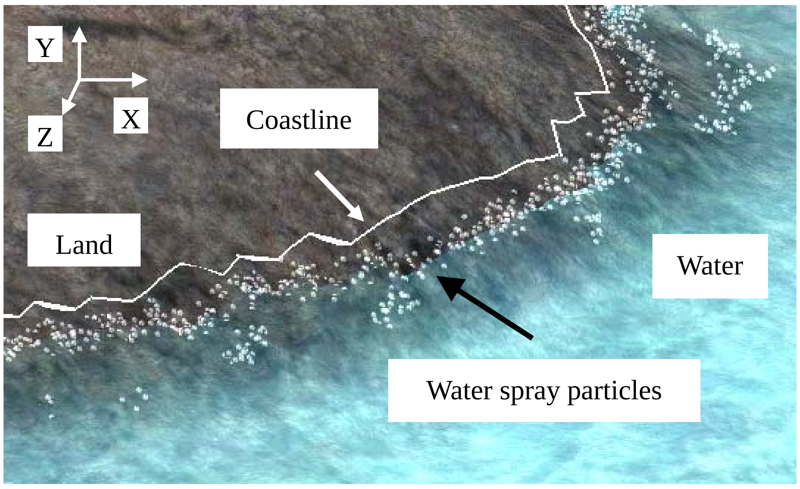


Figure 9.1: Water moving towards the land.

Figure 9.2 is a cross-sectional view of the water moving towards the land and producing water spray particles.

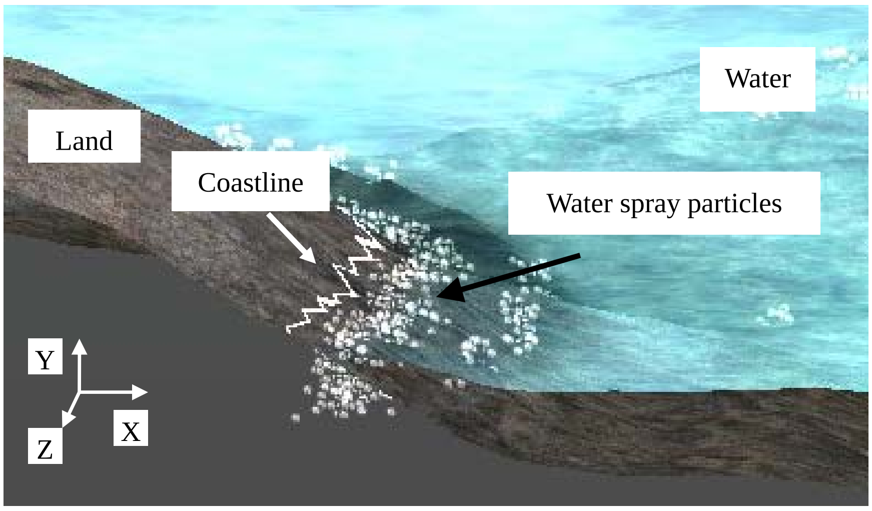


Figure 9.2: Water moving towards the land (cross-sectional view).

Figure 9.3 shows the land, coastline, wave curves, and water spray particles of Figure 9.2 from overhead. The red and white lines in the water mesh are wave curves 3 and 4, which are normally invisible. Wave springs and collision detection mean these wave curves can never cross-over.

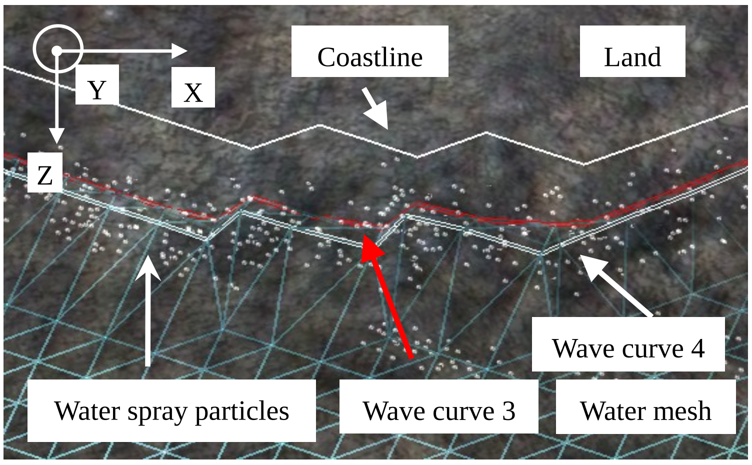


Figure 9.3: Wave curves moving towards the land.

Figure 9.4 shows the scene from Figure 9.1 after the water has rebounded from the land.

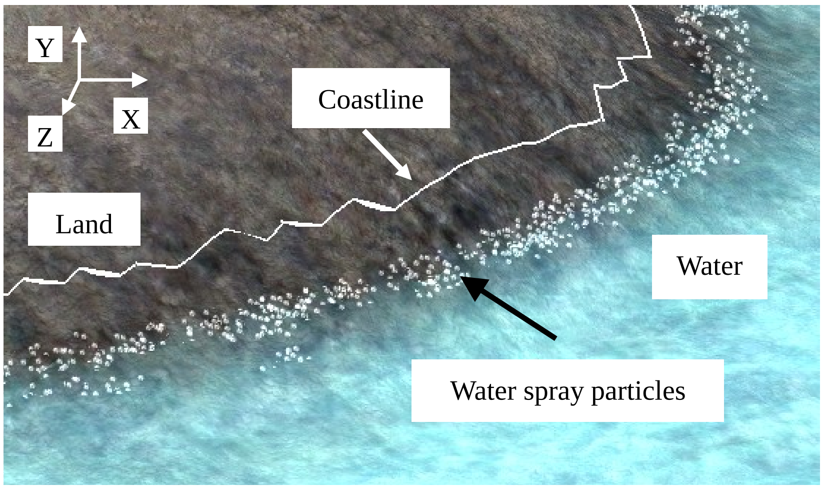


Figure 9.4: Water retreating from the land.

Figure 9.5 shows the cross-sectional view of the scene in Figure 9.4, as the water retreats from the land.

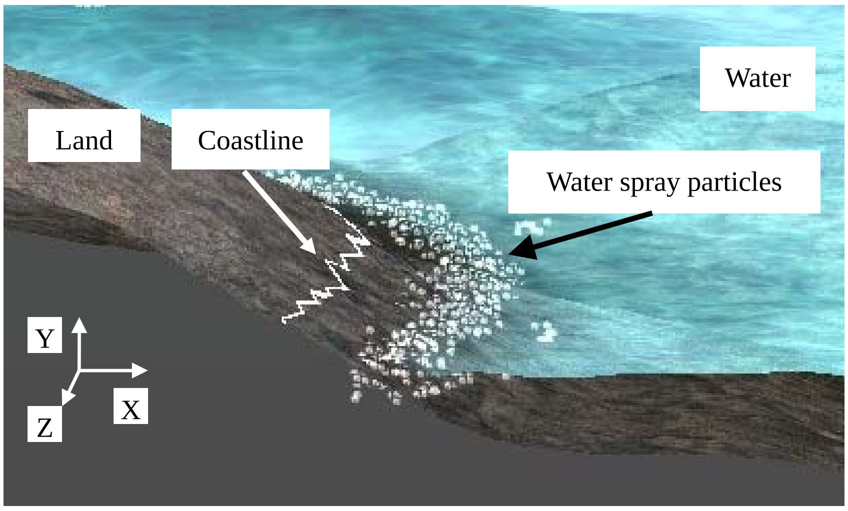


Figure 9.5: Water retreating from the land (cross-sectional view).

Figure 9.6 is a view of Figure 9.5 from overhead. It shows wave curves 3 and 4 being pulled back to their rest positions by their position springs. The interaction between wave curves 3 and 4 are controlled by wave springs and collision detection.

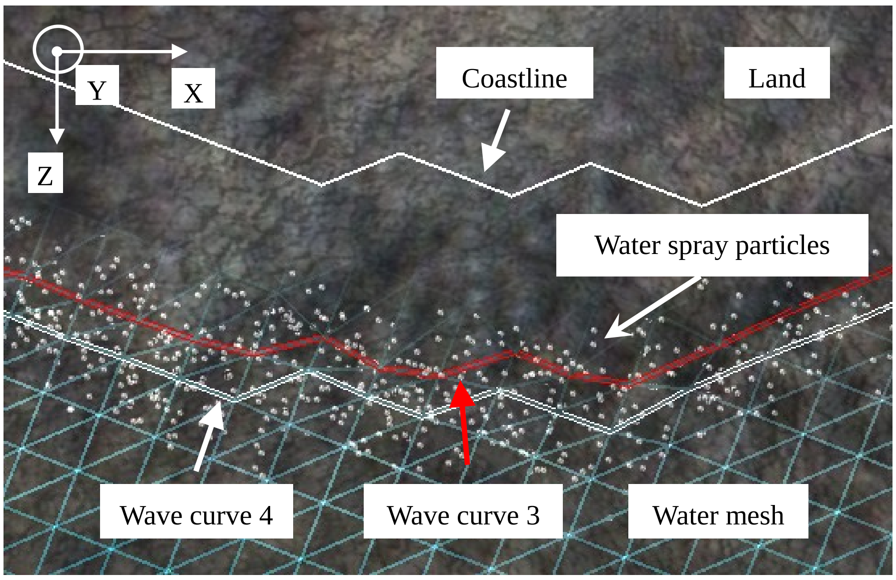


Figure 9.6: Wave curves rebounding from the land.

Figure 9.7 shows particles appearing on the crests of breaking waves.

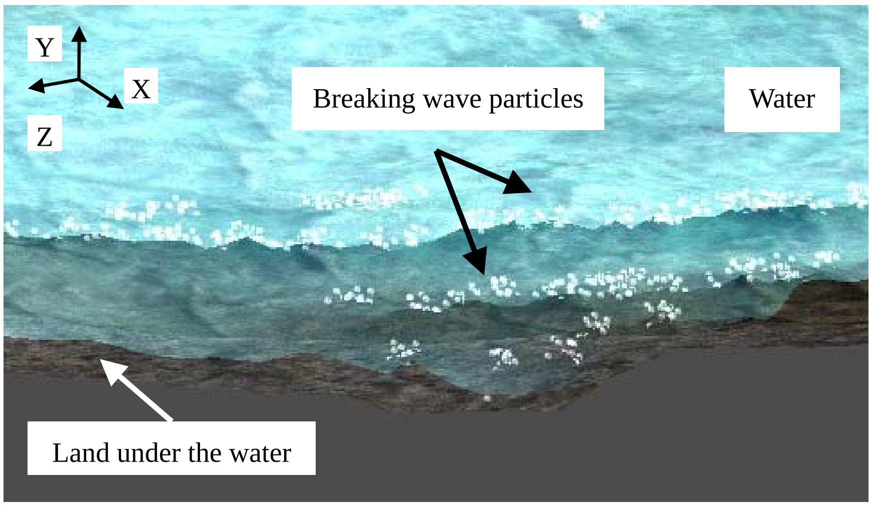


Figure 9.7: Breaking wave particles.

Figure 9.8 is a side view of Figure 9.7 showing particles covering the crests of the breaking waves, and also that the crests are darker than other parts of the waves.

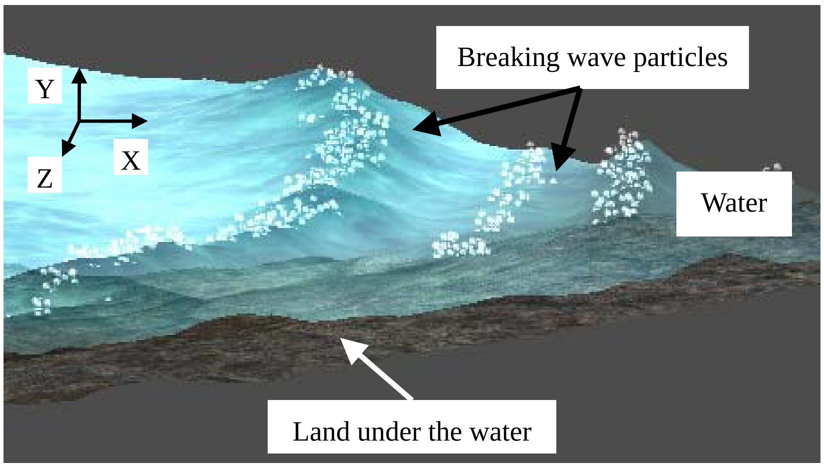


Figure 9.8: Breaking wave particles (side view).

REFERENCES

[1] Jerry Tessendorf, 1999, "Simulating Ocean Water", SIGGRAPH Course Notes, http://graphics.ucsd.edu/courses/rendering/2005/jdewall/tessendorf.pdf (last accessed Aug 26, 2010)

[2] A. Iglesias, November, 2004, "Computer Graphics for Water Modeling and Rendering: A Survey", *Future Generation Computer Systems*, http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V06-4CVX0RT-2&_user=267327&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000015658&_version=1&_urlVersion=0&_userid=267327&md5=34819df33b7809dc786b2110ddffd855 (last accessed Aug 26, 2010)

[3] Hugo Elias, "Perlin Noise", 1998, http://freespace.virgin.net/hugo.elias/models/m_perlin.htm (last accessed Aug 26, 2010)

[4] Darwyn R. Peachey, 1986, "Modeling Waves and Surf", *ACM SIGGRAPH Computer Graphics*, August, http://portal.acm.org/citation.cfm?id=15893&dl=ACM&coll=portal (last accessed Aug 26, 2010)

[5] Alain Fournier, William T. Reeves, 1986, "A Simple Model of Ocean Waves", *ACM SIGGRAPH Computer Graphics*, August, http://portal.acm.org/citation.cfm?id=15894 (last accessed Aug 26, 2010)

[6] Nick Foster, Ronald Fedkiw, 2001, "Practical Animation of Liquids", *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, http://physbam.stanford.edu/~fedkiw/papers/stanford2001-02.pdf (last accessed Aug 26, 2010)

[7] Douglas Enright, Stephem Marschner, Ronald Fedkiw, 2002, "Animation and Rendering of Complex Water Surfaces", *ACM Transaction on Graphics*, July, http://portal.acm.org/citation.cfm?id=566645 (last accessed Aug 26, 2010)

[8] Marcelo M. Maes, Tadahiro Fujimoto, Norishige Chiba, 2006, "Efficient Animation of

Water Flow on Irregular Terrains", *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, http://portal.acm.org/citation.cfm?id=1174447 (last accessed Aug 26, 2010)

[9] Stephen Manley, 1999, "OpenGL Fluid & Gel Modeling", http://www.nyx.net/~smanley/fluid/fluid.html (last accessed Aug 26, 2010)

[10] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, Mark Carlson, 2006, "Physically Based Deformable Models in Computer Graphics", *Computer Graphics Forum*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.4664&rep=rep1&type=pdf (last accessed Aug 26, 2010)

[11] David Halliday, Robert Resnick, Jearl Walker, 2005, *Fundamentals of Physics*, 7th Edition, Wiley.

[12] Dave Shreiner; Mason Woo; Jackie Neider; Tom Davis, 2007, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, 6th edition, Addison-Wesley Professional.

[13] Viorel Mihalef, Dimitris Metaxas, Mark Sussman, 2004, "Animation and control of breaking waves", *ACM SIGGRAPH/Eurographics symposium on Computer animation*, http://portal.acm.org/citation.cfm?id=1028523.1028565 (last accessed Oct 6, 2010)

[14] Julien Guertault, 2005, "Yet Another OpenGL Tutorial: Simple Water Rendering", http://zavie.free.fr/opengl/index.html.en (last accessed Aug 26, 2010)

[15] W.T.Reeves, 1983, "Particle Systems—a Technique for Modeling a Class of Fuzzy Objects", *ACM Transactions on Graphics*, Volume 2, Issue 2, pp. 91-108. April, http://portal.acm.org/citation.cfm?id=357320 (last accessed Aug 26, 2010)

[16] Edward Angel, 2005, *OpenGL A Primer*, Second Edition. Pearson Education.

[17] OpenGL Overview, 2010, http://www.opengl.org/about/overview/ (last accessed Aug 26, 2010)

[18] Dave Astle, 2005, *More OpenGL Game Programming*, 2th Edition, Course Technology PTR.

[19] JOGL document, v1.6.0, 2006, http://jogamp.org/jogl/www/ (last accessed Aug 26, 2010)

[20] Andrew Davison, 2007, *Pro Java 6 3D Game Development*, Apress, April.

[21] Claes Johanson, 2004, "Real-time Water Rendering Introducing the Projected Grid Concept", Master of Science Thesis, Lund University, March, http://habib.wikidot.com/projected-grid-ocean-shader-full-html-version (last accessed Aug 26, 2010)

[22] Ben Humphrey, 2001, "NeHe Productions OpenGL Lesson 34: Beautiful Landscapes by Means of Height Mapping", http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=34 (last accessed Aug 26, 2010)

[23] Kevin Hawkins, Dave Astle, 2002, *OpenGL Game Programming*, Course Technology PTR, May, http://glbook.gamedev.net/source.asp (last accessed Aug 26, 2010)

[24] Dave Astle, Kevin Hawkins, 2004, *Beginning OpenGL Game Programming*, Course Technology PTR, March, http://www.torrentreactor.net/torrents/1149913/Beginning-OpenGL-Game-Programming-Source-Code-rar (last accessed Aug 26, 2010)

[25] Keith A. Sverdrup, Alison B. Duxbury, Alyn C. Duxbury, 2006, *Fundamentals of Oceanography*, 5th edition, McGraw-Hill.

[26] Marcelo Alonso, Edward J. Finn, 1992, *Physics*, Addison-Wesley, June.

[27] Tzvetomir Vassilev, Bernhard Spanlang, 2002, "A Mass-Spring Model for Real Time Deformable Solids", East-West-Vision, September, http://www.cs.ucl.ac.uk/staff/b.spanlang/ (last accessed Aug 26, 2010)

[28] Trent Polack, 2002, *Focus On 3D Terrain Programming*, Course Technology PTR.

[29] Tobias Franke, 2001, "Terrain Texture Generation", http://www.flipcode.com/archives/Terrain_Texture_Generation.shtml (last accessed Aug 26, 2010)

[30] FRAPS, 2009, http://www.fraps.com/ (last accessed Sep 26, 2010)

[31] Emilio Lorca, Margot Recabarren, 2005, "Earthquakes and Tsunamis", ITIC, March, http://ioc3.unesco.org/itic/contents.php?id=155 (last accessed Aug 26, 2010)

APPENDICIES

Appendix A1: Simple.c

Display a white polygon using OpenGL in C.

Appendix A2: Simple.java

Display a white polygon using JOGL.

Appendix A3: Simple2.java, draw.vert, draw.frag

Display a polygon using JOGL, a vertex shader to scale it, and fragment shader to change it green.

Appendix A4: noise.c

Perlin noise for adding waves to a water surface.

Appendix A5: Loadpixels.java

Load a map, and use it’s blue pixels as height values.

Appendix A6: Mesh.java

Render land and water surface.

Appendix A7: Phase.java

Create a phase function for the heights of water vertices.

Appendix A8: WaveCurves.java

Calculate the coastline and wave curves.

Appendix A9: Velocities.java

Calculate the velocities for wave curves employing spring system.

Appendix A10: Collision.java

Process water/land collisions, and the collisions between waves.

Appendix A11: HeightOfWaveCurve.java

Adjust the height of wave curves 1-4.

Appendix A12: water.vert, water.frag

Vertex and fragment shaders for the water surface, which apply multitexturing, crest shading, and transparency.

Appendix A13: Particle.java

Water spray and breaking wave particle systems.

Appendix A14: par.vert, par.frag

Vertex and fragment shaders for the particle systems.

Appendix B: Published Paper

Rendering Water and Land Interaction Using a Spring System

```
/* Simple.c 2010/6/28
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Display a white polygon using OpenGL in C.
 * See section 2.1 for details.
 */

#include <GL/glut.h>
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT); //clears color buffer
   glColor3f(1,1,1); //set the color of the object
   glBegin(GL_POLYGON); //specifies the object type will be draw
     //set the vertices location of the polygon
     glVertex2f(0, 0);
     glVertex2f(0.5, 0);
     glVertex2f(0.5, 0.5);
     glVertex2f(0, 0.5);
   glEnd(); //end to specifies the vertices
   glFlush(); //forces to output the result immediately
}

int main(int argc,char** argv)
{
   glutInit(&argc,argv); //initializes GLUT
   glutCreateWindow("simple"); //create a window called simple
   glutDisplayFunc(display); //call the method display()


      glutMainLoop(); //enter an event-processing loop
}
```

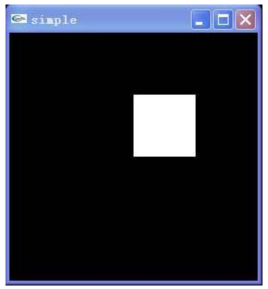


Figure A1.1: Output from OpenGL called from C.

Appendix A2

```
/* Simple.java 2010/6/28
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Display a white polygon using JOGL.
 * See section 2.2 for details.
 */

import javax.swing.*;
import javax.media.opengl.*;

public class Simple extends JFrame implements GLEventListener
{
   private GLCapabilities caps;
   private GLCanvas canvas;

   public Simple()
   {
      super("Simple"); //set the JFrame title
      //create GLcanvas
      caps = new GLCapabilities();
      canvas = new GLCanvas(caps);
      //open event listener
      canvas.addGLEventListener(this);
      getContentPane().add(canvas); //add canvas content into window
   }

   public void createwindow()
   {
      setSize(256, 256); //set window size
      //kill the process when the JFrame is closed
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);  //display result in window
   }

   public static void main(String[] args) //main function
   {
      //Create our Simple and run the method createwindow()
      new Simple().createwindow ();  }

   public void polygon(GL gl)
   {
      gl.glClear(GL.GL_COLOR_BUFFER_BIT); //clears color buffer
      gl.glColor3f(1,1,1); //set the color of the object
      //specifies the object type will be draw
      gl.glBegin(GL.GL_POLYGON);
         //set the vertices location of the polygon
         gl.glVertex2f(0,0);
         gl.glVertex2f(0.5f, 0);
         gl.glVertex2f(0.5f, 0.5f);
         gl.glVertex2f(0, 0.5f);
      gl.glEnd(); //end to specifies the vertices
      gl.glFlush(); //forces to output the result immediately
   }

   public void display(GLAutoDrawable drawable)
   {
      GL gl = drawable.getGL();
      polygon(gl); //Draw the model
   }

   //I do not need these methods
   public void init(GLAutoDrawable drawable) {}
   public void reshape(GLAutoDrawable drawable, int x, int y, int w,
                       int h) {}
   public void displayChanged(GLAutoDrawable drawable,
                              boolean modeChanged,
                              boolean deviceChanged) {}
}
```

