# Windy Awakening – CS488 Fall 2015 Final Project

Kevin Haslett - 20468033

## Installation Instructions

First unzip the "website" folder

If you already have a web server running you can simply drop the entire "website" folder into your hosting folder and run it from there.

ex: place "website" into "htdocs" (if you're running apache)

If you don't currently have a web server running, don't worry you don't need one. You can simply open the "index.html" file with your browser. However there may be some additional steps to follow to get everything to run this way, depending on which browser you use.

## Execution Instructions

Please use the latest version of either Google Chrome, or Firefox. I have tested with Google Chrome and Firefox on Windows, as well as Chromium on the graphics labs machines. All have worked without problems as far as I can tell.

I tested this with Google Chrome on Windows most extensively, so if you can, please test it there.

If you are using **Firefox** on Windows, everything should work just fine out of the box. Open the "index.html" file inside of the "website" folder and you're good to go.

If you are testing on **Google Chrome** or **Chromium** browser you will need to launch chrome with a special flag to allow for xhr requests to be made to and from "file://" urls.

First, close all open processes of Chrome. If you don't close all open Chrome processes this won't work.

In **Google Chrome** on Windows, create a shortcut to Chrome on your desktop.

1. Right Click -> Properties
2. In the "Target" field, add the following flag to the current target, *after* the quotations:
3. --allow-file-access-from-files
4. It should now look something like this:
5. "C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --allow-file-access-from-files
6. Now launch Chrome by clicking the shortcut.
7. Everything is now complete, simply open the "index.html" file with Chrome

In **Chromium** on Ubuntu (graphics lab),

1. Run the following command from the Terminal:
2. chromium-browser --allow-file-access-from-files
3. Everything is now complete, simply open the "index.html" file with Chrome

## Operating Instructions

The controls are as follows (they are also listed on the webpage to remind you):

- Click the canvas to capture the mouse (this one is important)
- WASD to move the boat
- Mouse to rotate camera
- Scroll wheel to zoom camera
- Space to freeze day/night cycle
- Hold Q to reverse day/night cycle
- Hold E to advance day/night cycle

You will also notice a number of checkboxes and dropdown options for demonstration purposes.

You can enable/disable certain features with the checkboxes.

You can also view some alternate frame buffers by selecting different options under the "Testing Mode" dropdown.

All of these controls should be easily understood by their names, and by playing with them a bit.

Nothing you do is irreversible so don't worry.

## Interesting Design Notes

I think one of the most interesting parts of the design is how I animated the geometry of the water. The water you see in the demo is a tessellated rectangle measuring 100x100 units. For reference, the length of the boat is about 1 unit long. This size was used because it appeared to be the minimum size that created the illusion of an infinite sea. With a density of 200x200 vertices, this amounts to 960 million vertices, so we can't make it significantly larger without it being too expensive. Unfortunately, at the current boat travel speed, this only takes about 30 seconds to reach the edge, making the illusion of an infinite sea break pretty quickly.

My solution to this problem was to *move* the geometry with you. Now, this would obviously cause some issues considering that the waves would move with you, breaking the illusion, but this actually isn't a problem because of how these waves were generated. After first attempting to update the water geometry every frame to animate the height of the waves, and determining that it was far too slow, it occurred to me to I could be doing this in the vertex shader, and for free! So in the vertex shader I have height of the water defined mathematically, as approximately: ***sin(x) * sin(z + time)***.

What this allowed me to do, was move the *x* and *z* positions of every vertex in the water geometry along with the boat, all in the vertex shader, and then calculate the height of the waves at that point in space. The effectively allows me to have an *infinite* sea with a *finite* amount of data in a static array buffer. You'll then get a much smaller (3x3 units) magic carpet of water you can ride around on. In order to animate the motion of the boat, I also have a copy of the wave height function. I determine the pitch rotation of the boat by calculating the height of the water at the front and back of the boat, and determining the angle with that.

Another interesting note about the water, is how the water Voronoi diagrams used for the water texture are generated. Each region in a Voronoi diagram is defined by a single defining point, and every point within that region is closer to its defining point than any other region's defining point. By exploiting this fact, we can represent each region as a three-dimensional cone. By intersecting the cones, and drawing them all with an orthographic projection, each pixel we draw will be closer to the cone of that colour than any other colour, by the nature of drawing only what is closest to the camera. This allows us to generate Voronoi diagrams very easily and very quickly.

You can view the Voronoi diagrams in my demo by unchecking the "Water Colouring", "Blur", and "Detect Edges" checkboxes above the canvas. As an aside, by unchecking them in that order you can see how the final texture is generated by unrolling the effects. I've explained this in more detail in my proposal. You will notice that the edges of the Voronoi regions are a bit jagged. This is the byproduct of using a small number of triangles around the surface of the cone, causing intersections that are less accurate to those of a mathematically perfect cone. In my demo I am using only 8. This is entirely intentional, because after processing the diagram we generate a more imperfect, wavier edge, which achieves a better affect to be used as waves. The regions' positions are then animated by having them "wander" at a slow speed in a random direction that gradually changes over time.

