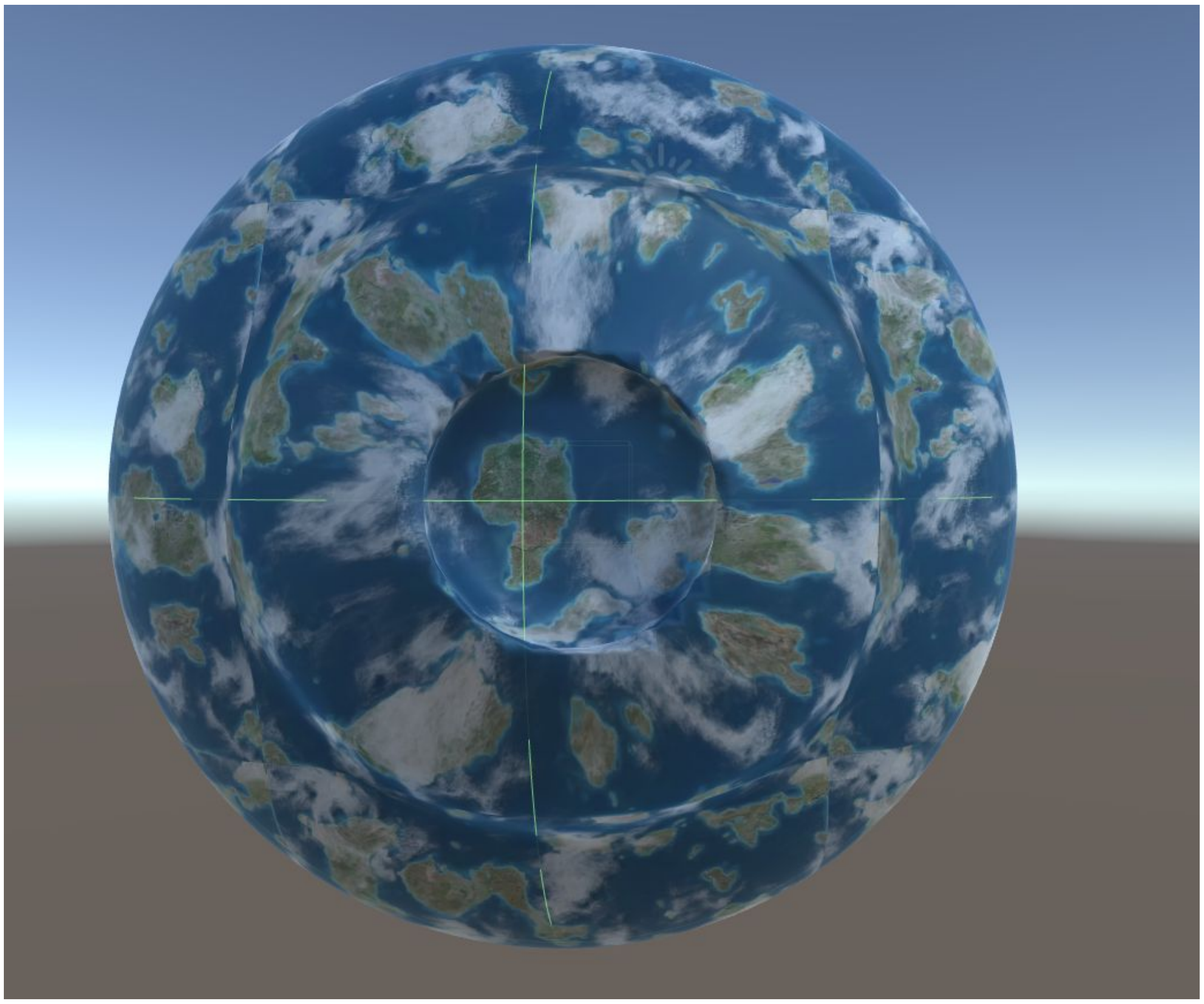
Game Lab - Assignment 1 - Raymond Becking 500785471



