Application of Sphere Generation Algorithms for Procedural Planets

Introduction

In this article we intend to take a closer look at a few different ways one can generate spheres. There are four main sphere generation algorithms that have been taken into consideration for this article. Before we even start implementing them, we will consider the currently understood benefits and drawbacks of each. From there we will select two of the best candidates for the job at hand – generating a procedural planet – and move on to implementation. We will look at multiple ways the sphere algorithms can be implemented and consider pros and cons for the different methods. We will consider each implementation and decide which are the best suited for the job. In the end we hope to answer "which sphere generation algorithm lends itself best for procedural planet generation?"

For the implementation Unity3D will be used for ease of visualization.

Context

In games, planets are often represented with different types of spheres, all spheres have benefits and drawbacks. The definition of a sphere is a 3D closed surface where every point on the sphere is at the same distance away from the centre. We can describe a sphere at the origin as:

$$x^2 + y^2 + z^2 = r^2$$

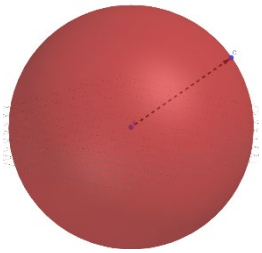
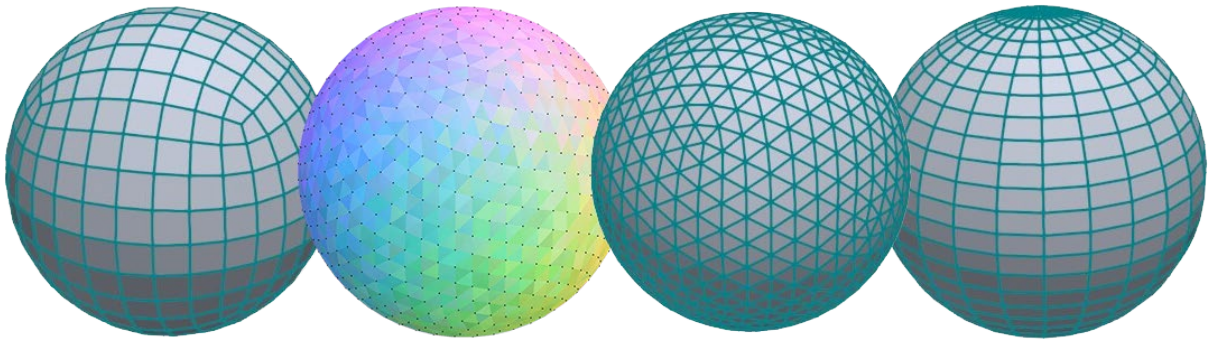


Figure 1. The Sphere Equation result

Since it is impossible to sample all the points on a sphere, we are satisfied with only sampling a limited number of points. There are quite a few methods that have been developed to help us select the most interesting points on a sphere. We can end up with points that make up a sphere in a variety of ways. In this article we will focus on four of these methods. The main purpose of these methods is to generate a selection of points that can be arranged to make up a sphere shape.



One solution to this is to generate a UV-sphere also known a radial sphere. The radial sphere attempts to solve this problem by separating a sphere into lateral bands going all the way around the sphere

and longitudinal bands running from pole to pole. With these bands we can group every point where these bands intersect. This can be calculated by deciding on how many stacks (longitude) and slices (lateral) is needed. The formula to find a points x, y, z coordinate can be determined with these calculations:

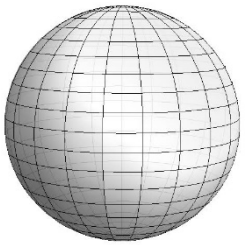


Figure 3. UV Sphere mesh

$$x = (r \cdot \cos\phi) \cdot \cos\theta \quad y = (r \cdot \cos\phi) \cdot \sin\theta \quad z = r \cdot \sin\phi$$

$$\text{Where } \theta = 2\pi \cdot \frac{\text{currentSlice}}{\text{maxSlices}} \text{ and } \phi = 2\pi \cdot \frac{\text{currentStack}}{\text{maxStacks}}$$

In his article [1] Song Ho Ahn explains how the formula can be implemented to find the points' coordinates.

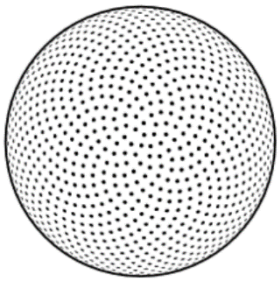


Figure 4. Point selection of Fibonacci sphere

Another method is known as the Fibonacci sphere, named after the Fibonacci ratio. This solution is discussed in the works of [2] González, where the generation of this sphere is explained by González's quote "The points of the Fibonacci lattice are arranged along a tightly wound generative spiral, with each point fitted into the largest gap between the precious points". The paper presents a piece of pseudocode for generating the coordinates in degrees for the points on the sphere. [3] Kogan in his paper offers a slightly less accurate alternative which is less computationally heavy. One drawback with the Fibonacci sphere is that there is no straight forward or simple method of triangulating the points gathered.

Two other methods of generating spheres are the cube sphere and the icosphere, these are two solutions to one problem. Keeping even distribution of vertex information on a spherical shape.

When [4] Richi, C. et al. discuss the use of a spherified cube, one of the benefits given is that one can avoid the singularities on both poles. Another advantage of using a spherified cube is that each of the 6 sides can be treated equally by using a rotation matrix to move each side into the same space when performing calculations. However, one of the drawbacks with this method is that the Sphere's UV map ends up with stretching along the center of each side, and the vertex density is noticeably denser around the corner compared to center of the side. A solution to this is suggested by Phil Nowell in [5] Mapping a Cube to a Sphere | Math Proofs. (n.d.). Another approach is to start with an icosahedral triangulation of the sphere instead of cubic, [6] Baumagerdner J. R. et al. takes us through the construction of a sphere of this type. One of the advantages brought up is that this shape starts out and preserves - throughout subdivision - an almost uniform triangulation of a sphere.

What to consider when choosing spheres

What makes a good candidate in this situation is a sphere that can be segmented into multiple parts without increasing the complexity, so that each part can be treated individually. A uniform triangle size and uniform spacing between the vertices would be a good quality to preserve even detail across the whole sphere.

The Fibonacci sphere

The Fibonacci sphere is likely the simplest sphere to generate programmatically. The points can be calculated with ease. With just a few lines of code you can generate an exact number of points on sphere and adjust it up and down one point at a time. This is a very neat feature of the Fibonacci sphere. And that's where it ends, for there is no simple way of triangulating the points after generating them. Another issue with the Fibonacci sphere is that it does not lend itself to be segmented into multiple parts. There are good aspects of this that makes it a good candidate, but then there are the other aspects that make it hard to consider as a candidate at all.

The spherified Icosahedron

Icosahedron is likely even simpler than the Fibonacci sphere to generate, however the base shape can be generated programmatically. It is much simpler to hardcode the position of each point and the order of the indices. The Icosahedron is a platonic solid, which is a convex regular polyhedron. This means that all 20 sides of the icosahedron are identical in shape and size. The points are spread relatively evenly across the sphere even after subdividing. All sides can be treated independently. All this sounds great but there is one drawback with this shape. When the shape is subdivided, the number of points grow exponentially. The shape starts with 12 points. After one subdivision, that grows to 42, after another it is 162, then 642 and so on. After 5 subdivisions the shape consists of 10242 points, and the jump in new points is over 30000 for another subdivision. This is not great. There is luckily more than one way to subdivide this sphere, but we will consider that when we get there.

