2D water simulation

# Introduction

In this report we’ll be going over what was done in this project, what it’s about, how it was done and a conclusion stating what went well with the project and what didn’t go well. The topic of this project is around 2D water simulation and ways to create a water simulation using the unity game engine. There were a few ways to tackle this topic but the way that was decided in this project was to use a line renderer, mesh renderers, triggers and particles using Hookes law and applying the Euler method to it to create a water simulation that shows waves and splashes.

Hookes law is a principle of physics that the force (F) needed to extend or compress a spring by some distance (X) is proportional to that distance. In an equation it looks like this F = kX where k is a constant factor characteristic of the spring. Hookes law is implemented into this project using a row of springs for the surface of the water, so as noted earlier the equation will be F = kX and in this project F will be the force produced by the springs, k is the spring constant and X is the displacement. We’ll discuss more about this in the implementation section of this report.

The Euler method is a first order numerical procedure for solving ordinary differential equations also known as ODEs with a given initial value. It is a more basic maybe even the most basic explicit method for integration of ODEs and it is the simplest Runge-Kutta method. What is meant by first order method is that the local error or error per steps is proportional to the square of the step size and the global error or error at a given time is proportional to the step size. This method is often used as the basis to build more complex methods such as the predictor-corrector method.

In this project we combine these two together to help create the simulation via colliders, springs and nodes. We’ll go over this in more detail in the code implementation section.

# Implementation

In this section we’ll be looking at the code and what was implemented and why. So to start off we’ll start from the very top, the surface of the water. We do this by using Unity’s line renderer and use nodes to appear as a continuous wave. To do this we’ll have to keep track of the position, velocities and acceleration of every node. To do this we used arrays for the x and y positions the velocities and the acceleration. We used the Linerenderer to store the nodes and outlines of the body of water.

float[] xPos;

float[] yPos;

float[] veloc;

float[] accel;

LineRenderer Body;

Next we needed to get some objects to contain our meshes and also set up the collider variable so that things can interact with the water.

GameObject[] meshobj;

Mesh[] meshes;

GameObject[] Col;

After that we set up our constants to store and give them base values. These constants include damping, spread, the z axis and the main spring constant as well.

const float springconst = 0.02f;

const float damping = 0.04f;

const float spread = 0.05f;

const float z = -1f;

After that we set up the variables for the body of water itself, so the base height the bottom and the sides. And also we set up the variables we need for our particle effects as well.

float baseheight;

float left;

float bottom;

// Particles

public GameObject splash;

public Material mat;

public GameObject watermesh;

Now that we’ve got everything set up variable wise it’s time to show off the creation of it all. We start off by determining how many nodes we want to use in this project.

int edgecount = Mathf.RoundToInt(width) \* 5;

int nodecount = edgecount + 1;

In this project we’ve decided to go with five per unit width to try and give a smooth motion that isn’t too demanding. This value can be changed but it does sacrifice performance. This gives us all of our lines and then we need, we also need the +1 for the extra node on the end.  
Now that’s been done we start to render the body of water using the Linerenderer component we built earlier.

Body = gameObject.AddComponent<LineRenderer>();

Body.material = mat;

Body.material.renderQueue = 1000;

Body.SetVertexCount(nodecount);

Body.SetWidth(0.1f, 0.1f);

Here we can actually vary how thick the line can be via the SetWidth, this allows the user to control how thick or thin the line is whether they want it to be noticeable or not.

After that we initialise all of our top variables and set up the arrays and the data they need.

xPos = new float[nodecount];

yPos = new float[nodecount];

veloc = new float[nodecount];

accel = new float[nodecount];

meshobj = new GameObject[edgecount];

meshes = new Mesh[edgecount];

Col = new GameObject[edgecount];

baseheight = top;

bottom = Bottom;

left = Left

This now sets up the arrays, what happens next is to actually set the values of the arrays by using a for loop.

for (int i = 0; i < nodecount; i++)

{

yPos[i] = top;

xPos[i] = Left + width \* i / edgecount;

Body.SetPosition(i, new Vector3(xPos[i], top, z));

accel[i] = 0;

veloc[i] = 0;

}

In this we set the y position to be at the top of the water and then we incrementally add nodes side by side. Initially the acceleration and velocities are set to 0 otherwise the water spawns in very strangely starting off bobbing up and down very fast. Setting them to 0 initially gives the image of still water as well.

Now the line has been made we move onto creating the mesh for the water itself. The meshes contain quite a few variables in them the first of which we’ll cover is the vertices.

