BUOYANCY SIMULATOR

A-LEVEL COMPUTER SCIENCE PROJECT BY THEO STEWART GRIFFITHS 13H

CANDIDATE NUMBER: XXXX
CENTRE NUMBER: XXXXX

CONTENTS

ANALYSIS

My project is a water-based physics simulation in which the user will be able to control a boat on an ocean surface. This concept is amenable to a computational solution because while physics simulations appear complex, they are a result of objects following simple rules. A computer is able to calculate the outcome of these simple rules mathematically on each object to produce the complete result. In this instance, the upthrust and drag can be calculated at every point on the boat.

Water simulations are very resource-intensive; I aim to produce a simulation which balances both simplicity and accuracy. I have opted to instead focus on the buoyancy aspect rather than the fluid dynamics as that requires mathematics out of the scope of this project and performance found on very few computers.

STAKEHOLDERS

BOAT ENGINEERS
Engineers can import test models of boats or other vehicles before they are manufactured to simulate how they would react in different conditions.

- *The simulation should work for various different objects as the simulation is vertex-based and real time*
- *Can be run on any computer which meets the requirements, which includes most modern computers*
- *The simulation will likely not be able to handle meshes with a large number of vertices*
- *The accuracy of the simulation is limited due to approximations in Unity and within the used algorithms*

PHYSICISTS
Physicists could investigate buoyant behaviour and model how fluids interact with objects immersed in them

- *Parametric behaviour means that the initial conditions can be changed to model different scenarios*
- *Physicists may be more interested in a fluid dynamic simulation rather than a buoyancy simulation*

AVERAGE USER
The average user would likely not have much use with this simulation however in the future the features could be used as the foundational mechanics of a game, however creating a game is out of the intended scope of this project.

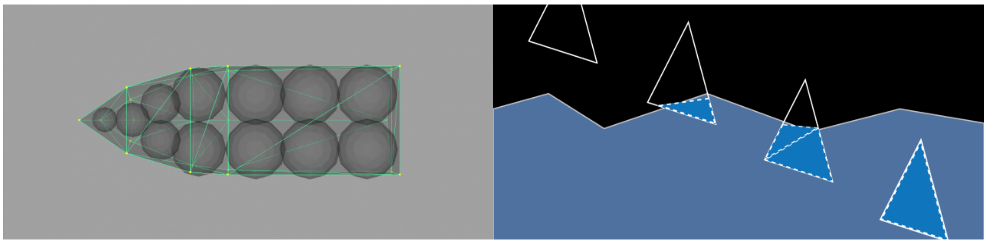
BUOYANCY

$$F_b = -\rho g V$$

Archimedes' Principle states that upthrust is equal to the product of the fluid's density, the strength of gravity and the volume of fluid displaced.

Upthrust is a force that acts on objects in a fluid, caused by the fluid beneath the object resisting compression. In the case of a floating object, the upthrust must be greater than the weight force, which allows them to float.

The total volume of the ship that is submerged must therefore be calculated. There are two ways to calculate this: the volumetric version, which approximates the area using cubes or spheres. This trivialises the rotational forces and therefore does not produce an accurate result. The surfacic version, which measures the displacement per vertex, is the method that makes the most sense in this context.



Volumetric method (left) and surfacic method (right)

SURFACIC METHOD

This algorithm has a time complexity of O(n²) and a space complexity of O(n). Transferring data from the GPU to CPU can cause a bottleneck and impact performance and Unity's physics engine has many limitations, but these should not impact the accuracy to a significant level.

- For each face on a floating object:
    - Calculate the distance, *d*, each vertex is underwater by subtracting the wave height from the vertex height.
    - For each vertex underwater, apply the force to the midpoint of the underwater vertices. The force's magnitude is equal to the amount of the triangle underwater - this can be approximated
    - Calculate the moment by applying the contact force to each vertex.
    - Apply drag forces.

EXISTING SOLUTIONS

https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games

The solution presented here is very complex, however it does produce an accurate result. The main problem is that it is completely CPU-bound, which means that the performance will not be great as the calculations are so complex.

WATER SHADER

$$X(a,b,t) = a + \frac{e^{kb}}{k}\sin(k(a+ct)),$$
$$Y(a,b,t) = b - \frac{e^{kb}}{k}\cos(k(a+ct)),$$

These functions provide the position of a vertex in a Gerstner wave at time t *at position* (a, b)*, with steepness* k *and speed* c.

Fluids in real life follow complex laws. Doing these graphical calculations with a shader will allow me to offload work from the CPU. The GPU is optimised to perform calculations on many vertices at once.

GERSTNER WAVES

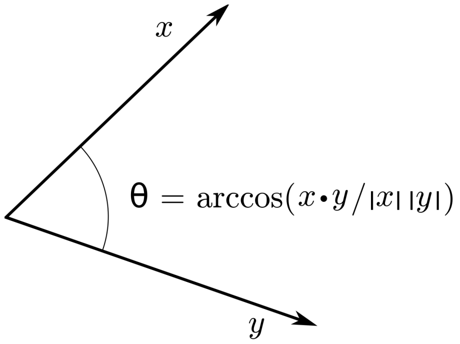
Gerstner waves are a type of trochoidal *wave.* They are often used to simulate the surface of fluids as they oscillate in both the x- and y-axis, like real fluids do. The direction and steepness of the main Gerstner wave will come from the direction and magnitude of the wind. The waves will be perpendicular to the wind and become steeper if the wind increases. To simulate turbulence, I may add extra, smaller Gerstner waves with different parameters.

The limitations of this model include a lack of inclusion of vorticity - the water is assumed to be uniformly moving in the same direction at a constant rate proportional to the speed of the wind. Additionally, the shape and speed of waves is determined by the depth of the water at a certain point, which is not accounted for in this model. These rules are outside of the scope of my project and I do not intend to include them as they do not have a significant impact on the accuracy of the simulation.

APPEARANCE

Water is transparent to light but not completely, resulting in fog the further down you go. This can be implemented by sampling the distance between the water's surface and the next object behind it from the point of the camera, and changing the alpha (transparency) value of the texture at that point accordingly.

SAILING



The strength of the forward force will be proportional to the cosine of the angle between the two vectors x *(the current forward vector of the boat) and* y *(the direction of the wind)*

The sailing system is based around the wind vector, which is the current direction of the waves.

STEERING

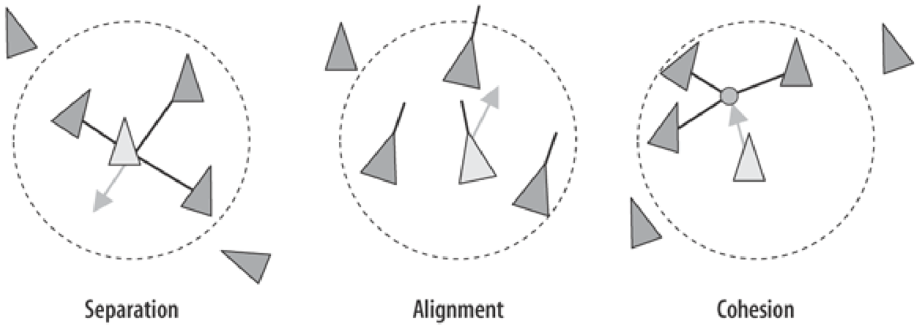
A boat with sails will move based on its orientation with the wind. The forward force the boat experiences is based on the cosine of the angle between the wind and the direction of the boat.

When the sails and wind are aligned, ***θ → 0*** and therefore ***cos θ → 1***, so the forward force will be strong. When the sails and wind are not aligned, ***θ → ±90*** and therefore ***cos θ → 0***. In summary, the boat will accelerate when the sails and wind line up, but slow down when they do not. The user's main controls will therefore be the ability to rotate the boat, and begin its movement, as well as orbit the camera about their selected object.

EXISTING SOLUTIONS

A similar concept exists in the game *Valheim*, in which players steer boats based on the direction of the wind. This method is very similar to what I envision, however I do not need most of the features (rowing, inventory, etc.)

BOIDS



Boids are an example of an emergent system - individuals following simple rules that give rise to complex patterns.

A *boid* is a simulated flock; the original explanation of the boid algorithm by its original author can be found here. Since the simulation is parametric, different values can be passed in to alter the behaviour of the overall flock, allowing me to create a flock of birds and a school of fish using the same code.

Boids take up a large amount of processing resources on the CPU, therefore this part of the project will require significant optimisation. Current options available for this include parallelisation either on the CPU or moving the workload instead to a compute shader for the GPU, which is optimised for vector calculations. The second method, while more efficient, will be significantly more difficult to achieve, and therefore I believe I will instead process the boids on the CPU.

COHESION

In a real flock, when one member comes across another they tend to stick together. This behaviour is represented by the cohesive behaviour of boids. Boids will detect others within their detection radius, and travel towards them. This, over time, causes larger flocks to absorb smaller ones creating an eventual single entity. To negate this, some randomness must also be added to create more chaotic, unpredictable and realistic results.

ALIGNMENT

Individuals tend to steer themselves in a certain direction based on the average of the direction of their neighbours to prevent them from colliding with each other. Over time, the average direction of every bird in the flock will become uniform, so some randomness will have to be added.

AVOIDANCE

Individuals also detect when other individuals get too close and steer to avoid a collision. This means that within a certain radius, a repulsive force must be applied.

FOCUS GROUP

My focus group understood my concept clearly and believed that my solution was a sensible and straightforward way to solve the problem. They agreed with my choice of stakeholders, namely engineers and physicists, but also suggested that the addition of extra features such as a sailing system and flocking system would mean that a game could be created using the features in my project.

CLOTH PHYSICS
The sails could be dynamic and respond to the wind by moving around like real sails would.

- *Cloth physics simulations are very complex*
- *Only an aesthetic feature rather than functional one*

GENERATED LANDMASSES
Landmasses could be generated using Perlin noise

- *Implementing land masses would be a good decision if the project were adapted into a game*

DIFFERENT TYPES OF BOAT
The user would have a choice to spawn different boats with different properties

- *This is a good idea, however I will not be implementing it*
- *2 or 3 different floating objects is enough for the project to function*

PROTOTYPE OVERVIEW

PROTOTYPE	FEATURES + SUCCESS CRITERIA
1	□ Buoyancy basics - sample height per vertex and apply force □ Water basics - model water using sine waves on CPU
2	□ Water shader - model water using Gerstner waves on GPU □ Buoyancy system - calculate upthrust based on volume immersed, including rotational forces
3	□ Sailing basics - wind vector sets direction of waves □ Waves - model water using Gerstner wave on CPU □ Implement wave parameters
X	**Possible features:** □ Water shader polish - add transparency, depth fog, etc. □ Sailing system □ Boids system

REQUIREMENTS

REQUIREMENT	REASON
OS (Windows 10)	The executable file requires Windows 10 to run.
GPU (integrated or discrete) with DX10-DX12 support	The water shader requires DirectX to run. This comes installed on most Windows 10 computers.
CPU with x64/x86 architecture, minimum 2 cores	Unity executables for the Windows platform require instructions found in the x64/x86 instruction set.
Standard peripherals - keyboard, mouse + monitor	For input/output - however essentially all computers have these.

DESIGN: PROTOTYPE 1

These success criteria have been chosen as they will produce a very simple prototype version of the final project. All of the implemented features will have to be refined in the future but as a proof-of-concept these objectives work well.

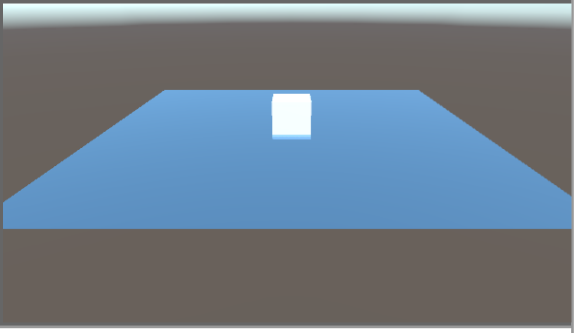
TASK	SUCCESS CRITERIA
Calculate offset of each vertex of an object and apply a force at point	□ Does the object rebound when dropped in water? □ Does the object eventually reach equilibrium on the water’s surface?
Create sine wave approximation	□ Is each vertex’s position modelled by a sine wave through time? □ Is the correct vertex height applied to each vertex?
Create static class instance for wave script	□ Is the wave manager script static?
Generate wave height per vertex individually	□ Does the wave height generate a y-coordinate from an x-coordinate?
Calculate offset of each vertex from wave height and apply force	□ Does the object oscillate with the wave?
Generate parametric water wave	□ Can the wavelength be set? □ Can the wave speed be set? □ Can the amplitude be set?
Move vertex manipulation to shader	□ Does the wave shader create the same wave as the CPU?
Upthrust force is proportional to the normal to the water’s surface	□ Do floating objects float perpendicular to the water’s surface?

DEVELOPMENT LOG: PROTOTYPE 1

ABSTRACTION/PSEUDOCODE

TASK 1: Each vertex on a given mesh needs to be sampled to check if it's height is below the water. Based on the height, a force will be applied at this location.

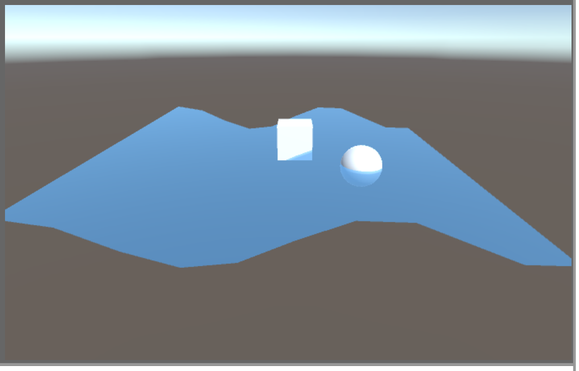
```
PROCEDURE buoyancy()
   FOREACH vertex IN object:
      IF vertex.globalPosition.y < 0
         ApplyForceAtPoint(vertex, vertex.y * Vector3.up)
   NEXT vertex
ENDPROCEDURE
```



- Any mesh in the simulation with the *BuoyantObject* component will now float and reach equilibrium when the gravitational force is equal to the sum of upthrust on each vertex.
- Some meshes with many vertices close together, such as a sphere, exhibit jittery behaviour. The solution is to increase the angular drag on the Rigidbody of the affected object.

TASK 2: The water's vertices must oscillate through the y-axis following a sine wave over time.

```
FUNCTION wave(Vector[] verts)
   Vector[] newVerts = verts;
   FOREACH vert IN newVerts:
      vert.y = sin(vert.x + time) * amplitude
   NEXT vertex
   RETURN newVerts
ENDFUNCTION
```



- The vertex manipulation will later be done on the GPU using a shader. GPUs offer better parallelisation, which is suited for graphical operations like this.

TASK 3: The wave manager class must be a static instance so that the wave height can be sampled by both the physics and graphics scripts.

- Using a singleton design pattern here means that I will be able to access the height of a wave at a certain point much faster, optimising the program significantly.

- Creating the singleton in the *OnEnable()* function did not correctly instantiate the singleton, however moving it to *Awake()* made it function correctly. This is because the *Awake()* function is called earlier.
- Singletons are very useful but often regarded as over time you can lose track of what is editing their values at runtime. In this project the singleton classes will not have many externally changing values.

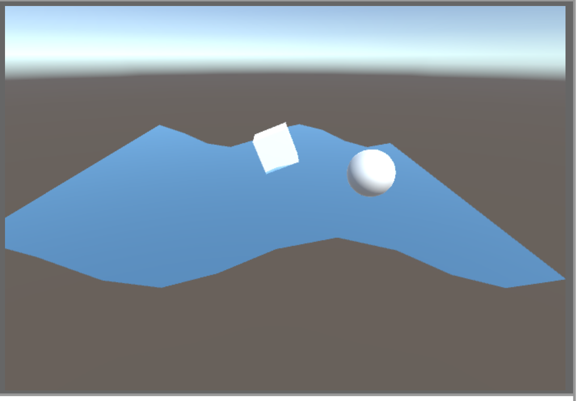
TASK 4: Instead of generating every vertex at the same time, one function should be used which gives the wave's height at a given x-coordinate.

```
FUNCTION getHeight(float x)
   RETURN sin(x - time) * amplitude
ENDFUNCTION
```

- The advantages of doing this are that the wave equation only needs to be changed in one place, and copied over to the shader. The wave equation can be used for both the CPU-based physics calculation as well as the GPU-based water simulation.

TASK 5: On a floating object, the height under the wave should be calculated by sampling the wave equation. The buoyant force should be applied if the vertex is underwater.