The UV sphere, also known as the Radial Sphere

This shape is easy to understand, as well as to generate. One can start at one of the poles and generate points in a circle stepwise until reaching the other pole. The vertical and horizontal slices can be adjusted independently. It is easy to unwrap, easy to generate and easy to understand. But the issue is that the UV sphere is the shape with the most uneven vertex distribution. When the vertical and horizontal slices grow to a large number, the majority of the vertices can be found near the poles. Noticeably fewer exist around the equator causing a lack of detail. Finally, like the Fibonacci sphere, this shape cannot easily be divided into parts that can be handled independently from each other.

The spherified Cube

The Cube is like the Icosahedron, for it is also a platonic solid. It shares the same benefits and drawbacks. However, they are not the same. The cube has less of a jump in vertex count whenever it is subdivided, and each side can be treated as a grid or a plane, independent of each other. It also lets us easily split each side into new independent parts if we wish. One drawback the spherified cube has

that differ from the icosahedron, is that the vertices become denser packed around the edges and corners of the would-be cube. There are solutions to this that makes it easy to spread the points on the spheres surface more evenly. This leads to a good point distribution.

Conclusion

	Granularity	Complexity	Point Uniformity	Modularity	Face Uniformity	Result
Icosahedron	Bad	Moderate	VeryGood	Good	Very Good	Good
UV Sphere	Good	Low	Bad	Bad	Bad	Bad
Fibonacci Sphere	Very Good	High	Good	Bad	Bad	Bad
Cube Sphere	Moderate	Moderate	Good	Good	Moderate	Good

With these results we will focus on the Icosahedron and the spherified cube for this project, as they scored the highest. We may look at the other shapes another time.

The Setup

This project was made with Unity 2020.3.26f1 (64-bit) on a Lenovo Legion 5 pro laptop. To follow along you will need a blank Unity3D Project.

Making a multi-mesh Cube-Sphere

Earlier in the article we talked about a cube with the advantage that one can in theory generate one side at a time and moving all sides to the same space before any calculation. Quote “Another advantage of using a spherified cube is that each of the 6 sides can be treated equally by using a rotation matrix to move each side into the same space when performing calculations.”. We will generate a cube this way, however instead of rotating each side into one local space, we will choose a direction vector that will act as the up-vector for each space. The vector will act as the local spaces up-vector, we will do this for each side before generating the quad.

Implementation

To create this Cube, we first need to create a new C# script. This script will have two classes, one for the side-meshes and another to hold the side-meshes and the info it needs such as resolution, radius, MeshFilter container etc. The side-mesh class will be tasked with generating the mesh after taking in its settings.

We start by defining a few variables that can be considered as “Cube-Settings”. Most importantly a container for holding MeshFilters, a container holding the side-meshes and a variable for subdivisions.

```
[Range(2, 255)]
public int resolution = 10; // subdivisions

MeshFilter[] meshFilters; // mesh container
SideMesh[] sideMeshes;

static const int maxFaces = 6; // amount of sides on a cube
```

Next, we create two methods. First the method, passes this information to each side.

```
void Initialize()
{
	if(meshFilters == null || meshFilters.Length == 0 )
	{
		meshFilters = new MeshFilter[maxFaces];
	}

	sideMeshes = new SideMesh[maxFaces];
	Vector3[] directions = { Vector3.up, Vector3.down, Vector3.left, Vector3.right, Vector3.forward, Vector3.back };

	for( int i = 0; i < maxFaces; ++i )
	{
		if(meshFilters[i] == null )
		{
			GameObject gameObject = new GameObject("SphereFace");
			gameObject.transform.parent = transform;

			gameObject.AddComponent<MeshRenderer>();
			gameObject.GetComponent<MeshRenderer>().sharedMaterial = new Material(Shader.Find("Standard"));
			meshFilters[i] = gameObject.AddComponent<MeshFilter>();
			meshFilters[i].sharedMesh = new Mesh();
		}
		sideMeshes[i] = new SideMesh(meshFilters[i].sharedMesh, resolution, directions[i]);
	}
}
```

This method sets up the container of MeshFilters if there isn't already one, creates a new container to hold the side-meshes, sets up a container of the local up-vectors we need. And then loops over each face and creates a mesh if it doesn't already exist, and then finally passes this information to a new side-mesh. The second method we need is one that generate the sides when we are ready.

```
void CreateSides()
{
	foreach(SideMesh side in sideMeshes)
	{
		side.CreateSide();
	}
}
```

For the SideMeshes class, we'll need a Mesh to hold the mesh it should affect, a variable to store the resolution, a container to hold vertices, UV-Coordinates, and indices. We also need to store the local axes for the side. We can calculate the other axes after we get a Up-Vector. In the constructor we thus ask for: A mesh, the preferred resolution, and a local-up vector. We then calculate and assign these respectively in the constructor.

```
Mesh mesh;
int resolution;

List<Vector3> vertices = new List<Vector3>();
List<Vector2> texCoords = new List<Vector2>();
List<int> indices = new List<int>();
Vector3 localUp;
Vector3 localForward;
Vector3 localRight;

public SideMesh(Mesh mesh, int resolution, Vector3 localUp)
{
	this.mesh = mesh;
	this.resolution = resolution;

	this.localUp = localUp;
	this.localForward = new Vector3( localUp.y, localUp.z, localUp.x );
	this.localRight = Vector3.Cross( this.localUp, localForward);
	this.mesh.Clear();
}
```

Now we make the “CreateSide” method mentioned earlier. This method generates the vertices in a grid like manner, and we calculate the winding order if the vertex is not in the last row or column. If the vertex is in the last row or column, the triangles that would be generated would be outside the mesh. At the end of the method, we assign the generated vertices and indices to the mesh.

```
public void CreateSide()
{
	int indice;
	for ( int y = 0; y < resolution; ++y)
	{
		for ( int x = 0; x < resolution; ++x)
		{
			indice = x + y * resolution;
			const Vector2 progress = new Vector2(x, y) / ( resolution - 1 );
			const Vector3 vPosition = localUp + localForward * (2 * progress.x - 1) + localRight * (2 * progress.y - 1);

			vertices.Add(vPosition);

			if(x != resolution - 1 && y != resolution - 1)
			{
				indices.Add(indice);
				indices.Add(indice + resolution + 1);
				indices.Add(indice + resolution);
				indices.Add(indice);
				indices.Add(indice + 1);
				indices.Add(indice + resolution + 1);
			}
		}
	mesh.vertices = vertices.ToArray();
	mesh.triangles = indices.ToArray();
}
```

We now have a functioning script which can be added to an empty game object, now if we call Initialize and CreateSides after each other a cube will be generated based on the resolution we give it.