Mesh Generation

https://www.youtube.com/watch?v=AfS9aKVa4eE

Idea

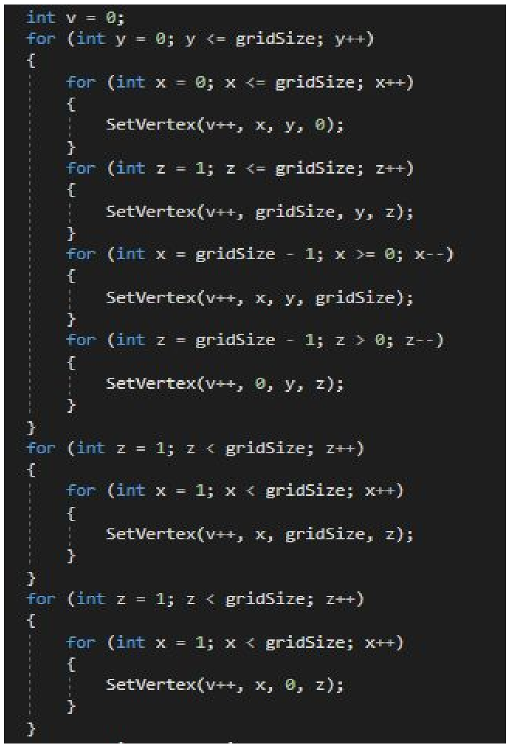
I wanted to create a mesh based effect to simulate a huge explosion or meteorite impact. So I wanted to create a mesh based sphere/planet where the impact would take place. When you click on the planet you create a “ripple” effect that looks like a shockwave, to do this I used mesh deformation in combination with a sinus formula. These are the two components I will explain

The sphere

The sphere i created is a cube sphere, it uses a cube as base. To go from a cube to a sphere, the sphere is projected from the cube.

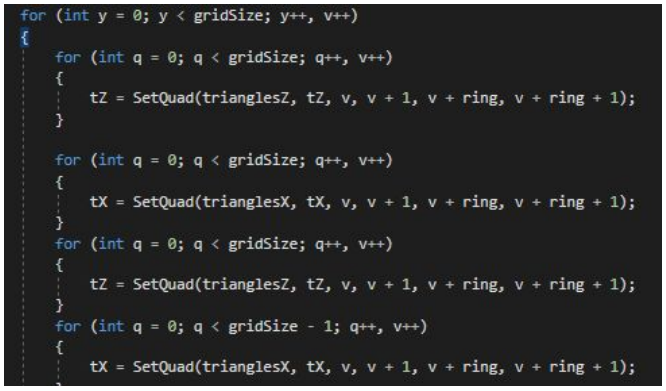
To project the sphere, the cube needs to be created first. To create the cube, the vertices need to be created first, each setvertex inside a for loop creates a side of the cube. The last 2 loops create the top and bottom of the cube.

Vertices:



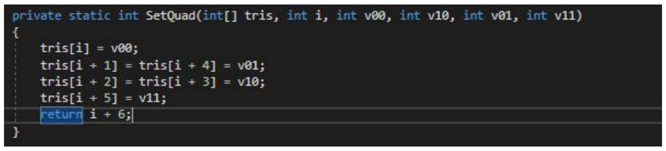
```
int v = 0;
for (int y = 0; y <= gridSize; y++)
{
    for (int x = 0; x <= gridSize; x++)
    {
        SetVertex(v++, x, y, 0);
    }
    for (int z = 1; z <= gridSize; z++)
    {
        SetVertex(v++, gridSize, y, z);
    }
    for (int x = gridSize - 1; x >= 0; x--)
    {
        SetVertex(v++, x, y, gridSize);
    }
    for (int z = gridSize - 1; z > 0; z--)
    {
        SetVertex(v++, 0, y, z);
    }
}
for (int z = 1; z < gridSize; z++)
{
    for (int x = 1; x < gridSize; x++)
    {
        SetVertex(v++, x, gridSize, z);
    }
}
for (int z = 1; z < gridSize; z++)
{
    for (int x = 1; x < gridSize; x++)
    {
        SetVertex(v++, x, 0, z);
    }
}
```