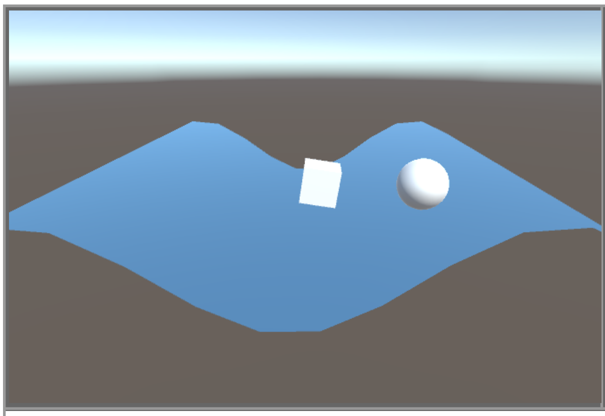
```
PROCEDURE buoyancy()
   FOREACH vertex IN object:
      d = vertex.y - getHeight(vertex.x)
      IF d < 0
         ApplyForceAtPoint(vertex, -d * Vector3.up)
   NEXT vertex
ENDPROCEDURE
```

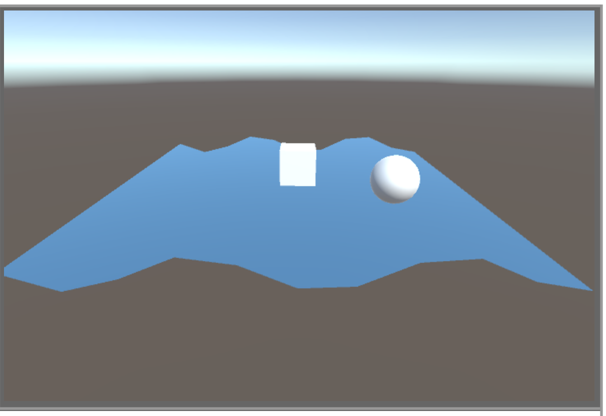


- The vertex manipulation will later be done on the GPU using a shader. GPUs offer better parallelisation.

TASK 6: Important parameters - the amplitude, wave speed and wavelength - should be controlled from the editor, by changing the wave equation to $a * sin(2\pi/\lambda(x - v\,t))$, where a = amplitude, λ = wavelength, v = wave speed

```
FUNCTION getHeight(float x)
   k = 2 * PI / wavelength
   RETURN amplitude * sin(k * (x - speed * time))
ENDFUNCTION
```

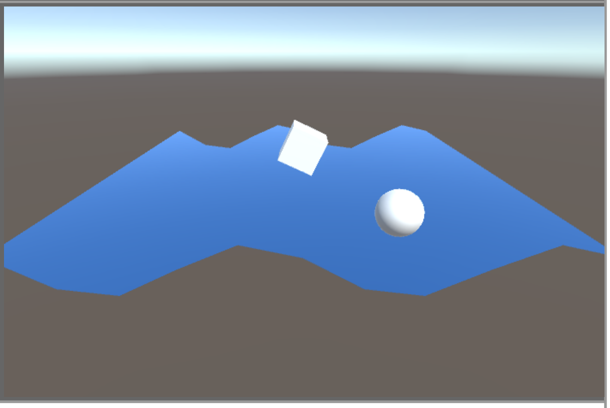




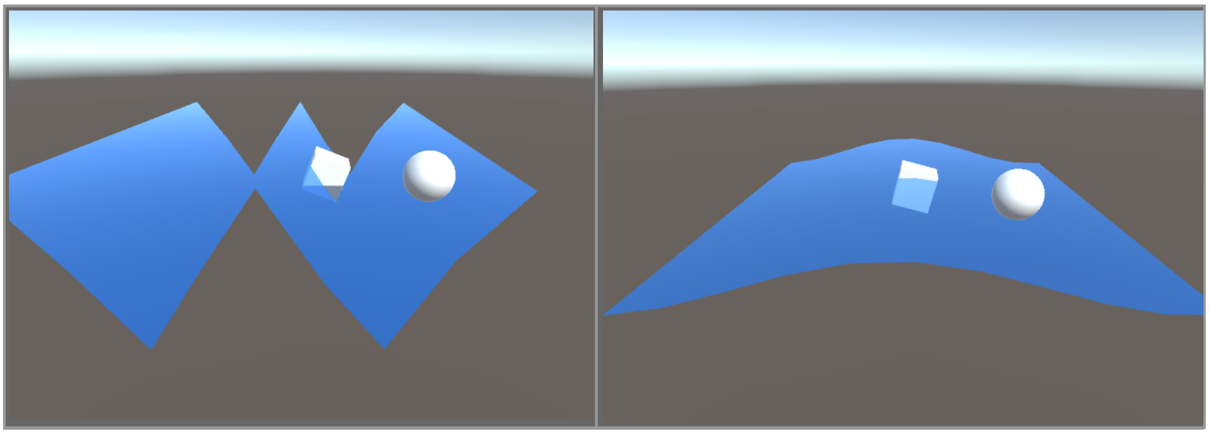
- This presented no problems as I already had the wave equation in my research. The results were in line with what I expected.
- Changing this to a Gerstner wave may present some problems as the x- and y-coordinate of a point are both defined by a function in trochoidal waves, so the relationship between y and x is not mapped by a function.

TASK 7: The GPU is better suited for vertex manipulation, so instead of manipulating the vertices on the CPU they should be changed on the GPU using a shader.

```
PROCEDURE vertex(vertexData v)
   k = 2 * PI / wavelength
   v.y = amplitude * sin(k * (v.x - speed * time))
   update vertex
ENDPROCEDURE
```



- After learning the basics of GLSL in preparation for this project, implementing the shader's basics was straightforward.
- The custom shader has a few issues to resolve:
    - ***Shadows are not cast properly on the water's surface***
    - ***The water's transparency does not work correctly (appears opaque in front of some objects)***
    - ***The CPU and GPU wave equations can be modified separately, causing them to create separate waves as shown below:***



TASK 8 [BUG #01]: The CPU wave equation should obtain its parameters directly from the GPU wave equation to prevent them from changing independently.

```
PROCEDURE OnEnable()
   IF material != NULL
      amplitude = material.amplitude
      speed = material.speed
      wavelength = material.wavelength
   ENDIF
ENDPROCEDURE
```

- Now, both forms of the wave function have identical parameters which are uniformly controlled by the water material, which uses the custom water shader.
- After testing, I found that if the material’s parameters are changed during runtime they will not be copied to the *WaveGenerator* script.
- Therefore, this needs to go the other way - the material’s parameters must be updated from the *WaveGenerator* script.

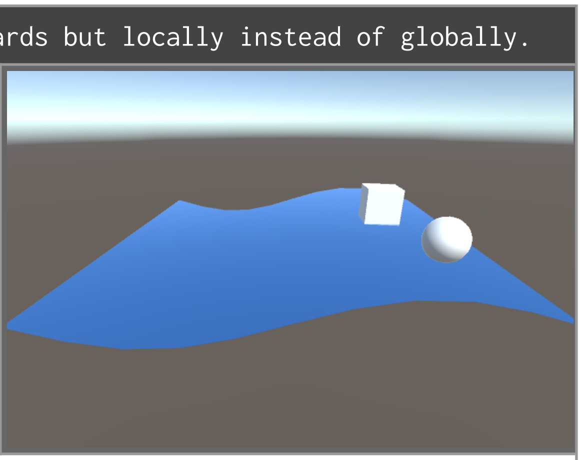
TASK 9: The GPU wave equation should obtain its parameters directly from the CPU wave equation to prevent them from changing independently.

```
PROCEDURE Update()
   IF material != NULL
      material.amplitude = amplitude
      material.speed = speed
      material.wavelength = wavelength
   ENDIF
ENDPROCEDURE
```

- To fix the previous issue I moved the previous code to the *Update()* function. The parameters will eventually be controlled by UI instead of sliders in the Editor.

TASK 10: The buoyant force should act upwards but locally instead of globally.

```
PROCEDURE buoyancy()
   FOREACH vertex IN object:
      d = vertex.y - getHeight(vertex.x)
      IF d < 0
         ApplyForceAtPoint(vertex, -d * Vector3.localUp)
      ENDIF
   NEXT vertex
ENDPROCEDURE
```

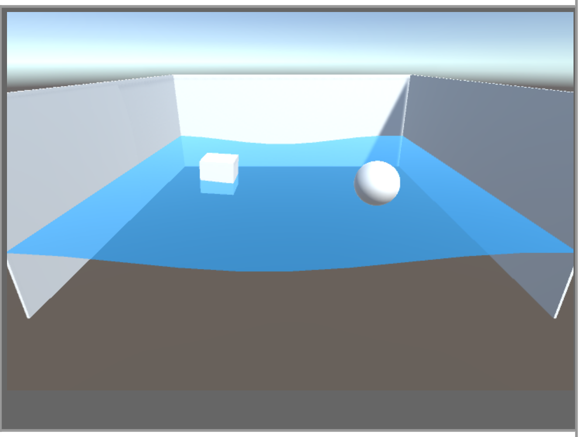


- Testing this produced expected results. Objects are moved up on the local y axis instead of the global y axis.
- This is a temporary approximation for the actual correct calculation - the ‘true’ upwards vector is the normal vector to the wave at a given point. For this prototype, this approximation satisfies the success criteria so it can be used instead.

TASK 11 [BUG #02]: Meshes with a high number of verts behave erratically.

- Meshes with a high number of vertices display a larger buoyant force than meshes with a low number of vertices. This is because in this prototype, I approximate the surfacic method by applying force per vertex rather than per face.
- There are 3 temporary fixes I have for this problem:
  - ***Increase the distance a vertex has to be beneath the water before a force is applied to it.***
  - ***Increase the drag on each of the rigidbodies to increase their resistance to motion.***
  - ***Increase the magnitude of the force to reduce the jittering.***
- I tested all three and the first and third produced unusual results, making the second the one I chose.
- In order to keep gravity functioning, I now had to apply the gravitational force per vertex in addition to the upthrust force.

```
PROCEDURE buoyancy()
   FOREACH vertex IN object:
      d = vertex.y - getHeight(vertex.x)
      IF d < 0
         ApplyForceAtPoint(vertex, -d * Vector3.localUp)
      ENDIF
      ApplyForceAtPoint(vertex, Vector3.down / object.vertices.length)
   NEXT vertex
ENDPROCEDURE
```



- Overall this produces a much smoother floating behaviour, however the jittering can return when the sphere lands on the crest of the wave. This completes the last success criteria for the first prototype.

TESTING

Iterative feature tests are documented in the development log

TEST	COMMENTS
INTERNAL	● I tested some of Unity's other inbuilt meshes such as the capsule and plane. They do float however tall objects which should tip over do not. This will be fixed in prototype 2.
USER 1	● The objects float on the water in a simple but quite realistic way. The water waves look quite simplistic but this is only the first prototype. ● I like how the simulation looks - sometimes there are glitches with the lighting. *The user is describing lighting artifacts. To fix this, I changed the Lighting settings in Unity by setting the colour space from linear to gamma.*
USER 2	● The sphere floats better than the cube but the difference is only noticeable if you look closely when the wave has a high amplitude. ● When I change the strength slider the objects are not affected. ● Also, the water casts a shadow beneath it which looks unnatural. *The variable* strength *on the* WaveGenerator *class should be used for every floating object. To fix this I moved it to the* WaveGenerator *singleton so that it was a static global variable. To fix the shadow issue I added the tag* noshadow *to the water.*

EVALUATION

CRITERIA	EVALUATION
□ Does the object rebound when dropped in water? □ Does the object eventually reach equilibrium on the water's surface?	These criteria were the first to be met in the prototype, and have been met completely.
□ Is each vertex's position modelled by a sine wave through time? □ Is the correct vertex height applied to each vertex?	These criteria are specific to the first prototype, however they are met.
□ Is the wave manager script static?	Singletons are generally considered bad practice in large projects however in this instance I believe that there is no issue with using them.
□ Does the wave height generate a y-coordinate from an x-coordinate?	At runtime, a static instance of this class is created to manage the water waves, by controlling the parameters of the CPU- and GPU-based models. However, as mentioned before, when the GPU approach is changed to a trochoidal wave, the *SampleWaveHeight()* function will also have to be changed. Trochoidal waves are produced from a combination of the oscillations in the x- and y-axis, so the approach used here to sample the value will not work on a Gerstner wave.
□ Does the object oscillate with the wave?	This criteria has been met however the result is unrealistic as the objects ignore rotational forces.
□ Can the wavelength be set? □ Can the wave speed be set? □ Can the amplitude be set?	These criteria have been met and the wave has variable parameters. In the next prototype I must establish a sensible range for these values as user 2 noticed that extreme values will affect the accuracy of the simulation.
□ Does the wave shader create the same wave as the CPU?	This has also taught me the basics of GLSL as I will have to add more to this shader in subsequent prototypes.
□ Do floating objects float perpendicular to the water's surface?	Objects definitely float perpendicular to the surface but as mentioned before this is not entirely accurate.

WaterShader Class (GLSL)

```
//water parameters
        float _Amplitude, _Wavelength, _Speed;

        // procedure applied to each vertex
        void vert (inout appdata_full vertexData)
        {
            float3 v = vertexData.vertex.xyz;

            //wave number
            float k = 2 * UNITY_PI / _Wavelength;
            v.y = _Amplitude * sin(k * (v.x - _Speed * _Time.y));
            vertexData.vertex.xyz = v;

        }
```

WaveGenerator Class (C#)

```
public class WaveGenerator : MonoBehaviour
{
    //### singleton
    public static WaveGenerator Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;
    }

    //### reference variables
    MeshFilter filter;
    public Material waterMaterial;
    //### wave equation parameters with default values;
    [Range(0f, 1f)] public float amplitude = 0.5f;
    [Range(1f, 20f)] public float wavelength = 5f;
    [Range(-10f, 10f)] public float speed = 1f;
    //### simulation parameters
    [Range(0.1f, 15f)] public float strength = 9.81f;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
    }

    void Update()
    {
        if (waterMaterial != null)
        {
            waterMaterial.SetFloat("_Amplitude", amplitude);
            waterMaterial.SetFloat("_Wavelength", wavelength);
            waterMaterial.SetFloat("_Speed", speed);
        }
    }

    Vector3[] GenerateWaveVertices(Vector3[] verts)
    {
        Vector3[] newVerts = verts;
        for (int i = 0; i < newVerts.Length; i++)
        {
            newVerts[i].y = SampleWaveHeight(newVerts[i].x);
        }
        return newVerts;
    }

    /// <summary>
    /// using wave equation 1
    /// f(x) = a * sin(k(x - ct))
    /// </summary>
    /// <param name="x">the x value</param>
    /// <returns>the y value</returns>
    public float SampleWaveHeight(float x)
    {
        float k = 2 * Mathf.PI / wavelength;
        return Mathf.Sin(k * (x - speed * Time.time)) * amplitude;
    }

    public Vector3 SampleNormal(float x)
    {
        float k = 2 * Mathf.PI / wavelength;
        float y = amplitude * Mathf.Cos(k * (x - speed * Time.time));
        Vector3 n = new Vector3(-y, 1, 0);
        return n.normalized;
    }
}
```

BuoyantObject Class (C#)

```
[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(Rigidbody))]
public class BuoyantObject : MonoBehaviour
{
    //### reference variables
    Rigidbody rb;
    MeshFilter filter;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
        rb = GetComponent<Rigidbody>();
    }

    public void Update()
    {
        ApplyBuoyancy();
    }

    /// <summary>
    /// calculates and applies buoyant force to each vertex
    /// </summary>
    void ApplyBuoyancy()
    {
        Vector3 gravity = WaveGenerator.Instance.strength *
                          Vector3.down / filter.mesh.vertices.Length;

        //for every vertex in the mesh
        foreach (Vector3 vertex in filter.mesh.vertices)
        {
            //convert local to global position
            Vector3 vt = transform.TransformPoint(vertex);
            float d = vt.y - WaveGenerator.Instance.SampleWaveHeight(vt.x);
            //if immersed, apply upward force
            if (d < 0f)
                                                     rb.AddForceAtPosition(-d    *
WaveGenerator.Instance.SampleNormal(vt.x) *
                                      WaveGenerator.Instance.strength,
                                      vt, ForceMode.Acceleration);

            //apply gravity
            rb.AddForceAtPosition(gravity, vt, ForceMode.Acceleration);
        }
    }
}
```

DESIGN: PROTOTYPE 2

After the success of the first prototype, the focus is now on improving the physics calculations to create a more accurate, more realistic simulation, as well as implementing a very useful feature with the camera controller.

TASK	SUCCESS CRITERIA
Upthrust force is proportional to the normal to the water's surface	□ Do floating objects float perpendicular to the water's surface?
Gerstner wave on the GPU	□ Does the water deform in a smooth parametric Gerstner wave?
Gerstner wave on the CPU	□ Can the x-position be approximated using any relatively fast approximation method □ Can the tangent function to the Gerstner wave be found? □ Can the normal function to the Gerstner wave be found? □ Do floating objects float perpendicular to the water's surface?
Camera controller	□ Can the user select different objects to follow? □ Can the user freely move the camera while in Free Mode? □ Can the user orbit the selected object in Follow Mode?

