Lab Activity 4

In this lab activity we are going to do some magic with our shader programs. Thus far we haven't really shown the power behind these programs. Head to the following sandbox **and fork it** to get started: https://codesandbox.io/s/lab-4-template-e2tgs3

The Environment

In this sandbox you'll find a floor and a sky created using a plane and a sphere, respectively. Both objects are shaded using their normal vectors. The `Plane` and `Sphere` classes are implemented for you and given in the `primitives` directory. There are also controls implemented such that you can rotate the camera using the mouse.

The Goal

Our goal here is to create a nice looking sky on our sphere to act as a skydome and to use our plane to create an ocean with waves. Notably, it would be difficult to do this using the same shader programs on both objects. So firstly we will need to set up our sandbox to handle different shaders for each object. We will also need to increase the amount of vertices that compose our plane such that we can simulate the waves. Then we will alter our shaders to generate our sky, and ocean.

Implement Multiple Shaders

Until now, we have been using a single shader for all of our objects. This is not ideal for a few reasons. Firstly, if we need different shading for multiple objects we have to use `if/then` statements in our shader which is bad for performance. GPU's are not designed to handle conditional logic as well as CPU's are. Secondly, our shaders get overly long and complex if we use the same one for each object.

Luckily we can compile many shaders and, when rendering, tell webGL to use any of them imperatively. To begin, head to the Plane class in `src/primitives/Plane.js` and create a place for our compiled shaders to live:

```
export default class Plane {
 constructor(widthSegments = 1, heightSegments = 1) {
   // buffers
   this.vertexBuffer = null;
   this.indexBuffer = null;
   this.uvBuffer = null;
   this.normalBuffer = null;

   // shader programs
   this.vertexShader = null; // ADD THESE
   this.fragmentShader = null;
   this.program = null;
   . . .
```

Then create a new function **in our Plane class** that takes the gl context as a parameter to create and compile our shader program. **Just copy the shaders from `index.js` for now**:

```
setProgram(gl) {
  this.vertexShader = `
    . . .
  `;

  this.fragmentShader = `
    . . .
  `;

  this.program = createProgram(gl, this.vertexShader, this.fragmentShader);
}
```

Note the createProgram() function must be imported from the textbook's library code (*do this at the top of the file*):

```
import { createProgram } from "../../lib/cuon-utils";
```

Now add the following to the render() function so that the shader program gets compiled on the first frame and webGL switches to that program when we render the object:

```
render(gl, camera) {
  // compile the shader if not already done so
  if (this.program === null) {
    this.setProgram(gl);
  }

  // tell webGL to use this
  gl.useProgram(this.program);
```

NOTE: We also need to change all references of `gl.program` to `this.program` (namely in the getUniformLocation/getAttributeLocation calls) in our render function:

```
 const position = gl.getAttribLocation(this.program, "position");
 const uv = gl.getAttribLocation(this.program, "uv");
 const normal = gl.getAttribLocation(this.program, "normal");
 const modelMatrix = gl.getUniformLocation(this.program, "modelMatrix");
 const normalMatrix = gl.getUniformLocation(this.program, "normalMatrix");
 const viewMatrix = gl.getUniformLocation(this.program, "viewMatrix");
 const projectionMatrix = gl.getUniformLocation(
   this.program,
   "projectionMatrix"
 );
```

You should now see something like this:

Note that the skydome is no longer rendering. That is because the render function in the Sphere class is still pointing to `gl.program`, which is no longer the active program. That is just a variable that the textbook helper function `initShaders()` sets for us, but is not set when we use `gl.useProgram()`.

Second Shader Program

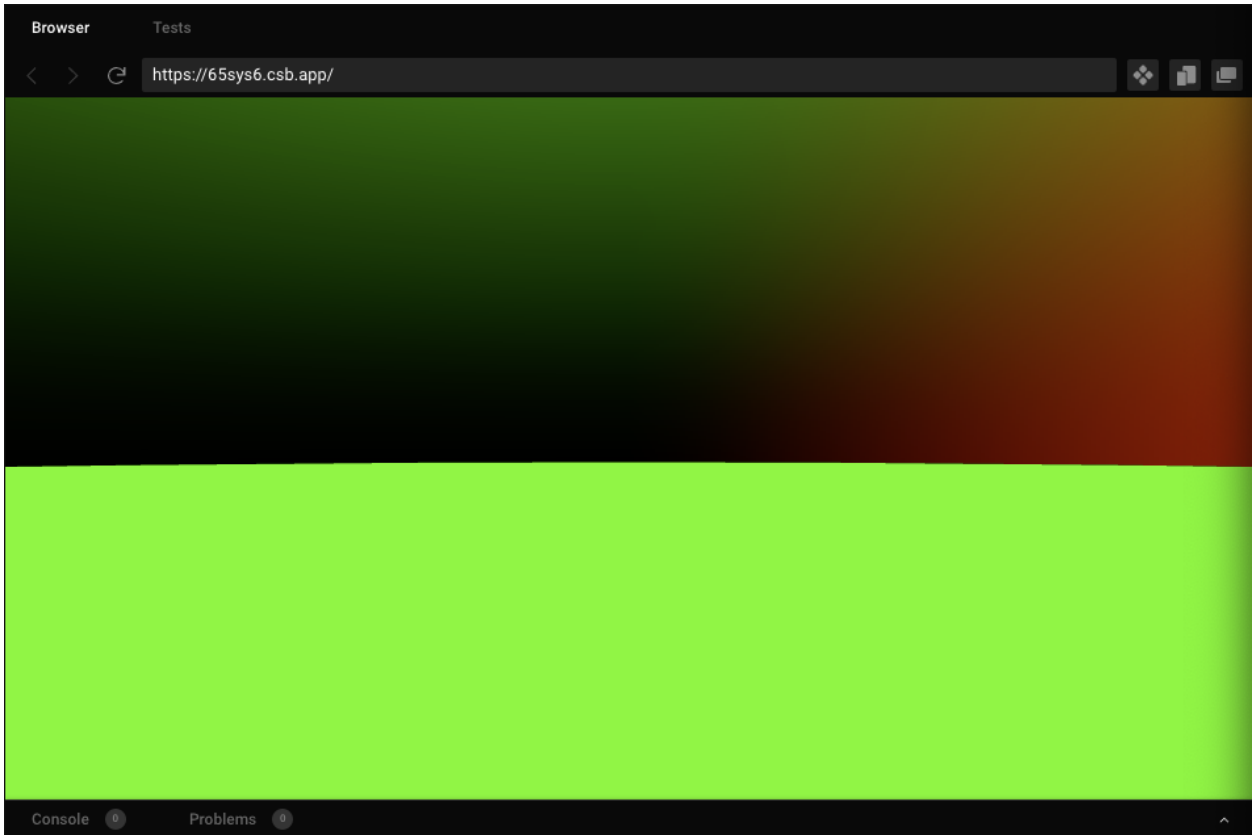
Let's go ahead and get the skydome working again. All we need to do is the same thing we did for our Plane class, but for the sphere class. Head to the src/primitives/Sphere.js file and add the same stuff that we added to the Plane class:

```
// Sphere.js
// import the library function
 import { createProgram } from "../../lib/cuon-utils";

// add shader stuff to the constructor
 constructor(radius = 0.5, widthSegments = 3, heightSegments = 2) {
   . . .

   // shader programs
   this.vertexShader = ``;
   this.fragmentShader = ``;
   this.program = null;

   . . .
   }

// add this function to the class
 setProgram(gl) {
   this.vertexShader = `
     . . .
   `;

   this.fragmentShader = `
     . . .
   `;

   this.program = createProgram(gl, this.vertexShader, this.fragmentShader);
   if (!this.program) console.error("Could not compile shaders for ", this);
 }
```