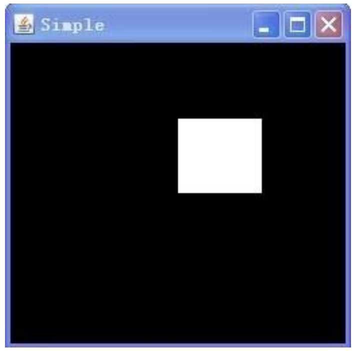


Figure A2.1: Output from OpenGL called from Java.

```
/* Simple2.java 2010/7/5
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Display a polygon using JOGL, a vertex shader to scale
 * it, and a fragment shader to change it green.
 * See section 2.3.3 for details
 */

import javax.swing.*;
import javax.media.opengl.*;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.nio.*;
import com.sun.opengl.util.*;

public class Simple2 extends JFrame implements GLEventListener
{
   private GLCapabilities caps;
   private GLCanvas canvas;
   private int drawVertex, drawFragment, drawShaderProgram;

   public Simple2()
   {
      super("Simple2"); //set the JFrame title
      //create GLcanvas
      caps = new GLCapabilities();
      canvas = new GLCanvas(caps);
      //open event listener
      canvas.addGLEventListener(this);
      getContentPane().add(canvas); //add canvas content into window
   }

   public void createwindow()
   {
      setSize(256, 256); //set window size
      //kill the process when the JFrame is closed
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);  //display result in window
   }

   public static void main(String[] args) //main function
   {
      //Create our Simple2 and run the method createwindow()
      new Simple2().createwindow ();  }

   public void init(GLAutoDrawable drawable)
   {
     GL gl = drawable.getGL();
      buildShader(gl);
   }

private void buildShader(GL gl)
{
   //build vertex shader
   drawVertex = gl.glCreateShader(GL.GL_VERTEX_SHADER);
   //build fragment shader
   drawFragment = gl.glCreateShader(GL.GL_FRAGMENT_SHADER);
   //build shader program
   drawShaderProgram = gl.glCreateProgram();
   try { drawShader(gl); } catch (IOException e) { }
}

private void drawShader(GL gl)throws IOException
{
   //load vertex shader source code
   BufferedReader brv =
              new BufferedReader(new FileReader("draw.vert"));
   String vsrc = "";
   String lineV;
   while ((lineV = brv.readLine()) != null)
   { vsrc += lineV + "\n"; }
   String Vsrc [] = new String [1];
   Vsrc [0] = vsrc;
   //replaces source code in the vertex shader
   gl.glShaderSource(drawVertex, 1, Vsrc, null);
   //compile vertex shader
   gl.glCompileShader(drawVertex);
   IntBuffer vertBuffer = BufferUtil.newIntBuffer(1);
   gl.glGetShaderiv(drawVertex, GL.GL_COMPILE_STATUS,
                    vertBuffer);

   //load fragment shader source code
   BufferedReader brf =
              new BufferedReader(new FileReader("draw.frag"));
   String fsrc = "";
   String line;
   while ((line = brf.readLine()) != null)
   { fsrc += line + "\n"; }
   String Fsrc [] = new String [1];
   Fsrc [0] = fsrc;
   //replaces source code in the fragment shader
   gl.glShaderSource(drawFragment, 1, Fsrc, null);
   //compile fragment shader
   gl.glCompileShader(drawFragment);
   IntBuffer fragBuffer = BufferUtil.newIntBuffer(1);
   gl.glGetShaderiv(drawFragment, GL.GL_COMPILE_STATUS,
                   fragBuffer);

   //attach vertex shader to the shader program
   gl.glAttachShader(drawShaderProgram, drawVertex);
   //attach fragment shader to the shader program
   gl.glAttachShader(drawShaderProgram, drawFragment);
   //links the program object specified by program
```

```
        gl.glLinkProgram(drawShaderProgram);
    }

    public void polygon(GL gl)
    {
        gl.glClear(GL.GL_COLOR_BUFFER_BIT); //clears color buffer
        gl.glColor3f(1,1,1); //set the color of the object
        gl.glUseProgram(drawShaderProgram); //use shaders
        //specifies the object type will be draw
        gl.glBegin(GL.GL_POLYGON);
            //set the vertices location of the polygon
            gl.glVertex2f(0,0);
            gl.glVertex2f(0.5f, 0);
            gl.glVertex2f(0.5f, 0.5f);
            gl.glVertex2f(0, 0.5f);
        gl.glEnd(); //end to specifies the vertices
        gl.glUseProgram(0); //end to use shaders
        gl.glFlush(); //forces to output the result immediately
    }

    public void display(GLAutoDrawable drawable)
    {
        GL gl = drawable.getGL();
        polygon(gl); //Draw the model
    }

    //I do not need these methods
    public void reshape(GLAutoDrawable drawable, int x, int y, int w,
                        int h) {}
    public void displayChanged(GLAutoDrawable drawable,
                               boolean modeChanged,
                               boolean deviceChanged) {}
}
```

Vertex Shader

```
/* draw.vert 2010/7/5
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn

 * Vertex shader for simple2.java
 * Scale polygon by multiply 0.5 in x, y, z coordinate
 * See section 2.3.2 for details
 */

void main(void)
{
    vec4 a = gl_Vertex; //get vertex position
    //scale vertex in x, y, z coordinates
    a.x = a.x * 0.5;
    a.y = a.y * 0.5;
    a.z = a.z * 0.5;
    //calculate the new vertex position
    gl_Position = gl_ModelViewProjectionMatrix * a;
}
```

Fragment Shader

```
/* draw.frag 2010/7/5
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn

 * Fragment shader for simple2.java
 * change the polygon color to green
 * See section 2.3.2 for details
 */

void main (void)
{
    //define the color to green
    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

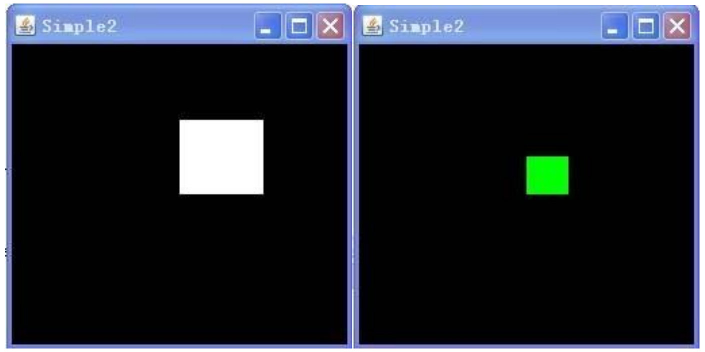


Figure A3.1: Output (left: without shaders; right: with shaders) from simple2.java.

```
/*==============================================================
**
** Perlin noise
** Copyright (C) 2005  Julien Guertault
**
** This program is free software; you can redistribute it and/or
** modify it under the terms of the GNU General Public License
** as published by the Free Software Foundation; either version 2
** of the License, or (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
** USA.
**
** See section 2.5.1 for details
**
** ============================================================ */

/*
** Improved Perlin noise.
** Original Perlin noise implementation can be found at :
**  http://mrl.nyu.edu/~perlin/doc/oscar.html#noise
*/

#include <stdlib.h>

#define MOD 0xff
Static int permut[256];
static const char gradient[32][4] =
{
   { 1,  1,  1,  0},{ 1,  1,  0,  1},{ 1,  0,  1,  1},{ 0,  1,  1,  1},
   { 1,  1, -1,  0},{ 1,  1,  0, -1},{ 1,  0,  1, -1},{ 0,  1,  1, -1},
   { 1, -1,  1,  0},{ 1, -1,  0,  1},{ 1,  0, -1,  1},{ 0,  1, -1,  1},
   { 1, -1, -1,  0},{ 1, -1,  0, -1},{ 1,  0, -1, -1},{ 0,  1, -1, -1},
   {-1,  1,  1,  0},{-1,  1,  0,  1},{-1,  0,  1,  1},{ 0, -1,  1,  1},
   {-1,  1, -1,  0},{-1,  1,  0, -1},{-1,  0,  1, -1},{ 0, -1,  1, -1},
   {-1, -1,  1,  0},{-1, -1,  0,  1},{-1,  0, -1,  1},{ 0, -1, -1,  1},
   {-1, -1, -1,  0},{-1, -1,  0, -1},{-1,  0, -1, -1},{ 0, -1, -1, -1},
};

Void InitNoise (void)
{
   unsigned int i = 0;
   while (i < 256)
      permut[i++] = rand () & MOD;
}

/*
** Function finding out the gradient corresponding to the coordinates */
static int Indice(const int i, const int j, const int k,
                 const int l)
{
   return (permut[(l + permut[(k + permut[(j + permut[i & MOD])
           & MOD]) & MOD]) & MOD] & 0x1f);
}

/*
** Functions computing the dot product of the vector and the gradient */
static float Prod (const float a, const char b)
{
   if (b > 0) return a;
   if (b < 0) return -a;
   return 0;
}

static float Dot_prod (const float x1, const char x2,
                       const float y1, const char y2,
                       const float z1, const char z2,
                       const float t1, const char t2)
{
   return (Prod (x1, x2) + Prod (y1, y2) + Prod (z1, z2)
           + Prod (t1, t2));
}

/* Functions computing interpolations */
static float Spline5 (const float state)
{
   /*
   ** Enhanced spline :
   ** (3x^2 + 2x^3) is not as good as (6x^5 - 15x^4 + 10x^3)
   */
   const float sqr = state * state;
   return state * sqr * (6 * sqr - 15 * state + 10);
}

static float Linear (const float start, const float end,
                     const float state)
{ return start + (end - start) * state; }

/* Noise function, returning the Perlin Noise at a given point */
Float Noise (const float x, const float y, const float z,
             const float t)
{
   /* The unit hypercube containing the point */
   const int x1 = (int) (x > 0 ? x : x - 1);
   const int y1 = (int) (y > 0 ? y : y - 1);
   const int z1 = (int) (z > 0 ? z : z - 1);
   const int t1 = (int) (t > 0 ? t : t - 1);
   const int x2 = x1 + 1;
   const int y2 = y1 + 1;
const int z2 = z1 + 1;
const int t2 = t1 + 1;

/* The 16 corresponding gradients */
const char * g0000 = gradient[Indice (x1, y1, z1, t1)];
const char * g0001 = gradient[Indice (x1, y1, z1, t2)];
const char * g0010 = gradient[Indice (x1, y1, z2, t1)];
const char * g0011 = gradient[Indice (x1, y1, z2, t2)];
const char * g0100 = gradient[Indice (x1, y2, z1, t1)];
const char * g0101 = gradient[Indice (x1, y2, z1, t2)];
const char * g0110 = gradient[Indice (x1, y2, z2, t1)];
const char * g0111 = gradient[Indice (x1, y2, z2, t2)];
const char * g1000 = gradient[Indice (x2, y1, z1, t1)];
const char * g1001 = gradient[Indice (x2, y1, z1, t2)];
const char * g1010 = gradient[Indice (x2, y1, z2, t1)];
const char * g1011 = gradient[Indice (x2, y1, z2, t2)];
const char * g1100 = gradient[Indice (x2, y2, z1, t1)];
const char * g1101 = gradient[Indice (x2, y2, z1, t2)];
const char * g1110 = gradient[Indice (x2, y2, z2, t1)];
const char * g1111 = gradient[Indice (x2, y2, z2, t2)];

/* The 16 vectors */
const float dx1 = x - x1;
const float dx2 = x - x2;
const float dy1 = y - y1;
const float dy2 = y - y2;
const float dz1 = z - z1;
const float dz2 = z - z2;
const float dt1 = t - t1;
const float dt2 = t - t2;

/* The 16 dot products */
const float b0000 = Dot_prod(dx1, g0000[0], dy1, g0000[1],
                             dz1, g0000[2], dt1, g0000[3]);
const float b0001 = Dot_prod(dx1, g0001[0], dy1, g0001[1],
                             dz1, g0001[2], dt2, g0001[3]);
const float b0010 = Dot_prod(dx1, g0010[0], dy1, g0010[1],
                             dz2, g0010[2], dt1, g0010[3]);
const float b0011 = Dot_prod(dx1, g0011[0], dy1, g0011[1],
                             dz2, g0011[2], dt2, g0011[3]);
const float b0100 = Dot_prod(dx1, g0100[0], dy2, g0100[1],
                             dz1, g0100[2], dt1, g0100[3]);
const float b0101 = Dot_prod(dx1, g0101[0], dy2, g0101[1],
                             dz1, g0101[2], dt2, g0101[3]);
const float b0110 = Dot_prod(dx1, g0110[0], dy2, g0110[1],
                             dz2, g0110[2], dt1, g0110[3]);
const float b0111 = Dot_prod(dx1, g0111[0], dy2, g0111[1],
                             dz2, g0111[2], dt2, g0111[3]);
const float b1000 = Dot_prod(dx2, g1000[0], dy1, g1000[1],
                             dz1, g1000[2], dt1, g1000[3]);
const float b1001 = Dot_prod(dx2, g1001[0], dy1, g1001[1],
                             dz1, g1001[2], dt2, g1001[3]);
const float b1010 = Dot_prod(dx2, g1010[0], dy1, g1010[1],
                             dz2, g1010[2], dt1, g1010[3]);
const float b1011 = Dot_prod(dx2, g1011[0], dy1, g1011[1],
                             dz2, g1011[2], dt2, g1011[3]);
const float b1100 = Dot_prod(dx2, g1100[0], dy2, g1100[1],
                             dz1, g1100[2], dt1, g1100[3]);
const float b1101 = Dot_prod(dx2, g1101[0], dy2, g1101[1],
                             dz1, g1101[2], dt2, g1101[3]);
const float b1110 = Dot_prod(dx2, g1110[0], dy2, g1110[1],
                             dz2, g1110[2], dt1, g1110[3]);
const float b1111 = Dot_prod(dx2, g1111[0], dy2, g1111[1],
                             dz2, g1111[2], dt2, g1111[3]);

/* Then the interpolations, down to the result */
const float idx1 = Spline5 (dx1);
const float idy1 = Spline5 (dy1);
const float idz1 = Spline5 (dz1);
const float idt1 = Spline5 (dt1);

const float b111 = Linear (b1110, b1111, idt1);
const float b110 = Linear (b1100, b1101, idt1);
const float b101 = Linear (b1010, b1011, idt1);
const float b100 = Linear (b1000, b1001, idt1);
const float b011 = Linear (b0110, b0111, idt1);
const float b010 = Linear (b0100, b0101, idt1);
const float b001 = Linear (b0010, b0011, idt1);
const float b000 = Linear (b0000, b0001, idt1);

const float b11 = Linear (b110, b111, idz1);
const float b10 = Linear (b100, b101, idz1);
const float b01 = Linear (b010, b011, idz1);
const float b00 = Linear (b000, b001, idz1);

const float b1 = Linear (b10, b11, idy1);
const float b0 = Linear (b00, b01, idy1);

return Linear (b0, b1, idx1);
}
```

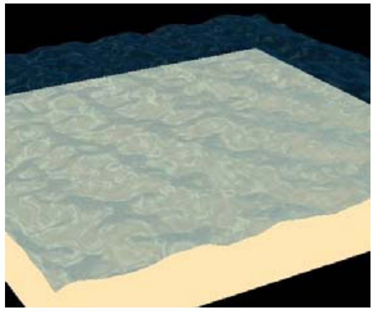


Figure A5.1: Guertault’s water surface.

Appendix A5

```
/* Loadpixels.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Load a map, and use it’s blue pixels as height values
 * See section 3.1 for details.
 */

package demos;

import java.awt.*;
import java.net.*;
import java.awt.event.*;
import java.awt.image.*;
import javax.swing.*;

public class Loadpixels extends JFrame
{
   private static int imageWidth, imageHeight, imageSize;
   private static int[] pixels;
   static int[] heightvalue;
   private static Image bufferImage[] = new Image[1];

   public void Loadpixels(String filename)
   {
      //call getImage method to load a image into image buffer
      bufferImage[0] = getImage(filename);
      imageWidth = bufferImage[0].getWidth(this); //get image width
      //get image height
      imageHeight = bufferImage[0].getHeight(this);
      imageSize = imageWidth* imageHeight; //compute image size

      //initialize pixels array of the image
      pixels=new int[imageSize];
      //initialize height value array
      heightvalue=new int[imageSize];
      //get pixels from image buffer and store into pixels array
      getImagePixels(bufferImage[0], pixels);
      //copy the blue value of the pixels array to height value array
      for(int i=0; i<imageSize; i++)
      {
         heightvalue[i]=pixels[i] & 0xff;
      }
   }

   private Image getImage(String filename)
   {
      //new a URLClassLoader method
      URLClassLoader urlLoader =
               (URLClassLoader)this.getClass().getClassLoader();
      URL url = null; //clear url
      Image image = null; //clear image
      //appoint a image's url by use its name
      url = urlLoader.findResource(filename);
      //load the image from url
      image = Toolkit.getDefaultToolkit().getImage(url);
      //new a MediaTracker
      MediaTracker mediatracker = new MediaTracker(this);
      try
      {
          //add a image into mediatracker's list
          mediatracker.addImage(image, 0);
          //start to load image and wait until finish
          mediatracker.waitForID(0);
      }
      //if it have exception catch the exception and let image return //null
      catch (InterruptedException _ex)
      { image = null; }
      //if the image id is error let image return null too
      if (mediatracker.isErrorID(0))
      { image = null; }
      return image;
   }

   private boolean getImagePixels(Image image, int pixels[])
   {
      //new a PixelGrabber method to get image pixels
      PixelGrabber pixelgrabber =
             new PixelGrabber(image, 0, 0, imageWidth, imageHeight,
                                  pixels, 0, imageWidth);
      //try to get image pixels
      try
      { pixelgrabber.grabPixels(); }
      //if it have exception catch the exception and return false
      catch (InterruptedException _ex)
      { return false; }
      return true;
   }
}
```

Appendix A6

```
/* Mesh.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Render land and water
 * surface.
 * See sections 3.1, 3.2, 6.1.2 for details.
 */
package demos;

import javax.swing.*;
import javax.media.opengl.*;
import javax.media.opengl.glu.*;
import java.awt.event.*;
import com.sun.opengl.util.*;
import java.awt.*;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.nio.*;
import java.io.BufferedReader;
import java.io.FileReader;


public class Mesh extends JFrame implements GLEventListener, KeyListener,
MouseMotionListener, MouseListener
{
   private FPSAnimator animator;
   private GLU glu;
   private GLUT glut;
   private GLCapabilities caps;
   private GLCanvas canvas;

   //land mesh
   private int landsize = 128; //land mesh size
   //land vertex coordinate
   private float landvalue[][][] = new float[landsize][landsize][3];
   //land vertex array
   private float landArray[] = new float [landsize*landsize*3];
```

```
//land vertex buffer
private FloatBuffer landBuffer
             = BufferUtil.newFloatBuffer(landsize*landsize*3);
//land color array
private float landCArray[] = new float [landsize*landsize*3];
//land color buffer
private FloatBuffer landCBuffer
             = BufferUtil.newFloatBuffer(landsize*landsize*3);
//land combine texture
private float combineTArray[] = new float [landsize*landsize*2];
private FloatBuffer combineTBuffer
             = BufferUtil.newFloatBuffer(landsize*landsize*2);
//land detail texture
private float detailTArray[] = new float [landsize*landsize*2];
private FloatBuffer detailTBuffer
             = BufferUtil.newFloatBuffer(landsize*landsize*2);
//land index
private int landIndex [] = new int [landsize*landsize*3*2];
private IntBuffer landIndexBuffer
             = BufferUtil.newIntBuffer(landsize*landsize*3*2);

//water mesh
private int watersize = 128; //water mesh size
//water vertex coordinate
private float watervalue[][][]
                          = new float[watersize][watersize][3];
//this is the copy of watervalue[][][] which will not move in X-Z
//plane, and it will not control by height function.
private float watervalue2[][][]
                         = new float[watersize][watersize][3];
//water vertex array
private float waterArray[] = new float [watersize*watersize*3];
//water vertex buffer
private FloatBuffer waterBuffer
           = BufferUtil.newFloatBuffer(watersize*watersize*3);
//water color
private float waterCArray[] = new float [watersize*watersize*3];
private FloatBuffer waterCBuffer
           = BufferUtil.newFloatBuffer(watersize*watersize*3);
//water combine texture
private float materialTArray[]
                          = new float [watersize*watersize*2];
```

```
private FloatBuffer materialTBuffer
           = BufferUtil.newFloatBuffer(watersize*watersize*2);
//water detail texture
private float waterTArray[] = new float [watersize*watersize*2];
private FloatBuffer waterTBuffer
           = BufferUtil.newFloatBuffer(watersize*watersize*2);
//water index
private int waterIndex [] = new int [watersize*watersize*3*2];
private IntBuffer waterIndexBuffer
           = BufferUtil.newIntBuffer(watersize*watersize*3*2);
//height of bottom, equal to the height of land at that vertex
private  float  bottomvalue[][]  =  new  float[watersize][watersize];
//height of bottom, equal to the height of land at that vertex
WaterData waterData = new WaterData();
//for water shaders
private int waterVertex, waterFragment, waterShaderProgram,
            texWParam1, texWParam2;

//for the camera, move and rotate the model
private float width, length, xdirection, ydirection,
              zdirection=20, rotatex, rotatey, rotatez, newx,
              newy, newx2, newy2;
private boolean button1 = false, button2 = false; //botton of mouse
private boolean fill=true, land=true, water=true, mesh=false,
   move=true, blend=true, wavecurves=false; //key of keyboard

private float t, tt = 0, T1=8, T2=4, T3=2, T4=1,
       profileHeight = 12, waterLevel = 19;
//coastline
private List< Points > points1 = new ArrayList< Points >();
//wave curve 1
private List< Points > points2 = new ArrayList< Points >();
//wave curve 2
private List< Points > points3 = new ArrayList< Points >();
//wave curve 3
private List< Points > points4 = new ArrayList< Points >();
//wave curve 4
private List< Points > points5 = new ArrayList< Points >();

private Velocities velocities = new Velocities();
//for wave curve 1
private List< Vel > velocity2 = new ArrayList< Vel >();
```

```
private List< Points > inNumbers1For2 =new ArrayList< Points >();
private List< Points > outNumbers4For2 =new ArrayList< Points >();
private List< FloatNum > originDistancew4For2
                                 = new ArrayList< FloatNum>();
private List<FloatNum> verticalVel2 = new ArrayList< FloatNum>();
//for wave curve 2
private List< Vel > velocity3 = new ArrayList< Vel >();
private List< Points > inNumbers1For3 =new ArrayList< Points >();
private List< Points > outNumbers4For3 =new ArrayList< Points >();
private List< FloatNum > originDistancew4For3
                                 = new ArrayList< FloatNum>();
private List<FloatNum> verticalVel3 = new ArrayList< FloatNum>();

//for wave curve 3
private List< Vel > velocity4 = new ArrayList< Vel >();
private List< Vel > originalVelocity4 = new ArrayList< Vel >();
private List< Points > positiveDirection4
                                  = new ArrayList< Points >();
private List< Points > outNumbers5For4 = new ArrayList< Points >();
private List< Points > inNumbers1For4 = new ArrayList< Points>();
private List< FloatNum > SlopeFor4 = new ArrayList< FloatNum>();

//for wave curve 4
private List< Vel > velocity5 = new ArrayList< Vel >();
private List< Vel > originalVelocity5 = new ArrayList< Vel >();
private List< Points > positiveDirection5
                                   = new ArrayList< Points >();
private List< Points > inNumbers1For5 = new ArrayList< Points>();
private List< Points > inNumbers4For5 = new ArrayList< Points>();
private List< FloatNum > SlopeFor5 = new ArrayList< FloatNum>();

//for textures
private int landTexture, detailTexture, waterDetailTexture,
           materialTexture, parTex, texParam;
//particle shaders
int parVertex, parFragment, parShaderProgram;

Particle particle = new Particle();
//water spray particle
private int waterSprayParSize = 25;
private List <Vel3D> parvalue3 = new ArrayList< Vel3D >();
private List <FloatNum> parLife3 = new ArrayList< FloatNum >();
private List <Vel3D> parVel3 = new ArrayList< Vel3D >();

private List <Vel3D> parvalue4 = new ArrayList< Vel3D >();
private List <FloatNum> parLife4 = new ArrayList< FloatNum >();
private List <Vel3D> parVel4 = new ArrayList< Vel3D >();

//breaking wave particle
private int breakingWaveParSize = 15;
private float breakingWaveParValue[][][][]
                = new float [128][128][breakingWaveParSize][3];
private List<Vel3D> breakingWaveParVel = new ArrayList <Vel3D> ();
private List<FloatNum> breakingWaveParLife
                                = new ArrayList <FloatNum> ();

private Mesh()
{
   super("mesh"); //set the JFrame title
   //create GLcanvas
   caps = new GLCapabilities();
   canvas = new GLCanvas(caps);
   //open event, keyboard, mouse and mouse motion listener
   canvas.addGLEventListener(this);
   canvas.addKeyListener(this);
   canvas.addMouseListener(this);
   canvas.addMouseMotionListener(this);
   getContentPane().add(canvas); //add canvas content into window
   //set fps of animator
   animator = new FPSAnimator(canvas, 999, true);
   animator.start(); //start animator
}


private void createWindow()
{
   setSize(800, 600); //set window size
   //set window in the middle of our desktop
   setLocationRelativeTo(null);
   //kill the process when the JFrame is closed
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setVisible(true);  //display result in window
   canvas.requestFocusInWindow(); //request window
}
```

```
public static void main(String[] args) //main function
{
   //Create mesh and run the method createwindow()
   new Mesh().createWindow ();
}


public void init(GLAutoDrawable drawable)
{
   GL gl = drawable.getGL();
   glu = new GLU();
   int textureSize = 512; //land and water texture map size

   //for load pixels, land textures, water textures
   Loadpixels loadpixels =new Loadpixels();
   LoadLandTexture loadLandTexture = new LoadLandTexture();
   LoadWaterTexture loadWaterTexture = new LoadWaterTexture();

   buildLandData(); //build data for land mesh and get wave curves
   buildWaterData(gl); //build data for water mesh

   gl.glShadeModel(GL.GL_SMOOTH); //set model to smooth
   gl.glEnable(GL.GL_DEPTH_TEST); //Enables Depth Testing
   // The Type Of Depth Testing To Do
   gl.glDepthFunc(GL.GL_LESS);

   //create land textures
   landTexture = genTexture(gl);
   detailTexture = genTexture(gl);
   ByteBuffer landTextureBuffer
     = BufferUtil.newByteBuffer(textureSize * textureSize * 3);
   ByteBuffer landDetailTextureBuffer
      = BufferUtil.newByteBuffer(textureSize * textureSize * 3);
   loadLandTexture.LoadLandTexture(gl, glu, landTexture,
                               detailTexture, landTextureBuffer,
                               landDetailTextureBuffer);

   //create water textures
   waterDetailTexture = genTexture(gl);
   materialTexture = genTexture(gl);
```

```
        ByteBuffer waterTextureBuffer
           = BufferUtil.newByteBuffer(textureSize * textureSize * 3);
        ByteBuffer materialTextureBuffer
           = BufferUtil.newByteBuffer(textureSize * textureSize * 3);
        loadWaterTexture.LoadWaterTexture(gl, glu,
           waterDetailTexture, materialTexture, waterTextureBuffer,
           materialTextureBuffer);