DEVELOPMENT LOG: PROTOTYPE 2

ABSTRACTION/PSEUDOCODE

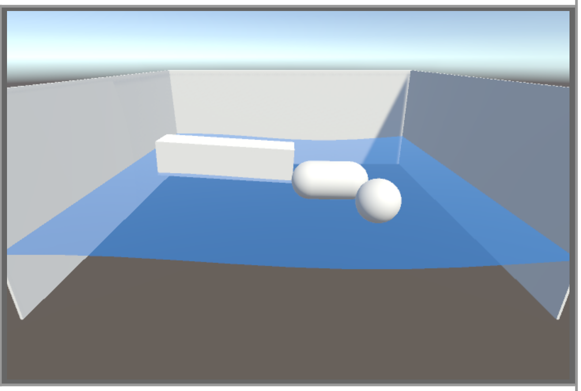
TASK 1: The buoyant force should be proportional to the normal to the water's surface. In prototype 1 this was done using the local 'up' vector on each vertex, but this is not equal to the normal when the object is rotated.

$$P(x) \;=\; [\,x,\; a\,sin[k(x - ct)]\,] \;\therefore\; P'(x) \;=\; [\,1,\; ka\,cos[k(x - ct)]\,]$$

$$P'(x) \bullet N(x) \;=\; 0$$

$$N(x) \;=\; [-\,P'y,\; P'x,\; 0] \;=\; [\;-\,ka\,cos[k(x - ct)],\; 1,\; 0\,]$$

```
FUNCTION getNormal(x)
   k = 2 * PI / wavelength
   y = amplitude * cos(k * (x - speed * time))
   Vector3 normal = (-y, 1, 0)
   RETURN normal.normalised;
ENDFUNCTION
```



- This calculates the normal to the water's surface, which produces much more accurate behaviour than before. Tall objects now rotate and fall over to land on their largest side.
- For now, the waves still use a sine wave. The maths will be more complex for the parametric wave but the principle will be the same.

TASK 2: The water shader also needs to contain the normal vectors for each vertex to create correct lighting.

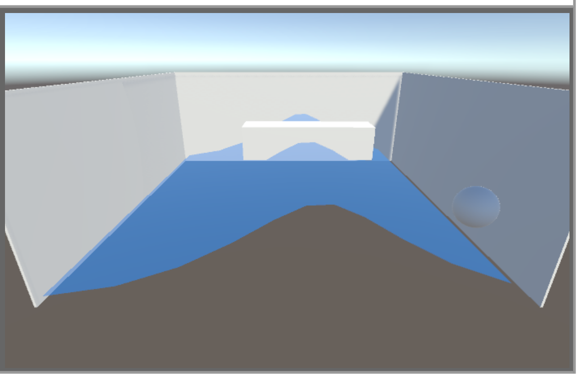
```
PROCEDURE vertex(vertexData v)
   k = 2 * PI / wavelength
   x = k * (v.x - speed * time)
   v.y = amplitude * sin(x)
   v.normal = new Vector3(-k * amplitude * cos(x), 1, 0).normalised;
   update vertex
ENDPROCEDURE
```

TASK 3: The water waves on the GPU should be modelled by a Gerstner wave instead of a sine wave.

$$P(x,\ y)\ = [\,x\ +\ a\cos[k(x\,-\,ct)],\ \ a\sin[k(x\,-\,ct)]\,]$$

View on desmos.com

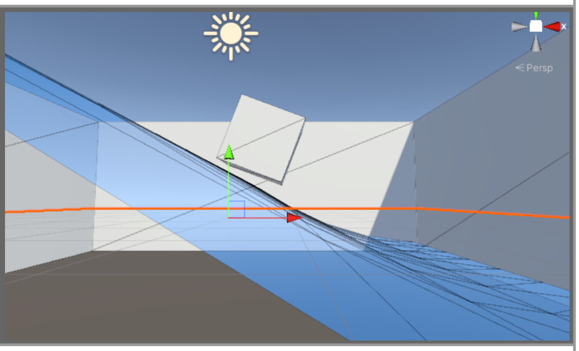
```
PROCEDURE vertex(vertexData v)
   k = 2 * PI / wavelength
   x = k * (v.x - speed * time)
   v.x += amplitude * cos(x)
   v.y = amplitude * sin(x)
   v.normals.calculate
   update vertex
ENDPROCEDURE
```



TASK 4: The CPU model must also use a Gerstner wave model to generate the heights.

- Previously, height was a function of x-position. Now, both the x- and y-positions are given by the multivariable function, so obtaining the height will be more difficult.
- Initially I will just sample the height however this will not produce an accurate result.

```
FUNCTION getHeight(x)
   k = 2 * PI / wavelength
   RETURN amplitude * sin(k * (x - speed * time))
ENDFUNCTION
```



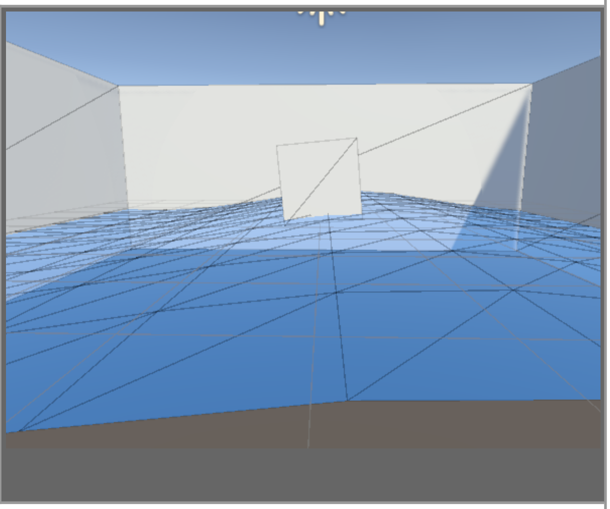
- As we can see above, the horizontal displacement which is a key component of the Gerstner wave is not accounted for in the calculation, which causes a noticeable inaccuracy.
- This discrepancy will be fixed in the future using approximations to reduce the horizontal offset.

TASK 5: The CPU model must also use a Gerstner wave model to generate the heights.

- I can use an approximation to find the vertical position of the Gerstner wave at a given coordinate, by using the Newton-Raphson method. This is where you can step back in a function to find a root of an equation.
- I intend to use one iteration for now as the time cost increases for each, providing diminishing returns.

$$x_{+2} = x_{+1} - dx,\ dx = x - x_{+1}$$

```
FUNCTION Gerstner(x)
   f = k(x - ct);
   Vector2 P,
   P.x = x + a * cos(f);
   P.y = a * sin(f);
   RETURN P
ENDFUNCTION

FUNCTION GetHeight(X0)
   X1 = Gerstner(X0).x
   X2 = 2 * X0 - X1
   RETURN Gerstner(X2).y
ENDFUNCTION
```



- This solves the main problem of the object's x-displacement being inaccurate however now we are back to the object being displaced just vertically rather than at a normal to the water's surface

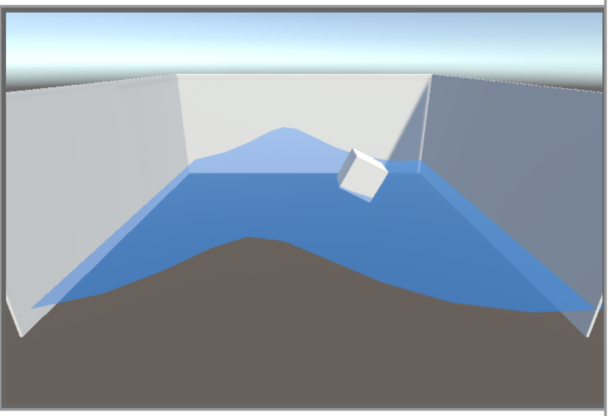
TASK 6: The CPU model must use a Gerstner wave model to generate the normals.

- The force applied to an object is always perpendicular to the water's surface. This must be accounted for in the simulation however the normal to a Gerstner wave requires more complex calculations.
- We must find the derivative of a Gerstner wave with respect to the x-coordinate, and then find the normal of the resulting tangent function.

$$P(x,\ y) = [\,x + a\cos[k(x - ct)],\ a\sin[k(x - ct)]\,,\ 0\,]$$
$$P'(x,\ y) = [\,1 - ka\sin[k(x - ct)],\ ka\cos[k(x - ct)]\,,\ 0\,]$$

$$N(x,\ y) = [\,-P'y,\ P'x,\ 0\,] =$$
$$[\,-ka\cos[k(x - ct)],\ ka\sin[k(x - ct)],\ 0]$$

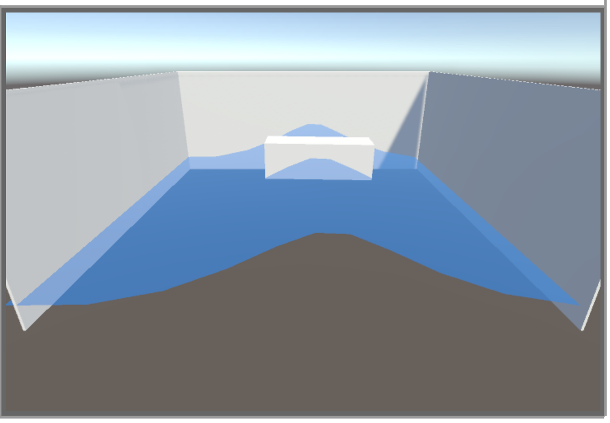
```
FUNCTION GetNormal(X0)
   X1 = Gerstner(X0).x
   X2 = 2 * X0 - X1
   f = k(x - ct);
   Vector2 P,
   P.x = 1 - k * a * sin(f);
   P.y = k * a * cos(f);
   RETURN (-P.y, P.x, 0)
ENDFUNCTION
```



- Using the approximation for x, we can find the normal vector at any given point on the wave. The normal force can then be calculated by multiplying this vector.

- Overall the accuracy of the resulting simulation is very good. I tested cubes, rectangles, spheres and capsules with increasing numbers of vertices and saw consistent, predictable behaviour. Furthermore, even with 5 high-vert objects the reported framerate did not drop below 100FPS.

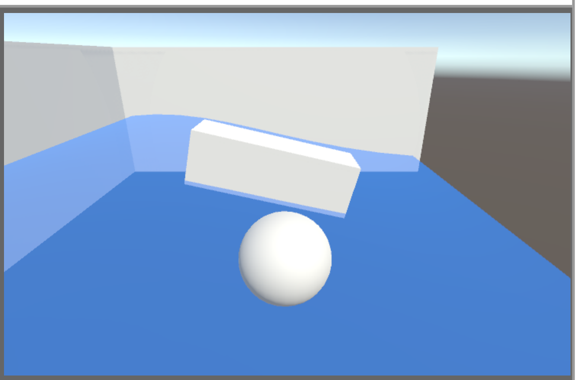
- The main problem so far is that for any object, the upthrust is based on the submerged vertices. In reality, the faces should also be included in the calculations.
- Ignoring faces, we may get situations such as the one shown to the right, where the wave can intersect with the object and cause inaccurate buoyancy.
- This is a limitation of using the vertex-based method.



TASK 7: The camera should focus on an object when it is clicked on.

- The camera system involves two modes: Free Mode, where the user will be free to move and rotate around however they want, and Control Mode, where the user can select and steer a boat in the scene.
- In order to begin implementing the controls, I will be using Unity's new event-based input system and importing it into the project from the Unity Package Manager.

```
PROCEDURE OnEnable()
  controls.Click += OnClick();
ENDPROCEDURE

PROCEDURE OnUpdate()
   IF target != NULL
      Camera.MoveTo(target);
   ENDIF
ENDPROCEDURE

PROCEDURE OnClick()
   Raycast from mouse point
   IF(ray.hit.object is of type BuoyantObject)
      target = ray.hit.object
   ENDIF
ENDPROCEDURE
```

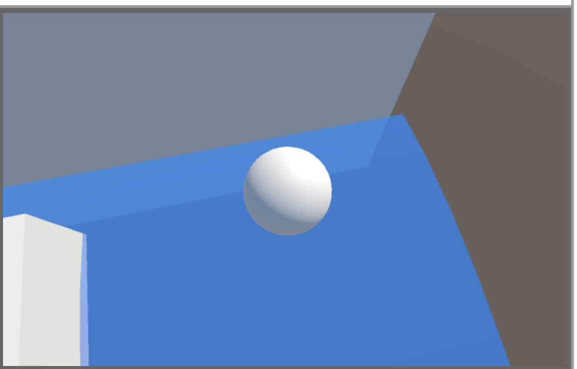


- The camera can now follow any buoyant object around.

TASK 8: Follow Mode camera can orbit the selected object.

- The camera needs to rotate around the selected target - x movement correlates to the rotation around the global y-axis, and y movement correlates to rotation around the local x-axis.

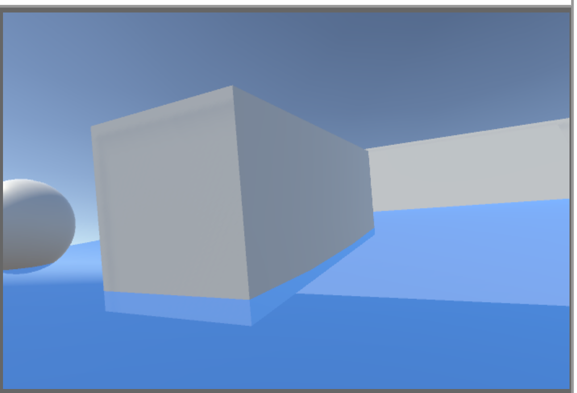
```
PROCEDURE OnUpdate()
   IF MMB.held
      Vector2 mouse =
Camera.ScreenPositionOfMouse
      Quaternion turnX =
AngleAxis(mouse.x, world.up)
      Quaternion turnY =
AngleAxis(mouse.y, local.right)
      rotation *= turnX * turnY
   ENDIF
ENDPROCEDURE
```



TASK 9 [BUG #03]: Follow Mode camera performs incorrectly at certain angles

- The camera does not perform as expected when it is near the poles of the object selected. This is because of the angle - it needs to be clamped between 5-80°. The problem with this is that Quaternions do not work the same as Euler rotations so my procedure will have to use Euler angles instead.

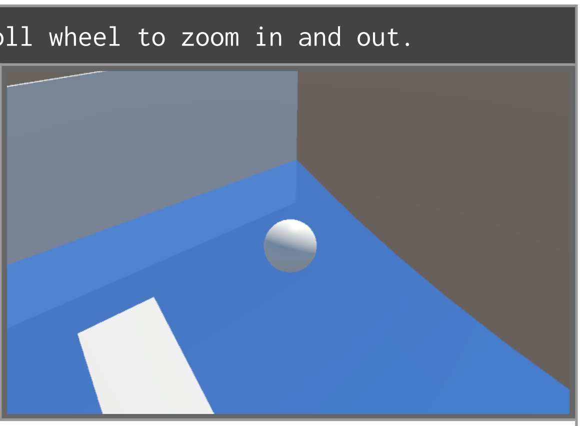
```
PROCEDURE OnUpdate()
   IF MMB.held
      Vector2 mouse =
Camera.ScreenPositionOfMouse
      turnXY.x += mouse.x
      turnXY.y += mouse.y
      turnXY.y = clamp(5, 80)
      rotation =
Quaternion.Euler(turnXY, 0)
      offset = Vector3.back * rotation
      …
   ENDIF
ENDPROCEDURE
```



- The camera is now able to rotate around and is clamped in the vertical direction.
- This bug also gave me the idea of implementing scroll zoom as the next feature.
- The range chosen for the angle of the camera, highlighted in red in the above pseudocode, was chosen so that the camera does not go too low and clip into the wave, and also cannot go too high as if it went above 90 the camera would be able to orbit the object upside down.

TASK 10: Follow Mode should allow the scroll wheel to zoom in and out.

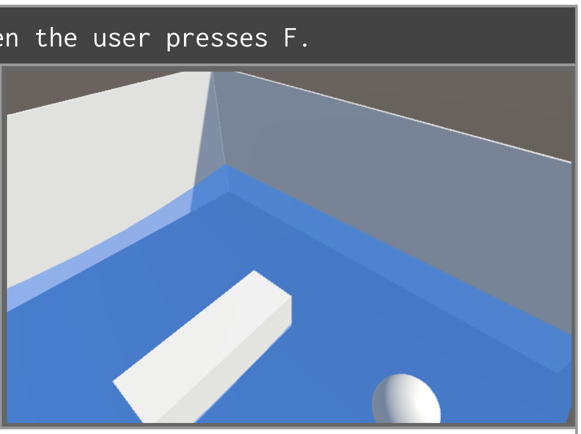
```
PROCEDURE OnEnable()
    ...
    WHEN Controls.scroll.performed =>
       distance += scrollwheel.value
       distance = clamp(2, 10)
    ...
ENDPROCEDURE
```



- The camera is now able to zoom in and out when the wheel is scrolled. The distance is clamped between two values, 2 and 10, so that the camera cannot clip into the object. These bounds are likely to be changed in the future but for now they are fine.

TASK 11: Free Mode should be activated when the user presses F.

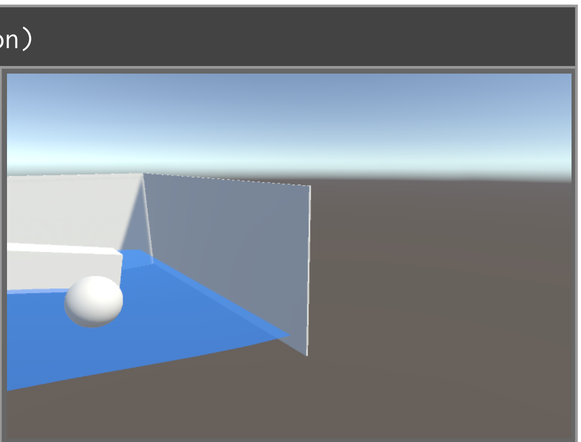
```
PROCEDURE OnEnable()
    ...
    WHEN Controls.F.performed =>
       FollowMode = Mode.Free
       target = NULL
    ...
ENDPROCEDURE
```



- The camera is now able to switch between free and follow mode.