You will also notice, upon zooming out a bit (using the scroll wheel), that the Voronoi diagram is actually a repeating texture. The way this is created is by duplicating the Voronoi regions on all sides of the existing regions, and then rendering the diagram with those mirror images included. This allows you to generate diagrams where each side seamlessly connects to the opposite, which is perfect for a repeating texture to use for my infinite sea.

One of the more artistic aspects of my demo is the sunset and sunrise that affects the background colour and light colour. In order to allow the colour changes to be determined in some user friendly way (because translating RGB colour values into GLSL and then writing code to interpolate them would be a nightmare to experiment with) is by using a gradient texture to sample the colour from. If you look in the "website/assets/textures" folder you'll find the gradient samples used for the colours of the sky, the sunlight, the sun, and the islands' colours, created in Photshop. The colour for the sun assets is determined based on the height of the sun, and using that to determine how far along the gradient texture to sample the colour used. The island heights is done in the same way, instead using the height of the point on the island.

I used a blur filter in both the water generation, and in the bloom shader. The way this was implemented, for efficiency's sake was to do a box blur in two passes, one horizontal and one vertical. I would then perform this box blur a second time to achieve an approximation of a Gaussian blur that was much cheaper than a real Gaussian blur.

The way I created my shadow mapping projection (which you can see by selecting "Shadow Map" from the "Testing Mode" dropdown), is by actually moving the camera used to render the light's perspective. The reason I can do this and still achieve correct shadows is because I am using a directional light to emulate the sun. I also decided to only generate shadow mapped shadows for the boat because that was the most intricate and interesting piece of geometry. It also allowed me to generate very precise and detailed shadows that didn't need to be smoothed, because I could use the entire resolution of the shadow map for just the boat.

## Third Party Code

For this project I only used one external library. The glMatrix library is the WebGL equivalent to using the glm library in C++ OpenGL. It provides all the functions necessary for manipulating matrices and vectors in JavaScript, saving you a lot of time implementing it yourself. This code can be found at "website/scripts/lib/gl-matrix.js" in my project. More info can be found here: http://glmatrix.net/

There are also a couple of widely used GLSL code snippets found online that I have used to help me generate the Perlin noise.

The following code snippet generates pseudorandom numbers between -1 and 1 using some very clever hashing techniques. This is a very common code snippet used by people who want to generate random numbers in GLSL:

```
float rand(vec2 co){
  return 2.0 * fract(sin(dot(co.xy ,vec2(12.9898,78.233))) * 43758.5453) - 1.0;
}
```

The other code snippet I used was the following function which transforms a linear interpolation t-value to a certain smooth curve interpolation that is commonly used when generating Perlin noise:

```
float fade(float t) {
  return t*t*t*(t*(t*6.0-15.0)+10.0);
}
```

## Navigating My Code

In order to navigate my code there are a few things to know. First of all, all of the shader code is located in the "index.html" file inside of script tags. They are found in pairs, tagged with a type property of either "x-shader/x-vertex" or "x-shader/x-fragment" indicating the type of shader it is. This is a very common idiom used in WebGL which also allows you to create new shader programs quite easily.

The rest of the code is found in the "website/scripts" folder. The "objParser.js" file contains of all the code I wrote to parse .OBJ and their associated .MTL files, and load their data into JavaScript objects. This parser also supports use of multiple material textures per model, as my models use.

Finally, the vast majority of the code is found in the "main.js" file. It is roughly ordered from top to bottom, in order of execution. It is organized into logical functions that encapsulate logic which should be easy to understand by their names. All code is initialized at the very bottom of the file by the call to the `main()` function, located inside the completion of the asynchronous calls used to load in all of the assets.

## Extra Objectives

1. **Animated Infinite Water Geometry**
   Creating water geometry to emulate animating waves was not originally an objective, but flat water was so boring that I felt I had to add some detail. A lot of thought also went into making this geometry also extend infinitely.
2. **Gradient Texture Sampling and Beautiful Sunset**
   A lot of work was put into fine tuning the colours and timing used to create the sunsets, which I think turned out quite well. I also came up with an interesting technique to sample a gradient texture to help me more easily fine tune the visual transitions. I think the end result proves that this was worth the time I spent.
3. **Rigging and Posing Models**
   As I mentioned in my original proposal, I didn't create the models used in this demo. I found a website online which had ripped the models from the original Game Cube version of Legend of Zelda: Windwaker. However, these models weren't posed into a sensible position for sailing. Link originally had his arms stretch out to his sides, and the boat's head was originally pointing straight in the air. The boat was also missing a model for the sail entirely. I used my past experience with 3DS Max to create a skeleton rig, and skin the models to deform properly. I then posed Link into a position similar to the one he was in while sailing in the original game. I the deformed the boat's head into a properly arched position, and created model geometry for the missing sale. As well as a texture similar to the one found in the original game. You can find these assets in the "design" folder. The original models I downloaded can be found in the "design/boat" folder and the "design/toonlink" folder. The 3DS max file can be found at "design/linkboat.max". And the sail texture I made can be found in the "design/sail" folder which contains the final .PNG file and the Photshop document I used to make it. On the following page you'll find a screenshot of the skeleton seen inside the posed model. A lot of work was put into polishing these models into their final result, and I think you can tell.