        //build data for particles
        buildParShader(gl);
        parTex = genTexture(gl);
        ByteBuffer waterSprayParTexBuffer
                           = BufferUtil.newByteBuffer(32 * 32 * 3);
        particle.buildBreakingWaveParData(breakingWaveParValue,
                breakingWaveParSize, landsize, watersize, watervalue,
                breakingWaveParVel, breakingWaveParLife);
        particle.buildWaterSprayParData(gl, glu, waterSprayParSize,
                points4, parvalue3, watervalue, parLife3, parVel3,
                parTex, waterSprayParTexBuffer);
        particle.buildWaterSprayParData(gl, glu, waterSprayParSize,
                points5, parvalue4, watervalue, parLife4, parVel4,
                parTex, waterSprayParTexBuffer);
     }

  /****************************Land*****************************/
     //build data for land and get wave curves
     private void buildLandData()
     {
        //Load a map and read its pixels
        Loadpixels loadpixels =new Loadpixels();
        loadpixels.Loadpixels("data/1285.jpg");
        //Set land vertices array
        for (int z = 0; z < landsize; z++)
        {
           for (int x = 0; x < landsize; x++)
           {
              landvalue[x][z][0] = x-(landsize>>1);
              landvalue[x][z][1]
                    = loadpixels.heightvalue[(z*landsize + x)]/8f;
              if(landvalue[x][z][1] == waterLevel+1)
                 //this number mutiply 8 must be an integer
                 landvalue[x][z][1]=19.875f;
              landvalue[x][z][2] = -z+(landsize>>1);
           }
        }

        //get wave curves
        new getWaveCurves().getWaveCurves(points1, points2, points3,
                points4, points5, landsize, landvalue, waterLevel+1);

        //put land data into array buffer for draw the land
        LandData landData = new LandData();
        //land vertex
        landData.triangleSArray(landsize, landsize, landvalue,
                               landArray, landBuffer);
        //land vertex index
        landData.getTriangleSIndex(landsize, landsize, landIndex,
                                 landIndexBuffer);
        //land color and textures
        landData.getColorAndTexture(landsize, landsize, landvalue,
              landCArray, combineTArray, detailTArray, landCBuffer,
              combineTBuffer, detailTBuffer);
     }


     //draw land surface
     private void buildLandPolygon(GL gl)
     {
        //define combine texture in texture unit 0
        gl.glActiveTexture(GL.GL_TEXTURE0);
        gl.glEnable(GL.GL_TEXTURE_2D); //enable 2D texture
        //bind texture
        gl.glBindTexture(GL.GL_TEXTURE_2D, landTexture);
        //set multitexture mode
        gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_TEXTURE_ENV_MODE,
                    GL.GL_COMBINE);
        gl.glTexEnvi(GL.GL_TEXTURE_ENV, GL.GL_COMBINE_RGB,
                    GL.GL_ADD_SIGNED);
        //define detail texture in texture unit 1
        gl.glActiveTexture (GL.GL_TEXTURE1);
        gl.glEnable(GL.GL_TEXTURE_2D);//enable 2D texture
        //bind texture
        gl.glBindTexture(GL.GL_TEXTURE_2D, detailTexture);
        //vertex array
        gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
        gl.glVertexPointer(3, GL.GL_FLOAT, 0, landBuffer);
        //color array
        gl.glEnableClientState(GL.GL_COLOR_ARRAY);
        gl.glColorPointer(3, GL.GL_FLOAT, 0, landCBuffer);
        //combine texture array
        gl.glClientActiveTexture(GL.GL_TEXTURE0);
        gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
        gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, combineTBuffer);
        //detail texture array
        gl.glClientActiveTexture (GL.GL_TEXTURE1);
        gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
        gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, detailTBuffer);

        //draw array
        for(int z=0; z<landsize-1; z++)
        {
           //draw land surface with triangle strip
           gl.glDrawElements(GL.GL_TRIANGLE_STRIP, landsize*2,
                             GL.GL_UNSIGNED_INT, landIndexBuffer);
           landIndexBuffer.position(z*landsize*2); //to next vertex
        }
        landIndexBuffer.rewind(); //rewind land index buffer

        //disable all
        gl.glDisableClientState(GL.GL_VERTEX_ARRAY);
        gl.glDisableClientState(GL.GL_COLOR_ARRAY);
        gl.glClientActiveTexture(GL.GL_TEXTURE0);
        gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
        gl.glClientActiveTexture(GL.GL_TEXTURE1);
        gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
        gl.glActiveTexture(GL.GL_TEXTURE0);
        gl.glDisable(GL.GL_TEXTURE_2D);
        gl.glActiveTexture(GL.GL_TEXTURE1);
        gl.glDisable(GL.GL_TEXTURE_2D);
        gl.glActiveTexture(GL.GL_TEXTURE0);
     }

/******************************Water**************************/
     //build data for water
     private void buildWaterData(GL gl)
     {
        //Set water vertices array
        for (int z = 0; z < watersize; z++)
        {
           for (int x = 0; x < watersize; x++)
           {
              watervalue[x][z][0] = x-(watersize>>1);
              watervalue2[x][z][0]=watervalue[x][z][0];
              watervalue[x][z][2] = -z+(watersize>>1);
              watervalue2[x][z][2]=watervalue[x][z][2];
              watervalue2[x][z][1]=19;
              bottomvalue[x][z] = 0; //build bottomvalue
           }
        }

        //put water data to array buffer for draw the water
        //water vertex index
        waterData.getTriangleSIndex(watersize, watersize,
                                    waterIndex, waterIndexBuffer);
        //water color and textures
        waterData.getColorAndTexture(watersize, watersize,
                  watervalue, landvalue, bottomvalue, waterCArray,
                  materialTArray, waterTArray, waterCBuffer,
                  materialTBuffer, waterTBuffer);

        //input bottomvalue by copy land height
        for(int z=(watersize-landsize)>>1;
            z<(watersize+landsize)>>1; z++)
        {
           for(int x=(watersize-landsize)>>1;
               x<(watersize+landsize)>>1; x++)
           {
              bottomvalue[x][z] = landvalue
                                  [x-((watersize-landsize)>>1)]
                                  [z-((watersize-landsize)>>1)][1];
           }
        }

        //get the nearest vertex of coastline for wave curve 4
        velocities.getNearestInsidePoints(points4, points1,
                                          landvalue, inNumbers1For4);
        //get the nearest vertex of coastline for wave curve 5
        velocities.getNearestInsidePoints(points5, points1,
                                  landvalue, inNumbers1For5);
    //get the nearest vertex of wave curve 4 for wave curve 5
    velocities.getNearestInsidePoints(points5, points4,
                                 landvalue, inNumbers4For5);
    //get the direction of velocity of wave curve 3
    velocities.getOriginalVelocity(points1, points4,
                   watervalue2, velocity4, originalVelocity4,
                   positiveDirection4);
    //get the direction of velocity of wave curve 4
    velocities.getOriginalVelocity(points1, points5,
                   watervalue2, velocity5, originalVelocity5,
                   positiveDirection5);
    //get the outside vertex for wave curve 3
    velocities.getNearestOutPoints(points4, points5, landvalue,
                                outNumbers5For4);
    HeightOfWaveCurve heightOfWaveCurve
                                  = new HeightOfWaveCurve();
    //get slope for wave curve 3 for adjust its height
    heightOfWaveCurve.getSlope(landvalue, points1, points4,
                                  inNumbers1For4, SlopeFor4);
    //get slope for wave curve 4 for adjust its height
    heightOfWaveCurve.getSlope(landvalue, points1, points5,
                                  inNumbers1For5, SlopeFor5);

    //create vertical velocity for wave curve 1
    for(int n=0; n<points2.size(); n++)
    {
       verticalVel2.add(new FloatNum(0));
    }
    //create vertical velocity for wave curve 2
    for(int n=0; n<points3.size(); n++)
    {
       verticalVel3.add(new FloatNum(0));
    }

    //create water shaders and shader program
    waterVertex = gl.glCreateShader(GL.GL_VERTEX_SHADER);
    waterFragment = gl.glCreateShader(GL.GL_FRAGMENT_SHADER);
    waterShaderProgram = gl.glCreateProgram();
    try { waterShader(gl); } catch (IOException e) { }
}
```

```
//build water shader
private void waterShader(GL gl)throws IOException
{
   //read vertex shader code, and put it into string
   BufferedReader brv
           = new BufferedReader(new FileReader("water.vert"));
   String vsrc = "";
   String lineV;
   while ((lineV = brv.readLine()) != null) {
      vsrc += lineV + "\n";
   }
   String Vsrc [] = new String [1];
   Vsrc [0] = vsrc;
   gl.glShaderSource(waterVertex, 1, Vsrc, null);
   gl.glCompileShader(waterVertex); //compile vertex shader
   //new vertex shader buffer
   IntBuffer vertBuffer = BufferUtil.newIntBuffer(1);
   gl.glGetShaderiv(waterVertex, GL.GL_COMPILE_STATUS,
                 vertBuffer); //get vertex shader information

   //read fragment shader code, and put it into string
   BufferedReader brf
            = new BufferedReader(new FileReader("water.frag"));
   String fsrc = "";
   String line;
   while ((line=brf.readLine()) != null)
   {
      fsrc += line + "\n";
   }
   String Fsrc [] = new String [1];
   Fsrc [0] = fsrc;
   gl.glShaderSource(waterFragment, 1, Fsrc, null);
   gl.glCompileShader(waterFragment); //compile fragment shader
   //new fragment shader buffer
   IntBuffer fragBuffer = BufferUtil.newIntBuffer(1);
   //get fragment shader information
   gl.glGetShaderiv(waterFragment, GL.GL_COMPILE_STATUS,
                 fragBuffer);

   //attach shader objects to shader program
   gl.glAttachShader(waterShaderProgram, waterVertex);
   gl.glAttachShader(waterShaderProgram, waterFragment);
```

```
    gl.glLinkProgram(waterShaderProgram); //link program

    //get the textures location for shaders
    texWParam1 = gl.glGetUniformLocation
                               (waterShaderProgram, "texture1");
    texWParam2 = gl.glGetUniformLocation
                               (waterShaderProgram, "texture2");
}


//draw water surface
private void buildWaterPolygon(GL gl)
{
    int X1, Z1, X2, Z2, X, Z;
    Phase phase=new Phase();

    //create wave profile using phase function
    for (int z = 0; z < watersize; z++)
    {
        for (int x = 0; x < watersize; x++)
        {
            //water over the land
            if(bottomvalue[x][z]<waterLevel)
            {
                //combine 4 wave profile functions to get basic wave
                //heiht
                watervalue[x][z][1]  =
                    (8*(float)Math.pow((phase.Phase(x, 0, T1, t,
                     bottomvalue[x][z])-0.5),2)-1)*phase.A
                   +(8*(float)Math.pow((phase.Phase(x, 0, T2, t,
                     bottomvalue[x][z])-0.5),2)-1)*phase.A*0.25f
                   +(8*(float)Math.pow((phase.Phase(x, 0, T3, t,
                     bottomvalue[x][z])-0.5),2)-1)*phase.A*0.0625f
                   +(8*(float)Math.pow((phase.Phase(x, 0, T4, t,
                     bottomvalue[x][z])-0.5),2)-1)*phase.A
                     *0.015625f;
                //plus water level
                watervalue[x][z][1] += waterLevel;
            }
            else //water under the land at height of water level
                watervalue[x][z][1] = waterLevel;
        }
```

```
}

//adjust height value of wave curves
HeightOfWaveCurve heightOfWaveCurve
                                  = new HeightOfWaveCurve();
//let the height of wave curve 3 always cover the land
heightOfWaveCurve.adjustHeightOfWaveCurve(landvalue,
                             watervalue, points4, SlopeFor4,
                             inNumbers1For4, points1, 0.2f);
//let the height of wave curve 4 always cover the land
heightOfWaveCurve.adjustHeightOfWaveCurve(landvalue,
                            watervalue, points5, SlopeFor5,
                            inNumbers1For5, points1, 0.5f);
//adjust height value of wave curve 1, 2
heightOfWaveCurve.adjustHeightOfW12(watervalue, points2,
                             verticalVel2, watervalue2);
heightOfWaveCurve.adjustHeightOfW12(watervalue, points3,
                             verticalVel3, watervalue2);
//give vertical velocity to wave curve 1 and 2
if((t-1)%8==0&&t!=0)
{
    for(int n=0; n<verticalVel3.size(); n++)
    {
        verticalVel3.get(n).N = 0.15f;
    }
    for(int n=0; n<verticalVel2.size(); n++)
    {
        verticalVel2.get(n).N = 0.25f;
    }
}
//keep wave curve 3&4 not under the water level to much
for(int c=0; c<points4.size(); c++)
{
    X=points4.get(c).X;
    Z=points4.get(c).Z;
    if(watervalue[X][Z][1]<=18.5f)
        watervalue[X][Z][1]=19f;
}
for(int c=0; c<points5.size(); c++)
{
    X=points5.get(c).X;
    Z=points5.get(c).Z;
```

```
   if(watervalue[X][Z][1]<=18.5f)
      watervalue[X][Z][1]=19f;
}
//put water vertex data into array buffer for draw it
waterData.triangleSArray(watersize, watersize, watervalue,
                         waterArray, waterBuffer);
gl.glUseProgram(waterShaderProgram);//use shaders
//set water detail texture in unit 0
gl.glActiveTexture(GL.GL_TEXTURE0);
gl.glEnable(GL.GL_TEXTURE_2D); //enable 2D texture
//bind texture
gl.glBindTexture(GL.GL_TEXTURE_2D, waterDetailTexture);
gl.glUniform1i(texWParam1, 0); //link to shaders
//set water material texture in unit 1
gl.glActiveTexture(GL.GL_TEXTURE1);
gl.glEnable(GL.GL_TEXTURE_2D); //enable 2D texture
//bind texture
gl.glBindTexture(GL.GL_TEXTURE_2D, materialTexture);
gl.glUniform1i(texWParam2, 1); //link to shaders
//vertex array
gl.glEnableClientState(GL.GL_VERTEX_ARRAY);
gl.glVertexPointer(3,  GL.GL_FLOAT,  0,  waterBuffer);
//color array
gl.glEnableClientState(GL.GL_COLOR_ARRAY);
gl.glColorPointer(3, GL.GL_FLOAT, 0, waterCBuffer);
//material texture array
gl.glClientActiveTexture(GL.GL_TEXTURE0);
gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, materialTBuffer);
//water texture array
gl.glClientActiveTexture (GL.GL_TEXTURE1);
gl.glEnableClientState(GL.GL_TEXTURE_COORD_ARRAY);
gl.glTexCoordPointer(2, GL.GL_FLOAT, 0, waterTBuffer);

//draw array
for(int z=0; z<watersize-1; z++)
{
   //draw water surface with triangle strip
   gl.glDrawElements(GL.GL_TRIANGLE_STRIP, watersize*2,
                     GL.GL_UNSIGNED_INT, waterIndexBuffer);
   waterIndexBuffer.position(z*watersize*2);//to next vertex
}
waterIndexBuffer.rewind(); //rewind water index buffer
//disable all
gl.glDisableClientState(GL.GL_VERTEX_ARRAY);
gl.glDisableClientState(GL.GL_COLOR_ARRAY);
gl.glClientActiveTexture(GL.GL_TEXTURE0);
gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
gl.glClientActiveTexture(GL.GL_TEXTURE1);
gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
gl.glClientActiveTexture(GL.GL_TEXTURE2);
gl.glDisableClientState(GL.GL_TEXTURE_COORD_ARRAY);
gl.glActiveTexture(GL.GL_TEXTURE0);
gl.glDisable(GL.GL_TEXTURE_2D);
gl.glActiveTexture(GL.GL_TEXTURE1);
gl.glDisable(GL.GL_TEXTURE_2D);
gl.glActiveTexture(GL.GL_TEXTURE0);
gl.glUseProgram(0);

gl.glDisable(GL.GL_BLEND); //disable blend
gl.glLineWidth(3); //set line width to 3
gl.glPointSize(6); //set point size to 6
//draw wave curves
if(wavecurves==true)
{
   //draw the wave curve 1
   gl.glColor3f(1,1,0); //wave curve 1 color
   gl.glBegin(GL.GL_LINE_STRIP);
   for (int c=0; c<points2.size(); c++)
   {
      X=points2.get(c).X;
      Z=points2.get(c).Z;
      gl.glVertex3f(watervalue[X][Z][0],
         watervalue[X][Z][1], watervalue[X][Z][2]);
   }
   gl.glEnd();
   //draw the wave curve 2
   gl.glColor3f(1,0,1); //wave curve 2 color
   gl.glBegin(GL.GL_LINE_STRIP);
   for (int c=0; c<points3.size(); c++)
   {
      X=points3.get(c).X;
      Z=points3.get(c).Z;
      gl.glVertex3f(watervalue[X][Z][0],
            watervalue[X][Z][1], watervalue[X][Z][2]);
        }
        gl.glEnd();
        //draw the wave curve 3
        gl.glColor3f(1,0,0); //wave curve 3 color
        gl.glBegin(GL.GL_LINE_STRIP);
        for (int c=0; c<points4.size(); c++)
        {
            X=points4.get(c).X;
            Z=points4.get(c).Z;
            gl.glVertex3f(watervalue[X][Z][0],
                watervalue[X][Z][1], watervalue[X][Z][2]);
        }
        gl.glEnd();
        //draw the wave curve 4
        gl.glColor3f(1,1,1); //wave curve 4 color
        gl.glBegin(GL.GL_LINE_STRIP);
        for (int c=0; c<points5.size(); c++)
        {
            X=points5.get(c).X;
            Z=points5.get(c).Z;
            gl.glVertex3f(watervalue[X][Z][0],
                watervalue[X][Z][1], watervalue[X][Z][2]);
        }
        gl.glEnd();
    }

    gl.glLineWidth(1); //set line width to 1
    gl.glPointSize(1); //set point size 1
    gl.glEnable(GL.GL_BLEND); //enable blend

    //velocity about the wave curve 3
    for(int n=0; n<points4.size(); n++)
    {
        X=points4.get(n).X;
        Z=points4.get(n).Z;
        //update x and z value of wave curve 3
        watervalue[X][Z][0]+=0.1f*velocity4.get(n).X;
        watervalue[X][Z][2]+=0.1f*velocity4.get(n).Z;
    }
    //update wave curve 3 velocity
    velocities.getVelocityForHittingWaveCurve(points4,
            watervalue, watervalue2, points1, points5,
            velocity4, positiveDirection4, outNumbers5For4,
            inNumbers1For4, originalVelocity4);
    //give velocity to wave curve 3 to keep it move
    for(int n=0; n<points4.size(); n++)
    {
        if((t-5)%8==0&&t!=0)
        {
            //update wave curve 3 original velocity
            velocities.getPlusVelocity(points1, inNumbers1For4,
                        points4, watervalue2, watervalue,
                        originalVelocity4);
            //plus original velocity to wave curve 3
            velocity4.get(n).X += 2f*originalVelocity4.get(n).X;
            velocity4.get(n).Z += 2f*originalVelocity4.get(n).Z;
        }
    }

    //velocity about the wave curve 4
    for(int n=0; n<points5.size(); n++)
    {
        X=points5.get(n).X;
        Z=points5.get(n).Z;
        //update x and z value of wave curve 3
        watervalue[X][Z][0]+=0.1f*velocity5.get(n).X;
        watervalue[X][Z][2]+=0.1f*velocity5.get(n).Z;
    }
    //update wave curve 4 velocity
    velocities.getVelocityForOtherWaveCurves(points5,
         watervalue, watervalue2, points4, velocity5,
         positiveDirection5, inNumbers4For5,originalVelocity5);
    //give velocity to wave curve 4 to keep it move
    for(int n=0; n<points5.size(); n++)
    {
        if((t-5)%8==0&&t!=0)
        {
            //update wave curve 4 original velocity
            velocities.getPlusVelocity(points1, inNumbers1For5,
                       points5, watervalue2, watervalue,
                       originalVelocity5);
            //plus original velocity to wave curve 4
            velocity5.get(n).X += 1f*originalVelocity5.get(n).X;
        velocity5.get(n).Z += 1f*originalVelocity5.get(n).Z;
    }
}

//create and copy wave curve 3&4 velocity for collision detction
List< Vel > copyOfvelocity4 = new ArrayList< Vel >();
List< Vel > copyOfvelocity5 = new ArrayList< Vel >();
for(int v4 = 0; v4 < velocity4.size(); v4++)
{
    copyOfvelocity4.add(new Vel(velocity4.get(v4).X,
                                velocity4.get(v4).Z));
}
for(int v5 = 0; v5 < velocity5.size(); v5++)
{
    copyOfvelocity5.add(new Vel(velocity5.get(v5).X,
                                velocity5.get(v5).Z));
}

//deal with water/land and waves collisions
Collision collision = new Collision();
collision.getVelocityForWaveCurves(points4, watervalue,
                points5, velocity4, copyOfvelocity4, velocity5,
                copyOfvelocity5, outNumbers5For4, watervalue2,
                originalVelocity4, originalVelocity5, points1,
                inNumbers1For4);

//check the new condition of breaking wave particles
boolean p1, p2, p3, p4, p5, p6, p7, p8;
for (int z = 1; z < watersize-1; z++)
{
    for (int x = 1; x < watersize-1; x++)
    {
        p1=false; p2=false; p3=false; p4=false; p5=false;
        p6=false; p7=false; p8=false;

        //find the highest vertex by compare neighbors
        if(watervalue[x-1][z][1]<watervalue[x][z][1])
            p1=true;
        if(watervalue[x-1][z+1][1]<watervalue[x][z][1])
            p2=true;
        if(watervalue[x][z+1][1]<watervalue[x][z][1])
            p3=true;
        if(watervalue[x+1][z+1][1]<watervalue[x][z][1])
            p4=true;
        if(watervalue[x+1][z][1]<watervalue[x][z][1])
            p5=true;
        if(watervalue[x+1][z-1][1]<watervalue[x][z][1])
            p6=true;
        if(watervalue[x][z-1][1]<watervalue[x][z][1])
            p7=true;
        if(watervalue[x-1][z-1][1]<watervalue[x][z][1])
            p8=true;

        //check the condition of breaking wave
        if(p1==true && p2==true && p3==true && p5==true
            && p6==true && p7==true && p8==true)
        {
            //check left front and right back vertices
            if(getTan(watervalue[x][z][1],
                    watervalue[x-1][z+1][1])
              + getTan(watervalue[x][z][1],
                     watervalue[x+1][z-1][1]) < 2.1f)
            {
                for(int i=0; i<breakingWaveParSize; i++)
                {
                    //give particle coordinate
                    breakingWaveParValue[x][z][i][0]
                        = watervalue[x][z][0]
                          + (float)Math.random()-0.5f;
                    breakingWaveParValue[x][z][i][1]
                        = watervalue[x][z][1]
                          + (float)Math.random()-0.5f;
                    breakingWaveParValue[x][z][i][2]
                        = watervalue[x][z][2]
                          + (float)Math.random()-0.5f;
                    //give particle vertical velocity
                    breakingWaveParVel.get(
                        (z-1)*(watersize-2)*breakingWaveParSize
                        + (x-1)*breakingWaveParSize + i).Y
                        = (float)Math.random()-0.5f;
                    //give particle life
                    breakingWaveParLife.get(
                        (z-1)*(watersize-2)*breakingWaveParSize
                         + (x-1)*breakingWaveParSize + i).N = 1;
    }
}
//check left and right vertices
else if(getTan(watervalue[x][z][1],
          watervalue[x-1][z][1])
          + getTan(watervalue[x][z][1],
          watervalue[x+1][z][1]) < 2.1f)
{
    for(int i=0; i<breakingWaveParSize; i++)
    {
        //give particle coordinate
        breakingWaveParValue[x][z][i][0]
            = watervalue[x][z][0]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][1]
            = watervalue[x][z][1]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][2]
            = watervalue[x][z][2]
              + (float)Math.random()-0.5f;
        //give particle vertical velocity
        breakingWaveParVel.get(
            (z-1)*(watersize-2)*breakingWaveParSize
            + (x-1)*breakingWaveParSize + i).Y
            = (float)Math.random()-0.5f;
        //give particle life
        breakingWaveParLife.get(
            (z-1)*(watersize-2)*breakingWaveParSize
            + (x-1)*breakingWaveParSize + i).N = 1;
    }
}
//check front and back vertices
else if(getTan(watervalue[x][z][1],
          watervalue[x][z+1][1])
          + getTan(watervalue[x][z][1],
           watervalue[x][z-1][1]) < 2.1f)
{
    for(int i=0; i<breakingWaveParSize; i++)
    {
        //give particle coordinate
        breakingWaveParValue[x][z][i][0]
            = watervalue[x][z][0]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][1]
            = watervalue[x][z][1]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][2]
            = watervalue[x][z][2]
              + (float)Math.random()-0.5f;
        //give particle vertical velocity
        breakingWaveParVel.get(
            (z-1)*(watersize-2)*breakingWaveParSize
             + (x-1)*breakingWaveParSize + i).Y
             = (float)Math.random()-0.5f;
        //give particle life
        breakingWaveParLife.get(
            (z-1)*(watersize-2)*breakingWaveParSize
            + (x-1)*breakingWaveParSize + i).N = 1;
    }
}
//check left back and right front vertices
else if(getTan(watervalue[x][z][1],
          watervalue[x-1][z-1][1])
          + getTan(watervalue[x][z][1],
          watervalue[x+1][z+1][1]) < 2.1f)
{
    for(int i=0; i<breakingWaveParSize; i++)
    {
        //give particle coordinate
        breakingWaveParValue[x][z][i][0]
            = watervalue[x][z][0]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][1]
            = watervalue[x][z][1]
              + (float)Math.random()-0.5f;
        breakingWaveParValue[x][z][i][2]
            = watervalue[x][z][2]
              + (float)Math.random()-0.5f;
        //give particle vertical velocity
        breakingWaveParVel.get(
            (z-1)*(watersize-2)*breakingWaveParSize
             + (x-1)*breakingWaveParSize + i).Y
             = (float)Math.random()-0.5f;
        //give particle life
```