TASK 12: Free Mode controls (mouse rotation)

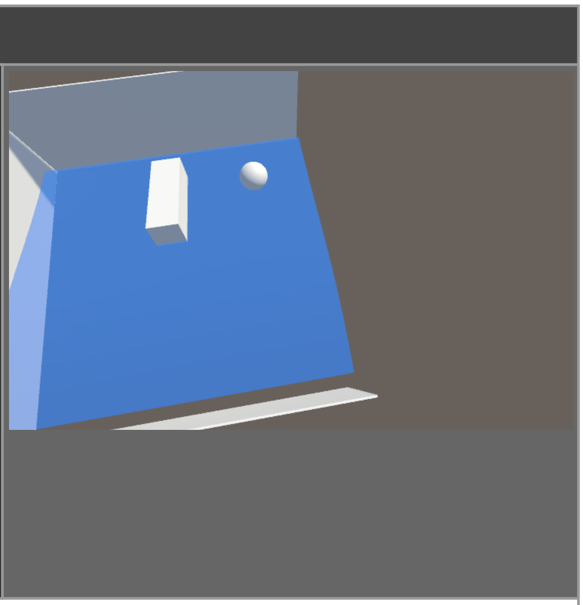
```
PROCEDURE OnUpdate()
   IF MMB.held && CameraMode = Free
      Vector2 mouse =
Camera.ScreenPositionOfMouse
      turnXY.x += mouse.x
      turnXY.y += mouse.y
      turnXY.y = clamp(5, 80)
      rotation =
Quaternion.Euler(turnXY, 0)
      ...
   ENDIF
ENDPROCEDURE
```



- The camera rotated on the vertical axis in the wrong way, so instead of adding the mouse y vector I subtracted it, fixing the problem.

TASK 13: Free Mode controls (WASD Motion)

```
PROCEDURE OnEnable()
    ...
    WHEN Controls.WASD.performed =>
       Vector2 dir = WASD.getVector
    ...
ENDPROCEDURE

PROCEDURE OnUpdate()
   IF CameraMode == Follow
      pos += transform.forward * dir.y
      pos += transform.right * dir.x
      ...
   ENDIF
ENDPROCEDURE
```



- The camera can now move around in space freely.

TASK 14: Adjusting the wave equation

$$a = \frac{s}{k},\ s = e^{kb} \qquad c = \sqrt{\frac{g}{k}} = \sqrt{\frac{g\lambda}{2\pi}}$$

- These new parameters (phase speed and steepness) improve the accuracy of the water waves compared to real physics.
- Using abstraction also makes my code much easier to read and prevents any calculations from being repeated which improves efficiency.

TESTING

Iterative feature tests are documented in the development log

TEST	COMMENTS
INTERNAL	● As mentioned in the tests for prototype 1, I added other 3D colliders into the scene to observe them. They now behaved much more realistically, obeying gravity and rotating rather than remaining in a fixed perpendicular position to the water.
USER 1	● The floating objects float much better than before. ● The camera being able to move is a nice addition but the controls feel a bit heavy and slow, and rotating the camera can be quite sensitive and jittery. ● There should be a way of adding new objects to the scene as having only 2 is not very interesting.

	A spawn menu is a good idea for a feature in prototype 3.
USER 2	● Sometimes the wave cuts through the middle of an object but this is only noticeable if you are zoomed in with the camera. ● Being able to follow different objects with the camera is a really good addition. You need to be careful when you click to select the right one.

EVALUATION

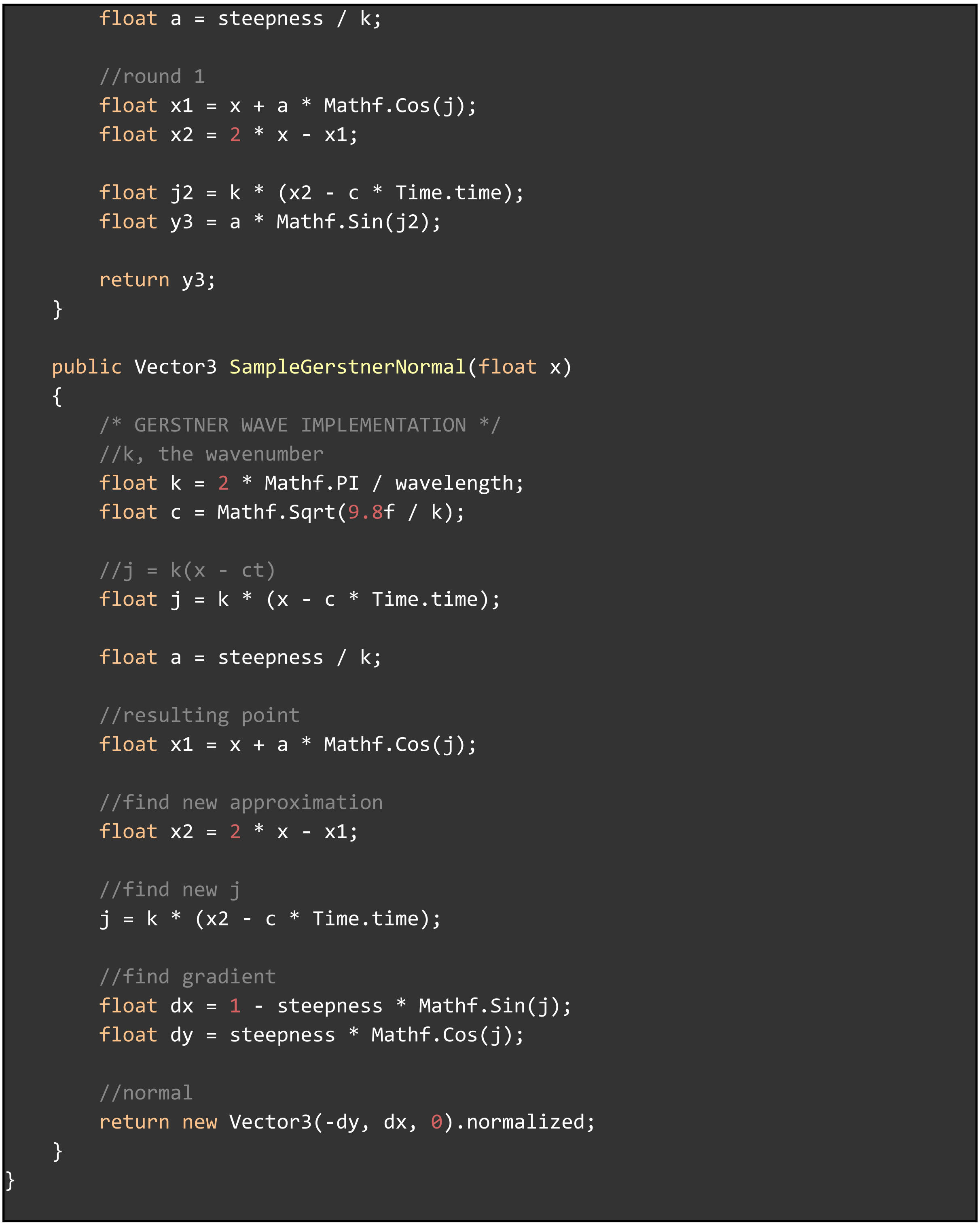
CRITERIA	EVALUATION
□ Do floating objects float perpendicular to the water's surface?	Objects now float perpendicular to the surface but also obey the laws of physics and can rotate/fall over. This makes the sailing mechanics redundant as objects will inevitably move in the same way as the wind as this is also the same way the water moves. I decided not to include the wind mechanics in the third prototype as a result.
□ Does the water deform in a smooth parametric Gerstner wave?	The water shader now manipulates the vertices with a Gerstner wave instead of a sine wave.
□ Can the x-position be approximated using any relatively fast approximation method □ Can the tangent function to the Gerstner wave be found? □ Can the normal function to the Gerstner wave be found? □ Do floating objects float perpendicular to the water's surface?	The first and last of these success criteria are only partially met. The approximation used needs to be improved in the third prototype, and objects do not float perpendicular to the water's surface but in a direction which is approximately perpendicular. The middle 2 were met by using calculus to find the derivative.
□ Can the user select different objects to follow? □ Can the user freely move the camera while in Free Mode? □ Can the user orbit the selected object in Follow Mode?	These criteria are technically met and the camera system does have complete functionality, but as user 1 reported it is not a 'comfortable' system to use. I decided to increase the movement sensitivity, decrease the mouse sensitivity and also interpolate all the values in the script which produced a smoother experience.

WaterShader Class (GLSL)

```
// procedure applied to each vertex
        void vert (inout appdata_full vertexData)
        {
            float3 v = vertexData.vertex.xyz;

            //wave number
            float k = 2 * UNITY_PI / _Wavelength;
            float c = sqrt(9.8 / k);
            float j = k * (v.x - c * _Time.y);
            float a = _Steepness / k;
            v.x += a * cos(j);
            v.y = a * sin(j);

            //tangent = (dx, dy, dz)
            float3 tangent = normalize(float3(1 - _Steepness * sin(j), _Steepness
* cos(j), 0));

            //normal = (-dy, dx, 0)
            float3 normal = float3(-tangent.y, tangent.x, 0);

            vertexData.vertex.xyz = v;
            vertexData.normal = normal;

        }
```

WaveGenerator Class (C#)

```
public class WaveGenerator : MonoBehaviour
{
    //### singleton
    public static WaveGenerator Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;
    }

    //### reference variables
    MeshFilter filter;
    public Material waterMaterial;

    //### wave equation parameters with default values;
    [Range(0f, 1f)] public float steepness = 0.5f;
    [Range(1f, 20f)] public float wavelength = 5f;

    //### simulation parameters
    [Range(0.1f, 15f)] public float strength = 9.81f;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
    }

    void Update()
    {
        if (waterMaterial != null)
        {
            waterMaterial.SetFloat("_Steepness", steepness);
            waterMaterial.SetFloat("_Wavelength", wavelength);
        }
    }


    public float SampleGerstnerHeight(float x)
    {
        /* GERSTNER WAVE IMPLEMENTATION */
        //k, the wavenumber
        float k = 2 * Mathf.PI / wavelength;
        float c = Mathf.Sqrt(9.8f / k);

        //j = k(x - ct)
        float j = k * (x - c * Time.time);
```



```
        float a = steepness / k;

        //round 1
        float x1 = x + a * Mathf.Cos(j);
        float x2 = 2 * x - x1;

        float j2 = k * (x2 - c * Time.time);
        float y3 = a * Mathf.Sin(j2);

        return y3;
    }

    public Vector3 SampleGerstnerNormal(float x)
    {
        /* GERSTNER WAVE IMPLEMENTATION */
        //k, the wavenumber
        float k = 2 * Mathf.PI / wavelength;
        float c = Mathf.Sqrt(9.8f / k);

        //j = k(x - ct)
        float j = k * (x - c * Time.time);

        float a = steepness / k;

        //resulting point
        float x1 = x + a * Mathf.Cos(j);

        //find new approximation
        float x2 = 2 * x - x1;

        //find new j
        j = k * (x2 - c * Time.time);

        //find gradient
        float dx = 1 - steepness * Mathf.Sin(j);
        float dy = steepness * Mathf.Cos(j);

        //normal
        return new Vector3(-dy, dx, 0).normalized;
    }
}
```

BuoyantObject Class (C#)

```
[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(Rigidbody))]
public class BuoyantObject : MonoBehaviour
{
    //### reference variables
    Rigidbody rb;
    MeshFilter filter;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
        rb = GetComponent<Rigidbody>();
    }

    public void Update()
    {
        ApplyBuoyancy();
    }

    /// <summary>
    /// calculates and applies buoyant force to each vertex
    /// </summary>
    void ApplyBuoyancy()
    {
        Vector3 gravity = WaveGenerator.Instance.strength *
                          Vector3.down / filter.mesh.vertices.Length;

        //for every vertex in the mesh
        foreach (Vector3 vertex in filter.mesh.vertices)
        {

            //convert local to global position
            Vector3 vt = transform.TransformPoint(vertex);

            /* GERSTNER WAVES */

            float d = vt.y - WaveGenerator.Instance.SampleGerstnerHeight(vt.x);
            if(d < 0f)
                rb.AddForceAtPosition(-d *
WaveGenerator.Instance.SampleGerstnerNormal(vt.x) *
                                      WaveGenerator.Instance.strength,
                                      vt, ForceMode.Acceleration);
            //apply gravity
            rb.AddForceAtPosition(gravity, vt, ForceMode.Acceleration);
        }
    }
}
```

CameraController Class (C#)

```
public class CameraController : MonoBehaviour
{
    //## reference variables
    Controls controls;
    Transform target;

    //##camera default offset
    public Vector3 offset;
    Vector3 currentOffset;

    //##camera state
    public CameraState currentState;

    float sensitivity = 0.25f;
    float distance = 3f;

    bool MMB = false;

    Vector2 turnXY = new Vector2(0, 0);
    Vector2 movDir = new Vector2(0, 0);


    void OnEnable()
    {
        currentState = CameraState.Free;
        currentOffset = offset;
        controls = new Controls();
        controls.Enable();
        controls.Camera.LMB.performed += OnLMB;
        controls.Camera.MMB.performed += ctx => { MMB = true; };
        controls.Camera.MMB.canceled += ctx => { MMB = false; };

        controls.Camera.Scroll.performed += ctx =>
        {
            float target = Mathf.Clamp(distance + (ctx.ReadValue<float>()) /
-120f, 2, 6);
            distance = Mathf.Lerp(distance, target, 0.5f);
        };

        controls.Camera.WASD.performed += ctx => { movDir =
ctx.ReadValue<Vector2>().normalized; };
        controls.Camera.WASD.canceled += ctx => { movDir = Vector2.zero; };

        controls.Camera.F.performed += ctx =>
        {
            if (currentState == CameraState.Follow)
            {
                Vector3 f = transform.forward;
                currentState = CameraState.Free;
                target = null;
                transform.rotation = Quaternion.LookRotation(f, transform.up);
            }
        };
    }

    /// <summary>
    /// method called on left mouse click
    /// </summary>
    /// <param name="context">the event details</param>
    public void OnLMB(InputAction.CallbackContext context)
    {
        Ray ray =
Camera.main.ScreenPointToRay(Mouse.current.position.ReadValue());
        RaycastHit hit;

        if(Physics.Raycast(ray, out hit, 100f))
        {
            if(hit.transform.gameObject.GetComponent<BuoyantObject>() != null)
            {
                target = hit.transform;
                currentState = CameraState.Follow;
                currentOffset = (3f * Vector3.back);
            }
        }
    }

    /// <summary>
    /// occurs every frame
    /// </summary>
    void Update()
    {
        HandleMotion();
    }

    void HandleMotion()
    {
        //if there is a current target
        if (currentState == CameraState.Follow)
        {
            //if the middle mouse button is being held
            if (MMB)
            {
                Vector2 mouse = Mouse.current.position.ReadValue();
                Vector2 offset = new Vector2(Screen.width / 2, Screen.height / 2);
                mouse = (mouse - offset) * Time.deltaTime * sensitivity;

                //set the mouse rotation
                turnXY.x += mouse.x;
                turnXY.y += mouse.y;
                turnXY.y = Mathf.Clamp(turnXY.y, -10f, 80f);

            }

            //set the offset vector
            currentOffset = Quaternion.Euler(turnXY.y, turnXY.x, 0) * (distance *
Vector3.back);
            transform.position = Vector3.Lerp(transform.position, target.position
+ currentOffset, 0.5f);
            transform.LookAt(target);
        }

        else
        {
            if (MMB)
            {
                Vector2 mouse = Mouse.current.position.ReadValue();
                Vector2 offset = new Vector2(Screen.width / 2, Screen.height / 2);
                mouse = (mouse - offset) * Time.deltaTime * sensitivity;

                //set the mouse rotation
                turnXY.x += mouse.x;
                turnXY.y -= mouse.y;
                turnXY.y = Mathf.Clamp(turnXY.y, -70f, 70f);
            }

            Vector3 pos = (transform.forward * movDir.y + transform.right *
movDir.x) * 0.1f ;
            pos += transform.position;

            //set the camera rotation
            Quaternion rot = Quaternion.Euler(turnXY.y, turnXY.x, 0);
            transform.rotation = Quaternion.Lerp(transform.rotation, rot, 0.4f);
            transform.position = Vector3.Lerp(transform.position, pos, 0.3f);
        }
    }

    void OnDisable()
    {
        controls.Disable();
    }
}

public enum CameraState
{
    Free,
    Follow
}
```

DESIGN: PROTOTYPE 3

TASK	SUCCESS CRITERIA
Wind direction	☐ Does the wave change direction? ☐ Can the direction be controlled by an angle variable? ☐ Does the wave change direction in real time?
Physics simulation	☐ One Newton-Raphson iteration ☐ Multiple Newton-Raphson iterations
UI	☐ Menu + Sliders for wave parameters ☐ Does the slider update the wave in real time?
Spawning + deleting objects	☐ Menu + list of spawnable objects ☐ Can objects be spawned in? ☐ Can objects be deleted?

