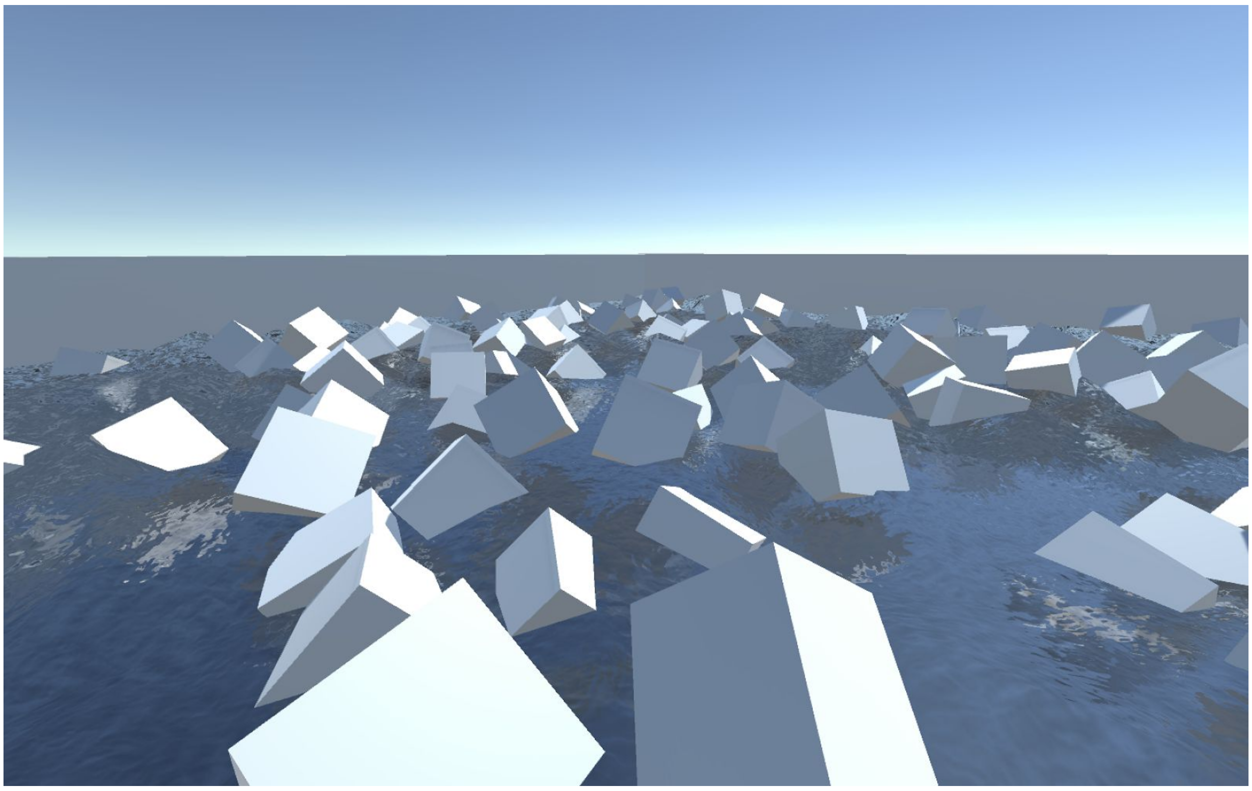
Hello!
We are Kang Sheng and Nikhil.
This is our graphics project.



Realistic boat simulation in Unity

50.017 Graphics & Visualisation

1000591	Tay Kang Sheng
1000635	Nikhil Sharma

Introduction

In this final project for 50.017 Graphics and Visualisation project, our group aims to build a computationally efficient simulation of objects in water suitable for video games and real time interactions. There are many models of implementation of fluids in computer graphics and one of the more common methods of implementation are physics based. One example of such implementation is the Smoothed-Particle Hydrodynamics (Monaghan, 2005). However, such model of implementations are complex to implement, computationally heavy and requires optimising data structures for real time and realistic graphical simulations. In our model of boat simulation, we aim to use a simplified model to create real time and passable interaction between objects and water suitable for video games and rapid prototyping.