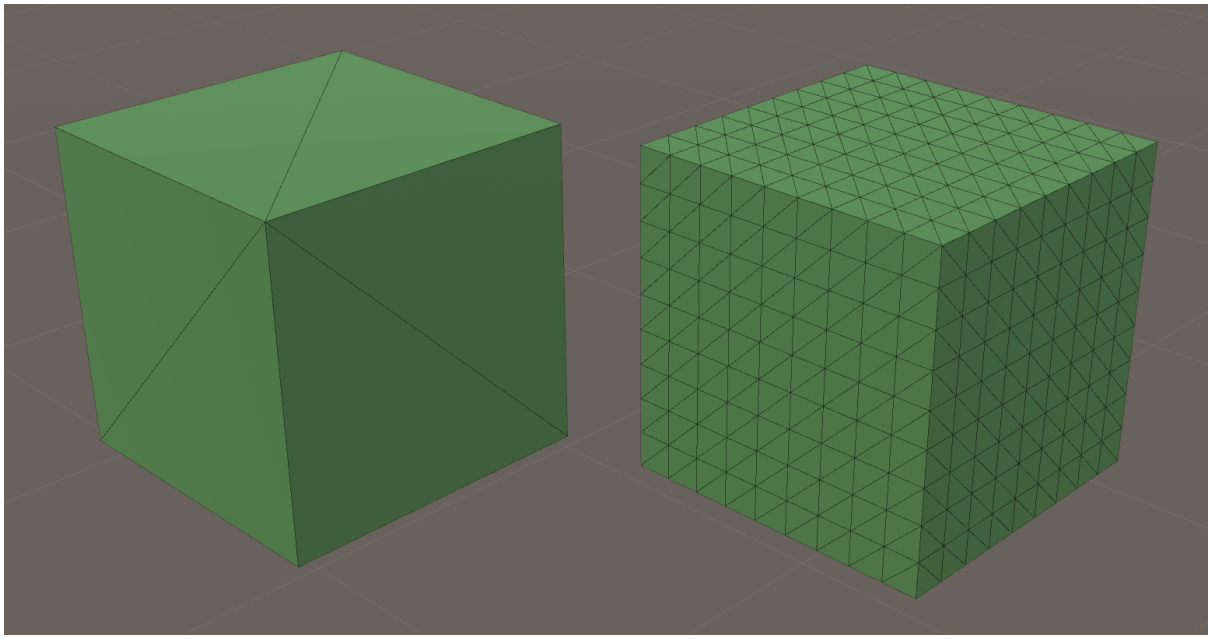


Figure 5. Cube with Resolution 2 (left), Cube with Resolution 10 (right).

We can turn these cubes into spheres by normalizing each vertex position then multiplying with a given radius. At the same time, we will add a function that can calculate the uv-coordinates of a vertex.

```
private Vector2 CalculateTexCoords(Vector3 p )
{
	Vector2 newUV = new Vector2();

	Vector3 nVec = p.normalized;
	newUV.x = ( Mathf.Atan2(nVec.z, nVec.x) / ( Mathf.PI * 2f ) ) + 0.5f;
	newUV.y = ( Mathf.Asin(nVec.y) / Mathf.PI ) + 0.5f;
	return newUV;
}
```

We update CreateSide() to reflect the new changes.

```
for ( int y = 0; y < resolution; ++y)
{
	for ( int x = 0; x < resolution; ++x)
	{
		// inside loop before //
		vertices.Add(vPosition);
		// above line is updated to the following //
		vertices.Add(vPosition.normalized * radius);
		texCoords.Add(CalculateTexCoords(vPosition));
	}
}
// we can add the uv-Coordinates to the mesh as well
mesh.vertices = vertices.ToArray();
mesh.uv = texCoords.ToArray();
mesh.triangles = indices.ToArray();
```

With the new changes the cubes are now spheres.

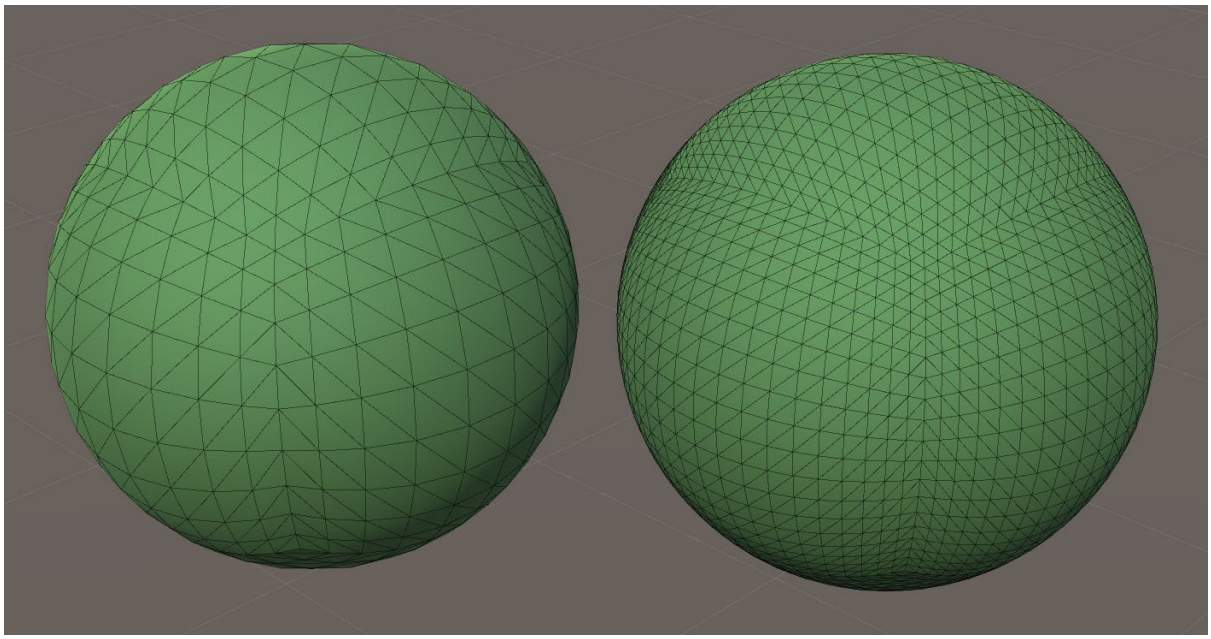


Figure 6. Cube-Sphere with Resolution 10 (left), Cube-Sphere with Resolution 20 (right). Both with a radius of 2.

We now have a script that generates Cube-Spheres for us. Note that the vertices are getting closer and closer together as they approach the edges of the sides. We address this next, as mentioned earlier there is a solution to this problem offered [5]. With a slightly heavier calculation, we get a better distribution. We add the method GetBetterPointDistribution. We pass in each vertex to GetBetterPointDistribution and add the returned vertex to vertices instead.

```
private static Vector3 GetBetterPointDistribution(Vector3 p)
{
	float x2 = p.x * p.x;
	float y2 = p.y * p.y;
	float z2 = p.z * p.z;
	float x1 = p.x * Mathf.Sqrt(1 - ( y2 + z2 ) / 2 + ( y2 * z2 ) / 3);
	float y1 = p.y * Mathf.Sqrt(1 - ( z2 + x2 ) / 2 + ( z2 * x2 ) / 3);
	float z1 = p.z * Mathf.Sqrt(1 - ( x2 + y2 ) / 2 + ( x2 * y2 ) / 3);
	return new Vector3(x1, y1, z1);
}


// before //
vertices.Add(vPosition.normalized * radius);
texCoords.Add(CalculateTexCoords(vPosition));

// after //

const Vector3 betterPoint = GetBetterPoint(vPosition);
vertices.Add(betterPoint.normalized * radius);
texCoords.Add(CalculateTexCoords(betterPoint));
```

We add a Material with a grid texture to see the changes to the spheres.