DEVELOPMENT LOG: PROTOTYPE 3

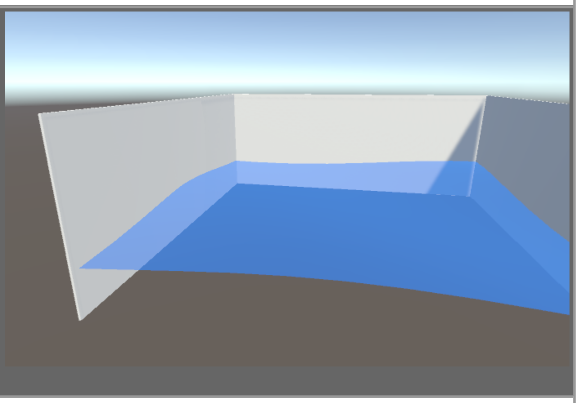
ABSTRACTION/PSEUDOCODE

TASK 1: Changing the direction of the wave

$$D(x,\ 0,\ z)\ =\ direction\ vector$$

$$P\ =\ (x\ +\ Dx\ \frac{s}{k}\ cos\ j,\ \frac{s}{k}\ sin\ j,\ z\ +\ Dz\ \frac{s}{k}\ cos\ j)$$

```
PROCEDURE vertex(vertexData v)
   k = 2 * PI / wavelength
   j = k * (dot(dir, v.xz) - speed * time)
   a = steepness / k
   v.x += dx * a * cos(j)
   v.y = a * sin(j)
   v.z += dz * a * cos(j)
   update vertex
ENDPROCEDURE
```



- The wave can now have its direction changed.

TASK 2: Changing the direction of the wave, part 2

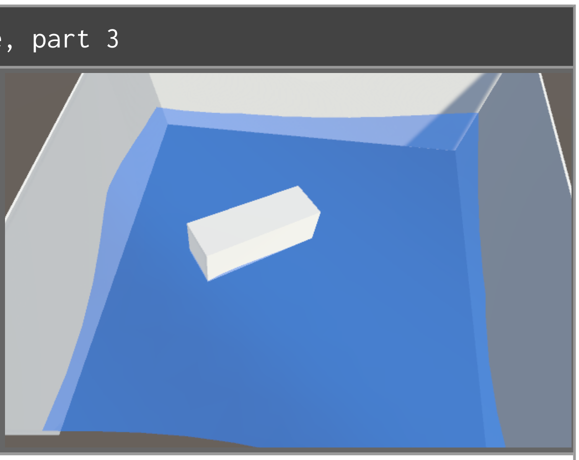
$$D = (\cos\theta,\ 0,\ \sin\theta)$$

```
PROCEDURE OnEnable()
   Vector4 vec = (cos(a), sin(a), 0, 0)
   waterMaterial.SetVector("Direction", vec)
ENDPROCEDURE
```

- The wave can now have its direction changed in realtime directly from the editor.
- The trigonometric functions in Unity take their parameters as radians rather than degrees. The angle has to be multiplied by a constant first (Mathf.Deg2Rad).

TASK 3: Changing the direction of the wave, part 3

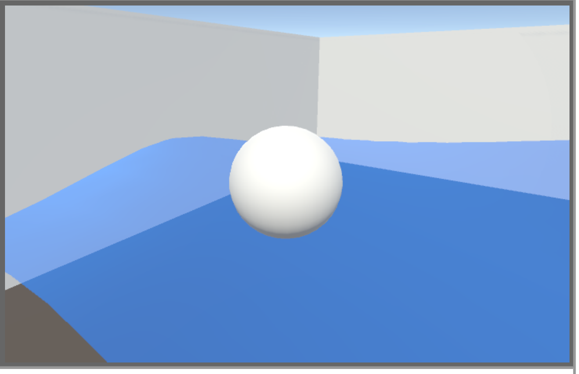
```
FUNCTION 2DGerstnerHeight(Vector3 p)
   k = 2 * PI / wavelength
   c = sqrt(9.8f / k)
   a = s / k
   v = (p.x, p.z);
   j = k( D.(x, z) - ct)
   RETURN a * sin(j);
ENDFUNCTION
```



- The wave's direction is now sampled in the height function.
- The dot product of the direction vector changes the input value of sin() depending on which direction the D vector is pointing in. The height of the wave now depends on both the x and z coordinate of the point.

TASK 4: Newton-Raphson approximation for height

```
FUNCTION 2DGerstnerHeight(Vector3 p)
   ...
   v.x += Dx * a * cos(j)
   v.y += Dy * a * cos(j)
   j = k( D.(x, z) - ct)
   ...
ENDFUNCTION
```

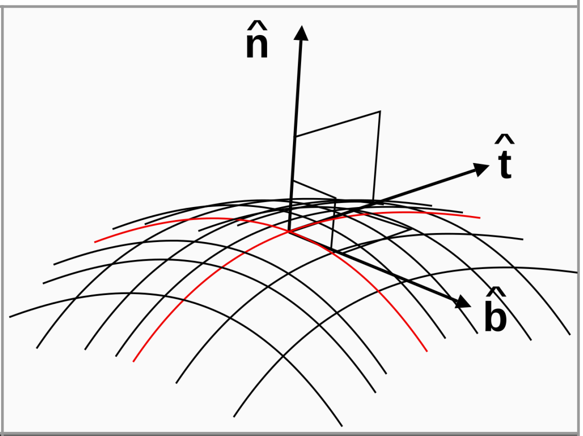


- The structure used for this means that I can now iterate through a specific number of Newton-Raphson approximations. This refactoring will be extremely useful in the later stages of this prototype.

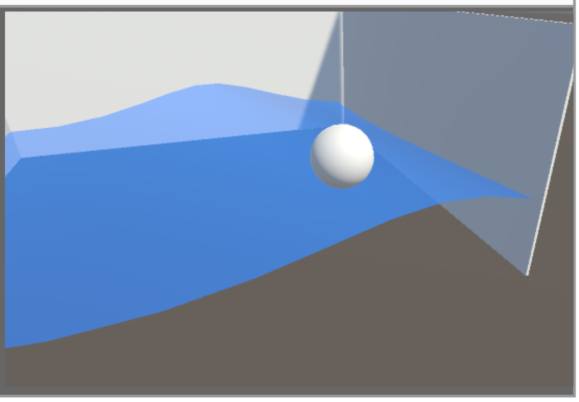
TASK 5: Newton-Raphson implementation in normal vector

$$T = P'x = (1 - D^2x * s \sin j, Dx * s \cos j, - DxDz * s \sin j)$$
$$B = P'z = (- DxDz * s \sin j, Dz * s \cos j, 1 - D^2z * s \sin j)$$
$$N = B \times T$$

- In order to calculate the normal, first I must find the partial derivative of the wave's surface in the x-direction (**tangent**) and also in the z-direction (**binormal**). The normal vector will be the cross product of these two vectors.



```
FUNCTION 2DGerstnerVector(Vector3 p)
   ...
   tangent = (1 - dx * dx * s * sin(j),
              dx * s * cos(j),
              -dx * dz * s * sin(j))
   binormal = ( -dx * dz * s * sin(j),
              dz * s * cos(j),
              1 - dz * dz * s * sin(j))
   RETURN cross(tangent, binormal)
ENDFUNCTION
```



TASK 6: Refactoring the wave function

- The wave script performs the same calculation twice a frame. I can increase the efficiency by combining the height and normal into a single Vector4.

```
FUNCTION GerstnerVector(Vector3 p)
   ...
   RETURN Vector4(n.x, n.y, n.z, h)
ENDFUNCTION
```

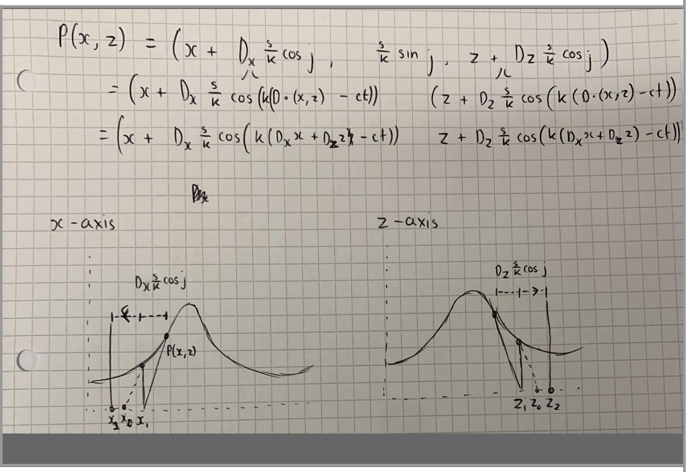
- Now the wave equation can be changed in one single location. The actual accuracy of the approximation will have to be increased as it still does not produce a completely accurate height, but the normal vector is now correct.

TASK 7: Approximating the x and z roots

- The approximation for the inverse function is now quite inaccurate. The new approximation must include both the x- and z- axis which makes it more complex. However since we have already calculated the tangent we just have to do the same calculation in 2 dimensions.

$$x_{n+1} = x_n - Dx * s * \cos j$$
$$z_{n+1} = z_n - Dz * s * \cos j$$

```
FOR i = 0 TO n
   v.x -= d.x * steepness *
cos(j)
   v.y -= d.y * steepness *
cos(j)
   j = k * (D.v - ct)
NEXT i
```

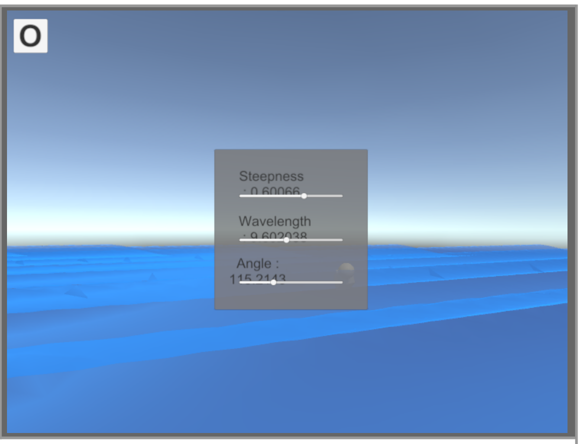

$P(x,z) = (x + D_x \frac{s}{k} \cos j, \frac{s}{k} \sin j, z + D_z \frac{s}{k} \cos j)$
$= (x + D_x \frac{s}{k} \cos(k(D \cdot (x,z) - ct))$
$(z + D_z \frac{s}{k} \cos(k(D \cdot (x,z) - ct))$
$= (x + D_x \frac{s}{k} \cos(k(D_x x + D_z z) - ct))$
$z + D_z \frac{s}{k} \cos(k(D_x x + D_z z) - ct))$
x - axis
z - axis
$D_x \frac{s}{k} \cos j$
P(x,z)
$x_2\ x_0\ x_1$
$D_z \frac{s}{k} \cos j$
$z_1\ z_0\ z_2$

- This approximation uses the fact that the step backwards is usually too large compared to the actual difference.
- It is hard to see whether this has significantly improved the accuracy of the simulation, however from inspection I can see that the objects lie closer to the surface of the water than before. For very high numbers of Newton-Raphson approximations efficiency decreases significantly.
- After some testing, I determined that a good number of iterations was 3, balancing time with accuracy and producing a reasonable result. When the wave parameters take extreme values, the accuracy decreases regardless of the number of iterations.

TASK 8: Implementing the UI

- The UI involves a button which toggles the menu, and a set of sliders which allow the user to set the parameters of the wave.

```
PROCEDURE ToggleMenu()
   menu.Active = !menu.IsActive
ENDPROCEDURE

PROCEDURE SetValue()
   Wave.value = slider.value
   Text = "Property: " +
toString(slider.value)
ENDPROCEDURE
```



Steepness : 0.60066

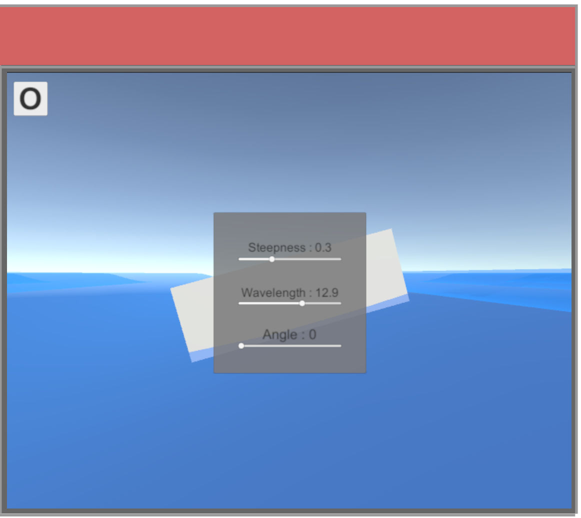
Wavelength : 9.602938

Angle : 115.2143

- This works correctly however as shown in the screenshot there are problems with the text width and also that the sliders are not initially set to the wave parameter, which means that as soon as the UI is opened and a slider is updated the wave's parameters are set to zero.

TASK 9 [BUG #04]: Fixing the UI

- Fixing the UI required the slider values and text to be initialised in the OnEnable() procedure in the UIManager class. The slider values are set first, the values are rounded and then the text is given the rounded value instead to prevent the number from occupying too much space in the text box.



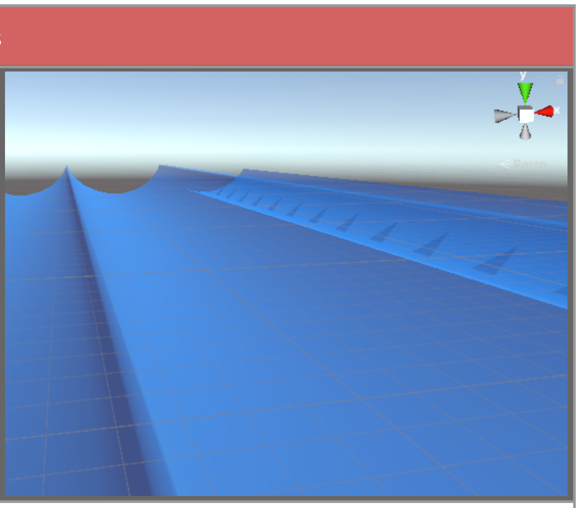
Steepness : 0.3

Wavelength : 12.9

Angle : 0

TASK 10 [BUG #05]: Limiting the parameters

- The values of the wave's parameters can take values which produce unrealistic behaviour, such as shading lighting errors, which are not representative of real physical results.
- The best way to handle this is to adjust the value's ranges so that they cannot take extreme values.

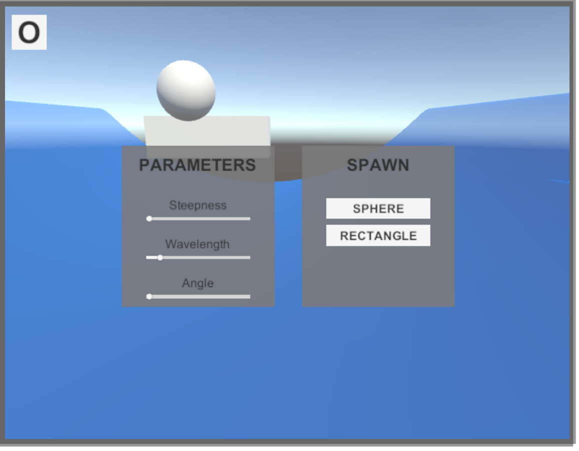


- For the amplitude, I chose 0-0.4 as this prevented any sharp edges from forming at the wave apex. Edges cannot have their normal vertices sampled and produce unexpected results, so removing them improves accuracy.

- For the wavelength, I chose 5-50. The vertices of the water are 1 unit apart so any wave smaller than about 5 will appear jagged instead of smooth. The wave plane is also 100x100 units so 50 allows 2 full waves to exist inside the plane.
- For the angle, the range is naturally between 0 and 360 as this is one full turn.

TASK 11: Spawn Menu

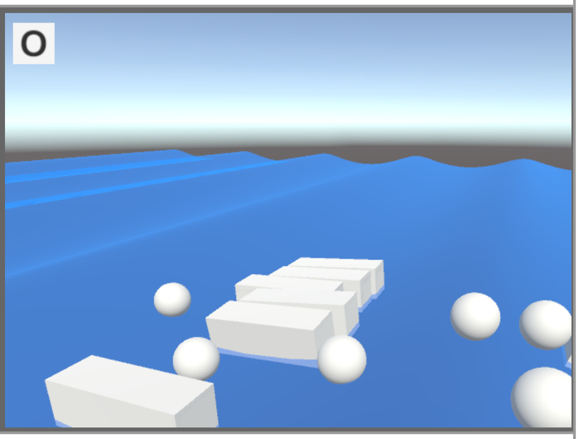
- The UI has been cleaned up and a new Spawn Menu section has been added.
- Adding this was very easy as the UI system uses modular elements and modular functions as I would imagine it will be expanded in the future.