Change from original proposal

Our initial idea for the project was to create a curved world in Unity that would look something like the curved world in the movie Inception. However, after some research, it turned out that replicating the world would not require a lot of coding, but rather, more time spent manually placing the buildings. Procedural modelling would have taken care of the generation of the cities, but we felt that generating the city was not the main part of the project. We wanted to make a project around fluids. It was originally to be implemented as wind blowing around the city but we decided that showing fluids in terms of water bodies would be more accurate as well as show off the effect on a more grandiose scale.

Research

Currently, Unity has no in-built buoyancy engine and hence most people use a buoyancy script that can be found on Unity's Asset Store. In our project, we want to recreate a simplified and effective model of the buoyancy effect and experiment its effects by floating the boat.

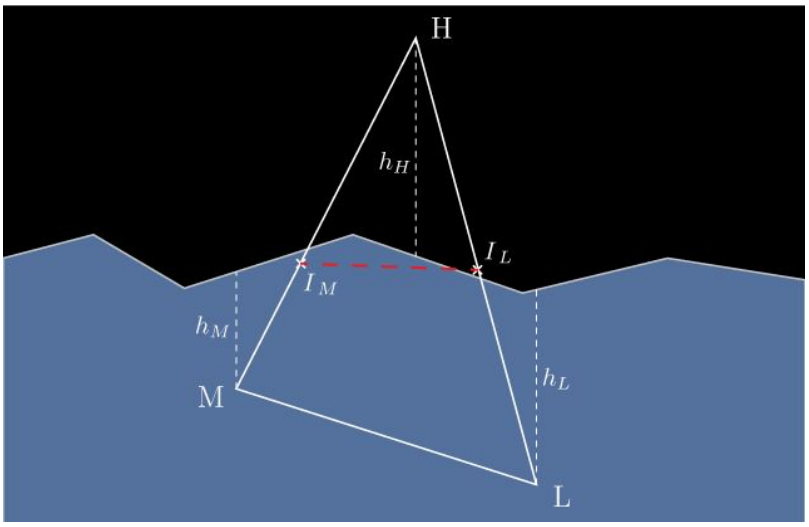
Objects float on water because of buoyancy. An object floats or sink depending on its relative density as compared to the medium it is in. If an object will sink if it is denser than its surrounding medium. If an object is placed in water, it will displace an amount of water equal to its density. The density of an object is how much mass an object has divided by its volume. For example, if we have two objects made from different materials, such as foam and wood, even though they are of the same size, the foam would be less dense than wood even though both objects have the same volume, they have different masses. The density of a boat depends on the hull and the air inside; it can float on the water by displacing an equivalent amount to that average.

$$F = p * g * A * h$$

F is the hydrostatic force acting on the floating object, keeping it afloat; ***p*** is the density of fluid; **A** is the surface area and **h** is the height of the object.

Hydrostatic force acts on the all parts of the surface under water. However, we only need to consider the vertical component of the force as the horizontal component of the horizontal components will all cancel out leaving just the vertical forces. Intuitively, if an object is a closed volume, there will always be another surface facing the opposite direction at the same depth. Therefore the overall buoyancy effect will be due to the different magnitude of the vertical forces acting on the different parts of the object that is under the water. These forces will result in a resultant force and torque with reference of the center of gravity of the object.

In order to calculate the parts that are underwater, we need to calculate what is the area of the object that is under the water. Since surface mesh is discretised into triangles, we can loop through all the triangles in the surface mesh and use a cutting algorithm on each triangle to approximate the parts of the triangle that is under the water. Adding up all the parts of the triangles that is under the water, we can put them together and form a mesh representing the underwater surface of the object. The cutting algorithm and the approximation is explained in more detail in this article.