```
                    breakingWaveParLife.get(
                        (z-1)*(watersize-2)*breakingWaveParSize
                         + (x-1)*breakingWaveParSize + i).N = 1;
                }
            }
        }
    }
    }
    //time interval
    tt+=1;
    t=tt/10;
}


//get arc tangent
private float getTan(float y, float y1)
{
    return (float)Math.atan(1/(y-y1));
}


//======================Partilces shader=======================
//build particle shaders and program
private void buildParShader(GL gl)
{
    parVertex = gl.glCreateShader(GL.GL_VERTEX_SHADER);
    parFragment = gl.glCreateShader(GL.GL_FRAGMENT_SHADER);
    parShaderProgram = gl.glCreateProgram();
    try { parShader(gl); } catch (IOException e) { }
}


//build particle shaders
private void parShader(GL gl)throws IOException
{
    //read vertex shader code, and put it into string
    BufferedReader brv = new BufferedReader(
                                    new FileReader("par.vert"));
    String vsrc = "";
    String lineV;
    while ((lineV = brv.readLine()) != null)
    {
        vsrc += lineV + "\n";
    }
    String Vsrc [] = new String [1];
    Vsrc [0] = vsrc;
    gl.glShaderSource(parVertex, 1, Vsrc, null);
    gl.glCompileShader(parVertex); //compile vertex shader
    //new vertex shader buffer
    IntBuffer vertBuffer = BufferUtil.newIntBuffer(1);
    gl.glGetShaderiv(parVertex, GL.GL_COMPILE_STATUS,
                    vertBuffer); //get vertex shader information

    //read fragment shader code, and put it into string
    BufferedReader brf = new BufferedReader(
                                    new FileReader("par.frag"));
    String fsrc = "";
    String line;
    while ((line=brf.readLine()) != null)
    {
        fsrc += line + "\n";
    }
    String Fsrc [] = new String [1];
    Fsrc [0] = fsrc;
    gl.glShaderSource(parFragment, 1, Fsrc, null);
    gl.glCompileShader(parFragment); //compile fragment shader
    //new fragment shader buffer
    IntBuffer fragBuffer = BufferUtil.newIntBuffer(1);
    //get fragment shader information
    gl.glGetShaderiv(parFragment, GL.GL_COMPILE_STATUS,
                    fragBuffer);
    //attach shader objects to shader program
    gl.glAttachShader(parShaderProgram, parVertex);
    gl.glAttachShader(parShaderProgram, parFragment);
    gl.glLinkProgram(parShaderProgram);

    //get the textures location for shaders
    texParam = gl.glGetUniformLocation(parShaderProgram,
                                        "texture");
}


/**********************Draw The Model***********************/
public void display(GLAutoDrawable drawable)
```

```
{
   GL gl = drawable.getGL();
   gl.glClearColor(0.3f,0.3f,0.3f,0f); //set clear color
   gl.glClearDepth(1.0f); //set clear depth
   //clear window
   gl.glClear(GL.GL_COLOR_BUFFER_BIT|GL.GL_DEPTH_BUFFER_BIT);
   gl.glMatrixMode(GL.GL_MODELVIEW); //At modelview mode
   //Set the current matrix to an identity matrix
   gl.glLoadIdentity();
   //Translatef in x, y direction with the value of
   //xdirection/40, ydirection/40
   gl.glTranslatef(xdirection/40, ydirection/40,0);
   //Scale the model in x,y,z direction with the value of
   //zdirection/40, zdirection/40, zdirection/40
   gl.glScalef(zdirection/40, zdirection/40, zdirection/40);
   gl.glScalef(0.3f, 0.3f, 0.3f);
   //Rotate 90 angle with a axes(0,0,0)to (1,0,0)
   gl.glRotatef(30,1,0,0);
   gl.glRotatef(rotatex,1,0,0);//Rotate rotatex angle with x axes
   gl.glRotatef(rotatey,0,1,0);//Rotate rotatey angle with y axes
   gl.glRotatef(rotatez,0,0,1);//Rotate rotatez angle with z axes

   buildCoordinate(gl); //Draw the coordinate axes

   if(land==true)
   {
      if(mesh==true)
         //set polygon mode to mesh
         gl.glPolygonMode(GL.GL_FRONT_AND_BACK,GL.GL_LINE);
      else
         //set polygon mode to surface
         gl.glPolygonMode(GL.GL_FRONT_AND_BACK,GL.GL_FILL);
      buildLandPolygon(gl); //Draw the land surface
   }

   if(fill==true)
      //set polygon mode to surface
      gl.glPolygonMode(GL.GL_FRONT_AND_BACK,GL.GL_FILL);
   else
      //set polygon mode to mesh
      gl.glPolygonMode(GL.GL_FRONT_AND_BACK,GL.GL_LINE);
```

```
   if(blend==true)
      gl.glEnable(GL.GL_BLEND); //enable blend
   //set blend mode
   gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE);

   if(water==true)
      buildWaterPolygon(gl); //Draw the water surface

   //draw breaking wave particle
   particle.buildBreakingWaveParticle(gl, parShaderProgram,
             parTex, texParam, watersize, breakingWaveParSize,
             breakingWaveParLife, breakingWaveParVel,
             breakingWaveParValue, watervalue);

   //draw water spray particle
   particle.buildWaterSprayParticle(gl, waterSprayParSize,
            points4, parvalue3, watervalue, velocity4, parLife3,
            parVel3, parShaderProgram, parTex, texParam);
   particle.buildWaterSprayParticle(gl, waterSprayParSize,
            points5, parvalue4, watervalue, velocity5, parLife4,
            parVel4, parShaderProgram, parTex, texParam);

   gl.glDisable(GL.GL_BLEND); //disable blend

   gl.glFlush(); //output the results immediately
   drawable.swapBuffers(); //swap buffers
}


private void buildCoordinate(GL gl) //draw 3D coordinate axes
{
   glut = new GLUT();
   gl.glLineWidth(3);
   //x axis
   gl.glColor3f(1,0,0);
   gl.glBegin(GL.GL_LINES);
      gl.glVertex3i(0, 30, 0);
      gl.glVertex3i(5, 30, 0);
   gl.glEnd();
   gl.glPushMatrix();
      gl.glTranslatef(5, 30, 0);
      gl.glRotatef(90, 0, 1, 0);
```

```
        glut.glutWireCone(0.3f, 1, 5, 5);
    gl.glPopMatrix();
    //y axis
    gl.glColor3f(0,1,0);
    gl.glBegin(GL.GL_LINES);
        gl.glVertex3i(0, 30, 0);
        gl.glVertex3i(0, 35, 0);
    gl.glEnd();
    gl.glPushMatrix();
        gl.glTranslatef(0, 35, 0);
        gl.glRotatef(-90, 1, 0, 0);
        glut.glutWireCone(0.3f, 1, 5, 5);
    gl.glPopMatrix();
    //z axis
    gl.glColor3f(0,0,1);
    gl.glBegin(GL.GL_LINES);
        gl.glVertex3i(0, 30, 0);
        gl.glVertex3i(0, 30, 5);
    gl.glEnd();
    gl.glPushMatrix();
        gl.glTranslatef(0, 30, 5);
        glut.glutWireCone(0.3f, 1, 5, 5);
    gl.glPopMatrix();
    gl.glLineWidth(1);
}


public void reshape(GLAutoDrawable drawable, int x, int y, int w,
                     int h)
{
    GL gl = drawable.getGL();
    gl.glViewport(0, 0, w, h); //Set area of viewport
    gl.glMatrixMode(GL.GL_PROJECTION); //At projection mode
    gl.glLoadIdentity();
    //Set area of projection view by window size
    if (w <= h)
      gl.glOrtho(-10.0, 10.0, -10.0 * (float) h / (float) w,
                10.0 * (float) h/ (float) w, -500.0, 500.0);
    else
    gl.glOrtho(-10.0 * (float) w / (float) h,
         10.0 * (float) w/ (float) h, -10.0, 10.0, -500.0, 500.0);
    gl.glMatrixMode(GL.GL_MODELVIEW); //At modelview mode
    gl.glLoadIdentity();
    width=w;
    length=h;
}


//for build texture
private int genTexture(GL gl)
{
    final int[] tmp = new int[1];
    gl.glGenTextures(1, tmp, 0);
    return tmp[0];
}


//for build buffer
private int genBuffer(GL gl)
{
    final int[] tmp = new int[1];
    gl.glGenBuffers(1, tmp, 0);
    return tmp[0];
}


public void keyPressed(KeyEvent e)
{
    switch (e.getKeyCode())
    {
        case KeyEvent.VK_UP: //up key is down
            rotatex -= 3f;
            break;
        case KeyEvent.VK_DOWN: //down key is down
            rotatex += 3f;
            break;
        case KeyEvent.VK_LEFT: //Left key is down
            rotatey -= 3f;
            break;
        case KeyEvent.VK_RIGHT: //right key is down
            rotatey += 3f;
            break;
        case KeyEvent.VK_Z: //Z key is down
            rotatez -= 3f;
            break;
         case KeyEvent.VK_X: //X key is down
            rotatez += 3f;
            break;
         case KeyEvent.VK_ESCAPE: //esc key is down
            System.exit(0);
            break;
         case KeyEvent.VK_PAGE_UP: //pageup key is down
            zdirection+=1f;
            break;
         case KeyEvent.VK_PAGE_DOWN: //pagedown key is down
            zdirection-=1f;
            break;
         case KeyEvent.VK_F: //F key is down
            fill=!fill;
            break;
         case KeyEvent.VK_W: //W key is down
            water=!water;
            break;
         case KeyEvent.VK_L: //L key is down
            land=!land;
            break;
         case KeyEvent.VK_M: //M key is down
            mesh=!mesh;
            break;
         case KeyEvent.VK_S: //S key is down
            move=!move;
            break;
         case KeyEvent.VK_B: //B key is down
            blend=!blend;
            break;
         case KeyEvent.VK_O: //O key is down
            wavecurves=!wavecurves;
            break;
      }
   }

   public void mousePressed(MouseEvent e)
   {
      //mouse left key is down
      if(e.getButton()==MouseEvent.BUTTON1)
      {
         button1=true;
         //update mouse coordinate
         newx=e.getX();newy=e.getY();
      }
      else button1=false;
      //mouse right key is down
      if(e.getButton()==MouseEvent.BUTTON3)
      {
         button2=true;
         newx2=e.getX();
         newy2=e.getY();
      }
      else button2=false;
   }

   public void mouseDragged(MouseEvent e)
   {
      if(button1)
      {
         //compute the change of the mouse coordinate
         xdirection+=e.getX()-newx;
         ydirection+=newy-e.getY();
         //check the mouse coordinate
         newx=e.getX();
         newy=e.getY();
      }
      if(button2)
      {
         //compute the change of the mouse coordinate
         rotatey+=(e.getX()-newx2)/5f;
         rotatex+=(e.getY()-newy2)/5f;
         if (rotatex>90)
            rotatex=90;
         if (rotatex<-90)
            rotatex=-90;
         //check the mouse coordinate
         newx2=e.getX();
         newy2=e.getY();
      }
   }
```

```
    //I do not need these methods
    public void displayChanged(GLAutoDrawable drawable,
                           boolean modeChanged, boolean deviceChanged){}
    public void keyReleased(KeyEvent key){}
    public void keyTyped(KeyEvent key){}
    public void mouseClicked(MouseEvent key){}
    public void mouseEntered(MouseEvent key){}
    public void mouseExited(MouseEvent key){}
    public void mouseReleased(MouseEvent key){}
    public void mouseMoved(MouseEvent e){}
}
```

Appendix A7

```
/* Phase.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Create a phase function for the heights of water vertices.
 * See section 4.1 for details.
 */

package demos;

import java.lang.Math.*;

public class Phase
{
   float g = 9.8f, A;
   public float Phase(float x, float z, float T, float t, float h)
   {
      float u, Ld, L, depth, deep;
      //deep-water wavelength
      Ld = g*(float)Math.pow(T,2)*(1/(2*(float)Math.PI));
      deep = Ld*0.5f; //condition of deep-water
      depth = 20-h; //depth of water
      if(depth >= deep)
         L = Ld;
      Else
         //shallow-water wavelength
         L = T*(float)Math.sqrt(g*depth);
      A = L*(1/Ld); //keeping amplitude over wavelength is a constant

      u = (float)Math.sqrt(x*x + z*z)*(1/L); //basic phase function
      u = u-(t*(1/T)); //phase function with time
      u = Math.abs(u%1); //make the phase value in the area of [0,1)

      //control the asymmetry of the wave profile
      if(depth < Ld*0.02f) //depth/L < 0.02f
         u = (float)Math.pow(u,2);
      return u;
   }
}
```

```
/* WaveCurves.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Calculate the coastline and wave curves.
 * See section 4.3 for details.
 */

package demos;

import java.util.List;
import java.util.ArrayList;

public class WaveCurves
{
   private List< Line > list = new ArrayList< Line >();
   private List< Line > list2 = new ArrayList< Line >();
   private List< Line > list3 = new ArrayList< Line >();
   private List< Line > list4 = new ArrayList< Line >();

   private boolean startPoint, endPoint;


   //put a line segment into list
   public void getLine(int x1, int z1, int x2, int z2)
   {
      //put line segment into a list
      list.add( new Line(x1, z1, x2, z2) );
   }


   //if each point of a line segment appear at least twice, get this
   //line segment
   private void compareLineSegment()
   {
      for(int k=0; k<list.size(); k++)
      {
         startPoint=false; endPoint=false;
         for(int i=0; i<list.size(); i++)
         {
            if(i==k) i++; if(i==list.size()) break;
            //compare the start point of a line segment
            compareStartPoint(list.get(k), list.get(i));
            if(startPoint==true)
            break;
         }
         for(int j=0; j<this.list.size(); j++)
         {
            if(j==k) j++; if(j==list.size()) break;
            //compare the end point of a line segment
            compareEndPoint(list.get(k), list.get(j));
            if(endPoint==true)
            break;
         }
         if(startPoint==true&&endPoint==true)
         {
            //put useful line segment into a new list
            list2.add(list.get(k));
         }
      }
      list.clear();
   }


   //make sure is or not the start vertex of line segment 1 can connect
   //line segment 2
   private void compareStartPoint(Line line1, Line line2 )
   {
      if(line1.X1==line2.X1 && line1.Z1==line2.Z1 ||
         line1.X1==line2.X2 && line1.Z1==line2.Z2)
         startPoint=true;
   }


   //make sure is or not the end vertex of line segment 1 can connect
   //line segment 2
   private void compareEndPoint(Line line1, Line line2)
   {
      if(line1.X2==line2.X2 && line1.Z2==line2.Z2 ||
         line1.X2==line2.X1 && line1.Z2==line2.Z1)
         endPoint=true;
   }


   //get all the continuous lines and connect them
   private void getOrder()
   {
      adjustOrder(list2, list3);
      while(!list2.isEmpty())
      {
         adjustOrder(list2, list4);
         list3.addAll(list3.size(), list4);
         list4.clear();
      }
   }


   //find out a continuous line and make the end point of a line segment
   //can connect to the start point of next line segment
   private void adjustOrder(List<Line> l1, List<Line> l2)
   {
      boolean add1=true, add2=true;
      l2.add(l1.get(0));
      l1.remove(0);

      while(add2==true)
```

```
{
   add2=false;
   for(int i=0; i<l1.size(); i++)
   {
      //compare the end point of the list l2 with the end
      //point of a line segment
      if(l2.get(l2.size()-1).X2==l1.get(i).X2 &&
         l2.get(l2.size()-1).Z2==l1.get(i).Z2)
      {
         //put the order line segment into the end of the list
         //l2 after turn it around
         l2.add(turnAround(l1.get(i)));
         l1.remove(i);
         add2=true;
         break;
      }
      //compare the end point of the list l2 with the start
      //point of a line segment
      if(l2.get(l2.size()-1).X2==l1.get(i).X1 &&
         l2.get(l2.size()-1).Z2==l1.get(i).Z1)
      {
         //put the order line segment into the end of the list
         //l2
         l2.add(l1.get(i));
         l1.remove(i);
         add2=true;
         break;
      }
   }
}

while(add1==true)
{
   add1=false;
   for(int i=0; i<l1.size(); i++)
   {
      //compare the start point of the list l2 with the start
      //point of a line segment
      if(l2.get(0).X1==l1.get(i).X1 &&
         l2.get(0).Z1==l1.get(i).Z1)
      {
         //put the order line segment into the start of the
         //list l2
         l2.add(0, turnAround(l1.get(i)));
         l1.remove(i);
         add1=true;
         break;
      }
      //compare the start point of the list l2 with the end
      //point of a line segment
      if(l2.get(0).X1==l1.get(i).X2 &&
         l2.get(0).Z1==l1.get(i).Z2)
      {
         //put the order line segment into the start of the
         //list l2
               l2.add(0, l1.get(i));
               l1.remove(i);
               add1=true;
               break;
            }
         }
      }
   }

   //turn around the line segment
   private Line turnAround(Line line)
   {
      int x, z;
      x=line.X1; line.X1=line.X2; line.X2=x;
      z=line.Z1; line.Z1=line.Z2; line.Z2=z;
      return line;
   }


   //get the start point from a line segment
   private void translateToPoints(List<Line> l, List<Points> p)
   {
      for(int i=0; i<l.size(); i++)
      {
         //put the start point of each line segment into point list
         p.add( new Points(l.get(i).X1, l.get(i).Z1) );
      }
      //put the end point of the last line segment into point list
      p.add( new Points(l.get(l.size()-1).X2,
                        l.get(l.size()-1).Z2) );
   }


   //if the continuous 3 points in a square, delete the middle point
   private void deleteSuperfluousPoint(List<Points> p)
   {
      int n=p.size();
      for(int i=0; i<n; i++)
      {
         if(i+2>n-1) break;
         if(checkConnective(p.get(i+2), p.get(i)))
         {
            p.remove(i+1);
            n--;
         }
      }
   }


   //check out is point a beside point b
   private boolean checkConnective(Points a, Points b)
   {
      if(   a.X==b.X+1 && a.Z==b.Z
        || a.X==b.X   && a.Z==b.Z+1
      || a.X==b.X+1 && a.Z==b.Z+1
      || a.X==b.X-1 && a.Z==b.Z
      || a.X==b.X   && a.Z==b.Z-1
      || a.X==b.X-1 && a.Z==b.Z-1
      || a.X==b.X-1 && a.Z==b.Z+1
      || a.X==b.X+1 && a.Z==b.Z-1  )
      return true;
   else
      return false;
}


//call other method to get coastline
public void getCoastline(List<Points> p1)
{
   //delete Useless Line Segment
   compareLineSegment();
   //dealing with Disordered Line Segments
   getOrder();
   //translate line segment to vertex
   translateToPoints(list3, p1);
   //delete useless vertex
   deleteSuperfluousPoint(p1);
   deleteSuperfluousPoint(p1);
   p1.remove(p1.size()-1);
}


//get wave curves
public void getWaveCurve(List<Points> p1, List<Points> p2,
                         List<Points> p3)
{
   List< Points > points = new ArrayList< Points >();
   List< Points > points2 = new ArrayList< Points >();
   List< Points > points3 = new ArrayList< Points >();
   boolean RepeatedPoints, SamePoints;
   //delete repeated points in list p1
   for(int i=0; i<p1.size(); i++)
   {
      RepeatedPoints=false;
      for(int j=i; j<p1.size()-1; j++)
      {
         //check repeated points
         if(p1.get(i).X==p1.get(j+1).X &&
            p1.get(i).Z==p1.get(j+1).Z)
         {
            RepeatedPoints=true;
            break;
         }
      }
      //get the point which is not repeated to new list points
      if(RepeatedPoints==false)
         points.add(p1.get(i));
   }
   p1.clear();
```

```
//delete same point with list p2
for(int m=0; m<points.size(); m++)
{
   SamePoints=false;
   for(int n=0; n<p2.size(); n++)
   {
      //check the same point
      if(points.get(m).X==p2.get(n).X &&
         points.get(m).Z==p2.get(n).Z)
      {
         SamePoints=true;
         break;
      }
   }
   //get the point which is not same to p2 to new list points2
   if(SamePoints==false)
      points2.add(points.get(m));
}

//for the wave curves 2, 3, 4
//delete same point with list p3
if(p2!=p3)
{
   for(int m=0; m<points2.size(); m++)
   {
      SamePoints=false;
      for(int n=0; n<p3.size(); n++)
      {
         //check the same point
         if(points2.get(m).X==p3.get(n).X &&
            points2.get(m).Z==p3.get(n).Z)
         {
            SamePoints=true;
            break;
         }
      }
      //get the point which is not same to p3 to new list points3
      if(SamePoints==false)
         points3.add(points2.get(m));
   }
   //adjust the order
   //put first two point to the wave curve list p1
   p1.add(points3.get(0));
   p1.add(points3.get(1));
   for(int m=1; m<points3.size(); m++)
   {
      for(int n=m+1; n<points3.size(); n++)
      {
         if(checkConnective(points3.get(m),
                            points3.get(n)))
         {
            //put the point to the wave curve list
            p1.add(points3.get(n));
            //cut the neighbor point on the neighbor order,
                    //so that the order for look through (first for
                    //loop) is right
                    points3.add(m+1, points3.get(n));
                    points3.remove(n+1);
                }
            }
        }
    }

    //for wave curve 1
    else
    {
        //adjust the order
        //put first two point to the wave curve list p1
        p1.add(points2.get(0));
        p1.add(points2.get(1));
        for(int m=1; m<points2.size(); m++)
        {
            for(int n=m+1; n<points2.size(); n++)
            {
                if(checkConnective(points2.get(m),
                                   points2.get(n)))
                {
                    //put the connectable point to the final wave
                    //curve list
                    p1.add(points2.get(n));
                    //copy the connectable point to the right
                    //position
                    points2.add(m+1, points2.get(n));
                    //remove the connectable point from the original
                    //position
                    points2.remove(n+1);
                }
            }
        }
    }
}
}
```

```
/* Velocities.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Calculate the velocities for wave curves employing spring system.
 * See section 5.6.1 for details.
 */

package demos;

import java.util.List;
import java.util.ArrayList;

public class Velocities
{
   private float t=0.1f, kw=0.2f, kp=0.5f, n=0f;
   private boolean hit=false;

   //spring for inside wave curves in X axis
   private float setInsideSpringForX(float x1, float x2)
   {
      float fx2=0, distanceX2=0;
      //force from inside wave curve x2
      distanceX2 = (float)Math.abs(x2-x1)-n;
      if(distanceX2>0 && x2!=x1)
         fx2 = kw*distanceX2*(x2-x1)*(1/(float)Math.abs(x2-x1));
      else if(distanceX2<0 && x2!=x1)
         fx2 = -kw*distanceX2*(x1-x2)*(1/(float)Math.abs(x1-x2));
      return fx2;
   }


   //spring for outside wave curves in X axis
   private float setOutsideSpringForX(float x0, float x1)
   {
      float fx0=0, distanceX0=0;
      //force from outside wave curve x0
      distanceX0 = (float)Math.abs(x1-x0)-n;
      if(distanceX0>0 && x0!=x1)
         fx0 = kw*distanceX0*(x0-x1)*(1/(float)Math.abs(x0-x1));
      else if(distanceX0<0 && x0!=x1)
         fx0 = -kw*distanceX0*(x1-x0)*(1/(float)Math.abs(x1-x0));
      return fx0;
   }


   //position spring in X axis
   private float setPSpringForX(float x1, float wx1)
   {
      return -kp*(x1-wx1);//force from origin
   }

   //spring for inside wave curves in Z axis
   private float setInsideSpringForZ(float z1, float z2)
   {
      float fz2=0, distanceZ2=0;
      //force from inside wave curve z2
      distanceZ2 = (float)Math.abs(z2-z1)-n;
      if(distanceZ2>0 && z2!=z1)
         //minus different from x axis
         fz2 = kw*distanceZ2*(z2-z1)*(1/(float)Math.abs(z2-z1));
      else if(distanceZ2<0 && z2!=z1)
         //minus different from x axis
         fz2 = -kw*distanceZ2*(z1-z2)*(1/(float)Math.abs(z1-z2));
      return fz2;
   }