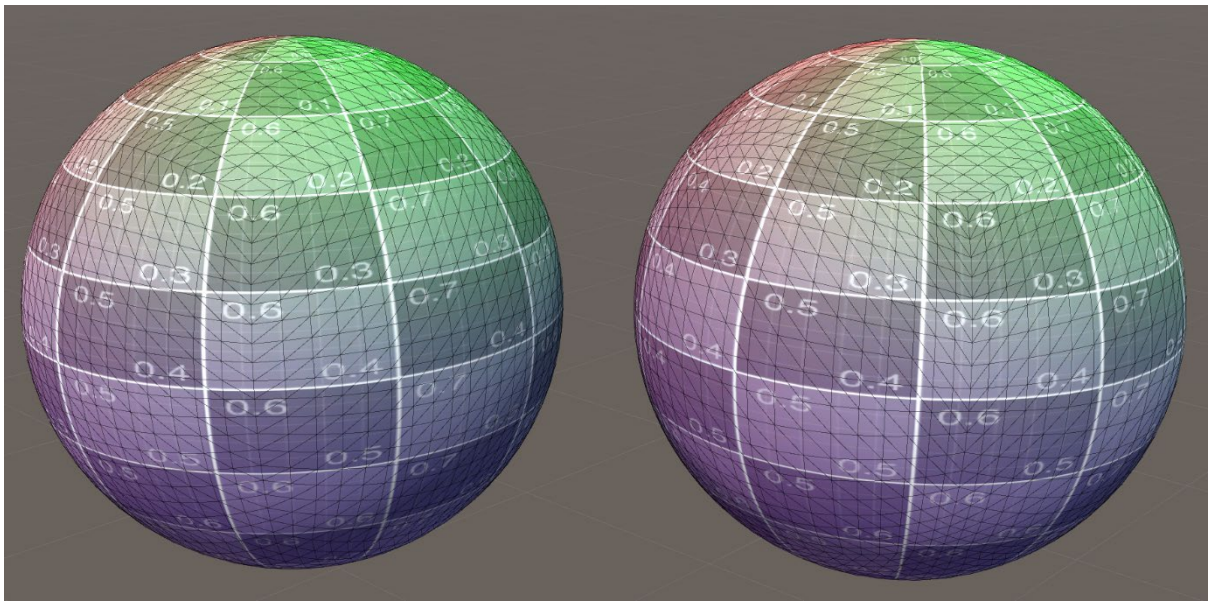


Figure 7. Cube-Sphere with points redistributed (left), Cube-Sphere with normal distribution (right). Both Cube-Spheres with a radius of 2 and Resolution 20.

Making a single-mesh Cube-Sphere

In contrast to the previous sphere, for this one, as it is quite a lot less straightforward, we will be following a guide made by [7] Jasper Flick, that we have modified to make it more focused. The Guide by Jasper Flick is made for a Rounded Box of any dimensions; however, we have simplified it to be just for cubes.

Implementation

We start much like the previous shape by creating a new C# script, this one will only hold one class, as this sphere is composed only of one mesh, there is no need to make an entire class for the mesh, and another to hold it. We start by declaring our settings, radius, resolution, MeshFilter and Mesh. We

also make the needed containers, vertex, uv and indices containers. One thing to note with this shape is that it only works with a resolution of multiples of two.

```
[Range(1, 52)]
private int resolution;

[Range(1f, 100f)]
public float radius = 10f;

MeshFilter meshFilter;
Mesh mesh;
List<Vector3> vertices = new List<Vector3>();
List<Vector2> texCoords = new List<Vector2>();
List<int> indices = new List<int>();
```

Similarly, to the previous shape we create two methods, one to initialize the needed values, and a second one to generate the mesh when we need it to. We start by checking if our MeshFilter already exist, in the case it does not we create a new game object, add a MeshRenderer to it and a MeshFilter which we store a new Mesh in. The second method is much like the previous SideMesh class, but more contained. We create new containers for vertices, uvs and indices, clear up the mesh we just created in case it holds old information. We create the needed vertices, uvs and indices and assign them to the mesh.

```
private void Initialize()
{
	if (meshFilter == null)
	{
		meshFilter = new MeshFilter();

		GameObject gameObject = new GameObject("CubeSphere");
		gameObject.transform.parent = transform;

		gameObject.AddComponent<MeshRenderer>();
		gameObject.GetComponent<MeshRenderer>().sharedMaterial = new Material(Shader.Find("Standard"));
		meshFilter = gameObject.AddComponent<MeshFilter>();
		meshFilter.sharedMesh = mesh = new Mesh();
	}
}

private void CreateSphere()
{
	vertices = new List<Vector3>();
	texCoords = new List<Vector2>();
	indices = new List<int>();

	if (mesh == null)
		return;

	mesh.Clear();

	CreateVertices();
	CreateTriangles();

	mesh.vertices = vertices.ToArray();
	mesh.uv = texCoords.ToArray();
	mesh.triangles = indices.ToArray();
}
```

We have made simplifications to [7] the guides process and distilled it to this. We utilize the familiar method to calculate texture coordinates.

```
private void CreateVertices()
{
	float offset = (resolution / 2f);
	float gap = 1f;
	// vertical sides
	for (float y = -offset; y <= offset; y += gap)
	{
		for (float x = -offset; x <= offset; x += gap)
		{
			vertices.Add(new Vector3(x, y, -offset));
		}
		for (float z = -offset + 1; z <= offset; z += gap)
		{
			vertices.Add(new Vector3(offset, y, z));
		}
		for (float x = offset - 1; x >= -offset; x -= gap)
		{
			vertices.Add(new Vector3(x, y, offset));
		}
		for (float z = offset - 1; z > -offset; z -= gap)
		{
			vertices.Add(new Vector3(-offset, y, z));
		}
	}
	//top
	for (float z = -offset + 1; z < offset; z += gap)
	{
		for (float x = -offset + 1; x < offset; x += gap)
		{
			vertices.Add(new Vector3(x, offset, z));
		}
	}
	// botom
	for (float z = -offset + 1; z < offset; z += gap)
	{
		for (float x = -offset + 1; x < offset; x += gap)
		{
			vertices.Add(new Vector3(x, -offset, z));
		}
	}

	for (int i = 0; i < vertices.Count; ++i)
	{
		vertices[i] = vertices[i].normalized * radius;
		texCoords.Add(CalculateTexCoords(vertices[i]));

	}
}
```

We have chosen to normalize our cube after all vertices have been assigned, which is a good point to introduce our radius as well. We now continue to the next method, concerning the triangulation of the generated vertices. We start by making our assignment method to simplify our code.

```
private int SetQuad(int i, int a, int b, int c, int d)
{
	indices.Add(a);
	indices.Add(c);
	indices.Add(b);
	indices.Add(b);
	indices.Add(c);
	indices.Add(d);
	return i + 6;
}
```