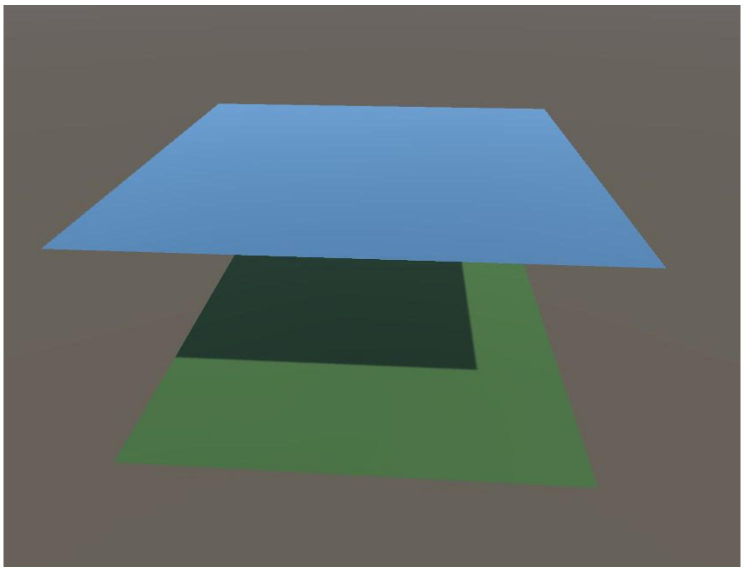


Putting together the underwater mesh and the formula for calculating hydrostatic force, we can sum up the total buoyancy force and apply it to the object.

Implementation

In this section, we will be providing a step by step guide to create the simplified boat simulation as shown in the cover page together with explanation of how and why each of these steps are important and relevant. All implementations are done using the Unity Engine so that we can abstract away from mundane openGL environment setup and focus on the hacks and model implementation of our project.

Step 1: Creating the environment

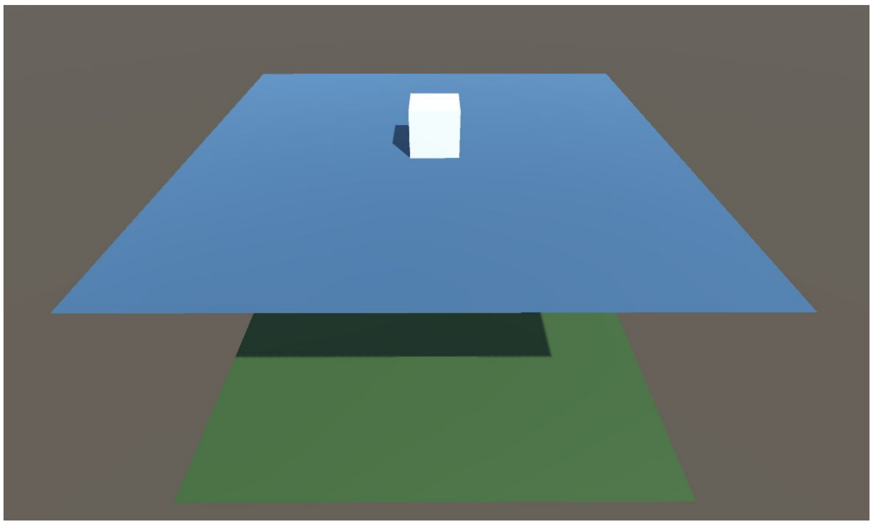


The first step to create our environment is to create the basic mesh for the surface of water and seabed. Water and seabed are represented using triangle meshes as points of the triangle meshes can be easily retrieved and manipulated to create simple wave like effects. A simple initialisation of the water and seabed mesh is by creating two 3D plane objects in Unity.

The default mesh of the plane object in unity is limited to a 10 by 10 square mesh and is not advisable to be used to represent the water surface because there is not enough vertices to create a smooth wave like effect. Therefore to create a large plane with enough vertices, it is advisable to write a function to generate your own flat plane mesh with enough vertices and triangles for a more realistic graphical representation of water. The default mesh collider on the 3D plane object for water should be removed so that objects will be able to drop through the plane just like how objects are able to drop under the surface of water. On the other hand, a simple seabed is a flat and static one. It only needs to detect collision with objects and stop objects from falling below the seabed. Therefore there is no need to create a more detailed mesh for the seabed and the default 10 by 10 square mesh of the 3D plane object will suffice. Meshes are recommended to be axis aligned so that it simplifies the calculation of forces in further parts of the report.