   //spring for outside wave curves in Z axis
   private float setOutsideSpringForZ(float z0, float z1)
   {
      float fz0=0, distanceZ0=0;
      //force from outside wave curve z0
      distanceZ0 = (float)Math.abs(z1-z0)-n;
      if(distanceZ0>0 && z0!=z1)
         fz0 = kw*distanceZ0*(z0-z1)*(1/(float)Math.abs(z0-z1));
      else if(distanceZ0<0 && z0!=z1)
         fz0 = -kw*distanceZ0*(z1-z0)*(1/(float)Math.abs(z1-z0));
      return fz0;
   }


   //position spring in Z axis
   private float setPSpringForZ(float z1, float wz1)
   {
      return -kp*(z1-wz1);//force from origin
   }


/****************************************************************/
   //deal with the wave curve which will hit the land
   public void getVelocityForHittingWaveCurve(
         List<Points> currentPoints, float Currentposition[][][],
         float originposition[][][], List<Points> neighborPoints,
         List<Points> outerNeighborPoints,  List<Vel> v,
         List<Points> positiveDirection,
         List<Points> outNumbers, List<Points> insideNearestVertex,
         List<Vel> originalVelocity)
   {
      float x=0, z=0, xo=0, zo=0, vx=0, vz=0, a, b, c, d, fx=0, fz=0;
      int dx=1, dz=1, dvx=1, dvz=1, o=0;
      for(int i=0; i<currentPoints.size(); i++)
      {
         fx=0; fz=0;
         //current position of current point
```

```
x = Currentposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][0];
x = Currentposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][2];
//two nearest outside vertices
if(outNumbers.get(i).Z != -1)
{
   int m=0, n=0;
   float x1=0, z1=0, x2=0, z2=0;
   m = outNumbers.get(i).X;
   //current position of the point 1 which will be hit by
   //current point
   x1 = Currentposition[outerNeighborPoints.get(m).X]
                       [outerNeighborPoints.get(m).Z]
                       [0];
   z1 = Currentposition[outerNeighborPoints.get(m).X]
                       [outerNeighborPoints.get(m).Z]
                       [2];
   //force of wave spring
   fx = setOutsideSpringForX(x1, x);
   fz = setOutsideSpringForZ(z1, z);

   n = outNumbers.get(i).Z;
   //current position of the point 2 which will be hit by
   //current point
   x2 = Currentposition[outerNeighborPoints.get(n).X]
                       [outerNeighborPoints.get(n).Z]
                       [0];
   z2 = Currentposition[outerNeighborPoints.get(n).X]
                       [outerNeighborPoints.get(n).Z]
   //force of wave spring
   fx += setOutsideSpringForX(x2, x);
   fz += setOutsideSpringForZ(z2, z);
   //velocity only in one axis
   if(originalVelocity.get(i).X == 0) fx = 0;
   else if(originalVelocity.get(i).Z == 0) fz = 0;
}
//one outside vertex
else
{
   int m=0;
   float x1=0, z1=0;
   m = outNumbers.get(i).X;
   //current position of the point which will be hit by
   //current point
   x1 = Currentposition[outerNeighborPoints.get(m).X]
                       [outerNeighborPoints.get(m).Z]
                       [0];
   z1 = Currentposition[outerNeighborPoints.get(m).X]
                       [outerNeighborPoints.get(m).Z]
                       [2];
   //force of wave spring
   fx = 2f*setOutsideSpringForX(x1, x);
   fz = 2f*setOutsideSpringForZ(z1, z);
   //velocity only in one axis
   if(originalVelocity.get(i).X == 0) fx = 0;
   else if(originalVelocity.get(i).Z == 0) fz = 0;
}

//force of position spring
fx += setPSpringForX(x,
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [0]);
fz += setPSpringForZ(z,
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [2]);
//normal velocity
vx = fx*t + v.get(i).X;
vz = fz*t + v.get(i).Z;

//current position of the point in coastline which will
//be hit by current point
o = insideNearestVertex.get(i).X;
xo = Currentposition[neighborPoints.get(o).X]
                    [neighborPoints.get(o).Z][0];
zo = Currentposition[neighborPoints.get(o).X]
                    [neighborPoints.get(o).Z][2];
//hitting: if the wave curve hit the coastline, let it come
//back to its origin
if(anotherGetDistance(xo, zo, x, z)<=0.5f)
{
   a = originposition[currentPoints.get(i).X]
                     [currentPoints.get(i).Z][0]-x;
   b = originposition[currentPoints.get(i).X]
                     [currentPoints.get(i).Z][2]-z;
   vx = 0.1f*a;
   vz = 0.1f*b;
}

//the direction of current position compare to origin
c = Currentposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][0]
    -originposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][0];
d = Currentposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][2]
    -originposition[currentPoints.get(i).X]
                   [currentPoints.get(i).Z][2];
if(c>0)       dx  =  1;
else if(c<0)  dx  = -1;
if(d>0)       dz  =  1;
else if(d<0)  dz  = -1;
if(vx>0)      dvx =  1;
else if(vx<0) dvx = -1;
if(vz>0)      dvz =  1;
else if(vz<0) dvz = -1;
//if current point come back to its origin and leave the
//bank, let it stop at origin quickly
```

```
        if((dx == -positiveDirection.get(i).X &&
            dvx == -positiveDirection.get(i).X)
           || (dz == -positiveDirection.get(i).Z &&
            dvz == -positiveDirection.get(i).Z))
        {
           vx*=0.8f;
           vz*=0.8f;
        }

        //final velocity
        v.get(i).X = vx;
        v.get(i).Z = vz;
     }
  }


  //deal with the other wave curves
  public void getVelocityForOtherWaveCurves(
         List<Points> currentPoints, float Currentposition[][][],
         float originposition[][][], List<Points> neighborPoints,
         List<Vel> v, List<Points> positiveDirection,
         List<Points> insideNearestVertex,
         List<Vel> originalVelocity)
  {
     float x=0, z=0, vx=0, vz=0, fx=0, fz=0, c, d;
     int dx=1, dz=1, dvx=1, dvz=1;
     for(int i=0; i<currentPoints.size(); i++)
     {
        fx=0; fz=0;
        //current position of current point
        x = Currentposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0];
        z = Currentposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][2];
        //two nearest inside vertices
        if(insideNearestVertex.get(i).Z != -1)
        {
           int p=0, q=0;
           float x3=0, z3=0, x4=0, z4=0;
           p = insideNearestVertex.get(i).X;
           //current position of the inpoint 1 which will be hit
           //by current point
           x3 = Currentposition[neighborPoints.get(p).X]
                               [neighborPoints.get(p).Z][0];
           z3 = Currentposition[neighborPoints.get(p).X]
                               [neighborPoints.get(p).Z][2];
           //force of wave spring
           fx += setInsideSpringForX(x, x3);
           fz += setInsideSpringForZ(z, z3);

           q = insideNearestVertex.get(i).Z;
           //current position of the inpoint 1 which will be hit
           //by current point
           x4 = Currentposition[neighborPoints.get(q).X]
                               [neighborPoints.get(q).Z][0];
           z4 = Currentposition[neighborPoints.get(q).X]
                               [neighborPoints.get(q).Z][2];
           //force of wave spring
           fx += setInsideSpringForX(x, x4);
           fz += setInsideSpringForZ(z, z4);
           if(originalVelocity.get(i).X == 0) fx = 0;
           else if(originalVelocity.get(i).Z == 0) fz = 0;
        }
        //one inside nearest vertex
        else
        {
           int p=0;
           float x3=0, z3=0;
           p = insideNearestVertex.get(i).X;
           //current position of the inpoint 1 which will be hit
           //by current point
           x3 = Currentposition[neighborPoints.get(p).X]
                               [neighborPoints.get(p).Z][0];
           z3 = Currentposition[neighborPoints.get(p).X]
                               [neighborPoints.get(p).Z][2];
           //force of wave spring
           fx += 2f*setInsideSpringForX(x, x3);
           fz += 2f*setInsideSpringForZ(z, z3);
           if(originalVelocity.get(i).X == 0) fx = 0;
           else if(originalVelocity.get(i).Z == 0) fz = 0;
        }

        //force of position spring
        fx += setPSpringForX(x,
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0]);
        fz += setPSpringForZ(z,
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [2]);
        //normal velocity
        vx = fx*t + v.get(i).X;
        vz = fz*t + v.get(i).Z;

        //the direction of current position compare to origin
        c = Currentposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0]
           - originposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0];
        d = Currentposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][2]
           - originposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][2];
        if(c>0)       dx  =  1;
        else if(c<0)   dx  = -1;
        if(d>0)       dz  =  1;
        else if(d<0)   dz  = -1;
        if(vx>0)      dvx =  1;
        else if(vx<0)  dvx = -1;
        if(vz>0)      dvz =  1;
        else if(vz<0)  dvz = -1;
        //if current point come back to its origin and leave the
        //bank, let it stop at origin quickly
        if((dx == -positiveDirection.get(i).X &&
            dvx == -positiveDirection.get(i).X)
           || (dz == -positiveDirection.get(i).Z &&
            dvz == -positiveDirection.get(i).Z))
        {
           vx*=0.8f;
           vz*=0.8f;
        }

        //final velocity
        v.get(i).X = vx;
        v.get(i).Z = vz;
    }
}


//get the direction of the velocity of currentPoints in each wave
//curve
public void getPlusVelocity(List<Points> neighborPoints,
             List<Points> inNumbers, List<Points> currentPoints,
             float originposition[][][],
             float currentposition[][][], List<Vel> v)
{
    float a=0, b=0;
    for(int i=0; i<currentPoints.size(); i++)
    {
        int m = inNumbers.get(i).X;
        //the direction of current position compare to origin
        a = originposition[neighborPoints.get(m).X]
                          [neighborPoints.get(m).Z][0]
           - currentposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0];
        b = originposition[neighborPoints.get(m).X]
                          [neighborPoints.get(m).Z][2]
           - currentposition[currentPoints.get(i).X]
                          [currentPoints.get(i).Z][2];
        v.get(i).X=0.5f*a;
        v.get(i).Z=0.5f*b;
    }
}


//get the direction of the original velocity of currentPoints in
//each wave curve
public void getOriginalVelocity(List<Points> neighborPoints,
         List<Points> currentPoints, float originposition[][][],
         List<Vel> v, List<Vel> v0,
         List<Points> positiveDirection)
{
    List< Points > point = new ArrayList< Points >();
    float a=0, b=0;
```

```
    int m=0;
    for(int i=0; i<currentPoints.size(); i++)
    {
       int X=0, Z=0;
       point.add(neighborPoints.get(0));
       for(int j=1; j<neighborPoints.size(); j++)
       {
          //find the nearest vertex
          if(anotherGetDistance(
                      originposition[currentPoints.get(i).X]
                                    [currentPoints.get(i).Z]
                                    [0],
                      originposition[currentPoints.get(i).X]
                                    [currentPoints.get(i).Z]
                                    [2],
                      originposition[neighborPoints.get(j).X]
                                    [neighborPoints.get(j).Z]
                                    [0],
                      originposition[neighborPoints.get(j).X]
                                    [neighborPoints.get(j).Z]
                                    [2])
          <= anotherGetDistance(
                        originposition[currentPoints.get(i).X]
                                      [currentPoints.get(i).Z]
                                      [0],
                        originposition[currentPoints.get(i).X]
                                      [currentPoints.get(i).Z]
                                      [2],
                        originposition[point.get(0).X]
                                      [point.get(0).Z][0],
                        originposition[point.get(0).X]
                                      [point.get(0).Z][2]))
          //get the nearest vertex’s index
          {
             point.add(0, neighborPoints.get(j));
             m=j;
          }
       }
       point.clear(); //clear list
       //the direction of current position compare to
       //nearest vertex
       a = originposition[neighborPoints.get(m).X]
                         [neighborPoints.get(m).Z][0]
          -originposition[currentPoints.get(i).X]
                         [currentPoints.get(i).Z][0];
       b = originposition[neighborPoints.get(m).X]
                         [neighborPoints.get(m).Z][2]
          -originposition[currentPoints.get(i).X]
                         [currentPoints.get(i).Z][2];
       v.add(new Vel(a*0.1f, b*0.1f )); //first velocity
       v0.add(new Vel(a*0.1f, b*0.1f )); //original velocity
       //get positiveDirection for current point
       if(a > 0)     X = 1;
       else if(a < 0) X = -1;
       if(b > 0)     Z = 1;
```

```
        else if(b < 0) Z = -1;
        positiveDirection.add(new Points(X, Z));
    }
}


//get nearest outside points for each wave curve
public void getNearestOutPoints(List<Points> currentPoints,
                                List<Points> outPoints,
                                float originposition[][][],
                                List<Points> outNumbers)
{
    List< Points > point = new ArrayList< Points >();
    List< Numbers > m = new ArrayList< Numbers >();
    for(int i=0; i<currentPoints.size(); i++)
    {
        point.add(outPoints.get(0));
        point.add(outPoints.get(1));
        m.add(0, new Numbers(0));
        for(int j=1; j<outPoints.size(); j++)
        {
            //find the nearest vertex
            if(anotherGetDistance(
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0],
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [2],
                     originposition[outPoints.get(j).X]
                                   [outPoints.get(j).Z][0],
                     originposition[outPoints.get(j).X]
                                   [outPoints.get(j).Z][2])
                <=anotherGetDistance(
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0],
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [2],
                     originposition[point.get(0).X]
                                   [point.get(0).Z][0],
                     originposition[point.get(0).X]
                                   [point.get(0).Z][2]))
            //get the nearest vertex's index
            {
                point.add(0, outPoints.get(j));
                m.add(0, new Numbers(j));
            }
        }
        //check if there are two nearest vertices
        if(anotherGetDistance(
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0],
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [2],
                     originposition[point.get(1).X]
                                   [point.get(1).Z][0],
                     originposition[point.get(1).X]
                                   [point.get(1).Z][2])
            == anotherGetDistance(
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0],
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [2],
                     originposition[point.get(0).X]
                                   [point.get(0).Z][0],
                     originposition[point.get(0).X]
                                   [point.get(0).Z][2]))
        {
            //put two nearest vertices in the list
            outNumbers.add(new Points(m.get(0).N, m.get(1).N));
        }
        else
        {
            //put one nearest vertex in the list
            outNumbers.add(new Points(m.get(0).N, -1));
        }
        point.clear(); //clear list
        m.clear(); //clear index list
    }
}


//get nearest inside points for each wave curve
public void getNearestInsidePoints(List<Points> currentPoints,
                                   List<Points> insidePoints,
                                   float originposition[][][],
                                   List<Points> insideNumbers)
{
    List< Points > point = new ArrayList< Points >();
    List< Numbers > m = new ArrayList< Numbers >();
    for(int i=0; i<currentPoints.size(); i++)
    {
        point.add(insidePoints.get(0));
        point.add(insidePoints.get(1));
        m.add(0, new Numbers(0));
        for(int j=1; j<insidePoints.size(); j++)
        {
            //find the nearest vertex
            if(anotherGetDistance(
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
                                   [0],
                     originposition[currentPoints.get(i).X]
                                   [currentPoints.get(i).Z]
```

```
                               [2],
                 originposition[insidePoints.get(j).X]
                               [insidePoints.get(j).Z]
                               [0],
                 originposition[insidePoints.get(j).X]
                               [insidePoints.get(j).Z]
                               [2])
      <=anotherGetDistance(
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [0],
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [2],
                 originposition[point.get(0).X]
                               [point.get(0).Z][0],
                 originposition[point.get(0).X]
                               [point.get(0).Z][2]))
   {
      //get the nearest vertex’s index
      point.add(0, insidePoints.get(j));
      m.add(0, new Numbers(j));
   }
}
//check if there are two nearest vertices
if(anotherGetDistance(
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [0],
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [2],
                 originposition[point.get(1).X]
                               [point.get(1).Z][0],
                 originposition[point.get(1).X]
                               [point.get(1).Z][2])
   == anotherGetDistance(
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [0],
                 originposition[currentPoints.get(i).X]
                               [currentPoints.get(i).Z]
                               [2],
                           originposition[point.get(0).X]
                               [point.get(0).Z][0],
                 originposition[point.get(0).X]
                               [point.get(0).Z][2]))
{
   //put two nearest vertices in the list
   insideNumbers.add(new Points(m.get(0).N,
                                m.get(1).N));
}
else
{
   //put one nearest vertex in the list
```

```
             insideNumbers.add(new Points(m.get(0).N, -1));
          }
          point.clear(); //clear list
          m.clear();//clear index list
      }
   }


   //get the distance of two points by input point’s list
   private float getDistance(Points p1, Points p2)
   {
      return  (float)Math.sqrt((float)Math.pow(p2.Z-p1.Z, 2)
                              +(float)Math.pow(p2.X-p1.X, 2));
   }


   //get the distance of two points by input x and z value
   private float anotherGetDistance(float x1, float z1, float x2,
                                    float z2)
   {
      return  (float)Math.sqrt((float)Math.pow(z2-z1, 2)
                              +(float)Math.pow(x2-x1, 2));
   }


   //get the distance of two line segments
   private float distanceBetweenPointAndLine(float x, float z,
                                             float x1, float z1,
                                             float x2, float z2)
   {
      return  (float)Math.abs((z2-z1)*x - (x2-x1)*z + z1*x2 - z2*x1)
                           *(1/anotherGetDistance(x1, z1, x2, z2));
   }


   //get square root
   private float sumOfXAndZ(float x, float z)
   {
      return  (float)Math.sqrt((float)Math.pow(x, 2)
                              +(float)Math.pow(z, 2));
   }
}
```

```
/* Collision.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Process water/land collisions, and the collisions between waves.
 * See section 5.6.2 for details.
*/

package demos;

import java.util.List;
import java.util.ArrayList;

public class Collision
{
   private float t=0.1f;
   //deal with the collision for wave curves. The vertex of outside
   //wave curve is a, the vertices of inside wave curve are b, c, d.
   //the wave spring is between a and c, and a will hit line segment
   //bc or cd.
   public void getVelocityForWaveCurves(List<Points> currentPoints,
         float Currentposition[][][],  List<Points> neighborPoints,
         List<Vel> velocitya, List<Vel> copyOfVelocitya,
         List<Vel> velocitybcd, List<Vel> copyOfVelocitybcd,
         List<Points> outsideNearestVertex,
         float originalposition[][][],List<Vel> originalVelocitya,
         List<Vel> originalVelocitybcd, List<Points> hitPoint,
         List<Points> insideNearestVertex)
   {
      int o;
      float x, z, x1, z1, xo, zo, intelx=0, intelz=0, longx, longz;
      boolean interaction;

      for(int i=0; i<currentPoints.size(); i++)
      {
         interaction=false;
         boolean hit = false;
         //current position of current point a
         x=Currentposition[currentPoints.get(i).X]
                          [currentPoints.get(i).Z][0];
         z=Currentposition[currentPoints.get(i).X]
                          [currentPoints.get(i).Z][2];
         //next position of current point a
         x1=x + t*velocitya.get(i).X;
         z1=z + t*velocitya.get(i).Z;
         longx = x + (x - x1)*100;
         longz = z + (z - z1)*100;

         //has one inside nearest vertex
```

```
if(insideNearestVertex.get(i).Z == -1)
{
   //current position of the point which will be hit by
   //current point
   o = insideNearestVertex.get(i).X;
   xo = Currentposition[hitPoint.get(o).X]
                       [hitPoint.get(o).Z][0];
   zo = Currentposition[hitPoint.get(o).X]
                       [hitPoint.get(o).Z][2];
   //hitting: if the wave curve hit the bank, let it come
   //back to its origin
   if(anotherGetDistance(xo, zo, x, z)<=0.5f)
   {
      float ox, oz;
      ox = originalposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0]-x;
      oz = originalposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][2]-z;
      velocitya.get(i).X = 0.2f*ox;
      velocitya.get(i).Z = 0.2f*oz;
      hit=true;
   }
}
//has two inside nearest vertices
else
{
   //current position of the point which will be hit by
   //current point
   o = insideNearestVertex.get(i).Z;
   xo = Currentposition[hitPoint.get(o).X]
                       [hitPoint.get(o).Z][0];
   zo = Currentposition[hitPoint.get(o).X]
                       [hitPoint.get(o).Z][2];
   //hitting: if the wave curve hit the bank, let it come
   //back to its origin
   if(anotherGetDistance(xo, zo, x, z)<=0.5f)
   {
      float ox, oz;
      ox = originalposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][0]-x;
      oz = originalposition[currentPoints.get(i).X]
                           [currentPoints.get(i).Z][2]-z;
      velocitya.get(i).X = 0.2f*ox;
      velocitya.get(i).Z = 0.2f*oz;
      hit=true;
   }
}

//has one outside nearest vertex
if(outsideNearestVertex.get(i).Z == -1)
{
   int b, c, d;
   float cx, cz, bx, bz, dx, dz, cx1, cz1, bx1, bz1, dx1,
        dz1;
   c = outsideNearestVertex.get(i).X;
```

```
//outside vertex’s left vertex
b=c-1; if(c==0) b=neighborPoints.size()-1;
//outside vertex’s right vertex
d=c+1; if(c==neighborPoints.size()-1) d=0;
//current position of the point c which will be hit by
//current point
cx=Currentposition[neighborPoints.get(c).X]
                  [neighborPoints.get(c).Z][0];
cz=Currentposition[neighborPoints.get(c).X]
                  [neighborPoints.get(c).Z][2];
//next position of point c
cx1=cx+ t*velocitybcd.get(c).X;
cz1=cz+ t*velocitybcd.get(c).Z;
//current position of point b beside c
bx=Currentposition[neighborPoints.get(b).X]
                  [neighborPoints.get(b).Z][0];
bz=Currentposition[neighborPoints.get(b).X]
                  [neighborPoints.get(b).Z][2];
//next position of point b
bx1=bx+ t*velocitybcd.get(b).X;
bz1=bz+ t*velocitybcd.get(b).Z;
//current position of point d beside c
dx=Currentposition[neighborPoints.get(d).X]
                  [neighborPoints.get(d).Z][0];
dz=Currentposition[neighborPoints.get(d).X]
                  [neighborPoints.get(d).Z][2];
//next position of point d
dx1=dx+ t*velocitybcd.get(d).X;
dz1=dz+ t*velocitybcd.get(d).Z;

//only waves collision
if(hit == false)
{
   //the interaction of a and bc happen
   if(TwoLineIsIntersect(longx, longz, x1, z1, cx1,
                cz1, bx1, bz1, intelx, intelz)==true)
   {
      interaction=true;
      float newVx , newVz ;

      //velocitices of b and c are not changed in X axis
      if(velocitybcd.get(c).X ==
        copyOfVelocitybcd.get(c).X
        && velocitybcd.get(b).X ==
        copyOfVelocitybcd.get(b).X)
      {
         //average velocity in X axis
         newVx = (velocitya.get(i).X
                 + velocitybcd.get(c).X
                 + velocitybcd.get(b).X)*0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
         else
            velocitya.get(i).X = newVx;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(c).X==0)
            velocitybcd.get(c).X = 0;
         else
            velocitybcd.get(c).X = newVx;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(b).X==0)
            velocitybcd.get(b).X = 0;
         else
            velocitybcd.get(b).X = newVx;
      }
      //velocity of b or c be changed in X axis
      else
      {
         //average velocity
         newVx = (velocitya.get(i).X
                 + copyOfVelocitybcd.get(c).X
                 + copyOfVelocitybcd.get(b).X)
                 *0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
         else
            velocitya.get(i).X = newVx;
         //if original velocity is 0, keep it, or update
         //it if the new velocity is bigger
         if(originalVelocitybcd.get(c).X==0)
            velocitybcd.get(c).X = 0;
         else if(Math.abs(newVx)
                 > Math.abs(velocitybcd.get(c).X))
            velocitybcd.get(c).X = newVx;
         //if original velocity is 0, keep it, or update
         //it if the new velocity is bigger
         if(originalVelocitybcd.get(b).X==0)
            velocitybcd.get(b).X = 0;
         else if(Math.abs(newVx)
                 > Math.abs(velocitybcd.get(b).X))
            velocitybcd.get(b).X = newVx;
      }

      //velocitices of b and c are not changed in Z axis
      if(velocitybcd.get(c).Z
         == copyOfVelocitybcd.get(c).Z
         && velocitybcd.get(b).Z
         == copyOfVelocitybcd.get(b).Z)
      {
         //average velocity in X axis
         newVz = (velocitya.get(i).Z
                 + velocitybcd.get(c).Z
                 + velocitybcd.get(b).Z)*0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).Z==0)
            velocitya.get(i).Z = 0;
         else
            velocitya.get(i).Z = newVz;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(c).Z==0)
               velocitybcd.get(c).Z = 0;
            else
              velocitybcd.get(c).Z = newVz;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(b).Z==0)
               velocitybcd.get(b).Z = 0;
            else
               velocitybcd.get(b).Z = newVz;
         }
         //velocity of b or c be changed in Z axis
         else
         {
            //average velocity
            newVz = (velocitya.get(i).Z
                    + copyOfVelocitybcd.get(c).Z
                    + copyOfVelocitybcd.get(b).Z)
                    *0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).Z==0)
               velocitya.get(i).Z = 0;
            else
               velocitya.get(i).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(c).Z==0)
               velocitybcd.get(c).Z = 0;
            else if(Math.abs(newVz)
                   > Math.abs(velocitybcd.get(c).Z))
               velocitybcd.get(c).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(b).Z==0)
               velocitybcd.get(b).Z = 0;
            else if(Math.abs(newVz)
                   > Math.abs(velocitybcd.get(b).Z))
               velocitybcd.get(b).Z = newVz;
         }
      }
      //the interaction of a and cd happens
      else if(TwoLineIsIntersect(longx, longz, x1, z1,
             cx1, cz1, dx1, dz1, intelx, intelz)==true)
      {
         interaction=true;
         float newVx, newVz;