After which we start the main method of triangulation. We begin by calculating how many points will make up the outer boarder of a segment, a ring if you prefer. As per [7] Jasper Flick, it will look something like this.

```
private void CreateTriangles()
{
	int ring = (resolution * 2) * 2;
	int t = 0, v = 0;
	for (int y = 0; y < resolution; y++, v++)
	{
		for (int q = 0; q < ring - 1; q++, v++)
		{
			t = SetQuad(t, v, v + 1, v + ring, v + ring + 1);
		}
		t = SetQuad(t, v, v - ring + 1, v + ring, v + 1);
	}
	t = CreateTopFace(t, ring);
	t = CreateBottomFace(t, ring);
}
```

We continue with creating the top and bottom faces.

```
private int CreateTopFace(int t, int ring)
{
	int v = ring * resolution;
	for (int x = 0; x < resolution - 1; x++, v++)
	{
		t = SetQuad(t, v, v + 1, v + ring - 1, v + ring);
	}
	t = SetQuad(t, v, v + 1, v + ring - 1, v + 2);

	int vMin = ring * (resolution + 1) - 1;
	int vMid = vMin + 1;
	int vMax = v + 2;

	for (int z = 1; z < resolution - 1; z++, vMin--, vMid++, vMax++)
	{
		t = SetQuad(t, vMin, vMid, vMin - 1, vMid + resolution - 1);
		for (int x = 1; x < resolution - 1; x++, vMid++)
		{
			t = SetQuad(t, vMid, vMid + 1, vMid + resolution - 1, vMid + resolution);
		}
		t = SetQuad(t, vMid, vMax, vMid + resolution - 1, vMax + 1);
	}
	int vTop = vMin - 2;
	t = SetQuad(t, vMin, vMid, vTop + 1, vTop);
	for (int x = 1; x < resolution - 1; x++, vTop--, vMid++)
	{
		t = SetQuad(t, vMid, vMid + 1, vTop, vTop - 1);
	}
	t = SetQuad(t, vMid, vTop - 2, vTop, vTop - 1);
	return t;
}
```

```
private int CreateBottomFace(int t, int ring)
{
	int v = 1;
	int vMid = vertices.Count - (resolution - 1) * (resolution - 1);
	t = SetQuad(t, ring - 1, vMid, 0, 1);
	for (int x = 1; x < resolution - 1; x++, v++, vMid++)
	{
		t = SetQuad(t, vMid, vMid + 1, v, v + 1);
	}
	t = SetQuad(t, vMid, v + 2, v, v + 1);

	int vMin = ring - 2;
	vMid -= resolution - 2;
	int vMax = v + 2;

	for (int z = 1; z < resolution - 1; z++, vMin--, vMid++, vMax++)
	{
		t = SetQuad(t, vMin, vMid + resolution - 1, vMin + 1, vMid);
		for (int x = 1; x < resolution - 1; x++, vMid++)
		{
			t = SetQuad(t, vMid + resolution - 1, vMid + resolution, vMid, vMid + 1);
		}
		t = SetQuad(t, vMid + resolution - 1, vMax + 1, vMid, vMax);
	}

	int vTop = vMin - 1;
	t = SetQuad(t, vTop + 1, vTop, vTop + 2, vMid);
	for (int x = 1; x < resolution - 1; x++, vTop--, vMid++)
	{
		t = SetQuad(t, vTop, vTop - 1, vMid, vMid + 1);
	}
	t = SetQuad(t, vTop, vTop - 1, vMid, vTop - 2);

	return t;
}
```

If we assign this C# script to an empty game object, we can now generate a sphere consisting of one mesh. With this mesh, when we generate vertices we have the opportunity to apply the slightly altered formula to more evenly distribute the vertices on this sphere. We update our method accordingly. GetBetterPoint need a slight adjustment for this sphere.

```
public static Vector3 GetBeterPoint( Vector3 p, int resolution )
{
	Vector3 v = new Vector3(p.x, p.y, p.z) * 2f / resolution;
	float x2 = v.x * v.x;
	float y2 = v.y * v.y;
	float z2 = v.z * v.z;

	float x1 = v.x * Mathf.Sqrt(1f - y2 / 2f - z2 / 2f + y2 * z2 / 3f);
	float y1 = v.y * Mathf.Sqrt(1f - z2 / 2f - x2 / 2f + z2 * x2 / 3f);
	float z1 = v.z * Mathf.Sqrt(1f - x2 / 2f - y2 / 2f + x2 * y2 / 3f);

	return new Vector3(x1, y1, z1);
}


// before
Vertices[i] = vertices[i].normalized * radius;
texCoords.Add(CalculateTexCoords(vertices[i]));
// new
Vertices[i] = GetBetterPoint(vertices[i], resolution).normalized * radius;
texCoords.Add(CalculateTexCoords(vertices[i]));
```

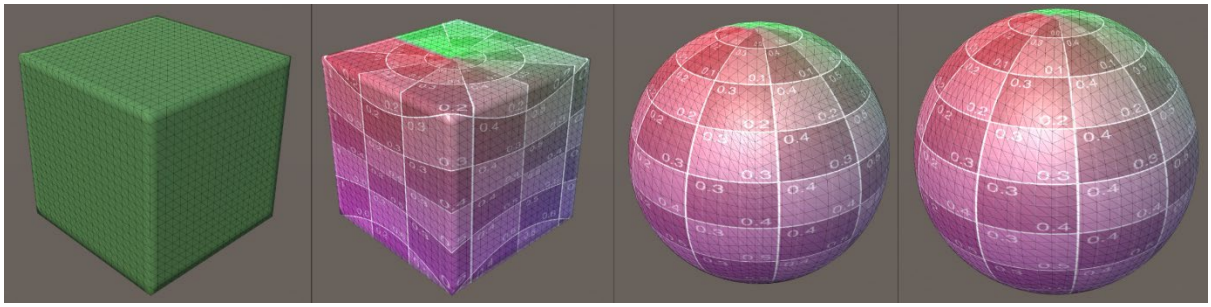


Figure 8. From left to right: (1) Initial vertices. (2) with a material after calculating UV. (3) After normalizing the vertex positions. (4) Redistributing the points on the sphere.

With this we can now generate a sphere much like the previous multi-mesh sphere, but only using one mesh. We apply a UV Grid Material to inspect our results.