And, of course, the `render()` function needs to be updated just like we did the Plane class's `render()` function. **Don't forget to change those `gl.program` references to `this.program`!**

```
render(gl, camera) {
  // compile the shader if not already done so
  if (this.program === null) {
    this.setProgram(gl);
  }

  // tell webGL to use this objects program
  gl.useProgram(this.program);

  . . .

  const position = gl.getAttribLocation(this.program, "position");
  const uv = gl.getAttribLocation(this.program, "uv");
  const normal = gl.getAttribLocation(this.program, "normal");
  const modelMatrix = gl.getUniformLocation(this.program, "modelMatrix");
  const normalMatrix = gl.getUniformLocation(this.program, "normalMatrix");
  const viewMatrix = gl.getUniformLocation(this.program, "viewMatrix");
  const projectionMatrix = gl.getUniformLocation(
    this.program,
    "projectionMatrix"
  );

  . . .
}
```

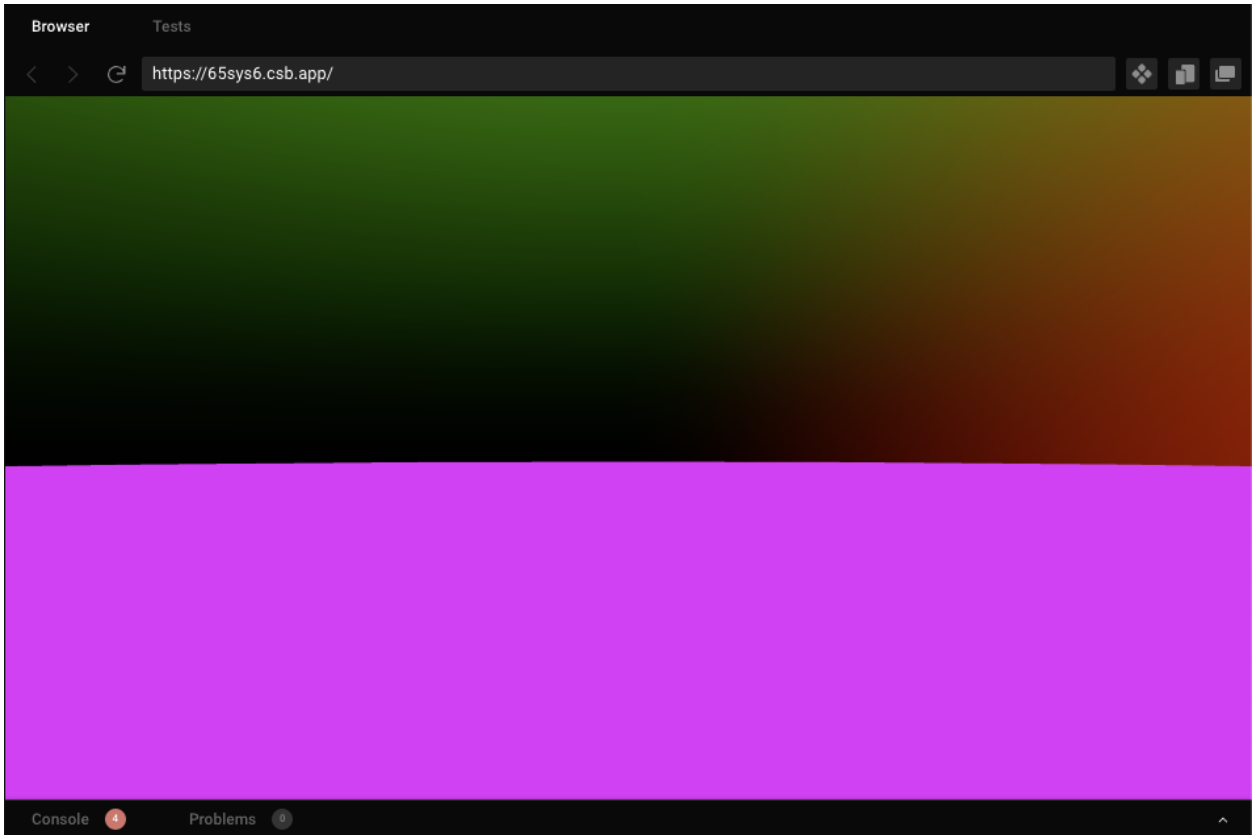
And our skydome should now be back:



To verify that we now have two functioning shader programs, change the Plane class's fragment shader to be a different color than the normal shading we are using currently:

```
// Plane.js in the setProgram() function
this.fragmentShader = `
precision mediump float;
varying vec3 vNormal;

void main() {
  vec3 norm = normalize(vNormal);

  vec3 color = 1.0 - norm; // new color based on normals

  gl_FragColor = vec4(color, 1.0); // note the change here
}
`;
```

You should now see this:



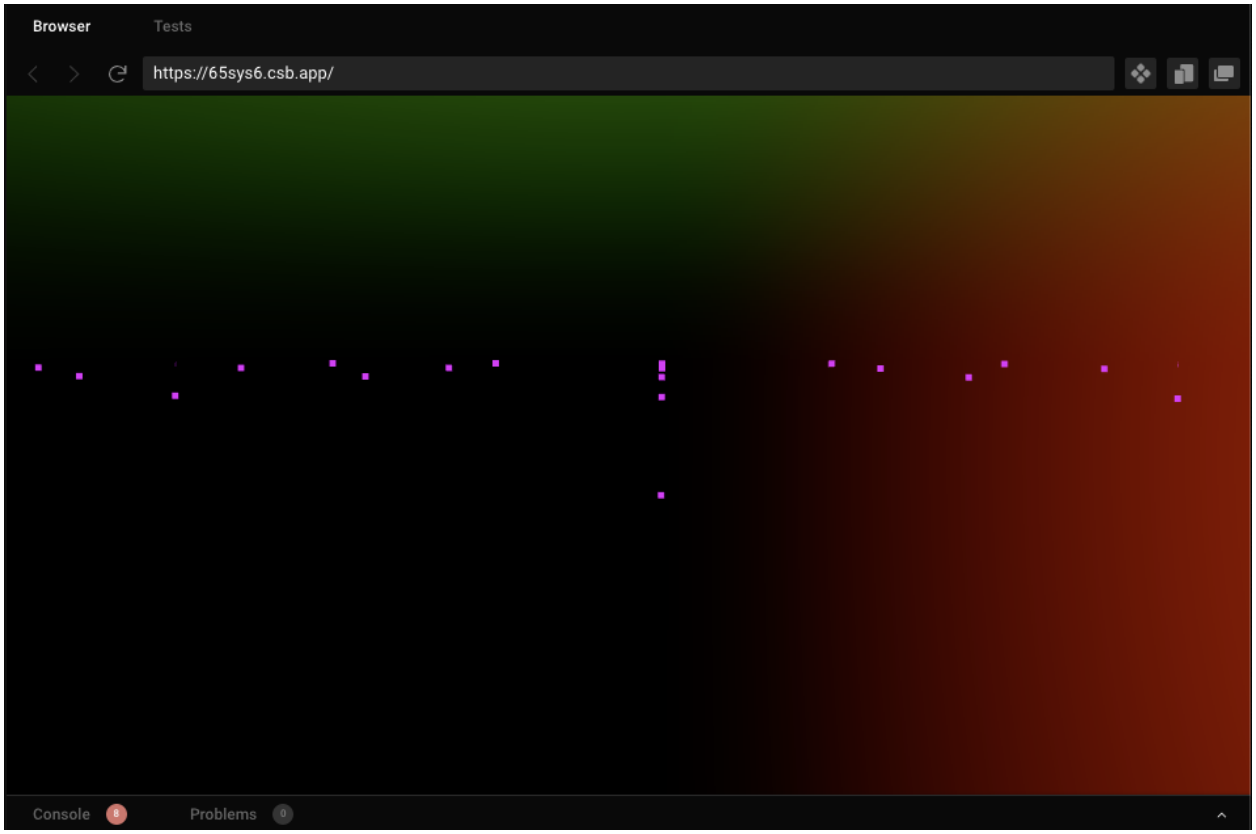
Nice! So we can now create two distinctly shaded objects without issue.

Create An Ocean

Our goal was to create a nice ocean scape with our shaders. So how do we create an ocean out of only a Plane? Well, we can have our plane be composed of not just 4 vertices, but many! Imagine subdividing our plane into an n-by-m-dimensional grid. Well, you don't have to imagine this because our plane class already can do this with arguments to the constructor! Head to where we created our floor Plane and add your desired resolution into the constructor like so (i suggest 10x10 for now):

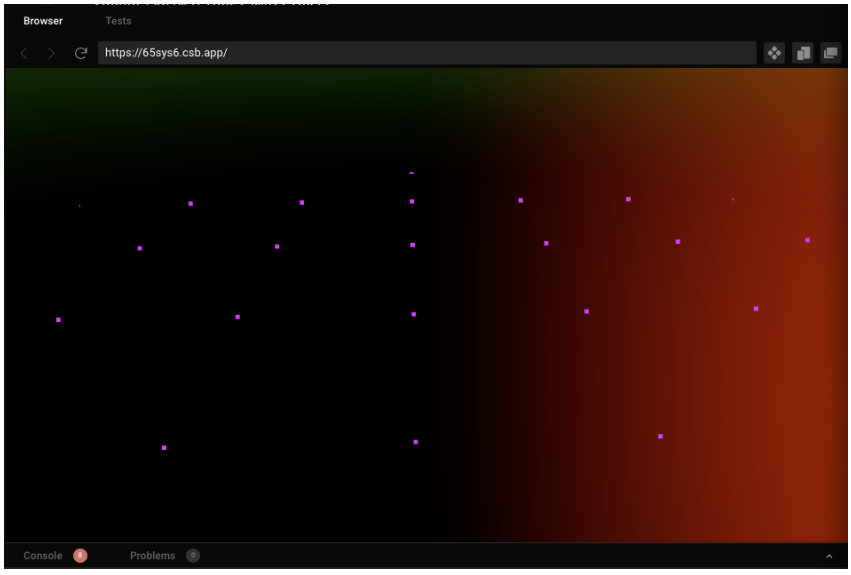
```
const floor = new Plane(10,10);
```

Now there's no visible difference, is there? Well head over to the `Plane.render()` function in `Plane.js` and change the draw mode in the last line from `gl.TRIANGLES` to `gl.POINTS` and you will see this (if they are too small add `gl_PointSize = 5.0;` anywhere in your vertex shader):



Bring your camera a bit higher on the Y-axis so you can see better:

```
// index.js
const camera = new Camera([0, 1, 5], [0, 1, 0]);
camera.position.elements[1] = 15;
```



Our plane is now composed of a 10x10 grid of planes rather than one large singular plane! We can now animate these vertex positions to simulate waves. Go ahead and change the draw mode back to `gl.TRIANGLES` in the `render()` function and bring your camera back down.