         //velocitices of c and d are not changed in X axis
         if(velocitybcd.get(c).X
            == copyOfVelocitybcd.get(c).X
            && velocitybcd.get(d).X
            == copyOfVelocitybcd.get(d).X)
         {
            //average velocity
            newVx = (velocitya.get(i).X
                    + velocitybcd.get(c).X
                    + velocitybcd.get(d).X)*0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).X==0)
               velocitya.get(i).X = 0;
            else
               velocitya.get(i).X = newVx;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(c).X==0)
               velocitybcd.get(c).X = 0;
            else
               velocitybcd.get(c).X = newVx;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(d).X==0)
               velocitybcd.get(d).X = 0;
            else
               velocitybcd.get(d).X = newVx;
         }
         //velocity of c or d be changed in X axis
         else
         {
            //average velocity
            newVx = (velocitya.get(i).X
                    + copyOfVelocitybcd.get(c).X
                    + copyOfVelocitybcd.get(d).X)
                      *0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).X==0)
               velocitya.get(i).X = 0;
            else
               velocitya.get(i).X = newVx;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(c).X==0)
               velocitybcd.get(c).X = 0;
            else if(Math.abs(newVx)
                   > Math.abs(velocitybcd.get(c).X))
               velocitybcd.get(c).X = newVx;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(d).X==0)
               velocitybcd.get(d).X = 0;
            else if(Math.abs(newVx)
                   > Math.abs(velocitybcd.get(d).X))
               velocitybcd.get(d).X = newVx;
         }

         //velocitices of c and d are not changed in Z axis
         if(velocitybcd.get(c).Z
           == copyOfVelocitybcd.get(c).Z
           && velocitybcd.get(d).Z
           == copyOfVelocitybcd.get(d).Z)
         {
            //average velocity
            newVz = (velocitya.get(i).Z
                    + velocitybcd.get(c).Z
                    + velocitybcd.get(d).Z)*0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).Z==0)
               velocitya.get(i).Z = 0;
            else
               velocitya.get(i).Z = newVz;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(c).Z==0)
               velocitybcd.get(c).Z = 0;
            else
               velocitybcd.get(c).Z = newVz;
            //if original velocity is 0, keep it, or update
            if(originalVelocitybcd.get(d).Z==0)
               velocitybcd.get(d).Z = 0;
            else
               velocitybcd.get(d).Z = newVz;
         }
         //velocity of c or d be changed in Z axis
         else
         {
            //average velocity
            newVz = (velocitya.get(i).Z
                    + copyOfVelocitybcd.get(c).Z
                    + copyOfVelocitybcd.get(d).Z)
                      *0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).Z==0)
               velocitya.get(i).Z = 0;
            else
               velocitya.get(i).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(c).Z==0)
               velocitybcd.get(c).Z = 0;
            else if(Math.abs(newVz)
                    > Math.abs(velocitybcd.get(c).Z))
               velocitybcd.get(c).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(d).Z==0)
               velocitybcd.get(d).Z = 0;
            else if(Math.abs(newVz)
                    > Math.abs(velocitybcd.get(d).Z))
               velocitybcd.get(d).Z = newVz;
         }
      }
   }
   ////////////////////////////////////////////////////
   //water/land and waves collisions happen at the same time
   else
   {
      //the interaction of a and bc happens
      if(TwoLineIsIntersect(longx, longz, x1, z1, cx1,
         cz1, bx1, bz1, intelx, intelz)==true)
      {
         interaction=true;
         //velocitices of c and b are not changed in X axis
         if(velocitybcd.get(c).X
            == copyOfVelocitybcd.get(c).X
            && velocitybcd.get(b).X
            == copyOfVelocitybcd.get(b).X)
         {
            //update to vertex a's velocity
            velocitybcd.get(c).X = velocitya.get(i).X;
            velocitybcd.get(b).X = velocitya.get(i).X;
         }
         //velocity of c or b be changed in X axis
         else
         {
            //keep the velocity if it is bigger than a's,
            //or update to a's velocity
            if(Math.abs(velocitya.get(i).X)
                     > Math.abs(velocitybcd.get(c).X))
               velocitybcd.get(c).X
                                 = velocitya.get(i).X;
            if(Math.abs(velocitya.get(i).X)
                     > Math.abs(velocitybcd.get(b).X))
               velocitybcd.get(b).X
                                 = velocitya.get(i).X;
         }
         //velocitices of c and b are not changed in Z axis
         if(velocitybcd.get(c).Z
            == copyOfVelocitybcd.get(c).Z
            && velocitybcd.get(b).Z
            == copyOfVelocitybcd.get(b).Z)
         {
            //update to vertex a's velocity
            velocitybcd.get(c).Z = velocitya.get(i).Z;
            velocitybcd.get(b).Z = velocitya.get(i).Z;
         }
         //velocity of c or b be changed in Z axis
         else
         {
            //keep the velocity if it is bigger than a's,
            //or update to a's velocity
            if(Math.abs(velocitya.get(i).Z)
               > Math.abs(velocitybcd.get(c).Z))
               velocitybcd.get(c).Z
                                 = velocitya.get(i).Z;
            if(Math.abs(velocitya.get(i).Z)
               > Math.abs(velocitybcd.get(b).Z))
               velocitybcd.get(b).Z
                                 = velocitya.get(i).Z;
         }
      }
      //the interaction of a and cd happens
      else if(TwoLineIsIntersect(longx, longz, x1, z1,
               cx1, cz1, dx1, dz1, intelx, intelz)==true)
      {
          interaction=true;

          //velocitices of c and d are not changed in X axis
          if(velocitybcd.get(c).X
             == copyOfVelocitybcd.get(c).X
             && velocitybcd.get(d).X
             == copyOfVelocitybcd.get(d).X)
          {
             //update to vertex a's velocity
             velocitybcd.get(c).X = velocitya.get(i).X;
             velocitybcd.get(d).X = velocitya.get(i).X;
          }
          //velocity of c or d be changed in X axis
          else
          {
             //keep the velocity if it is bigger than a's,
             //or update to a's velocity
             if(Math.abs(velocitya.get(i).X)
                > Math.abs(velocitybcd.get(c).X))
                velocitybcd.get(c).X
                                   = velocitya.get(i).X;
             if(Math.abs(velocitya.get(i).X)
                > Math.abs(velocitybcd.get(d).X))
                velocitybcd.get(d).X
                                   = velocitya.get(i).X;
          }
          //velocitices of c and d are not changed in Z axis
          if(velocitybcd.get(c).Z
             == copyOfVelocitybcd.get(c).Z
             && velocitybcd.get(d).Z
             == copyOfVelocitybcd.get(d).Z)
          {
             //update to vertex a's velocity
             velocitybcd.get(c).Z = velocitya.get(i).Z;
             velocitybcd.get(d).Z = velocitya.get(i).Z;
          }
          //velocity of c or d be changed in Z axis
          else
          {
             //keep the velocity if it is bigger than a's,
             //or update to a's velocity
             if(Math.abs(velocitya.get(i).Z)
                > Math.abs(velocitybcd.get(c).Z))
                velocitybcd.get(c).Z
                                   = velocitya.get(i).Z;
             if(Math.abs(velocitya.get(i).Z)
                > Math.abs(velocitybcd.get(d).Z))
                velocitybcd.get(d).Z
                                   = velocitya.get(i).Z;
          }
       }
    }
 }
  /////////////////////////////////////////////////////////
 //has two outside nearest vertices
```

```
else
{
   int b, c1, c2, d;
   float c1x, c1z, c2x, c2z, bx, bz, dx, dz, c1x1, c1z1,
         c2x1, c2z1, bx1, bz1, dx1, dz1;
   //index of two outsider vertices
   c1 = outsideNearestVertex.get(i).X;
   c2 = outsideNearestVertex.get(i).Z;
   c1 = Math.min(c1, c2);
   c2 = c1+1;
   //the vertex before c1
   b=c1-1; if(c1==0) b=neighborPoints.size()-1;
   //the vertex after c2
   d=c2+1; if(c2==neighborPoints.size()-1) d=0;
   //current position of the point c1 which will be hit by
   //current point
   c1x=Currentposition[neighborPoints.get(c1).X]
                      [neighborPoints.get(c1).Z][0];
   c1z=Currentposition[neighborPoints.get(c1).X]
                      [neighborPoints.get(c1).Z][2];
   //next position of point c1
   c1x1=c1x+ t*velocitybcd.get(c1).X;
   c1z1=c1z+ t*velocitybcd.get(c1).Z;
   //current position of the point c2 which will be hit by
   //current point
   c2x=Currentposition[neighborPoints.get(c2).X]
                      [neighborPoints.get(c2).Z][0];
   c2z=Currentposition[neighborPoints.get(c2).X]
                      [neighborPoints.get(c2).Z][2];
   //next position of point c2
   c2x1=c2x+ t*velocitybcd.get(c2).X;
   c2z1=c2z+ t*velocitybcd.get(c2).Z;
   //current position of point b beside c
   bx=Currentposition[neighborPoints.get(b).X]
                     [neighborPoints.get(b).Z][0];
   bz=Currentposition[neighborPoints.get(b).X]
                     [neighborPoints.get(b).Z][2];
   //next position of point b
   bx1=bx+ t*velocitybcd.get(b).X;
   bz1=bz+ t*velocitybcd.get(b).Z;
   //current position of point d beside c
   dx=Currentposition[neighborPoints.get(d).X]
                     [neighborPoints.get(d).Z][0];
   dz=Currentposition[neighborPoints.get(d).X]
                     [neighborPoints.get(d).Z][2];
   //next position of point d
   dx1=dx+ t*velocitybcd.get(d).X;
   dz1=dz+ t*velocitybcd.get(d).Z;

   //only waves collision
   if(hit == false)
   {
      //the interaction of a and bc1 happens
      if(TwoLineIsIntersect(longx, longz, x1, z1, c1x1,
                   c1z1, bx1, bz1, intelx, intelz)==true)
```

```
{
   interaction=true;
   float newVx, newVz;

   //velocitices of c1 and b are not changed in X
   //axis
   if(velocitybcd.get(c1).X
      == copyOfVelocitybcd.get(c1).X
      && velocitybcd.get(b).X
      == copyOfVelocitybcd.get(b).X)
   {
      //average velocity
      newVx = (velocitya.get(i).X
              + velocitybcd.get(c1).X
              + velocitybcd.get(b).X)*0.3333f;
      //if original velocity is 0, keep it, or update
      if(originalVelocitya.get(i).X==0)
         velocitya.get(i).X = 0;
      else
         velocitya.get(i).X = newVx;
      //if original velocity is 0, keep it, or update
      if(originalVelocitybcd.get(c1).X==0)
         velocitybcd.get(c1).X = 0;
      else
         velocitybcd.get(c1).X = newVx;
      //if original velocity is 0, keep it, or update
      if(originalVelocitybcd.get(b).X==0)
         velocitybcd.get(b).X = 0;
      else
         velocitybcd.get(b).X = newVx;
   }
   //velocity of c1 or b be changed in X axis
   else
   {
      //average velocity
      newVx = (velocitya.get(i).X
              + copyOfVelocitybcd.get(c1).X
              + copyOfVelocitybcd.get(b).X)
                *0.3333f;
      //if original velocity is 0, keep it, or update
      if(originalVelocitya.get(i).X==0)
         velocitya.get(i).X = 0;
      else
         velocitya.get(i).X = newVx;
      //if original velocity is 0, keep it, or update
      //it if the new velocity is bigger
      if(originalVelocitybcd.get(c1).X==0)
         velocitybcd.get(c1).X = 0;
      else if(Math.abs(newVx)
             > Math.abs(velocitybcd.get(c1).X))
         velocitybcd.get(c1).X = newVx;
      //if original velocity is 0, keep it, or update
      //it if the new velocity is bigger
      if(originalVelocitybcd.get(b).X==0)
         velocitybcd.get(b).X = 0;
```

```
      else if(Math.abs(newVx)
             > Math.abs(velocitybcd.get(b).X))
         velocitybcd.get(b).X = newVx;
   }
   //velocitices of c1 and b are not changed in Z
   //axis
   if(velocitybcd.get(c1).Z
      == copyOfVelocitybcd.get(c1).Z
      && velocitybcd.get(b).Z
      == copyOfVelocitybcd.get(b).Z)
   {
      //average velocity
      newVz = (velocitya.get(i).Z
              + velocitybcd.get(c1).Z
              + velocitybcd.get(b).Z)*0.3333f;
      //if original velocity is 0, keep it, or update
      if(originalVelocitya.get(i).Z==0)
         velocitya.get(i).Z = 0;
      else
         velocitya.get(i).Z = newVz;
      //if original velocity is 0, keep it, or update
      if(originalVelocitybcd.get(c1).Z==0)
         velocitybcd.get(c1).Z = 0;
      else
         velocitybcd.get(c1).Z = newVz;
      //if original velocity is 0, keep it, or update
      if(originalVelocitybcd.get(b).Z==0)
         velocitybcd.get(b).Z = 0;
      else
         velocitybcd.get(b).Z = newVz;
   }
   //velocity of c1 or b be changed in Z axis
   else
   {
      //average velocity
      newVz = (velocitya.get(i).Z
              + copyOfVelocitybcd.get(c1).Z
              + copyOfVelocitybcd.get(b).Z)
                *0.3333f;
      //if original velocity is 0, keep it, or update
      if(originalVelocitya.get(i).Z==0)
         velocitya.get(i).Z = 0;
      else
         velocitya.get(i).Z = newVz;
      //if original velocity is 0, keep it, or update
      //it if the new velocity is bigger
      if(originalVelocitybcd.get(c1).Z==0)
         velocitybcd.get(c1).Z = 0;
      else if(Math.abs(newVz)
             > Math.abs(velocitybcd.get(c1).Z))
         velocitybcd.get(c1).Z = newVz;
      //if original velocity is 0, keep it, or update
      //it if the new velocity is bigger
      if(originalVelocitybcd.get(b).Z==0)
         velocitybcd.get(b).Z = 0;
```

```
        else if(Math.abs(newVz)
                > Math.abs(velocitybcd.get(b).Z))
            velocitybcd.get(b).Z = newVz;
    }
}
//the interaction of a and c1c2 happens
else if(TwoLineIsIntersect(longx, longz, x1, z1,
      c1x1, c1z1, c2x1, c2z1, intelx, intelz)==true)
{
    interaction=true;
    float newVx , newVz ;

    //velocitices of c1 and c2 are not changed in X
    //axis
    if(velocitybcd.get(c1).X
        == copyOfVelocitybcd.get(c1).X
        && velocitybcd.get(c2).X
        == copyOfVelocitybcd.get(c2).X)
    {
        //average velocity
        newVx = (velocitya.get(i).X
                + velocitybcd.get(c1).X
                + velocitybcd.get(c2).X)*0.3333f;
        //if original velocity is 0, keep it, or update
        if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
        else
            velocitya.get(i).X = newVx;
        //if original velocity is 0, keep it, or update
        if(originalVelocitybcd.get(c1).X==0)
            velocitybcd.get(c1).X = 0;
        else
            velocitybcd.get(c1).X = newVx;
        //if original velocity is 0, keep it, or update
        if(originalVelocitybcd.get(c2).X==0)
            velocitybcd.get(c2).X = 0;
        else
            velocitybcd.get(c2).X = newVx;
    }
    //velocity of c1 or c2 be changed in X axis
    else
    {
        //average velocity
        newVx = (velocitya.get(i).X
                 + copyOfVelocitybcd.get(c1).X
                 + copyOfVelocitybcd.get(c2).X)
                   *0.3333f;
        //if original velocity is 0, keep it, or update
        if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
        else
            velocitya.get(i).X = newVx;
        //if original velocity is 0, keep it, or update
        //it if the new velocity is bigger
        if(originalVelocitybcd.get(c1).X==0)
            velocitybcd.get(c1).X = 0;
        else if(Math.abs(newVx)
                > Math.abs(velocitybcd.get(c1).X))
            velocitybcd.get(c1).X = newVx;
        //if original velocity is 0, keep it, or update
        //it if the new velocity is bigger
        if(originalVelocitybcd.get(c2).X==0)
            velocitybcd.get(c2).X = 0;
        else if(Math.abs(newVx)
                > Math.abs(velocitybcd.get(c2).X))
            velocitybcd.get(c2).X = newVx;
    }
    //velocitices of c1 and c2 are not changed in Z
    //axis
    if(velocitybcd.get(c1).Z
        == copyOfVelocitybcd.get(c1).Z
        && velocitybcd.get(c2).Z
        == copyOfVelocitybcd.get(c2).Z)
    {
        //average velocity
        newVz = (velocitya.get(i).Z
                + velocitybcd.get(c1).Z
                + velocitybcd.get(c2).Z)*0.3333f;
        //if original velocity is 0, keep it, or update
        if(originalVelocitya.get(i).Z==0)
            velocitya.get(i).Z = 0;
        else
            velocitya.get(i).Z = newVz;
        //if original velocity is 0, keep it, or update
        if(originalVelocitybcd.get(c1).Z==0)
            velocitybcd.get(c1).Z = 0;
        else
            velocitybcd.get(c1).Z = newVz;
        //if original velocity is 0, keep it, or update
        if(originalVelocitybcd.get(c2).Z==0)
            velocitybcd.get(c2).Z = 0;
        else
            velocitybcd.get(c2).Z = newVz;
    }
    //velocity of c1 or c2 be changed in Z axis
    else
    {
        //average velocity
        newVz = (velocitya.get(i).Z
                 + copyOfVelocitybcd.get(c1).Z
                 + copyOfVelocitybcd.get(c2).Z)
                   *0.3333f;
        //if original velocity is 0, keep it, or update
        if(originalVelocitya.get(i).Z==0)
            velocitya.get(i).Z = 0;
        else
            velocitya.get(i).Z = newVz;
        //if original velocity is 0, keep it, or update
        //it if the new velocity is bigger
        if(originalVelocitybcd.get(c1).Z==0)
```

```
            velocitybcd.get(c1).Z = 0;
         else if(Math.abs(newVz)
                > Math.abs(velocitybcd.get(c1).Z))
            velocitybcd.get(c1).Z = newVz;
         //if original velocity is 0, keep it, or update
         //it if the new velocity is bigger
         if(originalVelocitybcd.get(c2).Z==0)
            velocitybcd.get(c2).Z = 0;
         else if(Math.abs(newVz)
                > Math.abs(velocitybcd.get(c2).Z))
            velocitybcd.get(c2).Z = newVz;
      }
   }
   //the interaction of a and c2d happens
   else if(TwoLineIsIntersect(longx, longz, x1, z1,
         c2x1, c2z1, dx1, dz1, intelx, intelz)==true)
   {
      interaction=true;
      float newVx , newVz ;

      //velocitices of c2 and d are not changed in X
      //axis
      if(velocitybcd.get(c2).X
         == copyOfVelocitybcd.get(c2).X
         && velocitybcd.get(d).X
         == copyOfVelocitybcd.get(d).X)
      {
         //average velocity
         newVx = (velocitya.get(i).X
                  + velocitybcd.get(c2).X
                  + velocitybcd.get(d).X)*0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
         else
            velocitya.get(i).X = newVx;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(c2).X==0)
            velocitybcd.get(c2).X = 0;
         else
            velocitybcd.get(c2).X = newVx;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(d).X==0)
            velocitybcd.get(d).X = 0;
         else
            velocitybcd.get(d).X = newVx;
      }
      //velocity of c2 or d be changed in X axis
      else
      {
         //average velocity
         newVx = (velocitya.get(i).X
                  + copyOfVelocitybcd.get(c2).X
                  + copyOfVelocitybcd.get(d).X)
                    *0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).X==0)
            velocitya.get(i).X = 0;
         else
            velocitya.get(i).X = newVx;
         //if original velocity is 0, keep it, or update
         //it if the new velocity is bigger
         if(originalVelocitybcd.get(c2).X==0)
            velocitybcd.get(c2).X = 0;
         else if(Math.abs(newVx)
                > Math.abs(velocitybcd.get(c2).X))
            velocitybcd.get(c2).X = newVx;
         //if original velocity is 0, keep it, or update
         //it if the new velocity is bigger
         if(originalVelocitybcd.get(d).X==0)
            velocitybcd.get(d).X = 0;
         else if(Math.abs(newVx)
                > Math.abs(velocitybcd.get(d).X))
            velocitybcd.get(d).X = newVx;
      }

      //velocitices of c2 and d are not changed in Z
      //axis
      if(velocitybcd.get(c2).Z
         == copyOfVelocitybcd.get(c2).Z
         && velocitybcd.get(d).Z
         == copyOfVelocitybcd.get(d).Z)
      {
         //average velocity
         newVz = (velocitya.get(i).Z
                  + velocitybcd.get(c2).Z
                  + velocitybcd.get(d).Z)*0.3333f;
         //if original velocity is 0, keep it, or update
         if(originalVelocitya.get(i).Z==0)
            velocitya.get(i).Z = 0;
         else
            velocitya.get(i).Z = newVz;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(c2).Z==0)
            velocitybcd.get(c2).Z = 0;
         else
            velocitybcd.get(c2).Z = newVz;
         //if original velocity is 0, keep it, or update
         if(originalVelocitybcd.get(d).Z==0)
            velocitybcd.get(d).Z = 0;
         else
            velocitybcd.get(d).Z = newVz;
      }
      //velocity of c2 or d be changed in Z axis
      else
      {
         //average velocity
         newVz = (velocitya.get(i).Z
                  + copyOfVelocitybcd.get(c2).Z
                  + copyOfVelocitybcd.get(d).Z)
```

```
                    *0.3333f;
            //if original velocity is 0, keep it, or update
            if(originalVelocitya.get(i).Z==0)
               velocitya.get(i).Z = 0;
            else
               velocitya.get(i).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(c2).Z==0)
               velocitybcd.get(c2).Z = 0;
            else if(Math.abs(newVz)
                   > Math.abs(velocitybcd.get(c2).Z))
               velocitybcd.get(c2).Z = newVz;
            //if original velocity is 0, keep it, or update
            //it if the new velocity is bigger
            if(originalVelocitybcd.get(d).Z==0)
               velocitybcd.get(d).Z = 0;
            else if(Math.abs(newVz)
                   > Math.abs(velocitybcd.get(d).Z))
            velocitybcd.get(d).Z = newVz;
         }
      }
   }
   //////////////////////////////////////////////////
   //water/land and waves collisions at the same time
   else
   {
      //the interaction of a and bc1 happens
      if(TwoLineIsIntersect(longx, longz, x1, z1, c1x1,
                   c1z1, bx1, bz1, intelx, intelz)==true)
      {
         interaction=true;
         //velocitices of c1 and b are not changed in X
         //axis
         if(velocitybcd.get(c1).X
            == copyOfVelocitybcd.get(c1).X
            && velocitybcd.get(b).X
            == copyOfVelocitybcd.get(b).X)
         {
            //update to vertex a's velocity
            velocitybcd.get(c1).X = velocitya.get(i).X;
            velocitybcd.get(b).X = velocitya.get(i).X;
         }
         //velocity of c1 or b be changed in X axis
         else
         {
            //keep the velocity if it is bigger than a's,
            //or update to a's velocity
            if(Math.abs(velocitya.get(i).X)
               > Math.abs(velocitybcd.get(c1).X))
               velocitybcd.get(c1).X
                                  = velocitya.get(i).X;
            if(Math.abs(velocitya.get(i).X)
               > Math.abs(velocitybcd.get(b).X))
               velocitybcd.get(b).X
                                  = velocitya.get(i).X;
         }

         //velocitices of c1 and b are not changed in Z
         //axis
         if(velocitybcd.get(c1).Z
           == copyOfVelocitybcd.get(c1).Z
           && velocitybcd.get(b).Z
           == copyOfVelocitybcd.get(b).Z)
         {
            //update to vertex a's velocity
            velocitybcd.get(c1).Z = velocitya.get(i).Z;
            velocitybcd.get(b).Z = velocitya.get(i).Z;
         }
         //velocity of c1 or b be changed in Z axis
         else
         {
            //keep the velocity if it is bigger than a's,
            //or update to a's velocity
            if(Math.abs(velocitya.get(i).Z)
               > Math.abs(velocitybcd.get(c1).Z))
               velocitybcd.get(c1).Z
                                  = velocitya.get(i).Z;
            if(Math.abs(velocitya.get(i).Z)
               > Math.abs(velocitybcd.get(b).Z))
               velocitybcd.get(b).Z
                                  = velocitya.get(i).Z;
         }
      }