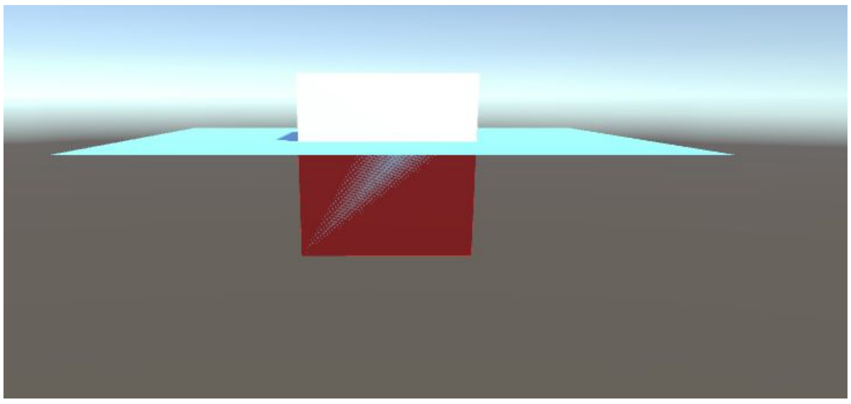
Drawing meshes in Unity is similar to drawing meshes in openGL. Generate a set of vertices and group these vertices in sets of three in the clockwise direction for Unity to draw these triangles on the screen.

Step 2: Creating a simple representation of a boat



For a start, our boat will be represented by a simple 3D cube. In the future implementation of this project, users can experiment with different 3D objects of different shape and sizes and play around with the interaction with the water mesh. Our boat should be subjected to gravity forces, able to drop under the water surface but not through the seabed. Create a default Unity 3D cube object to represent our boat, and add the rigidbody component to this object. If our boat is placed above the water mesh and upon starting the scene in Unity, our boat should drop through the water surface, collides and rest on top of the seabed. This is because we have not added any buoyancy forces to objects under the water and therefore our boat is only subjected to the effects of gravity.

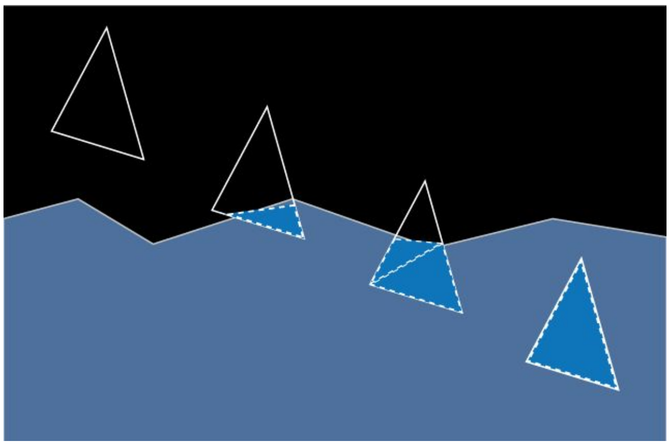
Step 3: Creating the underwater mesh



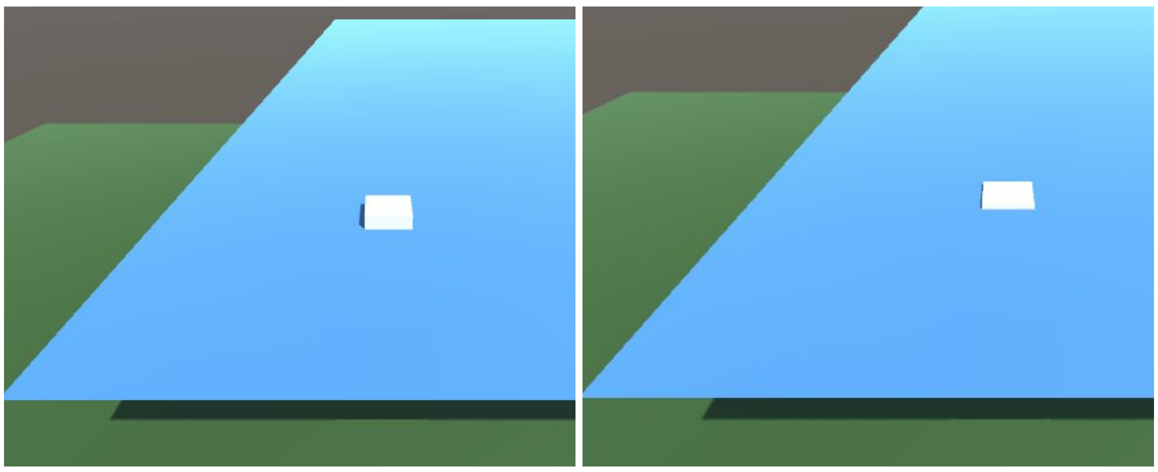
We would need to find the area of the game object that is underwater, so on every update cycle, we will loop through all of the triangles that make up the game object. For each triangle that is partially or fully submerged into the water, we will approximate the parts that are under the water and recalculate the triangles to add into the underwater mesh. We use the distance of each vertex of the triangle to the corner of the water line to check whether the triangle is partially or fully submerged into the water. There are four possibilities as to the direction of the triangle.