One thing we might want to do is make our ocean the

proper color, because ocean's tend not to be pink:

```
// Plane.js in the setProgram() function
this.fragmentShader = `
   precision mediump float;
   varying vec3 vNormal;

   void main() {
     vec3 norm = normalize(vNormal);

     vec3 color = vec3(0.15, 0.35, 0.75); // bluish color

     gl_FragColor = vec4(color, 1.0);
   }
   `;
```

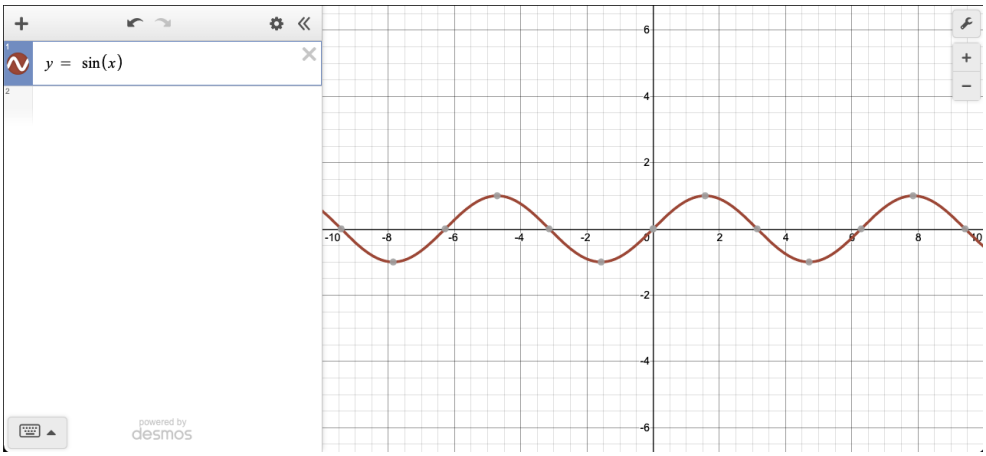
Now, let's get some waves! Head to the vertex shader, and we can alter the position of our vertices using some built in glsl functions. (**Note:** You can use anything you want here, I highly suggest playing around with this stuff, and the code we show below is a product of doing exactly that.)

```
void main() {
     vec4 transformedPosition = modelMatrix * vec4(position, 1.0);

     float waveIntensity = sin(transformedPosition.z) +
       cos(transformedPosition.x);

     transformedPosition.y += waveIntensity;

     gl_Position = projectionMatrix * viewMatrix * transformedPosition;
     vNormal = (normalMatrix * vec4(normal, 1.0)).xyz;
   }
```

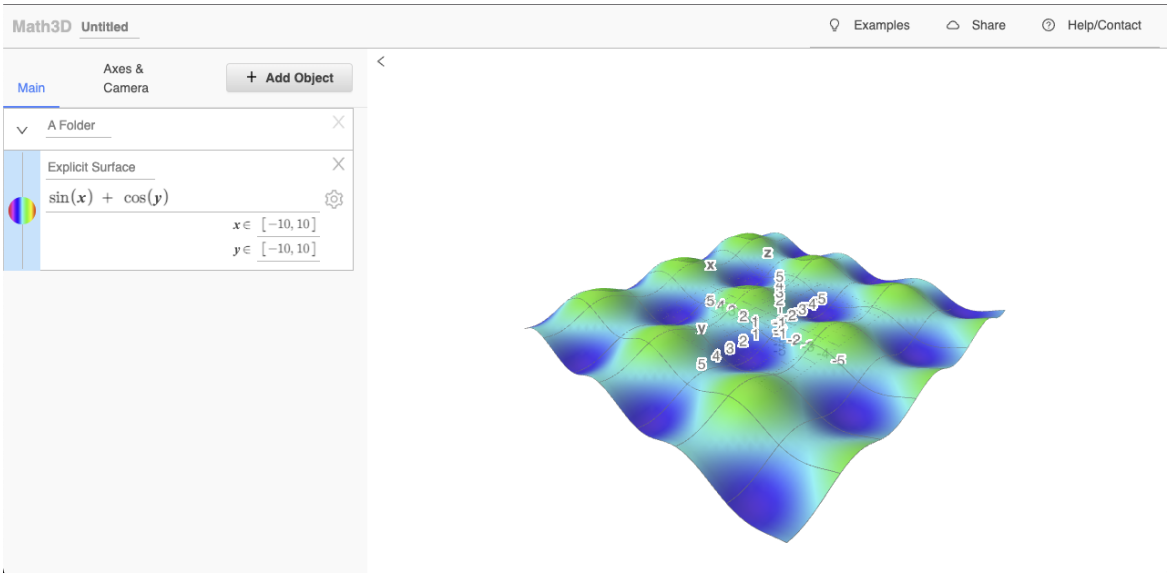
Now there is a couple things happening here:

- We are calculating the transformed position of the vertex first
    - This means we are applying the translations, rotations and scales to our object (e.g. the rotation on our floor in `index.js`)
- We then calculate the sin and cos of the z and x positions of our vertex (remember this plane is on the xz-plane after all transformations) and use that to generate some waves.