      //the interaction of a and c1c2 happens
      else if(TwoLineIsIntersect(longx, longz, x1,
        z1, c1x1, c1z1, c2x1, c2z1, intelx, intelz)==true)
      {
         interaction=true;

         //velocitices of c1 and c2 are not changed in X
         //axis
         if(velocitybcd.get(c1).X
            == copyOfVelocitybcd.get(c1).X
            && velocitybcd.get(c2).X
            == copyOfVelocitybcd.get(c2).X)
         {
            //update to vertex a's velocity
            velocitybcd.get(c1).X = velocitya.get(i).X;
            velocitybcd.get(c2).X = velocitya.get(i).X;
         }
         //velocity of c1 or c2 be changed in X axis
         else
         {
            //keep the velocity if it is bigger than a's,
            //or update to a's velocity
            if(Math.abs(velocitya.get(i).X)
               > Math.abs(velocitybcd.get(c1).X))
               velocitybcd.get(c1).X
```

```
                              = velocitya.get(i).X;
         if(Math.abs(velocitya.get(i).X)
            > Math.abs(velocitybcd.get(c2).X))
            velocitybcd.get(c2).X
                              = velocitya.get(i).X;
      }

      //velocitices of c1 and c2 are not changed in Z
      //axis
      if(velocitybcd.get(c1).Z
         == copyOfVelocitybcd.get(c1).Z
         && velocitybcd.get(c2).Z
         == copyOfVelocitybcd.get(c2).Z)
      {
         //update to vertex a’s velocity
         velocitybcd.get(c1).Z = velocitya.get(i).Z;
         velocitybcd.get(c2).Z = velocitya.get(i).Z;
      }
      //velocity of c1 or c2 be changed in Z axis
      else
      {
         //keep the velocity if it is bigger than a’s,
         //or update to a’s velocity
         if(Math.abs(velocitya.get(i).Z)
            > Math.abs(velocitybcd.get(c1).Z))
            velocitybcd.get(c1).Z
                              = velocitya.get(i).Z;
         if(Math.abs(velocitya.get(i).Z)
            > Math.abs(velocitybcd.get(c2).Z))
            velocitybcd.get(c2).Z
                              = velocitya.get(i).Z;
      }
   }

   //the interaction of a and c2d happens
   else if(TwoLineIsIntersect(longx, longz, x1, z1,
          c2x1, c2z1, dx1, dz1, intelx, intelz)==true)
   {
      interaction=true;

      //velocitices of c2 and d are not changed in X
      //axis
      if(velocitybcd.get(c2).X
         == copyOfVelocitybcd.get(c2).X
         && velocitybcd.get(d).X
         == copyOfVelocitybcd.get(d).X)
      {
         //update to vertex a’s velocity
         velocitybcd.get(c2).X = velocitya.get(i).X;
         velocitybcd.get(d).X = velocitya.get(i).X;
      }
      //velocity of c2 or d be changed in X axis
      else
      {
         //keep the velocity if it is bigger than a’s,
         //or update to a’s velocity
         if(Math.abs(velocitya.get(i).X)
            > Math.abs(velocitybcd.get(c2).X))
            velocitybcd.get(c2).X
                              = velocitya.get(i).X;
         if(Math.abs(velocitya.get(i).X)
            > Math.abs(velocitybcd.get(d).X))
            velocitybcd.get(d).X
                              = velocitya.get(i).X;
      }
      //velocitices of c2 and d are not changed in Z
      //axis
      if(velocitybcd.get(c2).Z
         == copyOfVelocitybcd.get(c2).Z
         && velocitybcd.get(d).Z
         == copyOfVelocitybcd.get(d).Z)
      {
         //update to vertex a’s velocity
         velocitybcd.get(c2).Z = velocitya.get(i).Z;
         velocitybcd.get(d).Z = velocitya.get(i).Z;
      }
      //velocity of c2 or d be changed in Z axis
      else
      {
         //keep the velocity if it is bigger than a’s,
         //or update to a’s velocity
         if(Math.abs(velocitya.get(i).Z)
            > Math.abs(velocitybcd.get(c2).Z))
            velocitybcd.get(c2).Z
                              = velocitya.get(i).Z;
         if(Math.abs(velocitya.get(i).Z)
            > Math.abs(velocitybcd.get(d).Z))
            velocitybcd.get(d).Z
                              = velocitya.get(i).Z;
      }
   }
  }
 }
}
}


//get distance of two points with points input
private float getDistance(Points p1, Points p2)
{
   return  (float)Math.sqrt((float)Math.pow(p2.Z-p1.Z, 2)
                          +(float)Math.pow(p2.X-p1.X, 2));
}


//get distance of two points with coordinate input
private float anotherGetDistance(float x1, float z1, float x2,
                                 float z2)
{
   return  (float)Math.sqrt((float)Math.pow(z2-z1, 2)
                          +(float)Math.pow(x2-x1, 2));
}


//get distance between a point and a line
private float distanceBetweenPointAndLine(float x, float z,
                        float x1, float z1, float x2, float z2)
{
   return (float) Math.abs((z2-z1)*x - (x2-x1)*z + z1*x2 - z2*x1)
                        *(1/anotherGetDistance(x1, z1, x2, z2));
}


//get square root
private float sumOfXAndZ(float x, float z)
{
   return (float)(Math.sqrt(Math.pow(x, 2)+Math.pow(z, 2)));
}


//check is two line segments intersect or not
private boolean TwoLineIsIntersect(float x0, float y0, float x1,
                        float y1, float x2, float y2, float x3,
                        float y3, float InterX, float InterY)
{
   //Two lines X0X1 AND X2X3
   float x, y;
   float Minx01 = Math.min(x0, x1);
   float Miny01 = Math.min(y0, y1);
   float Minx23 = Math.min(x2, x3);
   float Miny23 = Math.min(y2, y3);
   float Maxx01 = Math.max(x0, x1);
   float Maxy01 = Math.max(y0, y1);
   float Maxx23 = Math.max(x2, x3);
   float Maxy23 = Math.max(y2, y3);

   if(x1!=x0 && x2!=x3)
   {
      //slopes of two lines
      float k1 = (y1-y0)*(1/(x1-x0));
      float k2 = (y3-y2)*(1/(x3-x2));
      float Den = (y1-y0)*(x3-x2) - (y3-y2)*(x1-x0);
      if(k1==k2)
      {
         //parallel
         float d1 = Math.abs(y0*(x1-x0)-x0*(y1-y0)
                        -y2*(x3-x2)+x2*(y3-y2));
         if(d1==0)
         {
            //superposition
            if((x2>Minx01 && x2<Maxy01
                && y2>Miny01 && y2<Maxy01)
              ||(x3>Minx01 && x3<Maxy01
                && y3>Miny01 && y3<Maxy01)
              ||(x0>Minx23 && x0<Maxy23
                && y0>Miny23 && y0<Maxy23)
              ||(x1>Minx23 && x1<Maxy23
                && y1>Miny23 && y1<Maxy23))
            {
               //superposition is intersect
               return true;
            }
            else
            {
               return false;
            }
         }
         else
         {
            return false;
         }
      }
      //coordinate of the point of intersection
      x = ((y2-y0)*(x1-x0)*(x3-x2)+(y1-y0)*(x3-x2)*x0
            -(y3-y2)*(x1-x0)*x2)*(1/Den);
      y = ((y1-y0)*(x-x0))*(1/(x1-x0)) + y0;

      if(Minx01<=x && x<=Maxx01 && Miny01<=y && y<=Maxy01
         && Minx23<=x && x<=Maxx23 && Miny23<=y && y<=Maxy23)
      {
         //Intersect point
         InterX = x;
         InterY = y;
         return true;
      }
   }

   else if(x1==x0 && x2!=x3)
   {
      //coordinate of the point of intersection
      x = x0;
      y = ((y3-y2)*(x0-x2))*(1/(x3-x2)) + y2;
      if(Minx01<=x && x<=Maxx01 && Miny01<=y && y<=Maxy01
        && Minx23<=x && x<=Maxx23 && Miny23<=y && y<=Maxy23)
      {
         InterX = x;
         InterY = y;
         return true;
      }
   }

   else if(x1!=x0 && x2==x3)
   {
      //coordinate of the point of intersection
      x = x2;
      y = ((y1-y0)*(x2-x0))*(1/(x1-x0)) + y0;
      if(Minx01<=x && x<=Maxx01 && Miny01<=y && y<=Maxy01
         && Minx23<=x && x<=Maxx23 && Miny23<=y && y<=Maxy23)
      {
         InterX = x;
```

```
                InterY = y;
                return true;
            }
        }
        return false;
    }
}
```

```java
/* HeightOfWaveCurve.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Adjust the height of wave curves 1-4.
 * See section 5.6.3 for details.
 */

package demos;

import java.util.List;
import java.util.ArrayList;

public class HeightOfWaveCurve
{
   //increase the heights of wave curves 3 and 4 when they are close
   //to the land.
   public void AdjustHeightOfWaveCurve(float landValue[][][],
                 float waterValue[][][], List<Points> waveCurve,
                 List<FloatNum> Slope, List<Points> inNumbers,
                 List<Points> points, float plus)
   {
      for(int i = 0; i<waveCurve.size(); i++)
      {
         //coordinate of wave curve vertex’s current position
         float x = waterValue[waveCurve.get(i).X]
                           [waveCurve.get(i).Z][0];
         float z = waterValue[waveCurve.get(i).X]
                           [waveCurve.get(i).Z][2];
         //coordinate of land at the same position as wave curve
         //vertex’s rest position
         float xc = landValue[points.get(inNumbers.get(i).X).X]
                         [points.get(inNumbers.get(i).X).Z][0];
         float zc = landValue[points.get(inNumbers.get(i).X).X]
                         [points.get(inNumbers.get(i).X).Z][2];
         float yc = landValue[points.get(inNumbers.get(i).X).X]
                         [points.get(inNumbers.get(i).X).Z][1];
         //distance of wave curve vertex’s current and rest postion
         float distance = GetDistance(x, z, xc, zc);

         //adjust wave curve vertex’s height by make wave curve vertex
         //climb the land along the gradient of the land, and add
         //a height offset to keep them a little higher than the land
         //all the time
         waterValue[waveCurve.get(i).X][waveCurve.get(i).Z][1]
                  = yc - Slope.get(i).N * distance + plus;
      }
   }


   //height spring for wave curves 1 and 2
private float SpringOfW12(float x)
{
   float k = 3; //constant of height spring
   float f = -k*x; //force of height spring
   return f;
}


//control the height of wave curves 1 and 2 by using height spring
public void AdjustHeightOfW12(float landValue[][][],
                float waterValue[][][], List<Points> waveCurve,
                List<Points> inNumbers, List<Points> outnumbers,
                List<Points> points, List<Points> waveCurve4,
                List<FloatNum> originDistance,
                List<FloatNum> verticalVel)
{
   float t = 0.1f; //time interval
   for(int i=0; i<waveCurve.size(); i++)
   {
      //distance of wave curve vertex height from water level
      float x = waterValue2[waveCurve.get(i).X]
                           [waveCurve.get(i).Z][1]-19;

      //force from height spring
      float f = SpringOfW12(x);
      //wave curve vertex height change by its velocity
      waterValue2[waveCurve.get(i).X][waveCurve.get(i).Z][1]
               += verticalVel.get(i).N*t;
      //wave curve vertex velocity change by spring force and be
      //damp
      verticalVel.get(i).N = (verticalVel.get(i).N + f*t)*0.97f;
      //output the wave curve vertex height
      waterValue[waveCurve.get(i).X][waveCurve.get(i).Z][1]
               = waterValue2[waveCurve.get(i).X]
                             [waveCurve.get(i).Z][1];
   }
}

//get slope of the land
public void GetSlope(float landValue[][][],
                List<Points> coastline, List<Points> waveCurve,
                List<Points> inNumbers, List<FloatNum> Slope)
{
   float hCoastline, hWaveCurve, h, l;
   for(int i = 0; i < waveCurve.size(); i++)
   {
      //the height of current vertex’s nearest vertex in coastline
      hCoastline = landValue
                  [coastline.get(inNumbers.get(i).X).X]
                  [coastline.get(inNumbers.get(i).X).Z][1];
      //the height of current vertex in rest position
      hWaveCurve = landValue
                 [waveCurve.get(i).X][waveCurve.get(i).Z][1];
      //distance of wave curve vertex and its coastline vertex
      //in y axis
```

```
            h = hCoastline - hWaveCurve;
            //distance of wave curve vertex and its coastline vertex
            //in x-z plane
            l = GetDistance(
                    landValue[coastline.get(inNumbers.get(i).X).X]
                             [coastline.get(inNumbers.get(i).X).Z]
                             [0],
                    landValue[coastline.get(inNumbers.get(i).X).X]
                             [coastline.get(inNumbers.get(i).X).Z]
                             [2],
                    landValue[waveCurve.get(i).X]
                             [waveCurve.get(i).Z][0],
                    landValue[waveCurve.get(i).X]
                             [waveCurve.get(i).Z][2]);
            Slope.add(new FloatNum(h * (1/l))); //get the slope
        }
    }


    //get distance between two points
    private float GetDistance(float x1, float z1, float x2, float z2)
    {
        return ((float)Math.sqrt((float)Math.pow(z2-z1, 2)
                + (float)Math.pow(x2-x1, 2)));
    }
}
```

Appendix A12

Vertex Shader

```
/* water.vert 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * vertex shader for the water surface, which apply multitexturing, crest
 * shading, and transparency.
 * See section 6.2.5 for details.
 */

varying vec3 position;
varying vec3 landHeight;
void main(void)
{
   position = gl_Vertex.xyz; //read water coordinate
   landHeight = gl_Color.xyz; //read land coordinate (in color)

   //read the coordinate of water material texture
   gl_TexCoord[0] = gl_MultiTexCoord0;
   //read the coordinate of water detail texture
   gl_TexCoord[1] = gl_MultiTexCoord1;
   gl_Position = ftransform(); //output water coordinate
}
```

Fragment Shader

```
/* water.frag 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * fragment shader for the water surface, which apply multitexturing, crest
 * shading, and transparency.
 * See section 6.2.5 for details.
 */

varying vec3 position; //load water coordinate
varying vec3 landHeight; //load land coordinate
//load textures
uniform sampler2D texture1;
uniform sampler2D texture2;

void main(void)
{
   //get the water vertex height base on water level
   float waterVHeight = position.y - 19;
   //calculate the color value of water vertex
   float colorValue = -waterVHeight*0.25-0.1;

   //set pixel data using the texture coordinate
   vec4 tex1 = texture2D(texture1, gl_TexCoord[0].st);
   vec4 tex2 = texture2D(texture1, gl_TexCoord[1].st);

   float transparency = 1.0; //build transparency
   //the land height in the range (15, 19], the transparency in the
   //range (1, 0.01]
   if(landHeight.y>15 && landHeight.y<=19)
      transparency = 1.01-(landHeight.y-15)*0.25;
   // the land height equal and lower than 15, the transparency is
   // fixed to 1
   else if(landHeight.y<=15) transparency = 1;
   //The land height higher than 19, the transparency is fixed to 0.01
   else if(landHeight.y>19) transparency = 0.01;

   //sum a half of RGB pixel data of each texture and plus color pixel,
   //then add transparency
   gl_FragColor = vec4((tex1.r+tex2.r)*0.5f+colorValue,
                       (tex1.g+tex2.g)*0.5f+colorValue,
                       (tex1.b+tex2.b)*0.5f+colorValue,
                        transparency);
}
```

Appendix A13

```
/* Particle.java 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * Water spray and breaking wave particle systems.
 * See sections 7.1.1 and 7.2.1 for details.
 */


package demos;

import javax.media.opengl.*;
import javax.media.opengl.glu.*;
import java.util.List;
import java.util.ArrayList;
import java.nio.*;
import java.io.IOException;


public class Particle
{
   //build data for water spray particles
   public void buildWaterSprayParData(GL gl, GLU glu, int parsize,
                   List <Points> wavecurve, List<Vel3D> parvalue,
                   float watervalue [][][], List <FloatNum> parLife,
                   List<Vel3D> parVel, int waterSprayParTex,
                   ByteBuffer parTexBuffer)
   {
      //build particle array
      byte parTexArray [] = new byte [32*32*3];
      //set particle texture null
      TextureReader.Texture parTex = null;
      //read particle texture
      ReadTexture(parTex, "data/particle.bmp", parTexArray);
      parTexBuffer.put(parTexArray); //put par data into buffer
      parTexBuffer.rewind(); //rewind buffer

      //bind particle texture
      gl.glBindTexture(GL.GL_TEXTURE_2D, waterSprayParTex);
      //build particle texture
      glu.gluBuild2DMipmaps(GL.GL_TEXTURE_2D, GL.GL_RGB, 32, 32,
                  GL.GL_RGB, GL.GL_UNSIGNED_BYTE, parTexBuffer);
      //set texture parameters
      gl.glTexParameteri(GL.GL_TEXTURE_2D,
                     GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR);
      gl.glTexParameteri(GL.GL_TEXTURE_2D,
                     GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR);

      for(int i=0; i<wavecurve.size(); i++)
      {
         for(int j=0; j<parsize; j++)
         {
            //create particle position
            parvalue.add(new Vel3D(watervalue[wavecurve.get(i).X]
                                    [wavecurve.get(i).Z][0],
                                 watervalue[wavecurve.get(i).X]
                                    [wavecurve.get(i).Z][1],
                                 watervalue[wavecurve.get(i).X]
                                   [wavecurve.get(i).Z][2]));
            //create particle velocity
            parVel.add(new Vel3D(0, 0, 0));
            //create particle life
            parLife.add(new FloatNum(1));
         }
      }
   }


   //build water spray particles
   public void buildWaterSprayParticle(GL gl, int parsize,
                  List <Points> wavecurve, List<Vel3D> parvalue,
                  float watervalue [][][], List < Vel > velocity,
                  List <FloatNum> parLife, List<Vel3D> parVel,
                  int shaderProgram, int parTex, int texParam)
   {
      gl.glColor4f(0.7f, 0.7f, 0.7f, 1f); //set particle color
      //set parameter for point distance attenuation
      float att [] = new float [3];
      att [0] = 0.0f;
      att [1] = 1f;
      att [2] = 0.0f;
gl.glEnable (GL.GL_BLEND); //enable blend
//set blend function
gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE);
gl.glEnable(GL.GL_TEXTURE_2D); //enable texture
//enable vertex program point size
gl.glEnable(GL.GL_VERTEX_PROGRAM_POINT_SIZE);
gl.glPointSize(5); //set point size
//set point distance attenuation
gl.glPointParameterfv(GL.GL_POINT_DISTANCE_ATTENUATION,
                      att, 0);
//limit point size
gl.glPointParameterf(GL.GL_POINT_SIZE_MIN, 1);
gl.glPointParameterf(GL.GL_POINT_SIZE_MAX, 8);
//set point sprite parameter
gl.glTexEnvf(GL.GL_POINT_SPRITE,
             GL.GL_COORD_REPLACE, GL.GL_TRUE);
//set point fade threshold size
gl.glPointParameterf(GL.GL_POINT_FADE_THRESHOLD_SIZE, 8);
gl.glEnable(GL.GL_POINT_SPRITE); //enable point sprite

gl.glUseProgram(shaderProgram); //use shaders
gl.glBindTexture(GL.GL_TEXTURE_2D, parTex); //bind texture
gl.glUniform1i(texParam, 0); //transfer texture
for(int i=0; i<wavecurve.size(); i++)
{
   for(int j=0; j<parsize; j++)
   {
      gl.glPushMatrix();  //push matrix
         //draw particles in matrix
         gl.glBegin(GL.GL_POINTS);
            gl.glVertex3f(parvalue.get(i*parsize + j).X,
                          parvalue.get(i*parsize + j).Y,
                          parvalue.get(i*parsize + j).Z);
      gl.glEnd();
      gl.glPopMatrix(); //pop matrix
      //update particle attribute when it is alive
      if(parLife.get(i*parsize + j).N > 0)
      {
         //update the position
         parvalue.get(i*parsize + j).X
                       += 0.1f*parVel.get(i*parsize + j).X;
         parvalue.get(i*parsize + j).Y
                       += 0.1f*parVel.get(i*parsize + j).Y;
         parvalue.get(i*parsize + j).Z
                       += 0.1f*parVel.get(i*parsize + j).Z;
         //update the velocity
         parVel.get(i*parsize + j).Y -= 9.8f*0.1f;
         //limit the life
         if(parvalue.get(i*parsize + j).Y < 18)
            parLife.get(i*parsize + j).N = -1;
      }
      else //give new attribute to dead particle
      {
         //give new position around its vertex
         parvalue.get(i*parsize + j).X = watervalue
              [wavecurve.get(i).X][wavecurve.get(i).Z][0]
              +(float)(Math.random()-0.5f)*2;
         parvalue.get(i*parsize + j).Y = watervalue
              [wavecurve.get(i).X][wavecurve.get(i).Z][1]
              +(float)(Math.random())-0.5f;
         parvalue.get(i*parsize + j).Z = watervalue
              [wavecurve.get(i).X][wavecurve.get(i).Z][2]
              +(float)(Math.random()-0.5f)*2;
         //give new velocity equal to its vertex plus a random
         //offset in x and z axes, and give new random
         //velocity in y axis
         parVel.get(i*parsize + j).X = velocity.get(i).X
                             +(float)(Math.random())-0.5f;
         parVel.get(i*parsize + j).Y
            = (float)(Math.random())-0.5f;
         parVel.get(i*parsize + j).Z = velocity.get(i).Z
                             +(float)(Math.random())-0.5f;
         //give new life
         parLife.get(i*parsize + j).N = 1;
      }
   }
}
gl.glUseProgram(0); //end to use shaders
gl.glBindTexture(GL.GL_TEXTURE_2D, 0); //cancel to texture
gl.glDisable(GL.GL_POINT_SPRITE); //disable point sprite
}


//build breaking wave particle
```

```
public void buildBreakingWaveParticle(GL gl,
            int parShaderProgram, int parTex, int texParam,
            int watersize, int parSize, List <FloatNum> parLife,
            List < Vel3D > parVel, float parValue[][][][],
            float watervalue[][][])
{
   gl.glColor4f(0.7f, 0.7f, 0.7f, 1f); //set particle color
   //set parameter for point distance attenuation
   float att [] = new float [3];
   att [0] = 0.0f;
   att [1] = 1f;
   att [2] = 0.0f;

   gl.glEnable (GL.GL_BLEND); //enable blend
   //set blend function
   gl.glBlendFunc(GL.GL_SRC_ALPHA, GL.GL_ONE);
   gl.glEnable(GL.GL_TEXTURE_2D); //enable texture
   //enable vertex program point size
   gl.glEnable(GL.GL_VERTEX_PROGRAM_POINT_SIZE);
   gl.glPointSize(5); //set point size
   //set point distance attenuation
   gl.glPointParameterfv(GL.GL_POINT_DISTANCE_ATTENUATION,
                     att, 0);
   //limit point size
   gl.glPointParameterf(GL.GL_POINT_SIZE_MIN, 1);
   gl.glPointParameterf(GL.GL_POINT_SIZE_MAX, 8);
   //set point sprite parameter
   gl.glTexEnvf(GL.GL_POINT_SPRITE,
            GL.GL_COORD_REPLACE, GL.GL_TRUE);
   //set point fade threshold size
   gl.glPointParameterf(GL.GL_POINT_FADE_THRESHOLD_SIZE, 1);
   gl.glEnable(GL.GL_POINT_SPRITE); //enable point sprite

   gl.glUseProgram(shaderProgram); //use shaders
   gl.glBindTexture(GL.GL_TEXTURE_2D, parTex); //bind texture
   gl.glUniform1i(texParam, 0); //transfer texture

   for(int z = 1; z < watersize-1; z++)
   {
      for(int x = 1; x < watersize-1; x++)
      {
         for(int i=0; i<parSize; i++)
         {
            //update particle attribute when it is alive
            if(parLife.get((z-1)*(watersize-2)*parSize
              + (x-1)*parSize + i).N > 0)
            {
               gl.glPushMatrix(); //push matrix
                  //draw particles
                  gl.glBegin(GL.GL_POINTS);
                     gl.glVertex3f(parValue[x][z][i][0],
                                   parValue[x][z][i][1],
                                   parValue[x][z][i][2]);
                  gl.glEnd();
               gl.glPopMatrix(); //pop matrix

               //update the position
               parValue[x][z][i][1]
                       += 0.1f*parVel.get((z-1)*(watersize-2)
                             *parSize + (x-1)*parSize + i).Y;
               //update the velocity
               parVel.get((z-1)*(watersize-2)*parSize
                          + (x-1)*parSize + i).Y
                  -= 9.8f*0.1f;
               //limit the life
               if(parValue[x][z][i][1]
                  < watervalue[x][z][1]-0.5f)
                  parLife.get((z-1)*(watersize-2)*parSize
                               + (x-1)*parSize + i).N = -1;
            }
         }
      }
   }
   gl.glDisable(GL.GL_POINT_SPRITE); //disable point sprite
   gl.glUseProgram(0); //end to use shaders
}