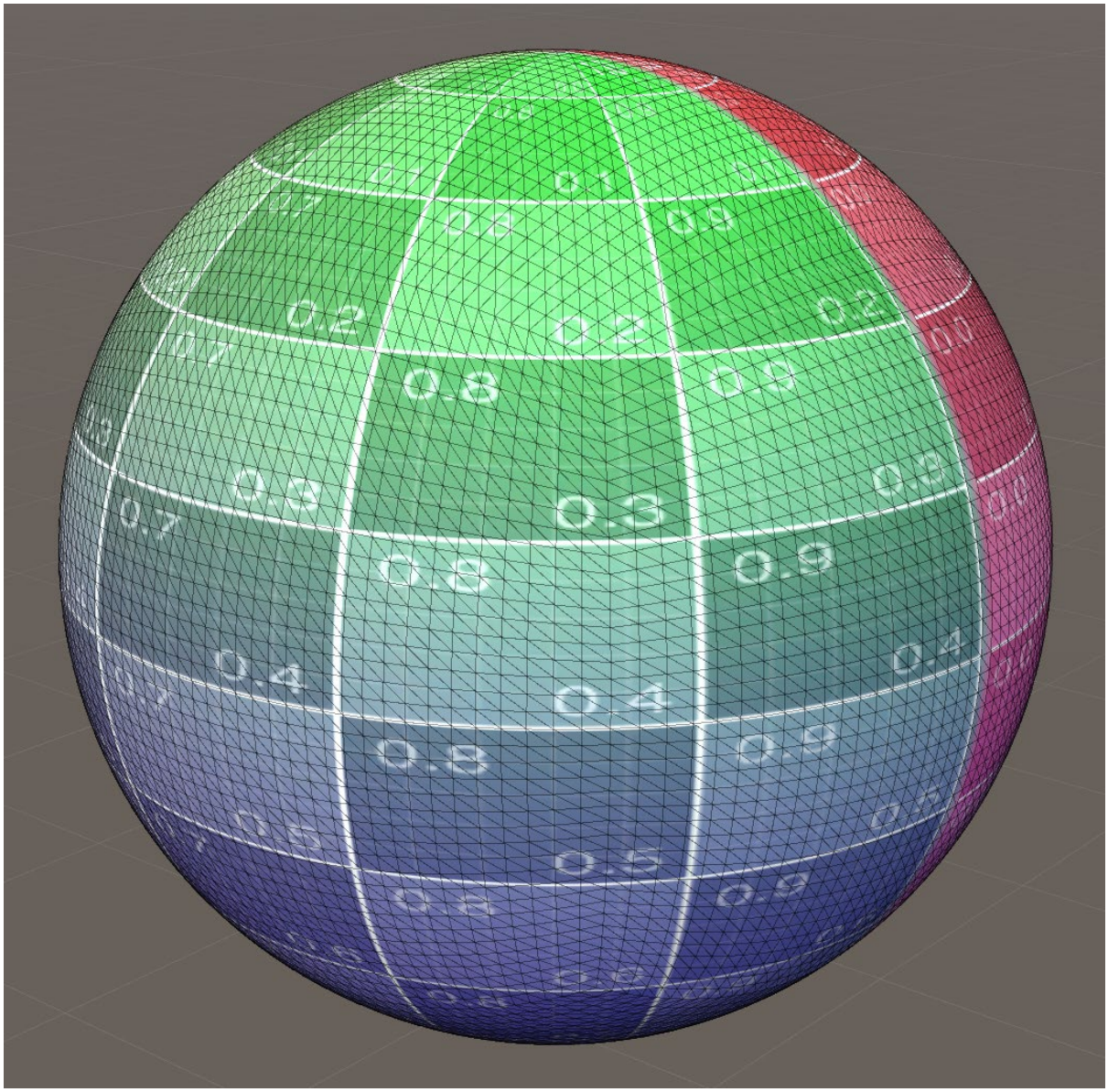


Figure 9. Single mesh Cube-Sphere with points redistributed, a radius of 10 and Resolution 20.

Making a spherified Icosahedron (IcoSphere)

The icosahedron is the Platonic solid with the most faces. It consists of twenty triangles that are identical in shape and size. It is this aspect that makes this shape interesting. It is true that the triangle faces making up the icosahedron stays congruent when we subdivide them. This aspect is however lost when we move each point an equal distance away from the shapes centre. It is lost, but the shape and size difference are so small that we can practically ignore it in this case, which will leave us with a shape that has all its vertices evenly distributed along its surface. There are multiple ways of generating the points that make up the icosphere, in this case we will implement a straightforward way of calculating each subdivision, this implementation lends itself well to separating the faces and handling them independently, similar to how the multi-mesh cube-sphere was structured.

[8] Andreas Kahler, outlines in his blogpost how one can go about creating this shape. We have adjusted and simplified the implementation to better suit our needs.

Implementation

Start by creating a new C# script. This script will hold two classes, the first class holds the settings, and a container of the meshes, in the same manner as the Multi-Mesh Cube-Sphere. The settings we need is the initial vertices, the initial indices, the subdivision level, the radius, the container for the TriangleFaces and some constants to help us describe the initial vertex positions.

```
[Range(0, 8)]
public int recursionLevel = 0;

[Range(1f, 100f)]
public float radius = 10f;

[SerializeField, HideInInspector]
MeshFilter[] meshFilters;
TriangleFace[] triFaces;

int maxFaces = 20;

private static float goldenRatio = (1f + Mathf.Sqrt(5f)) / 2f;
private static Vector3 diraction = new Vector3(1, goldenRatio, 0).normalized;
```

The array of the initial vertices and indices can be described like this.

```
Vector3[] vertices = new Vector3[] {
new Vector3( -diraction.x,  diraction.y, 0),
new Vector3(  diraction.x,  diraction.y, 0),
new Vector3( -diraction.x, -diraction.y, 0),
new Vector3(  diraction.x, -diraction.y, 0),

new Vector3(  0, -diraction.x, diraction.y),
new Vector3(  0,  diraction.x, diraction.y),
new Vector3(  0, -diraction.x,-diraction.y),
new Vector3(  0,  diraction.x,-diraction.y),

new Vector3(  diraction.y, 0, -diraction.x),
new Vector3(  diraction.y, 0,  diraction.x),
new Vector3( -diraction.y, 0, -diraction.x),
new Vector3( -diraction.y, 0,  diraction.x)
                                    };
```

```
int[] indices = new int[] {
                0, 11, 5,
                0, 5 , 1,
                0, 1 , 7,
                0, 7 , 10,
                0, 10, 11,

                1 , 5 , 9,
                5 , 11, 4,
                11, 10, 2,
                10, 7 , 6,
                7 , 1 , 8,

                3, 9, 4,
                3, 4, 2,
                3, 2, 6,
                3, 6, 8,
                3, 8, 9,

                4, 9, 5,
                2, 4, 11,
                6, 2, 10,
                8, 6, 7,
                9, 8, 1
                };
```

With this setup we Initialize each TriangleFace with the necessary information, this method is similar to the structure of the previous shapes, it differs mainly in what information needs to be sent to the TriangleFaces. The TriangleFace constructor takes in a Mesh, an array of vertices, the relevant indices, and how many times we want to subdivide the base shape.

```
void Initialize()
{
	if( meshFilters == null || meshFilters.Length == 0 )
	{
		meshFilters = new MeshFilter[maxFaces];
	}
	triFaces = new TriangleFace[maxFaces];

	for( int i = 0; i < maxFaces; ++i )
	{
		if( meshFilters[i] == null )
		{
			GameObject gameObject = new GameObject("TriangleFace");
			gameObject.transform.parent = transform;

			gameObject.AddComponent<MeshRenderer>();
			gameObject.GetComponent<MeshRenderer>().sharedMaterial = new Material(Shader.Find("Standard"));
			meshFilters[i] = gameObject.AddComponent<MeshFilter>();
			meshFilters[i].sharedMesh = new Mesh();
		}
		int[] idx = new int[3]{ indices[i * 3], indices[(i * 3) + 1], indices[(i * 3) + 2] };

		triFaces[i] = new TriangleFace(meshFilters[i].sharedMesh, vertices, idx, recursionLevel);
	}
}
```