To fill in the sides, the for loops loop around the cube. Each for loop creates a side based on the size set in Unity (gridSize). By adding two opposite sides to a single array, they will look the same when used as a submesh.



```
for (int y = 0; y < gridSize; y++, v++)
{
    for (int q = 0; q < gridSize; q++, v++)
    {
        tZ = SetQuad(trianglesZ, tZ, v, v + 1, v + ring, v + ring + 1);
    }

    for (int q = 0; q < gridSize; q++, v++)
    {
        tX = SetQuad(trianglesX, tX, v, v + 1, v + ring, v + ring + 1);
    }
    for (int q = 0; q < gridSize; q++, v++)
    {
        tZ = SetQuad(trianglesZ, tZ, v, v + 1, v + ring, v + ring + 1);
    }
    for (int q = 0; q < gridSize - 1; q++, v++)
    {
        tX = SetQuad(trianglesX, tX, v, v + 1, v + ring, v + ring + 1);
```

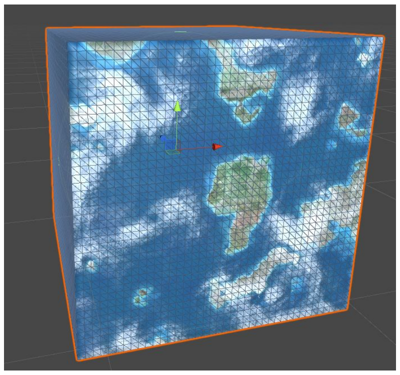
Before the triangles are placed, they are first transformed into a quad. To do this they need to be placed in a clockwise manner.



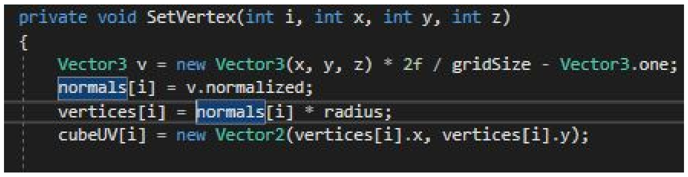
```
private static int SetQuad(int[] tris, int i, int v00, int v10, int v01, int v11)
{
    tris[i] = v00;
    tris[i + 1] = tris[i + 4] = v01;
    tris[i + 2] = tris[i + 3] = v10;
    tris[i + 5] = v11;
    return i + 6;
}
```

The top and bottom are filled in with quads as well.

Finally, the vertices and triangles are applied to the mesh
After all this is done, the cube looks like this:



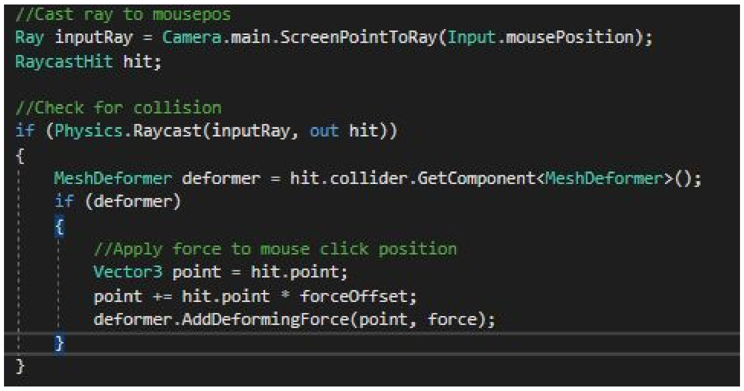
To transform this into a sphere, we need to project the sphere inside the cube. To do this we create a cube that exactly contains the sphere. By normalizing the vertices from the cube we create the roundness of the sphere, the length of these normalized vertices is based on the radius and thus allows resizing of the sphere.
The result can be seen on the front page.