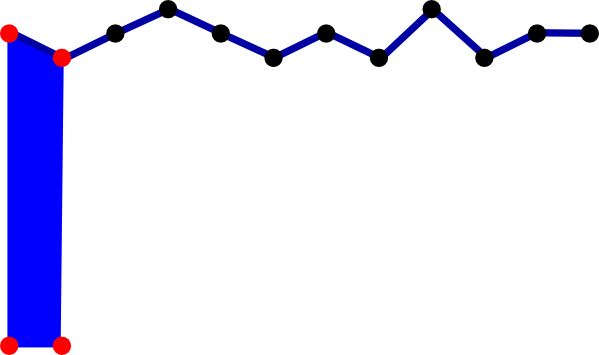


Figure 1: Example of how the mesh segments should look

Figure one shows how we build our mesh segments and how it looks on an individual basis. As seen in Figure 1 we have 4 vertices we want to set up. Another thing we set up here is the UVs for the mesh texture, the UVs choose which part of the textures we want to grab, so in this case we want the top right and left and bottom right and left. After we set up the vertices and the UVs we use the information we just set up we now need to tell the mesh how to draw out the triangles we need, as meshes are made up of triangles and we know that any quadrilateral can be made up of two triangles.

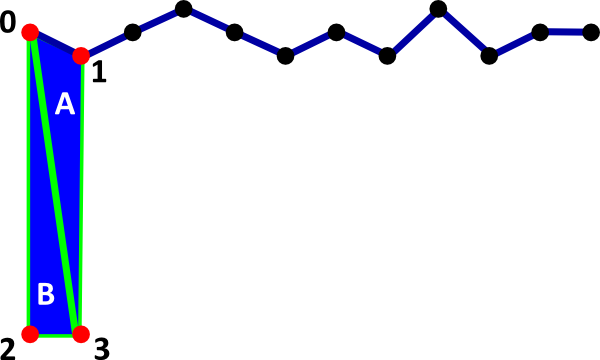


Figure 2: Example of how we want the triangles on the mesh

So looking at Figure 2 we can see with the node order labelled that triangle A connects nodes 1, 0 and 3 whilst triangle B connects 0,2 and 3. Now we know this we want to create an array of six integers reflecting that so we can create our quadrilateral. Once this was done we set up the mesh values and we make it so they’re created from a prefab which contains a mesh renderer and mesh filter.

The next step was setting up the colliders. Here we made them box colliders and gave them a specific name so that it was a bit more tidy in the scene. We then also make them a child of the main water gameobject and set them to be about half way between each node and set their size to fit accordingly, we then finish this off by adding a detection class to it so when a collision does happen it calls to that script and runs accordingly.

The next step was implementing that actual physics using Hookes law using the Euler method. So as said earlier Hookes law is F=kX, in this project the displacement is going to be the y position of each node minus the base height. We then add a damping factor proportional to the velocity of the force to dampen the force. Setting up the Euler method here is actually very simple, we just add the acceleration to the velocity and the velocity to the position every frame.

Next we create some wave propagation. We create two arrays. For each node we check the height of the previous node against the height of the current node and store the differences into one of the arrays. We then check the height of the subsequent node against the height of the node we’re checking and put that difference in the other array. We also multiply all value by a spread constant. This part of the code is all done in a for loop which loops a total of eight times. The reasoning behind this looping is because we want to run the process in small doses multiple times rather than one large calculation which could cause a less than fluid experience.

After setting all that up we moved to making the splash effect which is done via particles. We created another function called splash which checks the position of the splash and the velocity of whatever is hitting. This is public so it can use the colliders created earlier.

Once the particles have all been set up to detected when collision has happened and the life time of the particle effects have been determined, we moved onto the actual collision detection itself. We now go into the script that was mentioned earlier when setting up the colliders. In this script we calculate when objects have collided with our colliders and depending on the mass and velocity of the object what result we get from it, larger mass equalling a larger effect and splash.

With this all done and dusted the last part was spawning in objects to collide with the water. To do this we created a simple spawning mechanic which allows the user to spawn in two different objects, each with different mass to display the difference in the collision.