In the same way as with the other shapes we make a CreateSphere that can be called when needed.

```
void CreateSphere()
{
	foreach( TriangleFace face in triFaces )
	{
		face.CreateFaces();
	}
}
```

For the TriangleFace, we can re-use the structure of SideMesh. We remove the vectors we don't need and pick out the vertices we need before storing them in the TriangleFace's container. We also initialize the index list with hardcoded values '0, 1, 2', as that is in the order we store the vertices.

```
Mesh mesh;
int resolution; //subdivisions
List<Vector3> vertices = new List<Vector3>();
List<Vector2> texCoords = new List<Vector2>();
List<int> indices = new List<int>();

public TriangleFace( Mesh mesh, Vector3[] verts, int[] indis, int resolution)
{
	this.mesh = mesh;
	this.resolution = resolution;
	vertices = new List<Vector3> { verts[indis[0]], verts[indis[1]], verts[indis[2]] };
	indices = new List<int> { 0, 1, 2 };

	this.mesh.Clear();
}
```

The next step is to complete the CreateFace method. The TriangleFace's CrateFace method consists of multiple repeated steps. The way we subdivide this TriangleFace is to find the mid-point between each vertex that makes up the triangle and create a new vertex at that position. This is done for every edge segment. When the new vertices are created the old indices order is wrong. We re-triangulate the vertices by selecting the vert-index from each point of the original TriangleFace and add the nearest mid-points as the following indices, this makes up the outer triangles. Lastly to construct the centre triangle we just add the recently created mid-points in the order they were created. After all new indices have been added, we assign the list to the TriangleFace's indices container and repeat the previous steps for each subdivision. When finished, store the vertices in TriangleFace's vertex container, then assign the vertices and indices to the mesh.

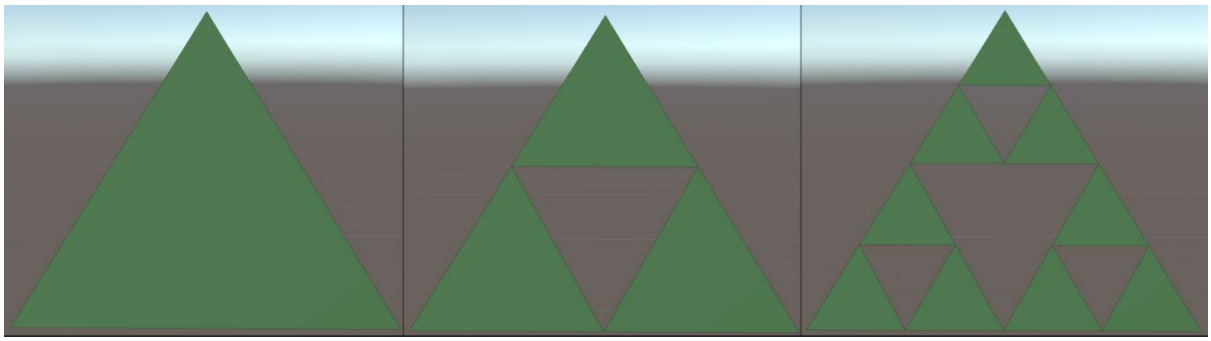


Figure 10. Illustration of the triangles that are made from the newly created mid-points with the original points.

```
private Vector3 GetMidPoint(Vector3 v1, Vector3 v2 )
{
	return Vector3.Lerp(v1, v2, 0.5f);
}

public void CreateFaces()
{
	List<Vector3> newVerts = vertices;
	for (int l = 0; l < resolution; ++l )
	{
		List<int> newIndices = new List<int>();
		for (int i = 0; i < indices.Count; i += 3 )
		{
			newVerts.Add(GetMidPoint(newVerts[indices[i]], newVerts[indices[i + 1]]));
			newVerts.Add(GetMidPoint(newVerts[indices[i + 1]], newVerts[indices[i + 2]]));
			newVerts.Add(GetMidPoint(newVerts[indices[i + 2]], newVerts[indices[i]]));


			newIndices.Add(indices[i]);
			newIndices.Add(newVerts.Count - 3);
			newIndices.Add(newVerts.Count - 1);

			newIndices.Add(indices[i + 1]);
			newIndices.Add(newVerts.Count - 2);
			newIndices.Add(newVerts.Count - 3);

			newIndices.Add(indices[i + 2]);
			newIndices.Add(newVerts.Count - 1);
			newIndices.Add(newVerts.Count - 2);

			newIndices.Add(newVerts.Count - 3);
			newIndices.Add(newVerts.Count - 2);
			newIndices.Add(newVerts.Count - 1);
		}
		indices = newIndices;
	}

	vertices = newVerts;

	mesh.vertices = vertices.ToArray();
	mesh.triangles = indices.ToArray();
}
```

When we add this script to an empty game object and call the necessary methods, we are presented with an icosphere, it can be subdivided as many times as needed. The number of vertices increase quite drastically from one subdivision to the next. We have limited the range from 0 to 8. The number of vertices grow at a rate of $10 \cdot 4^{n-1} + 2$, after only 10 subdivisions the Icosahedron consist of over 2.6 million vertices. Also keep in mind that the method used in this script has a duplicate of the vertices along each edge, which only adds to the issue.