1) All vertices are above the water (triangle above water)
2) All vertices are below the water (triangle fully submerged in water)
3) One vertex is under the water (triangle is partially submerged in water)
4) Two vertices are under the water (triangle is partially submerged in water)

Based on each possibility, we recreate triangles to form the surface under the water and add these triangles into the underwater mesh. Now we have a new mesh that represents the part of the boat that is underwater.



Step 4: Adding buoyancy forces to the boat



Adding the correct buoyancy forces to the object allows the object to float realistically in water. We would model the script according the equation stated above. This script would be attached to all the rigidbodies that would be created, thus allowing the objects to interact with the water mesh and bob up and down. To calculate the buoyancy force, we calculate the following:

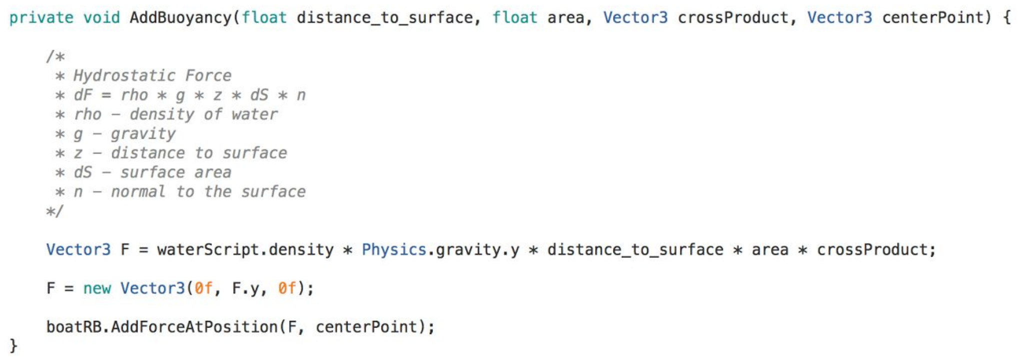
1) The center of the triangle, which is where we are going to add the force
2) The distance to surface from the center point
3) The triangle's normal vector
4) The area of the triangle

We then proceed to calculate the hydrostatic force.

$$dF = p * g * z * dS * n$$

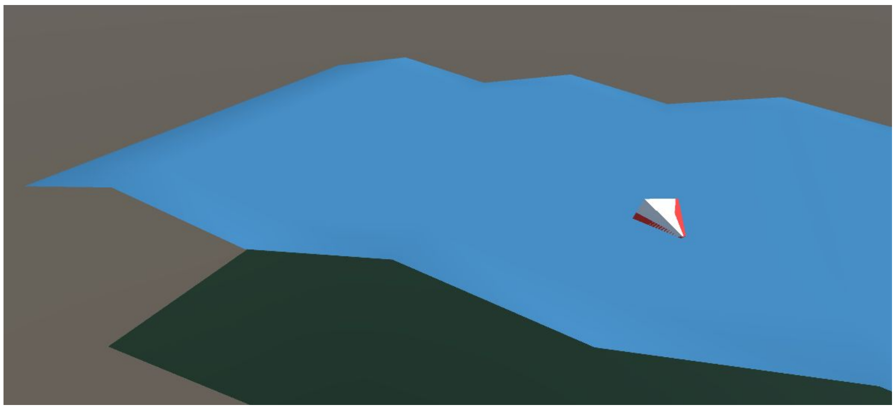
Where the **dF** is the change in hydrostatic force, ***p*** is the density, **z** is the distance to surface, **dS** is the surface area and **n** is the normal to the surface.