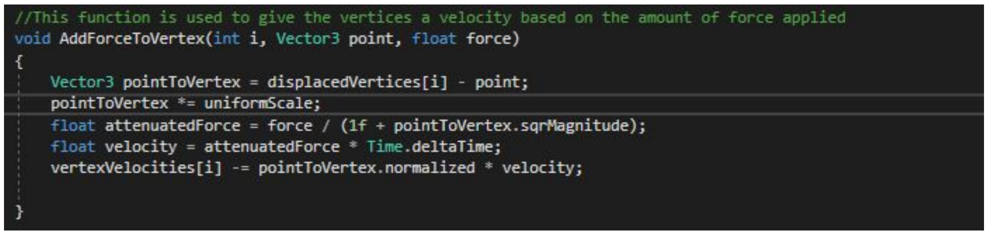
```
private void SetVertex(int i, int x, int y, int z)
{
    Vector3 v = new Vector3(x, y, z) * 2f / gridSize - Vector3.one;
    normals[i] = v.normalized;
    vertices[i] = normals[i] * radius;
    cubeUV[i] = new Vector2(vertices[i].x, vertices[i].y);
```

The mesh deforming

To deform on mouse input, a ray is cast from the camera to the mouse position.. If the raycast hits an object that has a deformer script, force is applied to that location. To make sure force is applied over a larger surface area, the hit position of the raycast is multiplied by a force offset.
The force is applied to all displaced vertices. This is then used to calculate a velocity based on the force applied, this will change how fast the vertices will be moved. To make sure the vertices move the right direction, the location where the mouse clicked is inverted and normalized.

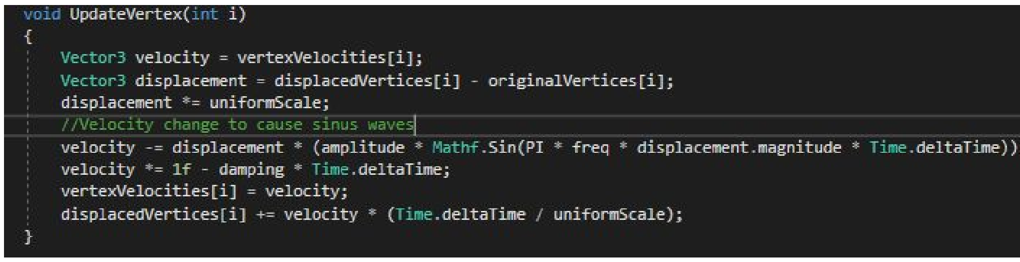


```
//Cast ray to mousepos
Ray inputRay = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit hit;

//Check for collision
if (Physics.Raycast(inputRay, out hit))
{
    MeshDeformer deformer = hit.collider.GetComponent<MeshDeformer>();
    if (deformer)
    {
        //Apply force to mouse click position
        Vector3 point = hit.point;
        point += hit.point * forceOffset;
        deformer.AddDeformingForce(point, force);
    }
}
```



```
//This function is used to give the vertices a velocity based on the amount of force applied
void AddForceToVertex(int i, Vector3 point, float force)
{
    Vector3 pointToVertex = displacedVertices[i] - point;
    pointToVertex *= uniformScale;
    float attenuatedForce = force / (1f + pointToVertex.sqrMagnitude);
    float velocity = attenuatedForce * Time.deltaTime;
    vertexVelocities[i] -= pointToVertex.normalized * velocity;

}
```

Based on the velocity the vertices are constantly updated. The velocity is also adjusted in the update to create a ripple effect. To do this the displaced vertices are multiplied by a sinus function. The amount of ripples and the size can be adjusted by changing the frequency and amplitude of this sinus function.



```
void UpdateVertex(int i)
{
    Vector3 velocity = vertexVelocities[i];
    Vector3 displacement = displacedVertices[i] - originalVertices[i];
    displacement *= uniformScale;
    //Velocity change to cause sinus waves
    velocity -= displacement * (amplitude * Mathf.Sin(PI * freq * displacement.magnitude * Time.deltaTime))
    velocity *= 1f - damping * Time.deltaTime;
    vertexVelocities[i] = velocity;
    displacedVertices[i] += velocity * (Time.deltaTime / uniformScale);
}
```