- Just like with a regular y = sin(x) function, where as x increases/decreases we create a wave along the y axis:



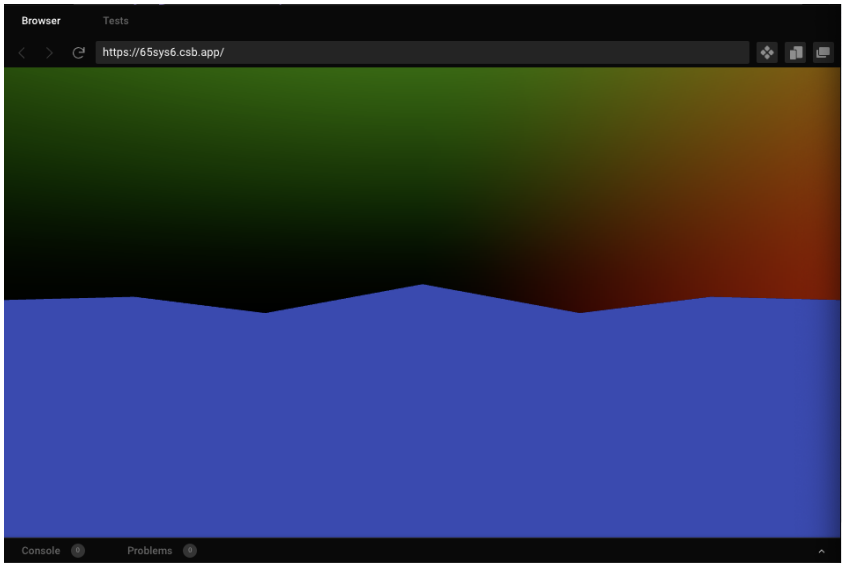
- We can also create 2-dimensional waves across our plane using x *and* z (note that most 3d calculators, including the one below, have z as the vertical axis):



- We then add that generated ‘wave height’ to the y component of our vertex to raise/lower it.

Note: It’s important to know here that the normal vectors are no longer representative of our plane, because the displacement is generative and the normal vector cannot easily be transformed to represent it’s plane’s new orientation. Thus, the normal vectors will still act as if the plane is flat all the way across.

This results in waves that should look similar to these:



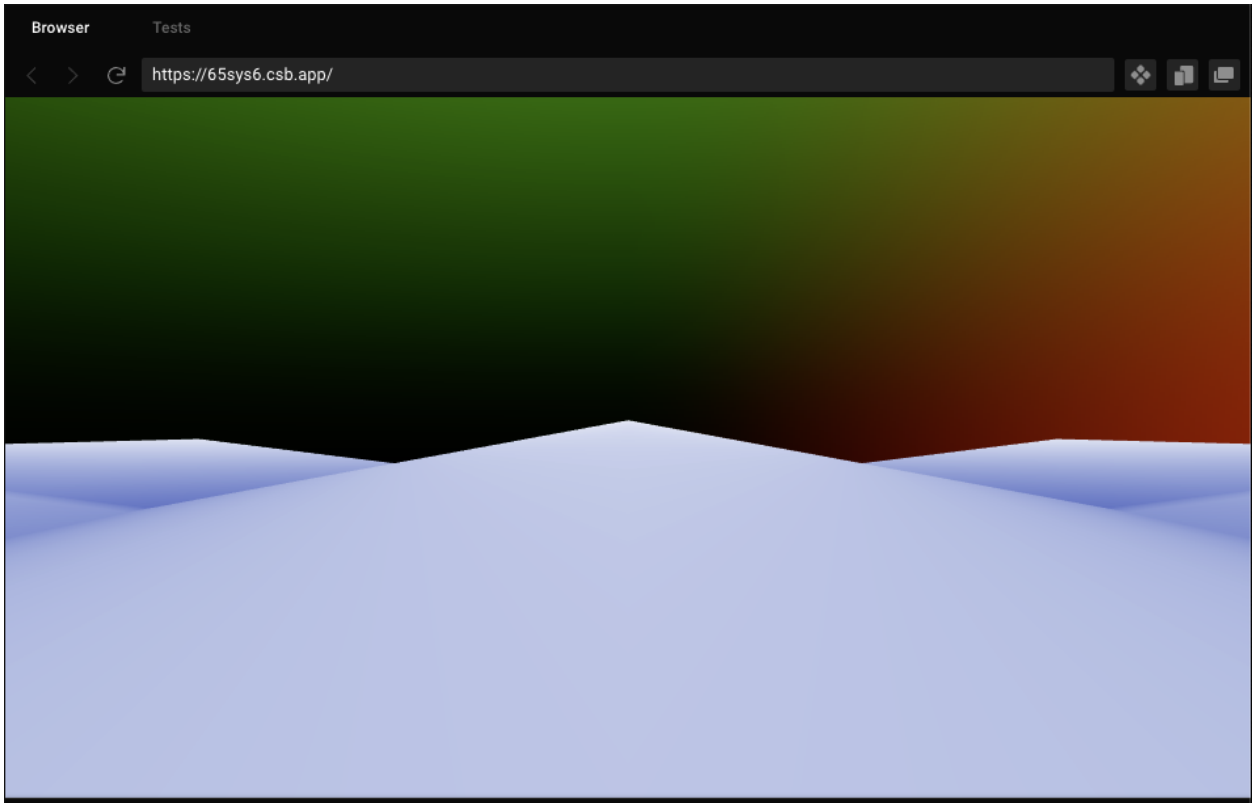
Obviously, this is not very pretty because the entire ocean is the same color. Let's shade it lighter the higher the water is! To do this, we can just pass the `waveIntensity` to the fragment shader as a varying. Note that this intensity can be as low as -2 and as high as 2; ranges in [0,1] are much easier to work with in GLSL and are more intuitive as percentages:

```
// **VERTEX** SHADER
varying float vWaveHeight; // add this

void main() {
  . . .
  vWaveHeight = (waveIntensity + 2.0) * 0.25; // map from [-2,2] to [0,1]
}
```