After adding this force to all our objects, the objects will start floating on the water.



```
private void AddBuoyancy(float distance_to_surface, float area, Vector3 crossProduct, Vector3 centerPoint) {

    /*
     * Hydrostatic Force
     * dF = rho * g * z * dS * n
     * rho - density of water
     * g - gravity
     * z - distance to surface
     * dS - surface area
     * n - normal to the surface
    */

    Vector3 F = waterScript.density * Physics.gravity.y * distance_to_surface * area * crossProduct;

    F = new Vector3(0f, F.y, 0f);

    boatRB.AddForceAtPosition(F, centerPoint);
}
```

Step 5: Adding wave forces to the water

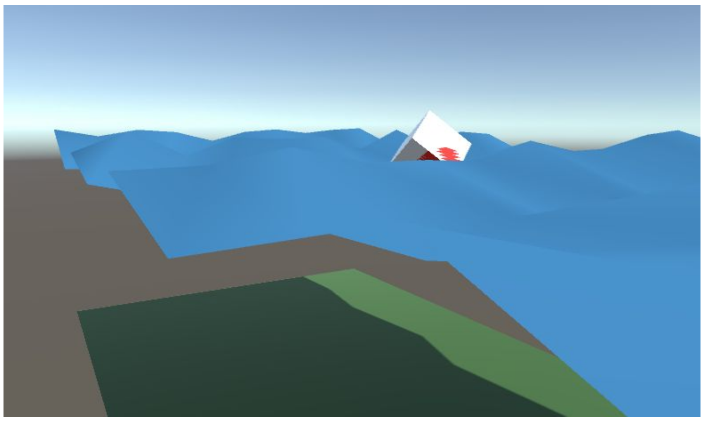


After making the objects float, we need to add more dynamism to the scene. As such, we wanted to create waves to have a more realistic ocean scene. We went about this by modelling the wave on a sinusoidal wave. We began by creating a list to hold all the y-coordinates of the vertices of the water mesh. We would then modify the vertices by applying a sinusoidal function that changes with respect to time. We would then save these new vertices. This function would keep repeating to have constant waves.

$$yCoordinate \mathrel{+}= Mathf.Sin((Time.time * speed + zCoordinate) / waveDistance) * scale;$$

This function includes the z coordinate to add a constant that changes. Speed was a constant that would increase how fast the waves were flowing. Wave distance allowed us to change the distance between the crests of the wave. Scale was how high the wave was allowed to be.

Step 6: Applying Perlin Noise to water surface

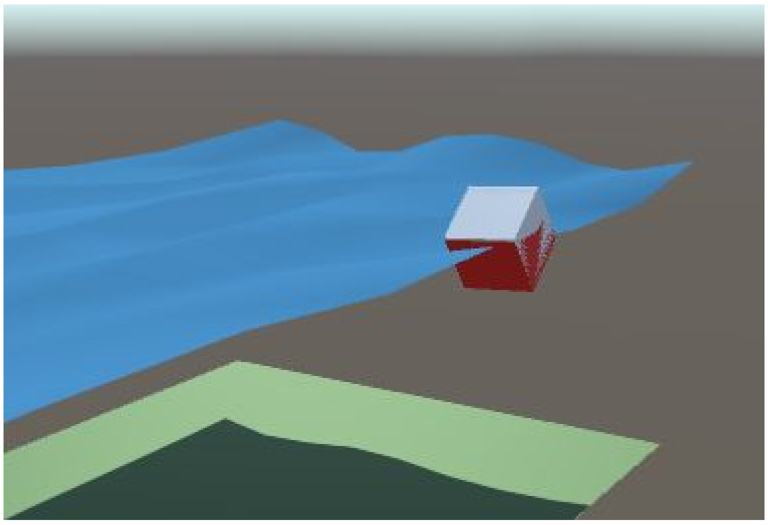


In order to make the water look more realistic, we decided to add perlin noise that makes the water surface appear as though it was being slightly buffeted by passing winds. This was done by using Unity's in-built Perlin Noise generator, which we made more dynamic by changing it with respect to time, similar to the wave generation function that was defined above. The function that we used was as below.

$$yCoordinate \mathrel{+}= Mathf.PerlinNoise(x\_coord, z\_coord + Mathf.Sin(Time.time * 0.1f);$$

The idea of using perlin noise was found on a unity forum where users would post tips on how to improve scenes to make them more realistic.