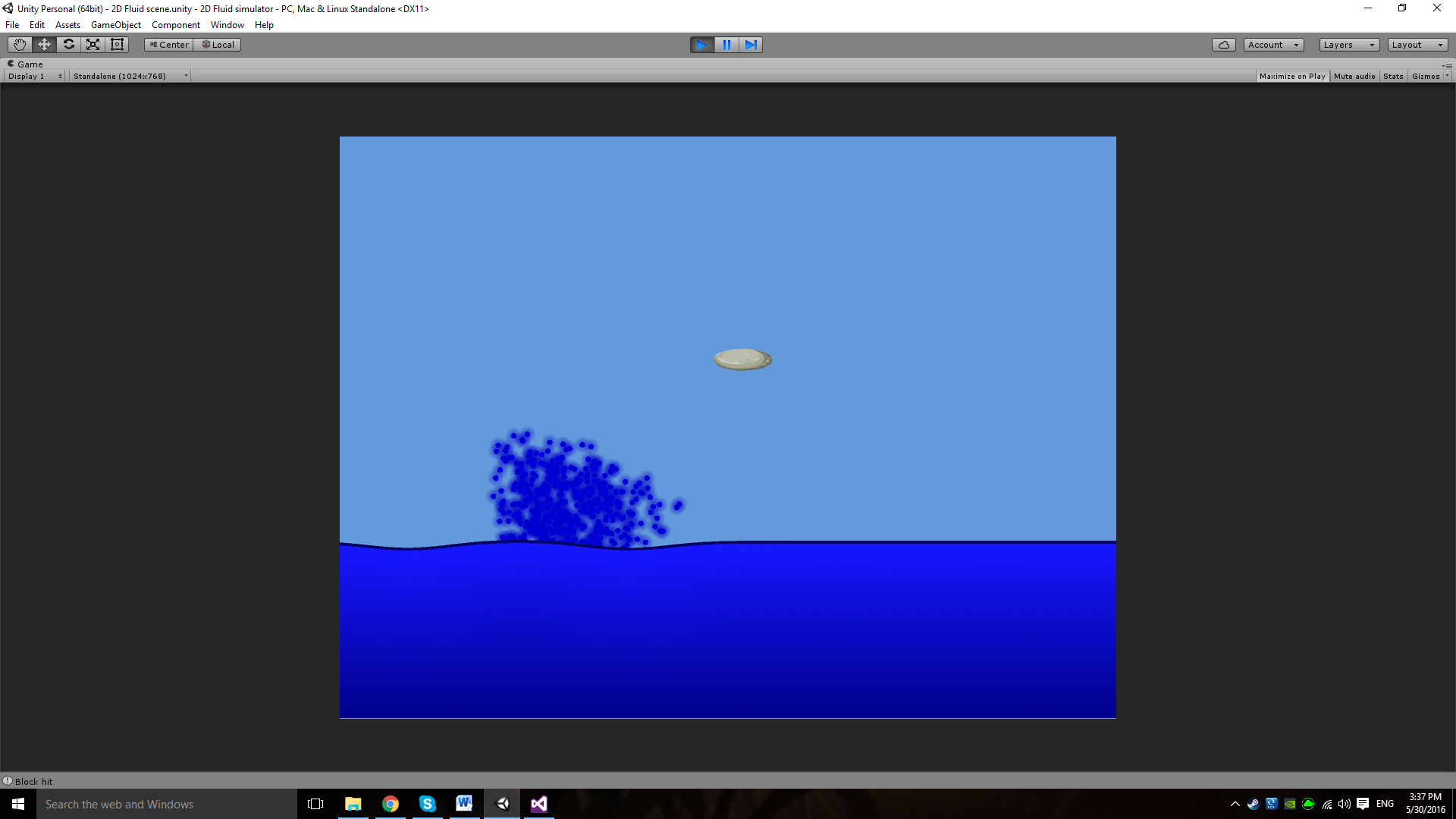


Figure 3: An example of the project in action#

Figure 3 shows off the working demo in progress, showing an object that has just collided with the water and has created a ripple effect based on the mass of the object.

If we had longer to work on this project some we’d like to implement would be some form of buoyancy to allow objects based on mass and velocity the possibility of floating on the surface. There is a component in unity called Buoyancy Effector 2D which does simulate it in a fashion similar to the way we’d want, however there are a few issues with using this. Firstly when applying the effector it requires a rigid body to be applied to the object it’s on and has the effector ticked. When we attempted this it spawned an inviable 2D box in the middle of the water that the objects would collide with which made them look silly as they’d hit it and stay just above the water. We could not find a way to reposition this box and decided it was not worthwhile as the second reason we didn’t use it was because it’s a simple add-on rather than hard code which would have been the more interesting aspect, however again we could not figure out how to implement this in the time given.

# Conclusion

Whilst we understand the project done is fairly simple in terms of what it is, we’re very happy with the outcome and what has been achieved in this project. If this project could be done again firstly we’d tackle other possible ways of handling the physics, for example rather than using the simple Euler method, another method that could have been used is the Verlet Integration for more precise physics. However because of the use of damping and because of the simplicity it was decided to stick with the Euler but if given time and another opportunity it’d be an interesting change to see this project running with the Verlet integration instead. Another thing that’d be added would be different scenes containing possible different types of liquids, so for example in this project we focused on water and tried to make the physics as water like as possible, what would be quite nice to add would be other types of liquids as maybe something like lava, or acid and try and simulate those kinds of fluids and build it around how they would react to collisions. The last thing that was one of the bigger things that was wanted to be tackled was the use of metaballs to create the splash rather than use particles. This could have created a really nice simulation of a splash, however with all the other parts of the project that was being researched there was not enough time to really get a good look at how to implement it into the methods used in this project. All in all whilst a very simple result has been produced here we’re still very proud of what has been created in the time for this project.

# Refrences

BBC (2006) *GCSE Bitesize: Hooke’s law*. Available at: http://www.bbc.co.uk/schools/gcsebitesize/science/add\_aqa/forces/forceselasticityrev2.shtml (Accessed: 25 May 2016).

Dummer, J. (no date) *Verlet explained*. Available at: http://lonesock.net/article/verlet.html (Accessed: 30 May 2016).

Khan Academy (2015) *Euler’s method*. Available at: https://www.khanacademy.org/math/differential-equations/first-order-differential-equations/eulers-method-tutorial/v/eulers-method (Accessed: 30 May 2016).

The Editors of Encyclopædia Britannica (2016) ‘Hooke’s law | physics’, in *Encyclopædia Britannica*. Available at: http://www.britannica.com/science/Hookes-law (Accessed: 26 May 2016).