TASK 12: Spawn Menu functionality

- The Spawn Manager class will be a singleton that links to the spawn menu and when a button is clicked a corresponding object will be spawned in.

```
PROCEDURE SpawnObject(index)
   InstantiateNew(objects[index], (0, 0, 0))
ENDPROCEDURE
```

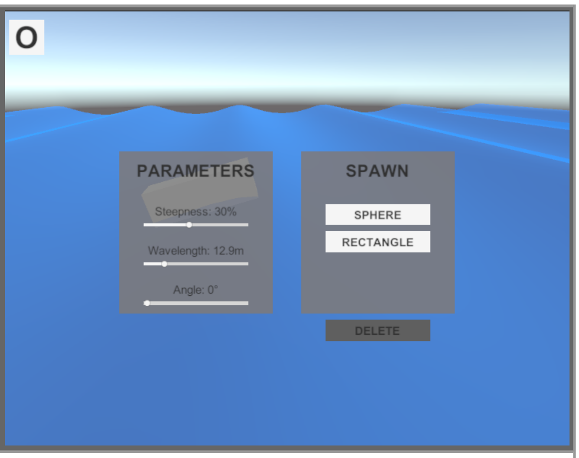


- Each button passes a different index to the class which then selects which object is spawned in. Objects will also need to be deleted.

TASK 13: Delete Selected Item functionality

- The delete button should only appear when there is an item being followed. When clicked, it should return the camera to free mode and delete the object from the scene. For this to work the *CameraController* class must be made into a singleton so the selected floating object can be easily accessed as it will be static.

```
PROCEDURE DeleteObject()
   Camera.Unlink()
   Destroy(Camera.target)
ENDPROCEDURE
```



Steepness: 30%

Wavelength: 12.9m

Angle: 0°

- The delete button does not vanish when an object is deleted, nor does it appear when a new object is selected.

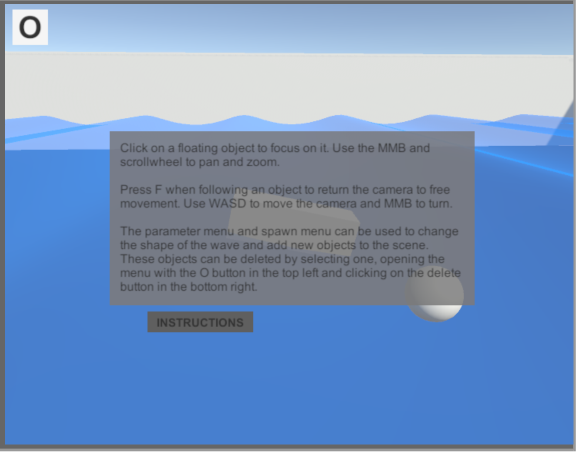
TASK 14 [BUG #06]: Setting delete button

```
PROCEDURE SetButton(targetState)
   IF Camera.State == Follow AND spawnMenu is active:
     deleteButton.state = targetState;
   ELSE
     deleteButton.state = inactive;
   ENDIF
ENDPROCEDURE
```

- I moved the delete button logic to a new public procedure that gets called under the two circumstances mentioned in the previous task.
- It takes in a target state for the delete button (active or inactive) and determines if the delete button should be in that state or not.

TASK 15: Spawn Menu

- I added an instruction menu in the UI after showing my project to my focus group, which was a good addition to improve the usability of the program. The instructions are clear and concise.



Click on a floating object to focus on it. Use the MMB and scrollwheel to pan and zoom.

Press F when following an object to return the camera to free movement. Use WASD to move the camera and MMB to turn.

The parameter menu and spawn menu can be used to change the shape of the wave and add new objects to the scene. These objects can be deleted by selecting one, opening the menu with the O button in the top left and clicking on the delete button in the bottom right.

TASK 16: Cleanup

- My code needed to be commented on and refactored in some places. I also got rid of obsolete sections, as I had commented out obsolete functions and variables instead of deleting them just in case I needed to roll back to a previously working section of code.

END USER TESTING

With the testing, I asked my end users to attempt to break the simulation and do things that I would not have thought to do. Below are their comments as well as what I did to fix any serious issues which they encountered.

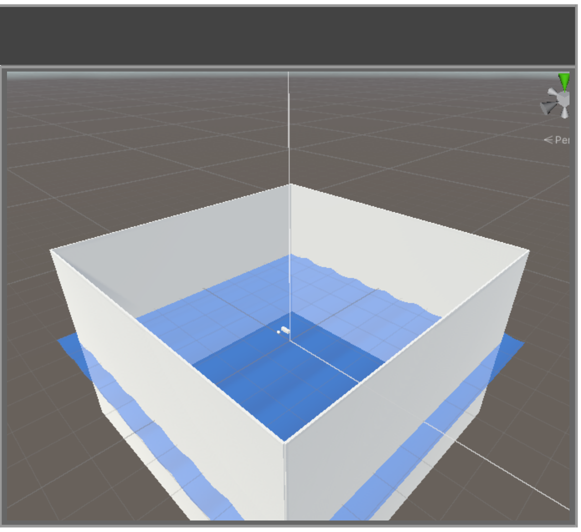
TEST	COMMENTS
USER 1	● The simulation looks quite accurate especially with spheres. They float well. ● The UI looks very sleek and simplistic and is not crowded. ● I like how the changes to the wave can be seen in real time rather than having to close the menu, but the sliders are quite sensitive if they are dragged very quickly. ● I can cause the frame rate to drop very low if I spawn in a large number of objects.
USER 2	● Even though the graphics are quite simple I can see the potential behind the simulation. ● The range of spawnable objects is limited but they each demonstrate how the simulation works. ● The instruction menu makes the controls easy to understand and the menu is organised very logically. ● If the wave sliders are changed very quickly objects can be launched over the edge of the walls.

POLISHING

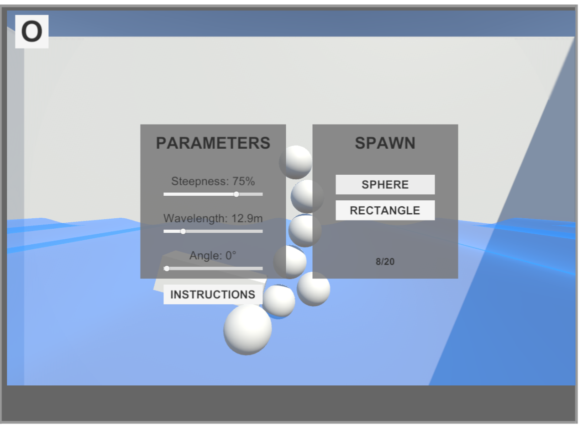
TASK 1: Increasing wall height

- I scaled up the walls around the water as well as their colliders on the y-axis to prevent any objects reaching over the top.

- I decided that a scale factor of x10 vertically was enough to produce a high enough wall to prevent any objects from escaping the scene.



TASK 2: Implementing spawn limit

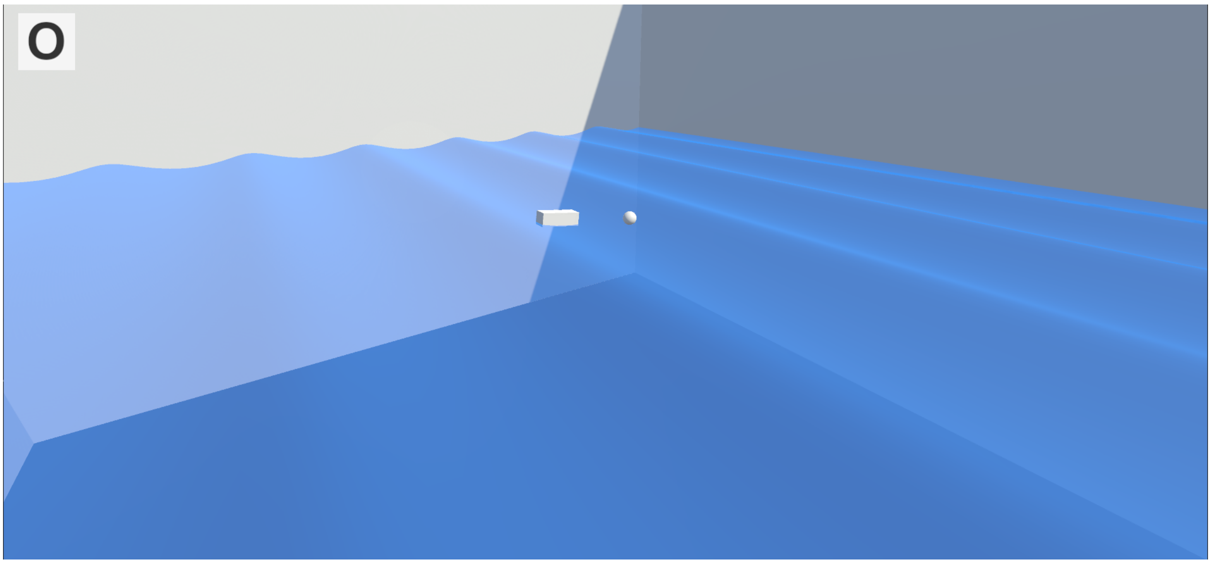
- I added a counter that was increased or decreased when the user spawned or deleted an object. This appears as a counter in the spawn menu as shown in the picture on the right. 20 objects seemed like a reasonable upper limit.	 O PARAMETERS Steepness: 75% Wavelength: 12.9m Angle: 0° INSTRUCTIONS SPAWN SPHERE RECTANGLE 8/20

EVALUATION

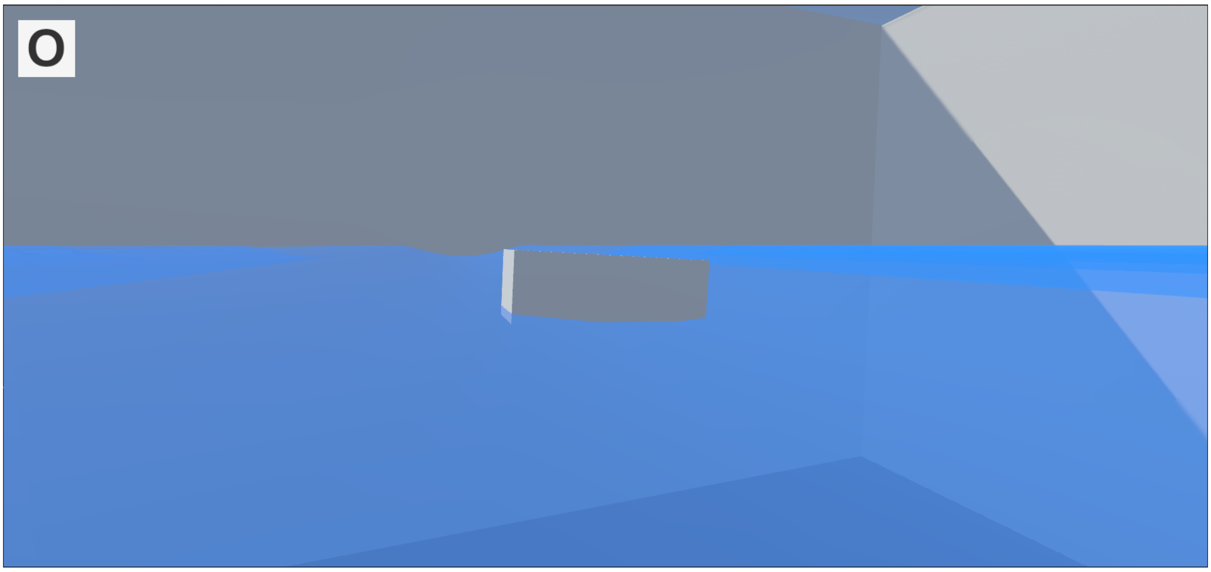
CRITERIA	EVALUATION
□ Does the wave change direction? □ Can the direction be controlled by an angle variable? □ Does the wave change direction in real time?	These success criteria are all met. The wave does change direction and so do objects on the wave's surface in real time when the slider in the menu is changed.
□ One Newton-Raphson iteration □ Multiple Newton-Raphson iterations	The Newton-Raphson loop can have any upper limit but as discussed earlier 3 was chosen as a sensible balance between accuracy and time.
□ Menu + Sliders for wave parameters □ Does the slider update the wave in real time?	There is a slider for amplitude, wavelength and angle, each of which has a formatted text value and updates the wave in real time.
□ Menu + list of spawnable objects □ Can objects be spawned in? □ Can objects be deleted?	Objects can be spawned and deleted in real time very easily using the spawn menu.

EVALUATION

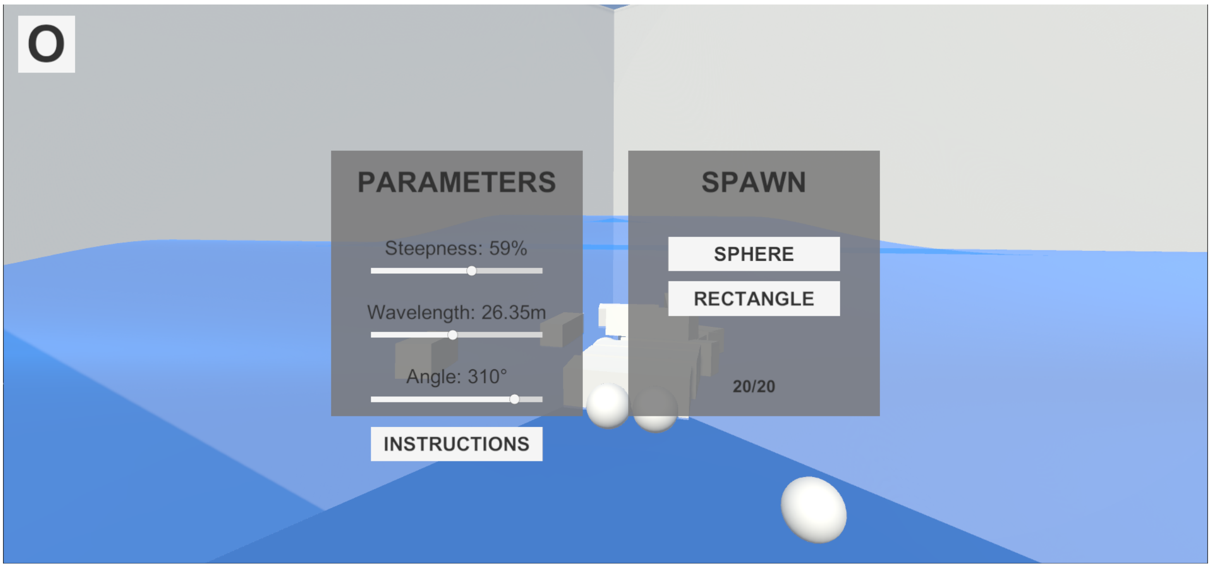
GALLERY



The starting scene with the 2 default objects - cuboid and sphere.



The camera is orbiting a single object.



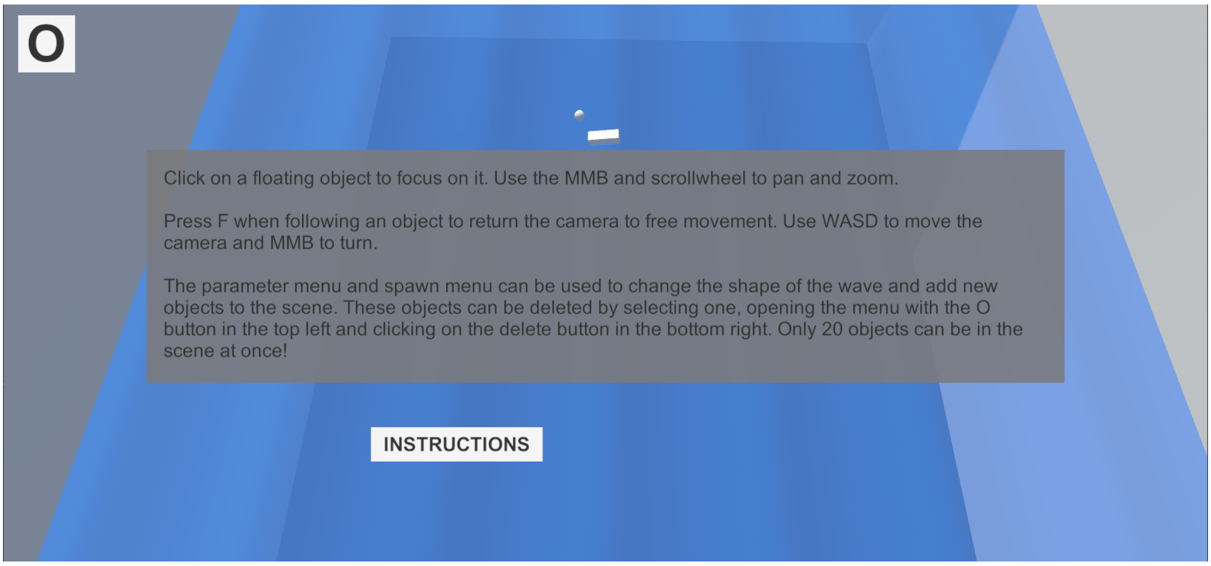
Steepness: 59%

Wavelength: 26.35m

Angle: 310°

20/20

20 objects have been spawned in - this is the maximum

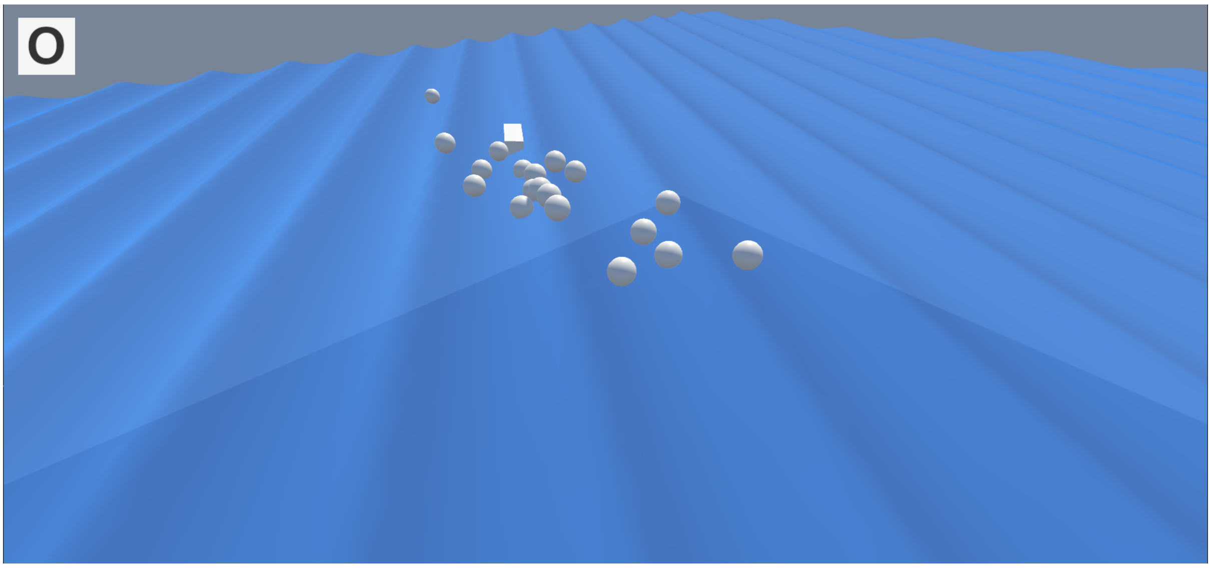


Click on a floating object to focus on it. Use the MMB and scrollwheel to pan and zoom.