And the fragment shader:

```
// **FRAGMENT** SHADER
varying float vWaveHeight;

void main() {
  . . .
  vec3 color = vec3(0.15, 0.35, 0.75); // bluish color
  color = mix(color, vec3(1.0), vWaveHeight); // linear interpolation

  gl_FragColor = vec4(color, 1.0);
}
```

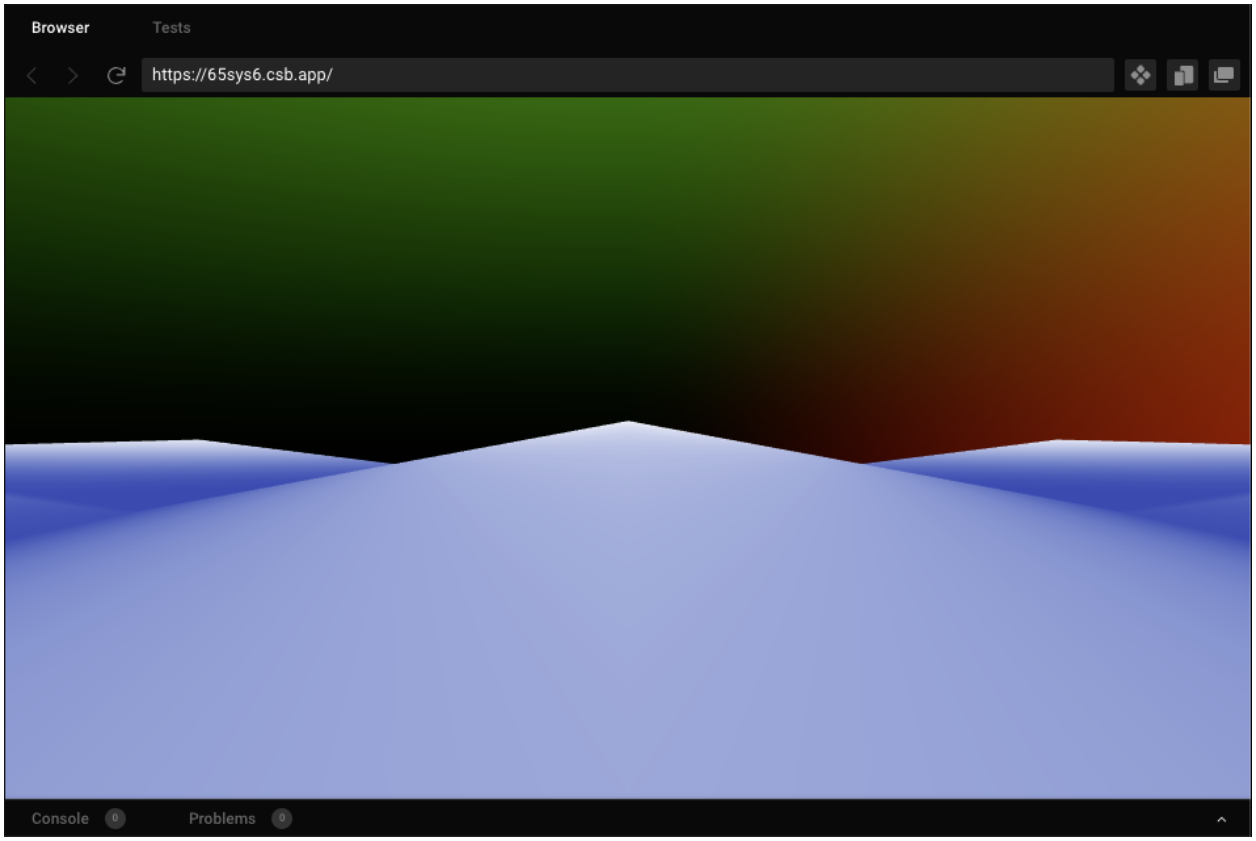
Take a look at those waves now!



Feel free to play with the colors a bit, I suggest using the `smoothstep()` function to ease the `vWaveHeight` varying in the fragment shader and clamp all values below 0.35 to 0.0:

```
color = mix(color, vec3(1.0), smoothstep(0.35, 1.0, vWaveHeight));
```

(Learn more about smoothstep here: https://thebookofshaders.com/glossary/?search=smoothstep)



Now let's get these guys animated. To do this, we will need to pass the current time in as a uniform. Add that uniform in the ***vertex shader*** like so:

```
this.vertexShader = `
  . . .

  uniform mat4 modelMatrix;
  uniform mat4 normalMatrix;
  uniform mat4 viewMatrix;
  uniform mat4 projectionMatrix;
  uniform float uTime; // ADD THIS

  . . .
```

In the `render()` function, we will set this uniform equal to the current time (note that `performance.now()` is in ms and seconds are easier to play with so we divide):

```
render(gl, camera) {
   . . .
   const uTime = gl.getUniformLocation(this.program, 'uTime'); // ADD THIS
   const position = gl.getAttribLocation(this.program, "position");
   const uv = gl.getAttribLocation(this.program, "uv");
   const normal = gl.getAttribLocation(this.program, "normal");
   const modelMatrix = gl.getUniformLocation(this.program, "modelMatrix");
   const normalMatrix = gl.getUniformLocation(this.program, "normalMatrix");
   const viewMatrix = gl.getUniformLocation(this.program, "viewMatrix");
   const projectionMatrix = gl.getUniformLocation(
     this.program,
     "projectionMatrix"
   );

   gl.uniform1f(uTime, performance.now() / 1000); // ADD THIS
   gl.uniformMatrix4fv(modelMatrix, false, this.modelMatrix.elements);
   gl.uniformMatrix4fv(normalMatrix, false, this.normalMatrix.elements);
   gl.uniformMatrix4fv(viewMatrix, false, camera.viewMatrix.elements);
   gl.uniformMatrix4fv(
     projectionMatrix,
     false,
     camera.projectionMatrix.elements
   );

   . . .
 }
```

Now modify the vertex shader to make use of this:

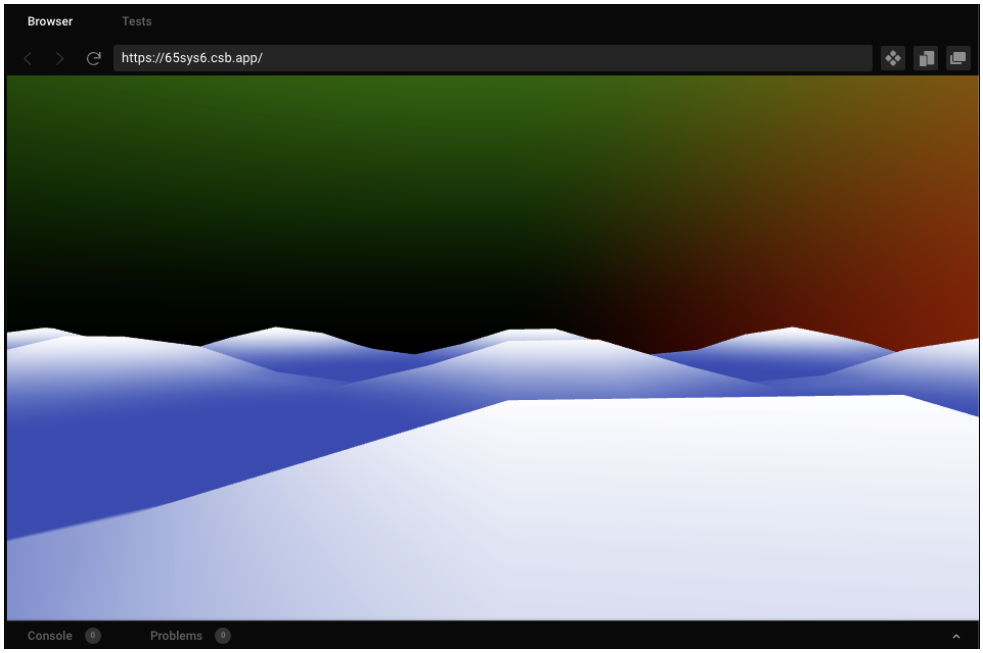
```
void main() {
    vec4 transformedPosition = modelMatrix * vec4(position, 1.0);

    float waveZ = transformedPosition.z + uTime * 0.5; // add this
    float waveX = transformedPosition.x - uTime * 0.2; // add this
    float waveIntensity = sin(waveZ) + cos(waveX); // change this

    transformedPosition.y += waveIntensity;

    gl_Position = projectionMatrix * viewMatrix * transformedPosition;
    vNormal = (normalMatrix * vec4(normal, 1.0)).xyz;
    vWaveHeight = (waveIntensity + 2.0) * 0.25; // map from [-2,2] to [0,1]
  }
```

And you should see your waves moving!

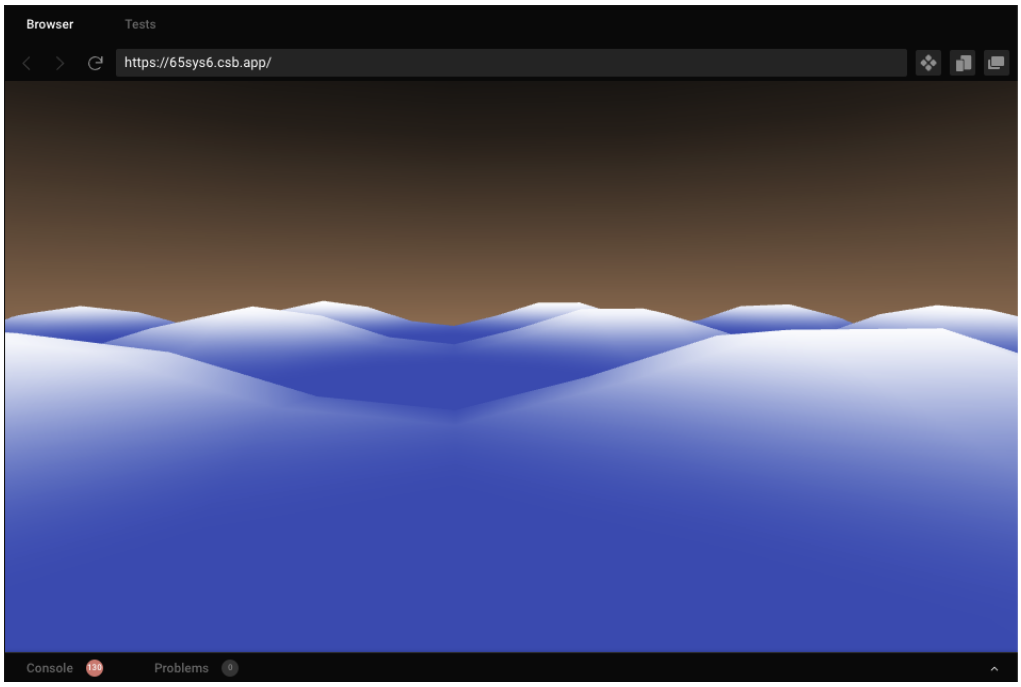
Extras

If you want to play with this further:

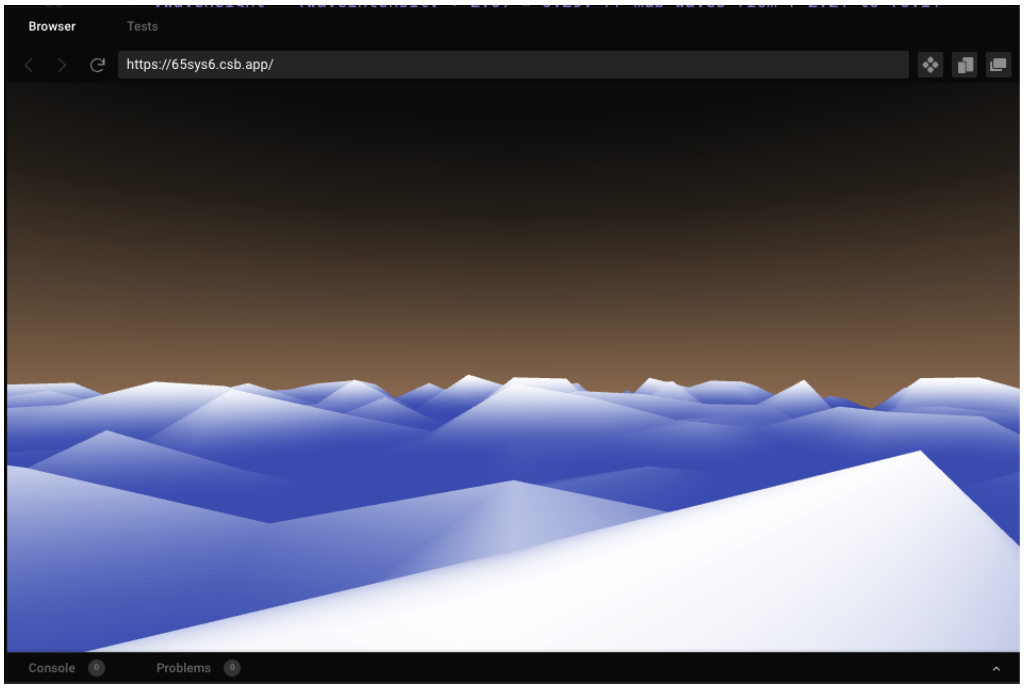
- Try increasing the resolution of your plane and changing the amplitude of the trigonometric functions in the vertex shader for smoother waves (don't fry your computer like I did!!!! 😎):



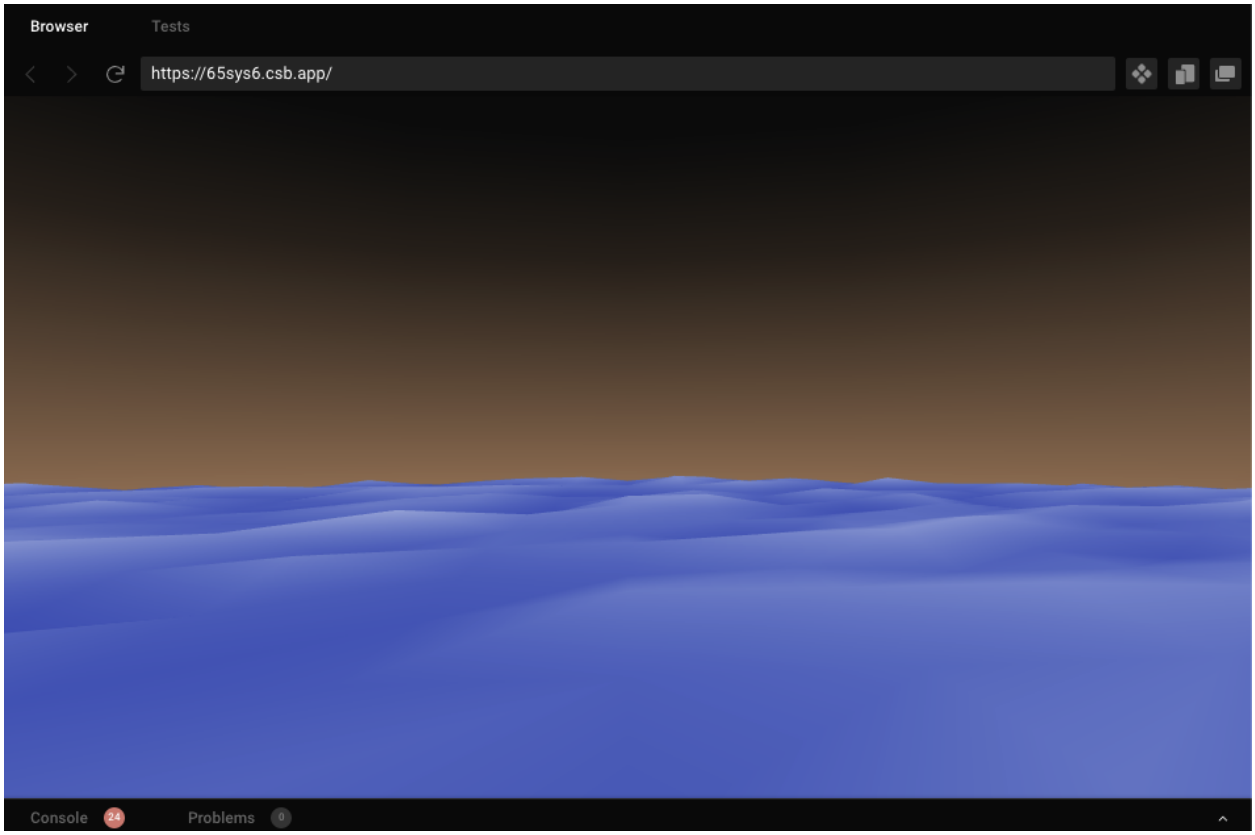
- Do something cool with your skydome shader, for example mix() based on the height of that fragment:



- Add some randomness to the waves using a random function (google glsl random):



- Use noise functions to generate the wave intensity (google glsl noise functions):

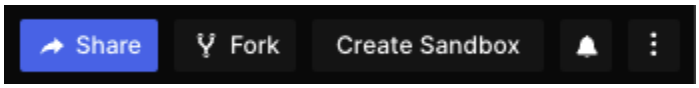


- Use the built in uv's or normals to do some texture mapping or phong lighting

Congrats, You're done for the week!

Submission

To submit this lab assignment, make sure you submit your ***own*** codesandbox link for the lab. This means that you will need to fork the sandbox that you and your group worked on together such that each individual has their own. The fork button is on the top right corner of the page (if not here, it will be in the drop-down menu at the top left under *file* > *fork sandbox*):



Then, you will submit the following on canvas:

- Link to your sandbox (**click on the blue share button and click copy link at bottom**)
- List of groupmates names
- Optionally, if you worked on this alone rather than in the lab section group please comment that here