//build breaking wave particle data
public void buildBreakingWaveParData(
                float breakingWaveParValue[][][][], int parSize,
           int landsize, int watersize, float watervalue[][][],
                 List < Vel3D > parVel, List <FloatNum> parLife)
{
      for(int z = 1; z < watersize-1; z++)
      {
         for(int x = 1; x < watersize-1; x++)
         {
            for(int i=0; i<parSize; i++)
            {
               //create particle position
               breakingWaveParValue[x][z][i][0]
                  = watervalue[x][z][0];
               breakingWaveParValue[x][z][i][1]
                  = watervalue[x][z][1];
               breakingWaveParValue[x][z][i][2]
                  = watervalue[x][z][2];
               //create particle velocity
               parVel.add(new Vel3D(0, 0, 0));
               //create particle life
               parLife.add(new FloatNum(1));
            }
         }
      }
   }


   //read texture from a file
   private void ReadTexture(TextureReader.Texture texutre,
                           String textureName, byte textureArray[])
   {
      try
      {
         //read texture with file name
         texutre = TextureReader.readTexture(textureName);
      }
      catch (IOException e)
      {
         e.printStackTrace();
         throw new RuntimeException(e);
      }
      //get texture pixels in a array
      texutre.getPixels().get(textureArray, 0,
                              textureArray.length);
   }
}
```

Appendix A14

Vertex Shader

```
/* par.vert 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * vertex shader for the particle systems.
 * See sections 7.1.1 and 7.2.1 for details.
 */

void main(void)
{
   vec4 position = gl_Vertex; //read particle coordinate
   //read particle texture coordinate
   gl_TexCoord[0] = gl_MultiTexCoord0;

   gl_PointSize = 8-(position.y-19)*3; //change particle size
   if(gl_PointSize < 2) gl_PointSize = 2; //limit particle size
   else if (gl_PointSize > 5) gl_PointSize = 5; //limit particle size

   gl_Position = ftransform();  //output particle coordinate
}
```

Fragemnt shader

```
/* par.frag 2010/7/8
 * By Sui Yifan
 * E-mail: suisuige@yahoo.com.cn
 *
 * fragment shader for the particle systems
 * See sections 7.1.1 and 7.2.1 for details.
 */

uniform sampler2D texture;

void main(void)
{
   //set pixel data using the texture coordinate
   vec4 tex = texture2D(texture, gl_TexCoord[0].st);
   //output particle color by using coordinate
   gl_FragColor = vec4(tex.r, tex.g, tex.b, 1);
}
```

RENDERING WATER AND LAND INTERACTION USING A SPRING SYSTEM

Yifan Sui
Andrew Davison
Department of Computer Engineering
Faculty of Engineering
Prince of Songkla University
Hat Yai, Songkla, Thailand
E-mail: suisuige@yahoo.com.cn
E-mail: ad@fivedots.coe.psu.ac.th

KEYWORDS
Spring system, wave curve, collision detection, mesh, height value.

ABSTRACT

This paper describes a spring-based model for the interaction of water and land, which reconciles realism and fast rendering. The system controls the motion and interdependences of water vertices utilizing two kinds of springs and collision detection. As a consequence, a wave's movement affects the waves around it, and a wave 'hits' the land, rebounding with a suitably changed height and velocity.

1. INTRODUCTION

The rendering of large areas of water is well understood (Tessendorf 1999; Johanson 2004), and has become common in games. However, there is little physics-based interaction with the shoreline as waves move up and down, and generate spray and foam.

Most 3D systems employ Perlin noise functions (Johanson 2004), although some ocean effects (such as refraction and obstacle collision) have been utilized (Iglesias 2004). For instance, Peachey (Peachey 1986) and Fournier and Reeves (Fournier and Reeves 1986) model waves that approach and break on a sloping beach. However, there is no real force connection between the water and the land, since the wave profile is changed according to wave steepness and water depth.

Foster and Fedkiw (Foster and Fedkiw 2001), and Enright, Marschner and Fedkiw (Enright et al. 2002) simulate breaking waves using a combination of textures and particles. The computational cost of the former is several minutes per frame on a PentiumII 500MHz.

Mass-spring systems are arguably the simplest and most intuitive of all deformable models for simulating fluid (Nealen et al. 2006). Using a spring system to simulate the motion of water over a coastline is compulationally feasible, as this paper illustrates.

2. WAVE CURVES AND THE COASTLINE

The land is a 128*128 textured mesh contoured with a height map. The waves in the water mesh are modeled using Peachey's method (Peachey 1986), so the height of the water vertices varies as a sum of water level and wave height. The color of each vertex is based on its current height. Our prototype was created using JOGL (a Java binding for OpenGL (JOGL, 2009)). Figure 1 shows a screenshot of the model with its elements.

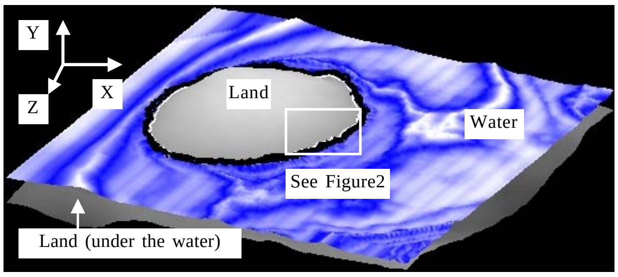


Figure 1: An overview of the model.

The coastline is the series of water vertices that are closest to the land, (see Figure 2). Wave curve 1 is the line of vertices one mesh interval away from the coastline, wave curve 2 is the line of vertices one mesh interval away from wave curve 1. In this way, we define a coastline and four wave curves, which are linked with springs as explained in section 3.

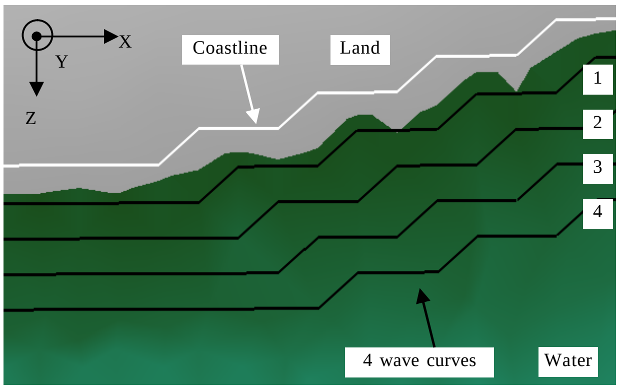


Figure 2: A view of the model from overhead, showing the coastline and four wave curves.

2.1 Vertical Water Mesh Movement
In shallow water, the water vertices move up and down by employing a summed combination of four versions of the height function (1):

$$Y_i = 8 * \left(\frac{\sqrt{x_i^2 + z_i^2}}{L_i} - \frac{t}{T} - \frac{1}{2}\right)^2 - 1 \qquad (1)$$

where *i* is the index of a vertex, *Y* is the wave height of the vertex, *x* and *z* are the (X, Z) coordinate values of the vertex, *t* is the time which increases by 0.1 in each frame. *T* is the period of the function, equal to 80 frames to look realistic. *L* is the wavelength at the vertex position.

When a wave enters the shallows, where the depth is less than one-twentieth of the wavelength, the wave length *L* is determined by Equation (2):

$$L = T\sqrt{gd} \qquad (2)$$

where *d* is the depth of water, and *g* the gravity (Alonso and Finn 1992; Sverdrup 2006).

Figure 3 illustrates how these equations affect the height of the water mesh as it approaches the shallows.

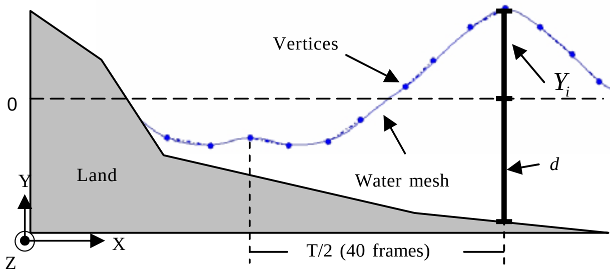


Figure 3: The height of water vertices in shallow water.

2.2 The Coastline
The coastline is the line of water mesh vertices closest to the land mesh, as shown in Figure 2 and Figure 4.

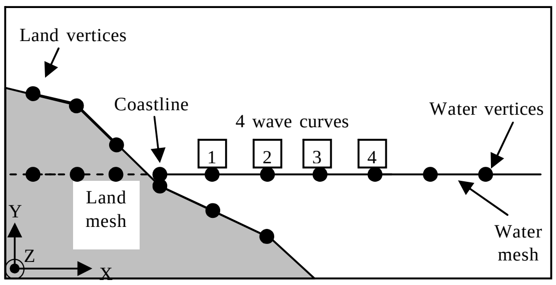


Figure 4: The position of the coastline.

The coastline boundary moves up and down due to waves, but doesn't shift in the X-Z plane.

2.3 Wave Curves
Wave curve 1 is the line of water mesh vertices adjacent to the coastline, but one mesh interval away from the land. Wave curve 2 is the line of water mesh vertices adjacent to wave curve 1, but one mesh interval further away. Wave curve 3 and 4 are calculated in a similar way. Our model is limited to four wave curves as a balance between interaction realism and computational efficiency.

Wave curves 3 and 4 can move in the X-Z plane, towards and away from the coastline. Each vertex in the curves has a movement direction pointing from its original position toward the nearest coastline vertex. The vertices of wave curve 3 can move up to the coastline, while the vertices of wave curve 4 can move up to wave curve 1 (see Figure 5). Wave curve 4 can not easily pass through wave curve 3. These restrictions on curve interaction produce realistic wave behavior, and are implemented using our spring system and collision detection, as detailed in sections 3 and 4.

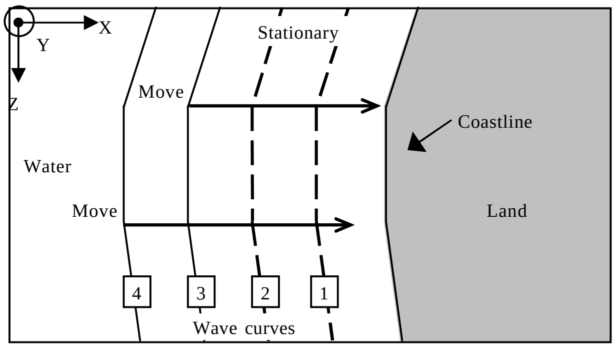


Figure 5: The movement of wave curves.

Wave curves 1 and 2 can not move in the X-Z plane, which means that the ebb and flow of the water against the coastline is driven by wave curves 3 and 4.

3. INTERACTION BETWEEN WAVE CURVES

The interaction between the water and land use *position springs* and *wave springs* to modify the X- and Z- velocities of the vertices in wave curves 3 and 4.

3.1 Position Springs
A position spring ensures that a vertex is pulled back to its original position after moving towards the land. Every vertex in wave curve 3 and 4 has its own position spring.

Figure 6 shows a vertex N_s. At time 0, it is at its rest position, labeled as $N_{s,0}$. At time t, it has moved to be at position $N_{s,t}$. The position spring P extends from the $N_{s,0}$ rest position and will pull N_s back from its $N_{s,t}$ position.

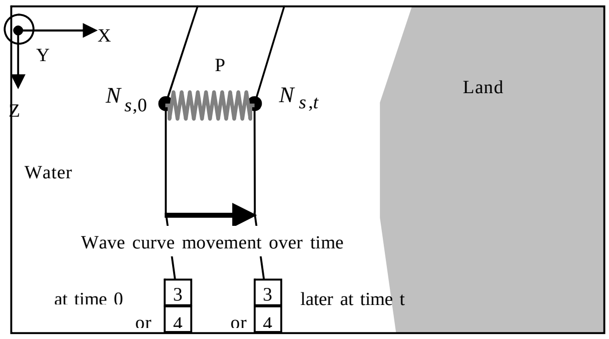


Figure 6: A position spring P for vertex N_s.

3.2 Wave Springs
Every neighboring pair of vertices in wave curves 3 and 4 are linked by a wave spring. For example, Figure 7 shows a wave spring W linking the vertices N_s and N_r of wave curves 3 and 4.

Wave springs help to deal with crossover behavior when vertices in wave curve 4 are moving faster than those in wave curve 3, and attempt to pass through it. Wave springs slow down wave curve 4 vertices as they approach wave curve 3.

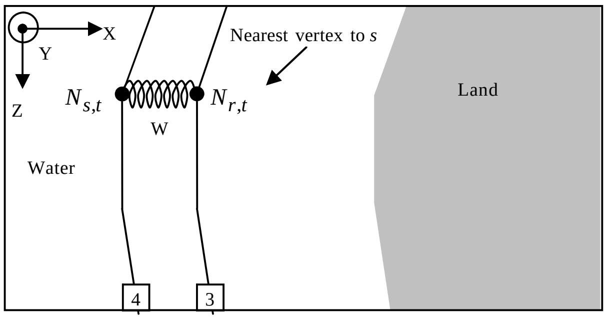


Figure 7: A Wave spring W between vertices N_s and N_r in wave curves 3 and 4.

4. COLLISION DETECTION

Our model deals with two kinds of collision:

1) between the water and the land, as represented by wave curve 3 and the coastline;

2) between waves, as represented by wave curves 3 and 4.

Our approach builds upon real time collision detection (Ericson 2005) by applying it in the context of our spring system.

4.1 Water and Land Collision
The collision detection algorithm is simplified by utilizing the coastline to represents the land, and wave curve 3 as the leading edge of the water.

Each coastline vertex is surrounded by a bounding sphere, whose diameter is equal to the initial inter-mesh spacing.

If a wave curve 3 vertex moves inside the bounding sphere of a coastline vertex, a collision has happened. The velocity of the wave curve 3 vertex is reversed, to make it head back towards its rest position.

Figure 8 shows a vertex N_s in wave curve 3. At time 0, it is at position $N_{s,0}$, then moves towards the coastline and 'hits' the coastline vertex C_p at time t. The velocity of N_s, $V_{s,t}$, is reversed to be $-V_{s,t+1}$ at the next time interval t+1. A scaling factor also reduces the velocity, to take account of the way a wave loses energy when rebounding.

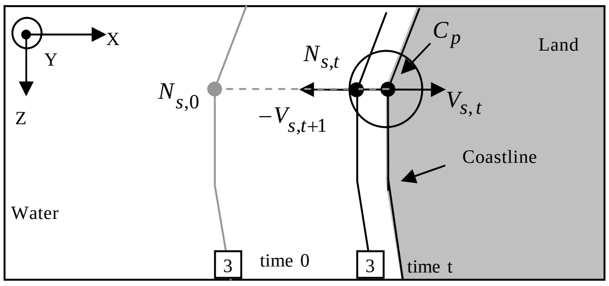


Figure 8: Water and land collision.

4.2 Wave Collision
As explained in section 3, wave springs implement crossover slowdown, but if the velocity of wave curve 4 is much higher than wave curve 3 then crossover could still occur. This is prevented by collision detection between the vertices of wave curves 3 and 4. When a vertex in wave curve 4 hits wave curve 3, their velocities are equalized, so the two wave curve segments will move together, or perhaps separate. This is implemented by updating the velocity of the vertex in wave curve 4 and its nearest neighbor in wave curve 3.

Figure 9 shows the case when vertex N_s is about to hit the wave curve segment V1-V2. A collision is detected between N_s and the segment, and the velocities of N_s and V2 are modified.

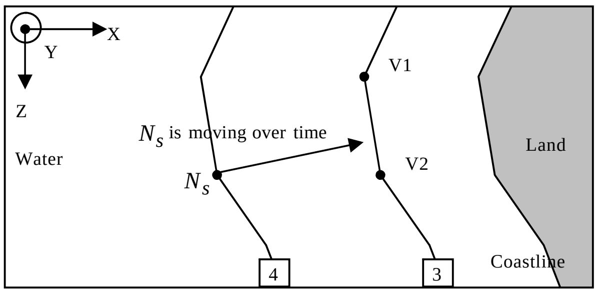


Figure 9: Wave Collision.

The overall behavior of N_s will be more complicated than this (and more realistic) by also being affected by a wave spring linking it to V2 (its nearest neighbor in wave curve 3), which is not shown in Figure 9.

5. TESTING

On a single core 1.73 GHz 2GB DDRII-533 RAM Intel GMA 950, the model executes at about 70 FPS; on a two-core 1.86 GHz 1GB DDRII-533 RAM X300 graphics card, about 140 FPS are achieved, and our OS both are Windows XP SP3 Professional. When we extend the mesh size to 256*256, the model executes on two machines at about 54 FPS and 26 FPS.

Figure 10 is a cross-sectional view of the model showing water moving towards the land.

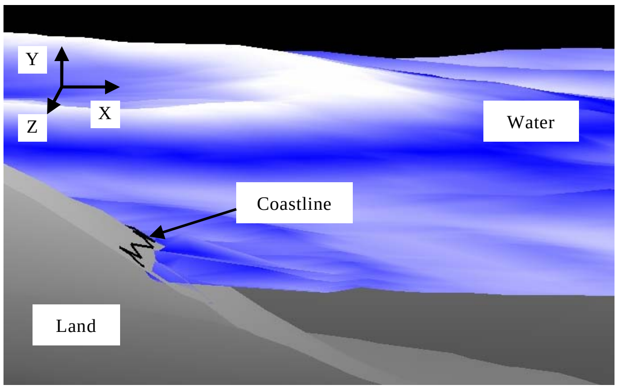


Figure 10: Water moving towards the land.

Figure 11 shows the land, coastline, and wave curves of Figure 10 from overhead.

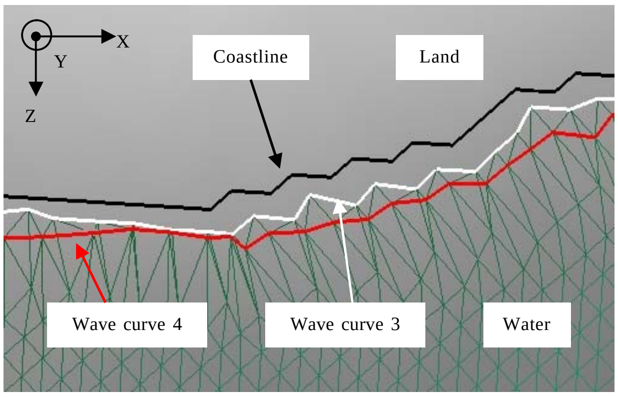


Figure 11: Wave curves moving towards the land.

Figure 12 shows the scene later after the water has rebounded from the land.

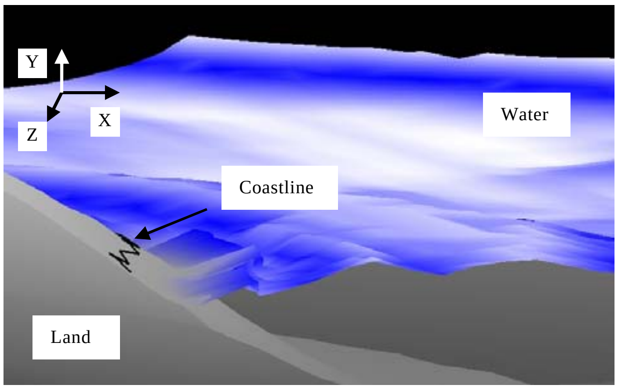


Figure 12: Water retreating from the land.

Figure 13 is a view of Figure 12 from overhead. It shows that the position springs in wave curves 3 and 4 are drawing their vertices back to their rest positions. The interaction between wave curves 3 and 4, as controlled by wave springs and collision detection, is also visible.

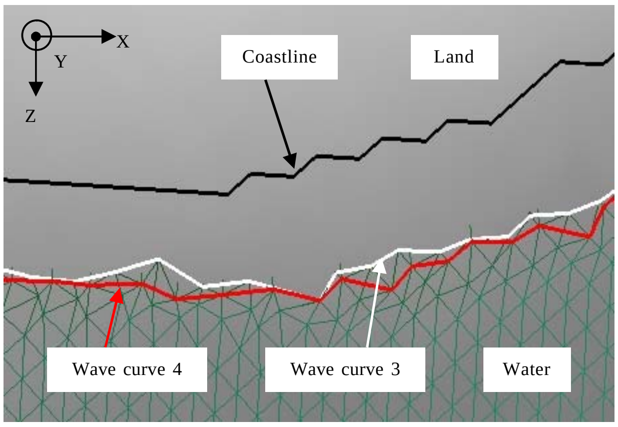


Figure 13: Wave curves rebounding from the land.

Figure 13 illustrates that crossover still occurs, as it does in real waves, but is a rare event, and is soon followed by the waves either moving in unison or pulling apart.

6. CONCLUSIONS

Our system models the interaction of water and land using a novel combination of two types of springs (position and wave springs) and two forms of collision detection. The simulation exhibits realistic behavior between waves and the coastline, and between the waves themselves, while rendering at very acceptable speeds. The spring system is relatively simple to understand and fine-tune, and is based on the physical characteristics of real waves.

We plan to improve the visualization by adding particle-based foam and spray. It will appear on wave crests, the coastline, and wherever collisions occur.

When the water recedes from the land, the exposed areas should look wet. We intend to color these areas accordingly, and let the color gradually fade over time.

Our long term goal is to use this approach to model Tsunami-land interaction. The spring system will need to be modified to deal with large waves (over 30m in height) moving at very high speeds (more than 800 km/h) (Kaitoku 2008). The coastline interaction will need to be more complicated to deal with the way a tsunami can wash over a large body of land.

REFERENCES

A. Iglesias. 2004. “Computer Graphics for Water Modeling and Rendering: A Survey”. *Future Generation Computer Systems*, Volume 20, Issue 8, (Nov), 1355-1374. http://www.sciencedirect.com/science?_ob=MImg&_imagekey=B6V06-4CVX0RT-2-3&_cdi=5638&_user=267327&_orig=search&_coverDate=11%2F01%2F2004&_sk=999799991&vie

w=c&wchp=dGLbVzz-zSkzV&md5=c4eea2bb7547c6e532e217a9ec196151&ie=/sdarticle.pdf (last accessed Oct 30, 2009)

Alain Fournier and William T. Reeves. 1986. “A Simple Model of Ocean Waves”. *ACM SIGGRAPH Computer Graphics*, Volume 20, Issue 4, (Aug), 75-84. http://portal.acm.org/citation.cfm?id=15894 (last accessed Oct 30, 2009)

Andrew Nealen; Matthias Müller; Richard Keiser; Eddy Boxerman and Mark Carlson. 2006. “Physically Based Deformable Models in Computer Graphics”. *Computer Graphics Forum*, Volume 25, Number 4, (Dec), 809-836. http://www.matthiasmueller.info/publications/egstar2005.pdf (last accessed Oct 30, 2009)

Christer Ericson. 2005. Real-Time Collision Detection. Morgan Kaufmann.

Claes Johanson. 2004. "Real-time Water Rendering- Introducing the Projected Grid Concept". Master’s thesis. Department of Computer Science, Lund University, (Mar). http://fileadmin.cs.lth.se/graphics/theses/projects/projgrid/ (last accessed Oct 30, 2009)

Darwyn R. Peachey. 1986. “Modeling Waves and Surf”. *ACM SIGGRAPH Computer Graphics*, Volume 20, Issue 4, (Aug), 65-74. http://portal.acm.org/citation.cfm?id=15893 (last accessed Oct 30, 2009)

Douglas Enright; Stephem Marschner and Ronald Fedkiw. 2002. “Animation and Rendering of Complex Water Surfaces”. *ACM Transaction on Graphics*, Volume 21, Issue 3, (Jul), 736-744. http://physbam.stanford.edu/~fedkiw/papers/stanford2002-03.pdf (last accessed Oct 30, 2009)

Jerry Tessendorf. 1999. “Simulating Ocean Water”. *SIGGRAPH Course Notes*. http://graphics.ucsd.edu/courses/rendering/2005/jdewall/tessendorf.pdf (last accessed Oct 30, 2009)

JOGL. 2009. http://kenai.com/projects/jogl/pages/Home (last accessed Oct 30, 2009)

Keith A. Sverdrup; Alison B. Duxbury and Alyn C. Duxbury. 2006. “Waves and Tides”. In *Fundamentals of Oceanography*, 5th edition, McGraw-Hill, 180-192

Marcelo Alonso and Edward J. Finn. 1992. “Wave Motion”. In *Physics*, Addison-Wesley, 747-766

Nick Foster and Ronald Fedkiw. 2001. “Practical Animation of Liquids”. *SIGGRAPH 2001*, Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 23-30. http://physbam.stanford.edu/~fedkiw/papers/stanford2001-02.pdf (last accessed Oct 30, 2009)

Tammy Kaitoku. 2008. *Tsunami, the Great Waves*. 5th edition, International Tsunami Information Center. http://ioc3.unesco.org/itic/contents.php?id=169 (last accessed Oct 30, 2009)

VITAE

Name Mr. Sui Yifan

Student ID 5010120159

Educational Attainment

Degree	Name of Institution	Year of Graduation
Bachelor of Engineering	JiangXi University of Science and Technology	2006

List of Publication and Proceedings

[1] Sui, Y. and Davison, A. 2009. "Rendering Water and Land Interaction using a Spring System", Game-On 2009: 10th Int. Conf. on Intelligent Games and Simulation, Dusseldorf, Germany, Nov. 26-28th, pp.25-29.