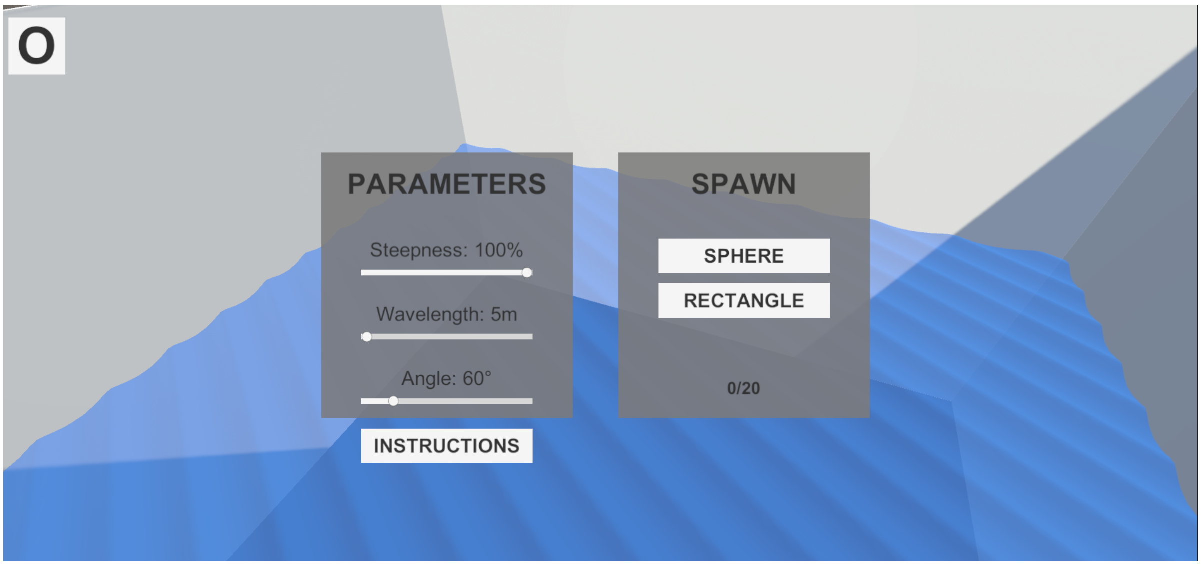
Press F when following an object to return the camera to free movement. Use WASD to move the camera and MMB to turn.

The parameter menu and spawn menu can be used to change the shape of the wave and add new objects to the scene. These objects can be deleted by selecting one, opening the menu with the O button in the top left and clicking on the delete button in the bottom right. Only 20 objects can be in the scene at once!

The instructions



Objects floating on a wave with parameters a=0.2, w=5, d=45



The same scene, but every object has been deleted using the delete button which appears in the menu when an object is selected.

SUCCESS CRITERIA

SCRIPT	ANALYSIS + CRITERIA MET
WaveGenerator	● The script meets the success criteria established during the analysis, as well as the criteria established in each prototype. The conversion of the Vector3 and float function which returned the normal vector and target height of the wave to a single Vector4 function increased the efficiency of the program significantly, as the same calculations were repeated twice. The return data could be reduced to a single variable with the w-component representing the target height. ● This Vector4 has to be separated into a Vector3 and float anyway, but there is a function built into Unity which removes the w-component so this process is still very efficient. ● Calculating the actual solution instead of using the Newton-Raphson approximation would have required much more complexity and probably resulted in a much slower program, as the calculations have to be done for every single vertex. It is quite obvious that this approach was better, but the performance difference for most modern computers is probably very small. This is an example of the tradeoff between time and space in Computer Science. ● An obvious drawback of the approximation is a lack of accuracy. To prevent the performance from being affected significantly, the number of iterations of the Newton-Raphson algorithm must be kept to around 3, so the actual normal vector/height above the wave is only accurate for relatively small waves. To address this problem I reduced the range of the wave parameters to prevent giant waves with unrealistic values being simulated. ● These methods involve matrix multiplication as they perform calculations on vectors which are a type of matrix. GPUs are optimised for matrix and tensor maths so as a result it may have been easier to parallelise this on the GPU using a compute shader. If this was a high-level commercial project then this is what I would have done, as it would have led to a significantly better end product. However, the requirements would have been much higher as the GPU power needed for such calculations exceeds the average PC.
UIManager	● The UIManager in my opinion does exactly what it needs to. It is by nature very modular as it must perform particular actions for each button. Even though these functions are very similar, the fact that they each operate on a different object means that I could not refactor them into a single function, which is what I usually attempt to do with subprograms that have identical functions.

	● I think that the UI's minimalist nature fits the program very well. I made sure that UI/UX was not a high priority in the development however ensuring that there is a consistent UI experience is very important for all software. This was the hardest part to test as I had to ensure each sequence of controls/buttons led to the correct outcome. ● In terms of usability, the UI is very straightforward and clear. Each button has a textbox which tells the user what it does, as well as instant feedback on the sliders which tell the user what the current settings are. The separation of the UI into a wave parameter and spawn menu element makes it very clear what each part does.
SpawnManager	● The spawn system is fairly rudimentary and works well, however it may not scale to extra items. If I continued to develop the software this is one of the areas I would like to improve on, giving the user the option to either import their own models or choose from a selection of inbuilt models rather than just cubes and spheres. The success criteria have been met however as this is a variety of objects which can easily be expanded in the future. ● One problem I have identified with the spawn system is the fact that technically there is no upper limit on the number of spawned items. The user could spawn in an unlimited number of buoyant objects - since simulating each has such a large performance cost, this could potentially cause the instance of the program to crash. As a problem this would be easy to fix - every spawned item should be added to a list and if the list exceeds a certain length then no new items can be spawned.
CameraController	● The camera movements have been tested thoroughly through my use of the program, as they were needed to navigate and interact with objects while I was testing the program. I tweaked this script during development to make it easier for me to use my own project. ● Examples include limiting the rotation of the camera by clamping it between two values, as over-rotation of the camera led to it clipping into the water and into other objects. In a way this is the system that has been tested the most due to its frequent use. I initially did not consider the camera system as important and it came as an add-on at the end of prototype 2. This is a reflection on how development of software can throw unexpected problems which require a project to change its direction and new features may end up becoming the focus of the program. ● Overall the camera controller meets almost all of the success criteria. According to some of my focus group who tested the final project, it can feel heavy (due to the interpolation of the motion), sluggish (due to the latency

	of the input system) and counterintuitive (the camera can accelerate and jitter if the mouse is moved too quickly). There is more polishing which could be done with this system.
BuoyantObject	● The buoyant object system meets the success criteria I set initially and during each prototype even though the solution I chose did not match the one I initially devised. This is because during development I realised that my original algorithm using the surfacic approach would have a very high performance overhead as new vertices would have to be generated and used for extra accuracy every time the simulation added a new object. I instead opted to approximate the surfacic method by measuring the number of vertices and their distance underwater, applying a force normal to the water's surface at these points. ● As mentioned, this meets the success criteria - objects produce accurate results and float in a reasonably realistic manner, the accuracy of which increases for higher vertex meshes such as spheres. This is because the number of sample points increases with the number of vertices. ● The limitations of this approach occur when an object has a low number of vertices, such as the rectangle object. In the simulation, if the parameters are set to their maximum values, only the corners of the shape will appear to float on the water's surface.

FUTURE FEATURES

FEATURE	ANALYSIS
Sailing mechanics	● The sailing mechanic is not compatible with the current wave generation system. Objects will move in the same direction as the wind anyway as that is the way which the wave travels. ● In the future, larger objects travelling on smaller waves could utilise a sailing mechanic to travel, or instead of a sailing mechanic they could have propellers like actual boats. This system would be very easy to implement - the object could be selected by the camera, rotated using the A and D keys, and the W and S keys could be used to travel forwards and backwards respectively. ● This feature was originally suggested to me by a focus group, who said that the core concept could easily be applied to a game.

Boids	● The boids system was not implemented in the final project as I decided to instead spend my time focusing on the important features of the project. ● This feature, like the wave system, is very resource-intensive and as a result if it were implemented would likely have a significant performance cost making the project appear less polished. ● As a purely aesthetic feature it was sensible to opt out of including it, however in a future prototype I think it would be a very suitable feature to include to further take the project down the game route.
Landmass generation	● Using Perlin noise or other sampled noise, landmasses could be made in order to create an infinite sailing simulation. ● As a feature to add this would not be extremely complex but would definitely improve the quality of the project and make it more aesthetically pleasing.

CONCLUSION

Overall, my project meets all of the success criteria to some degree in functionality, maintainability and expandability. My methods of tackling each decomposed problem using modular code means that development was very straightforward and I produced a final project which functions very well. This project has required me to learn new skills, such as GLSL to produce the shader and also the use of multivariable calculus to determine the derivative of a wave function. There are plenty of different directions I could also choose to continue this project in the future if I were to develop a fourth prototype, for example, which would integrate well with the foundations of the project I have created.

FINAL CODE

WaveGenerator Class (C#)

Manages the wave on the CPU so its height can be sampled.

```
public class WaveGenerator : MonoBehaviour
{
    //### singleton
    public static WaveGenerator Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;
    }

    //### reference variables
    MeshFilter filter;
    public Material waterMaterial;

    //### parameters with default values
    [Range(0f, 0.4f)]   public float steepness = 0.1f;
    [Range(5f, 50f)]    public float wavelength = 10f;
    [Range(0f, 360f)]   public float angle = 0f;
    [Range(0.1f, 15f)]  public float strength = 9.81f;

    //### direction vector
    Vector2 d;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
    }

    void Update()
    {
        if (waterMaterial != null)
        {
            waterMaterial.SetFloat("_Steepness", steepness);
            waterMaterial.SetFloat("_Wavelength", wavelength);

            //calculate the direction vector from the angle
            float theta = angle * Mathf.Deg2Rad;
            d = new Vector2(Mathf.Cos(theta),Mathf.Sin(theta)).normalized;
            Vector4 vec = new Vector4(d.x, d.y, 0, 0);
            waterMaterial.SetVector("_Direction", vec);
        }
    }

    /// <summary>
    /// calculates the normal vector and height for a given x and z position
    /// </summary>
    /// <param name="x">input x - position</param>
    /// <param name="z">input z - position</param>
    /// <returns>vector4 (normal.x, normal.y, normal.z, height)</returns>
    public Vector4 SampleGerstnerWave(float x, float z)
    {
        //k, the wave number
        float k = 2 * Mathf.PI / wavelength;
        //c, the wave speed
        float c = Mathf.Sqrt(9.8f / k);
        //a = s / k
        float a = steepness / k;
        Vector2 v = new Vector2(x, z);

        //j = k( D.(x, z) - ct)
        float j = k * (Vector2.Dot(d, v) - c * Time.time);

        //2 Newton-Raphson iterations
        for (int i = 0; i < 3; i++)
        {
            v.x -= d.x * steepness * Mathf.Cos(j);
            v.y -= d.y * steepness * Mathf.Cos(j);
            j = k * (Vector2.Dot(d, v) - c * Time.time);
        }

        //tangent
        Vector3 tangent = new Vector3(1 - (d.x * d.x * steepness * Mathf.Sin(j)),
                                      d.x * steepness * Mathf.Cos(j),
                                      -d.x * d.y * steepness * Mathf.Sin(j));
        //binormal
        Vector3 binormal = new Vector3(-d.x * d.y * steepness * Mathf.Sin(j),
                                       d.y * steepness * Mathf.Cos(j),
                                       1 - (d.y * d.y * steepness *
Mathf.Sin(j)));
        //normal
        Vector3 normal = Vector3.Cross(binormal, tangent).normalized;

        return new Vector4(normal.x, normal.y, normal.z, a * Mathf.Sin(j));
    }
}
```

UIManager Class (C#)

Manages the functionality of the UI (buttons & sliders)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using System;

public class UIManager : MonoBehaviour
{
    //### singleton
    public static UIManager Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;
    }

    //### reference variables
    public GameObject parameterMenu;
    public GameObject spawnMenu;
    public GameObject deleteButton;
    public GameObject instructionButton;
    public GameObject instructionMenu;
    public Text counterText;

    public Slider steepnessSlider;
    public Text steepnessText;
    public Slider wavelengthSlider;
    public Text wavelengthText;
    public Slider angleSlider;
    public Text angleText;

    void OnEnable()
    {
        //initialise all UI elements
        parameterMenu.SetActive(false);
        spawnMenu.SetActive(false);
        deleteButton.SetActive(false);
        instructionButton.SetActive(false);
        instructionMenu.SetActive(false);

        steepnessSlider.value = WaveGenerator.Instance.steepness;
        SetSteepnessText();
        wavelengthSlider.value = WaveGenerator.Instance.wavelength;
        SetWavelengthText();
        angleSlider.value = WaveGenerator.Instance.angle;
        SetAngleText();

    }

    /// <summary>
    /// sets the menu state
    /// </summary>
    public void ToggleMenu()
    {
        //is the menu meant to be on or off
        bool targetState = !parameterMenu.activeSelf;
        parameterMenu.SetActive(targetState);
        spawnMenu.SetActive(targetState);
        SetDeleteButton(targetState);
        instructionButton.SetActive(targetState);
        instructionMenu.SetActive(false);
    }


    /// <summary>
    /// sets the delete button state
    /// </summary>
    /// <param name="target">the target state - on or off</param>
    public void SetDeleteButton(bool target)
    {
        //delete button only on if object is being followed
        if (CameraController.Instance.currentState == CameraState.Follow &&
spawnMenu.activeSelf)
            deleteButton.SetActive(target);
        else
            deleteButton.SetActive(false);
    }

    /// <summary>
    /// called when the instruction button is pressed
    /// </summary>
    public void InstructionButtonPressed()
    {
        //if the instructions are closed
        if (!instructionMenu.activeSelf)
        {
            spawnMenu.SetActive(false);
            parameterMenu.SetActive(false);
            instructionMenu.SetActive(true);
            SetDeleteButton(false);
        }
        else
        {
            spawnMenu.SetActive(true);
            parameterMenu.SetActive(true);
            instructionMenu.SetActive(false);
            SetDeleteButton(true);
        }
    }

    /// <summary>
    /// sets steepness UI + value
    /// </summary>
    public void SetSteepness()
    {
        WaveGenerator.Instance.steepness = steepnessSlider.value;
        SetSteepnessText();
    }

    /// <summary>
    /// sets wavelength UI + value
    /// </summary>
    public void SetWavelength()
    {
        WaveGenerator.Instance.wavelength = wavelengthSlider.value;
        SetWavelengthText();
    }

    /// <summary>
    /// sets angle UI + value
    /// </summary>
    public void SetAngle()
    {
        WaveGenerator.Instance.angle = angleSlider.value;
        SetAngleText();
    }

    /// <summary>
    /// formats steepness text
    /// </summary>
    void SetSteepnessText()
    {
        steepnessText.text = String.Format(
                             "Steepness: {0}%",
                             (Math.Round(steepnessSlider.value / 0.4f, 2) *
100).ToString());
    }