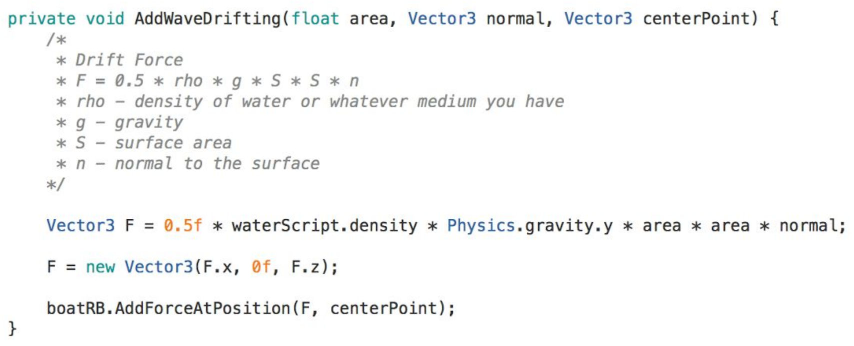
Step 7: Applying Drift Force to game objects



Now our water looks realistic with waves and our cube floats. However, the cube does not move with the waves for now. We have to add a drift force to the cube in order for it to move with the waves.

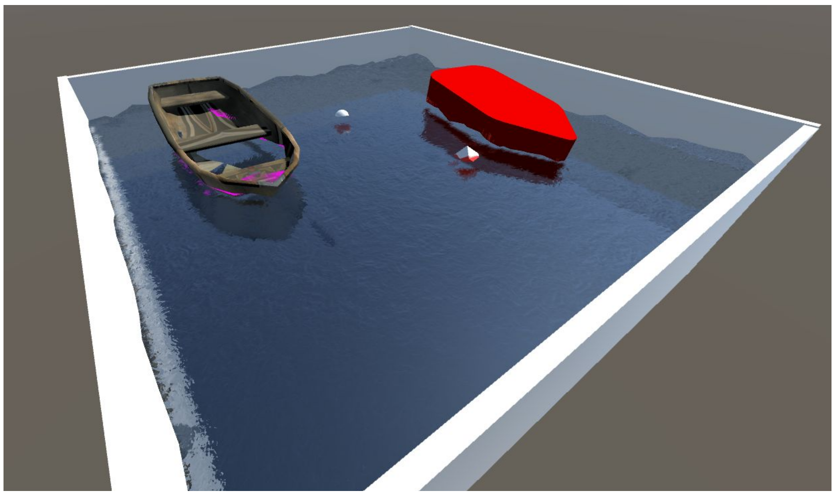
$$F = 0.5 * p * g * S * S * n$$

The drifting force was referenced from the WikiWaves article. We simply added this equation in the game objects file. This would allow the floating object to drift with the water.



```
private void AddWaveDrifting(float area, Vector3 normal, Vector3 centerPoint) {
    /*
     * Drift Force
     * F = 0.5 * rho * g * S * S * n
     * rho - density of water or whatever medium you have
     * g - gravity
     * S - surface area
     * n - normal to the surface
    */

    Vector3 F = 0.5f * waterScript.density * Physics.gravity.y * area * area * normal;

    F = new Vector3(F.x, 0f, F.z);

    boatRB.AddForceAtPosition(F, centerPoint);
}
```