    /// <summary>
    /// formats wavelength text
    /// </summary>
    void SetWavelengthText()
    {
        wavelengthText.text = String.Format(
                              "Wavelength: {0}m",
                              Math.Round(wavelengthSlider.value, 2).ToString());
    }

    /// <summary>
    /// formats angle text
    /// </summary>
    void SetAngleText()
    {
        angleText.text = String.Format(
                         "Angle: {0}°",
                         Math.Round(angleSlider.value, 0).ToString());
    }
}
```

SpawnManager Class (C#)

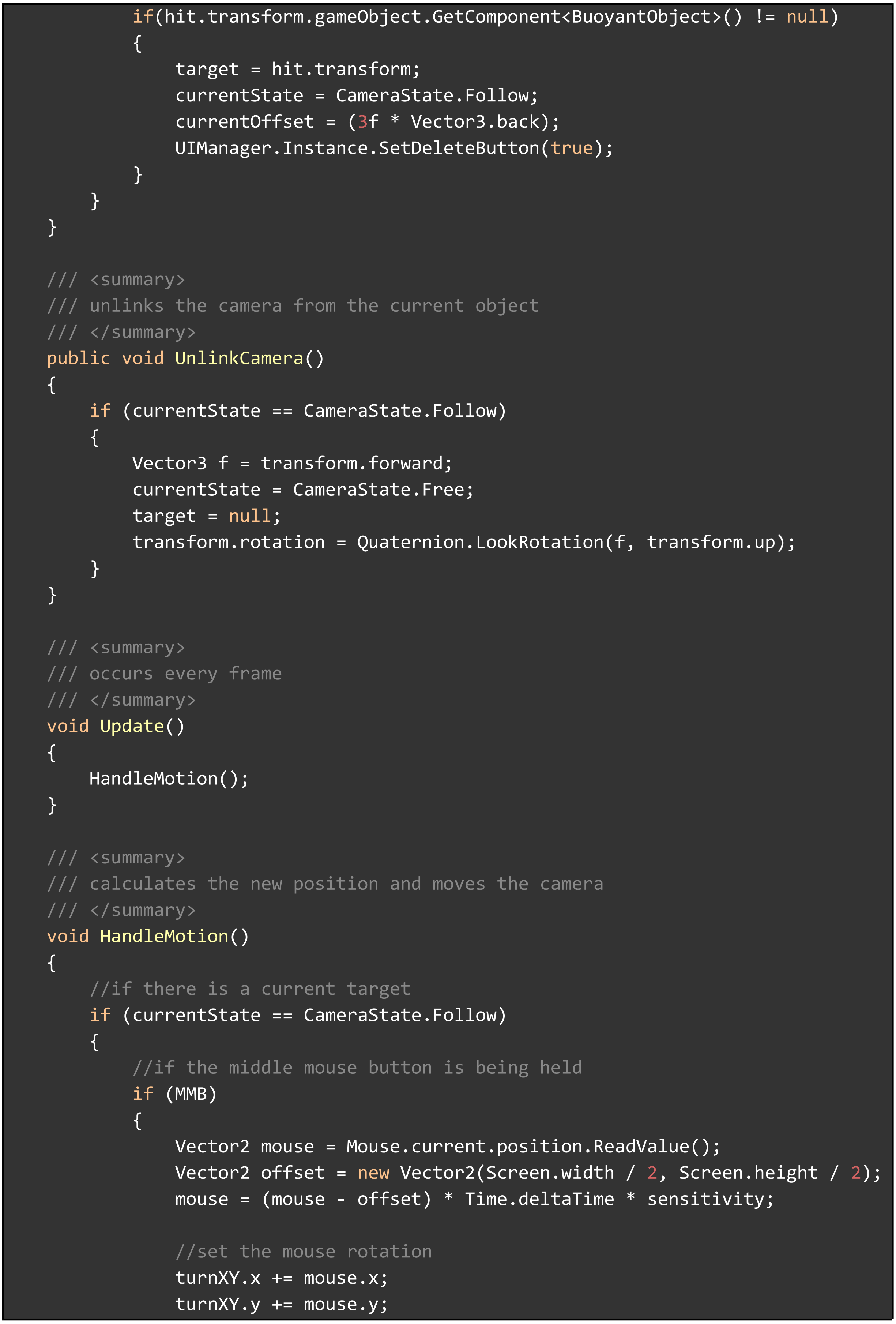
Manages the creation and destruction of floating objects

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SpawnManager : MonoBehaviour
{

    //### singleton
    public static SpawnManager Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;

        objectsInScene = Object.FindObjectsOfType<BuoyantObject>().Length;
        UpdateCounterText();
    }

    //list of spawnable objects
    public GameObject[] spawns = new GameObject[2];
    [SerializeField]   private int objectsInScene;

    /// <summary>
    /// spawns an object from the array
    /// </summary>
    /// <param name="n">the index of the item to spawn</param>
    public void SpawnObject(int n)
    {
        if(objectsInScene < 20)
        {
            GameObject go = Instantiate(spawns[n], Vector3.up * 3,
Quaternion.identity);
            objectsInScene++;
            UpdateCounterText();
        }
    }

    /// <summary>
    /// deletes the current
    /// </summary>
    public void DeleteObject()
    {
        Transform t = CameraController.Instance.target;
        CameraController.Instance.UnlinkCamera();
        Destroy(t.gameObject);
        UIManager.Instance.SetDeleteButton(false);
        objectsInScene--;
        UpdateCounterText();
    }

    public void UpdateCounterText()
    {
        UIManager.Instance.counterText.text = objectsInScene.ToString() + "/20
objects";
    }
}
```

CameraController Class (C#)

Manages the input system to move the camera and select objects

```
public class CameraController : MonoBehaviour
{
    //### singleton
    public static CameraController Instance;
    void Awake()
    {
        if (Instance != null && Instance != this)
            Destroy(this);
        else
            Instance = this;
    }

    //### reference variables
    Controls controls;
    public Transform target;

    //### parameters with default values
    Vector3 defaultOffset = new Vector3(0, 2, -3);
    float sensitivity = 0.25f;
    //### states
    public CameraState currentState;
    bool MMB = false;
    float distance = 3f;
    Vector3 currentOffset;
    Vector2 turnXY = new Vector2(0, 0);
    Vector2 movDir = new Vector2(0, 0);

    void OnEnable()
    {
        //setup controls and state of camera
        currentState = CameraState.Free;
        currentOffset = defaultOffset;
        controls = new Controls();
        controls.Enable();
        controls.Camera.LMB.performed += OnLMB;
        controls.Camera.MMB.performed += ctx => { MMB = true; };
        controls.Camera.MMB.canceled += ctx => { MMB = false; };

        //set scroll wheel behaviour
        //scroll wheel changes distance to targeted object
        controls.Camera.Scroll.performed += ctx =>
        {
            float target = Mathf.Clamp(distance + (ctx.ReadValue<float>()) /
-120f, 2, 12);
            distance = Mathf.Lerp(distance, target, 0.5f);
        };

        controls.Camera.WASD.performed += ctx => { movDir =
ctx.ReadValue<Vector2>().normalized * 2f; };
        controls.Camera.WASD.canceled += ctx => { movDir = Vector2.zero; };

        //set F button behaviour
        //releases camera from followed object
        controls.Camera.F.performed += ctx => UnlinkCamera();
    }

    /// <summary>
    /// method called on left mouse click
    /// </summary>
    public void OnLMB(InputAction.CallbackContext context)
    {
        Ray ray =
Camera.main.ScreenPointToRay(Mouse.current.position.ReadValue());
        RaycastHit hit;

        //if clicked object is a buoyant object set it as camera target
        if(Physics.Raycast(ray, out hit, 100f))
        {
```



```
            if(hit.transform.gameObject.GetComponent<BuoyantObject>() != null)
            {
                target = hit.transform;
                currentState = CameraState.Follow;
                currentOffset = (3f * Vector3.back);
                UIManager.Instance.SetDeleteButton(true);
            }
        }
    }

    /// <summary>
    /// unlinks the camera from the current object
    /// </summary>
    public void UnlinkCamera()
    {
        if (currentState == CameraState.Follow)
        {
            Vector3 f = transform.forward;
            currentState = CameraState.Free;
            target = null;
            transform.rotation = Quaternion.LookRotation(f, transform.up);
        }
    }

    /// <summary>
    /// occurs every frame
    /// </summary>
    void Update()
    {
        HandleMotion();
    }

    /// <summary>
    /// calculates the new position and moves the camera
    /// </summary>
    void HandleMotion()
    {
        //if there is a current target
        if (currentState == CameraState.Follow)
        {
            //if the middle mouse button is being held
            if (MMB)
            {
                Vector2 mouse = Mouse.current.position.ReadValue();
                Vector2 offset = new Vector2(Screen.width / 2, Screen.height / 2);
                mouse = (mouse - offset) * Time.deltaTime * sensitivity;

                //set the mouse rotation
                turnXY.x += mouse.x;
                turnXY.y += mouse.y;
                turnXY.y = Mathf.Clamp(turnXY.y, 5f, 80f);
            }

            //set the offset vector
            currentOffset = Quaternion.Euler(turnXY.y, turnXY.x, 0) * (distance *
Vector3.back);
            transform.position = Vector3.Lerp(transform.position, target.position
+ currentOffset, 0.5f);
            transform.LookAt(target);
        }

        else
        {
            //if there is no target and middle mouse button is held
            if (MMB)
            {
                Vector2 mouse = Mouse.current.position.ReadValue();
                Vector2 offset = new Vector2(Screen.width / 2, Screen.height / 2);
                mouse = (mouse - offset) * Time.deltaTime * sensitivity;

                //set the mouse rotation
                turnXY.x += mouse.x;
                turnXY.y -= mouse.y;
                turnXY.y = Mathf.Clamp(turnXY.y, -30f, 70f);
            }

            Vector3 pos = (transform.forward * movDir.y + transform.right *
movDir.x) * 0.1f ;
            pos += transform.position;

            //set the camera rotation
            Quaternion rot = Quaternion.Euler(turnXY.y, turnXY.x, 0);
            transform.rotation = Quaternion.Lerp(transform.rotation, rot, 0.4f);
            transform.position = Vector3.Lerp(transform.position, pos, 0.3f);
        }
    }

    void OnDisable()
    {
        controls.Disable();
    }
}

public enum CameraState
{
    Free,
    Follow
}
```

BuoyantObject Class (C#)

Manages the physics of any floating object

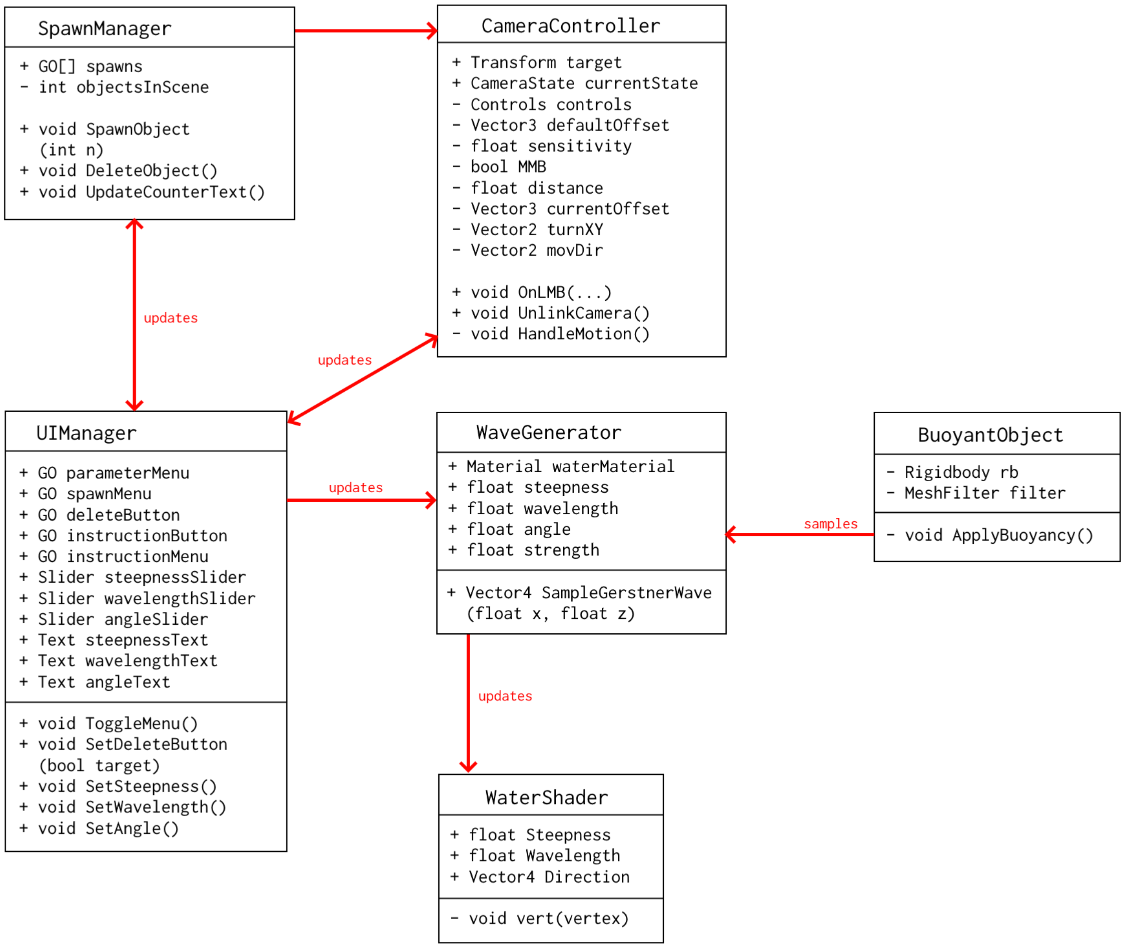
```
public class BuoyantObject : MonoBehaviour
{
    //### reference variables
    Rigidbody rb;
    MeshFilter filter;

    void OnEnable()
    {
        filter = GetComponent<MeshFilter>();
        rb = GetComponent<Rigidbody>();
    }

    public void Update()
    {
        ApplyBuoyancy();
    }

    /// <summary>
    /// calculates and applies buoyant force to each vertex
    /// </summary>
    void ApplyBuoyancy()
    {
        Vector3 gravity = WaveGenerator.Instance.strength *
                          Vector3.down / filter.mesh.vertices.Length;

        //for every vertex in the mesh
        foreach (Vector3 vertex in filter.mesh.vertices)
        {

            //convert local to global position
            Vector3 vt = transform.TransformPoint(vertex);

            /* GERSTNER WAVES */
            Vector4 g = WaveGenerator.Instance.SampleGerstnerWave(vt.x, vt.z);
            float d = vt.y - g.w;

            //apply buoyant force if underwater
            if(d < 0f)
            {
                rb.AddForceAtPosition(-d * (Vector3)g *
                                      WaveGenerator.Instance.strength,
                                      vt, ForceMode.Acceleration);
            }

            //apply gravity
            rb.AddForceAtPosition(gravity, vt, ForceMode.Acceleration);
        }
    }
}
```

WaterShader Class (GLSL)

Manages the GPU side of the wave by rendering the vertices

```
Shader "Custom/WaterShader"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0

        _Steepness ("Steepness", Range(0,1)) = 0.5
        _Wavelength ("Wavelength", Range(1,20)) = 5
        _Direction ("Direction (2D)", Vector) = (1, 0, 0, 0)
    }
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Standard alpha vertex:vert
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input
        {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;

        //water parameters
        float _Steepness, _Wavelength;
        float2 _Direction;

        UNITY_INSTANCING_BUFFER_START(Props)
        UNITY_INSTANCING_BUFFER_END(Props)

        // procedure applied to each vertex
        void vert (inout appdata_full vertexData)
        {
            float3 v = vertexData.vertex.xyz;
            //wave number
            float k = 2 * UNITY_PI / _Wavelength;
            float c = sqrt(9.8 / k);
            float2 d = normalize(_Direction);
            float j = k * (dot(d, v.xz) - c * _Time.y);
            float a = _Steepness / k;
            v.x += d.x * (a * cos(j));
            v.y = a * sin(j);
            v.z += d.y * (a * cos(j));

            //tangent = (dx, dy, dz)
            float3 tangent = normalize(float3(1 - _Steepness * sin(j), _Steepness
* cos(j), 0));

            //normal = (-dy, dx, 0)
            float3 normal = float3(-tangent.y, tangent.x, 0);

            vertexData.vertex.xyz = v;
            vertexData.normal = normal;

        }

        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            // Albedo comes from a texture tinted by color
            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            o.Albedo = c.rgb;
            // Metallic and smoothness come from slider variables
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
}
```

UML DIAGRAM



BIBLIOGRAPHY

Sources used/referenced in the creation of my project:

- https://www.gamedeveloper.com/programming/water-interaction-model-for-boats-in-video-games
- https://www.youtube.com/watch?v=1WCPFwxZpVg
- https://catlikecoding.com/unity/tutorials/flow/waves/
- https://en.wikipedia.org/wiki/Trochoidal_wave
- https://www.red3d.com/cwr/boids/