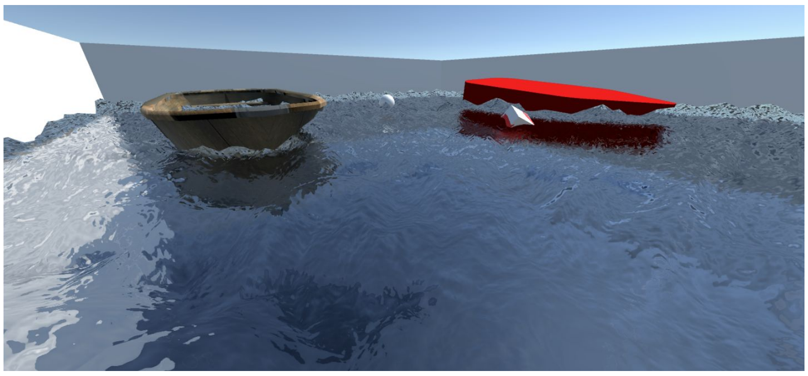
Step 8: Beautifying the scene



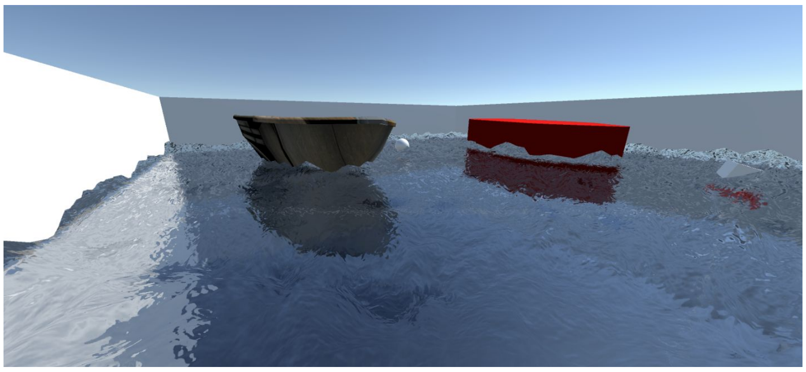
After the above steps were completed, we had a working scene. However, the water was an unnatural blue and was not very realistic. For a more presentable water scene, we use a water texture from the Unity Asset Store. The Unity Asset Store has many assets created by users for other Unity developers to use. After applying the texture to the water material, the scene came to life. We also downloaded a boat asset off the Asset Store, and replaced our simple white cube with the boat asset. All we had to do was apply the game object script to the boat in order for it to have buoyancy as well as be affected by the wave and drift forces. In the end, we had a realistic boat floating in water, with waves gently buffeting the boat, and causing the boat to drift.

Experiments

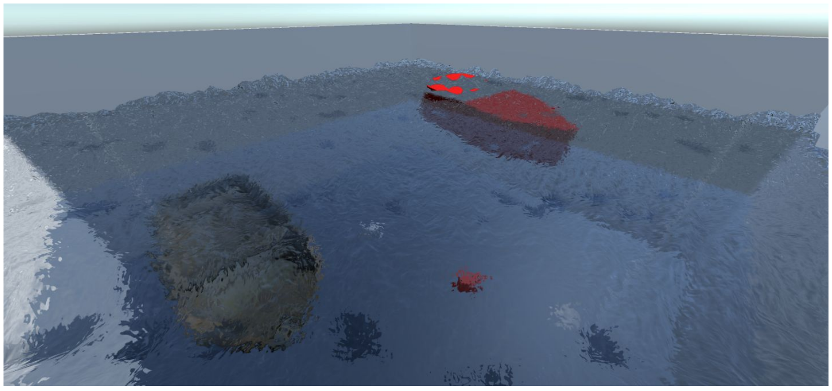
In order to test out the physics capabilities of our project, as well as to show the realism of the world that we created, we run the scene multiple times with different water densities. As stated above, the density of the water affects whether an object can float or sink in it.



The first experiment that we conducted was keeping the water density at 1,000 kg/m^3. This is the actual density of water. Both boats were partially submerged in the water. Boats are usually at this height in water as the weight of the boat displaces the same amount of water.



The second experiment that we conducted was changing the density of water to 500 kg/m^3. From the picture, we can see that the boats were practically floating above the water. The cube was also more susceptible to drift forces which caused the cube to move further away from it's spawn location as compared to the first photo.



The final experiment that we conducted, involved changing the density of water to 2,000 kg/m^3. All the game objects immediately sank. This would of course be changed if we were to adjust the weight of the game objects.

Thus, the experiment shows that the modelling of the water as well as the game objects were keeping in line with the laws of physics, and hence is a realistic simulation.

Conclusion

This was a very interesting project to do. We learned how to develop a scene using the Unity Game Engine at a more advanced level, as well applying concepts that we learned in classes on a higher level. Unity is a powerful tool that allows us to abstract some of the lower level components out and spend more time on the actual implementation of the scene. We also learned how to apply the Physics concepts we were taught a few years back, in a real-world simulation. It was amazing to see our physics knowledge tie in with our coding knowledge to make a beautiful, realistic scene come alive. You can also find our project on GitHub. We love CG!

References

Monaghan. J J (2005). Smoothed particle hydrodynamics.
http://iopscience.iop.org/article/10.1088/0034-4885/68/8/R01/meta

Kerner (2015). Water interaction model for boats in video games
http://gamasutra.com/view/news/237528/Water_interaction_model_for_boats_in_video_games.php