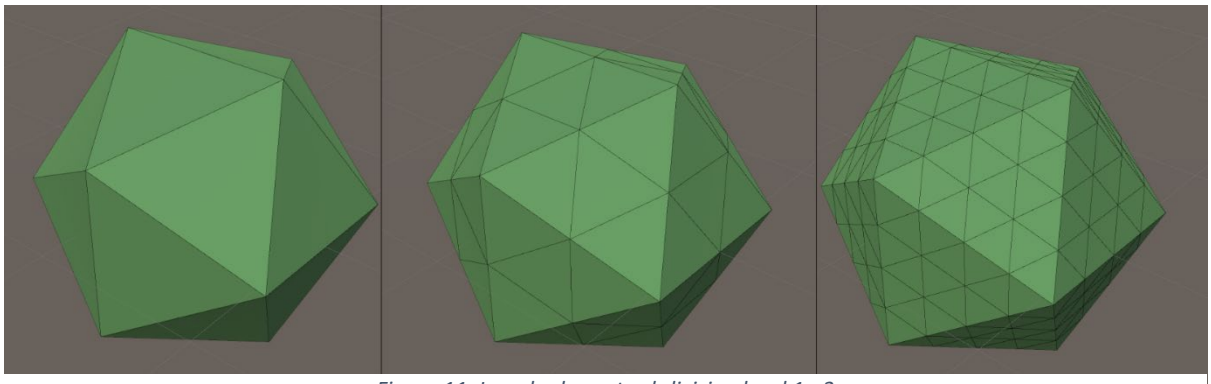


Figure 11. Icosahedron at subdivision level 1 - 3

To make this into a sphere we update the method by normalizing the result of GetMidPoint and multiplying it with the radius. At the same time, we can calculate the uv-coordinates of each point with the same method we've been using for this so far, CalculateTexCoords(). We must add the original three vertices uv-coordinates before the subdivision, this can be done at any point before the icosphere is subdivided.

```
// Before
newVerts.Add(GetMidPoint(newVerts[indices[i]], newVerts[indices[i + 1]]).normalized * radius);
newVerts.Add(GetMidPoint(newVerts[indices[i + 1]], newVerts[indices[i + 2]]));
newVerts.Add(GetMidPoint(newVerts[indices[i + 2]], newVerts[indices[i]]));

// After
newVerts.Add(GetMidPoint(newVerts[indices[i]], newVerts[indices[i + 1]]).normalized * radius);
texCoords.Add(CalculateTexCoords(newVerts[newVerts.Count - 1]));

newVerts.Add(GetMidPoint(newVerts[indices[i + 1]], newVerts[indices[i + 2]]).normalized * radius);
texCoords.Add(CalculateTexCoords(newVerts[newVerts.Count - 1]));

newVerts.Add(GetMidPoint(newVerts[indices[i + 2]], newVerts[indices[i]]).normalized * radius);
texCoords.Add(CalculateTexCoords(newVerts[newVerts.Count - 1]));

// also assign the uv-coordinates to the mesh
mesh.uv = texCoords.ToArray();
```

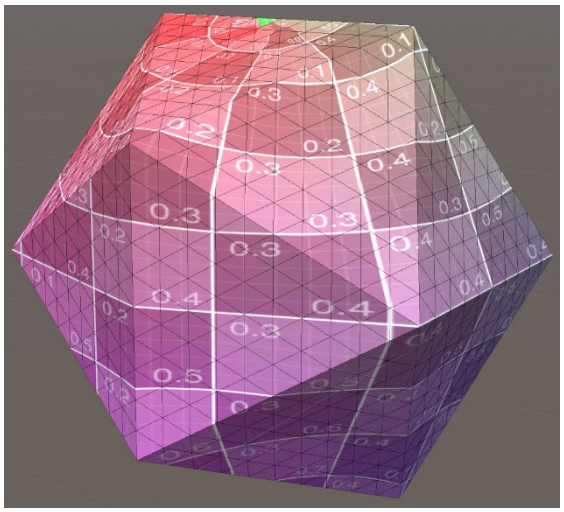


Figure 12. Icosphere subdivided 3 times. Only adding the UVs.

After adding this, UV Grid material has been added to the meshes to inspect if the UVs are correct.

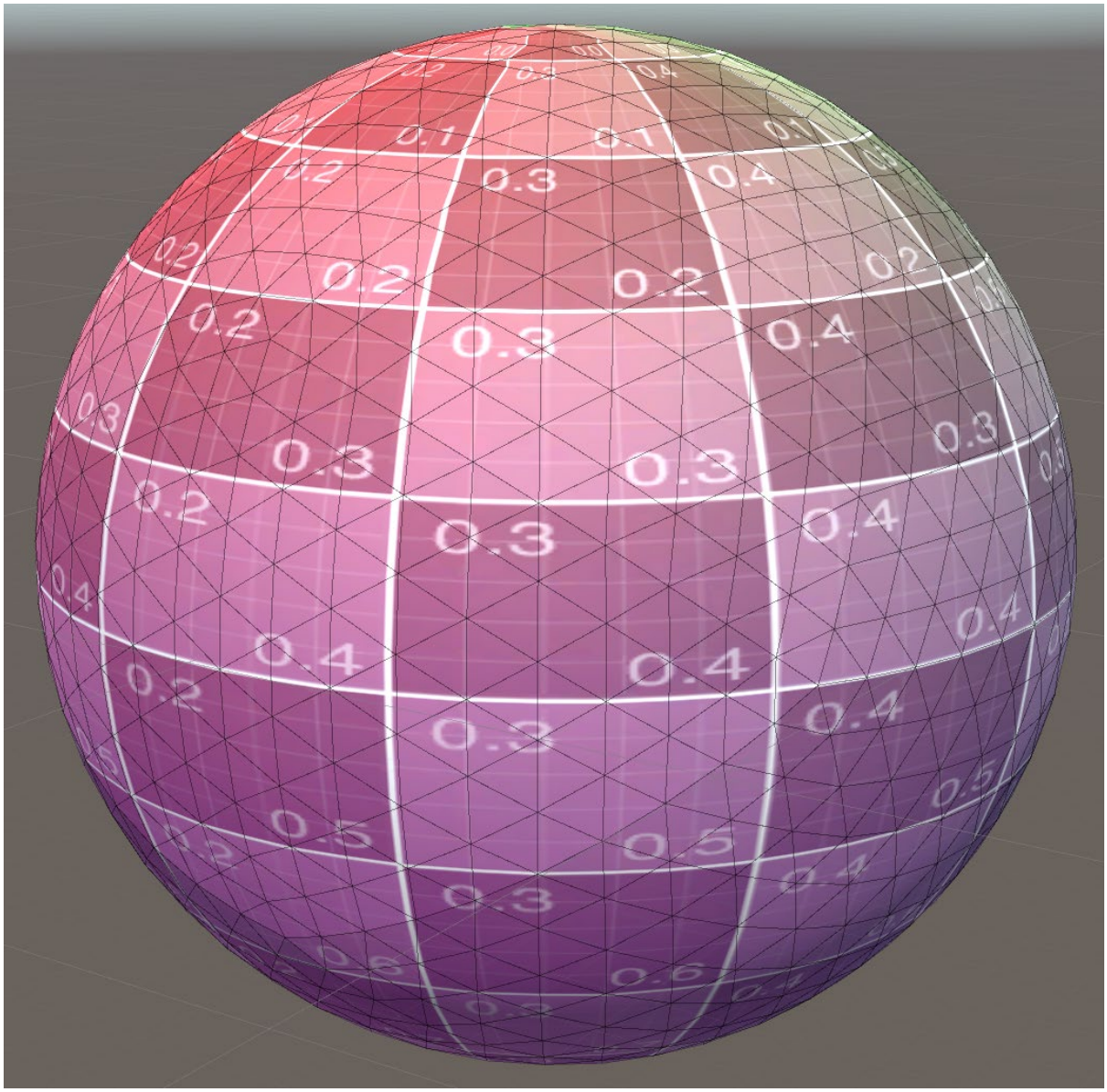


Figure 13. Icosphere subdivided 3 times. Radius 10. UV Grid-texture

Making a IcoSphere - Alternative Method

There is an alternative method for making an icosphere. This alternative method grants a lot more control over how many vertices should be generated per TriangleFace. A drawback with this is that the CreateFace method is a lot more involved. It is a relatively long process, which we will go through step by step.

Implementation

Start by creating a new C# script. This script will hold two classes, the first class holds the settings, and a container of the meshes, in the same manner as the Multi-Mesh Cube-Sphere. The settings we need is the initial vertices, the initial indices, the subdivision level, the radius, the container for the IcoFaces and some constants to help us describe the initial vertex positions. The setup for this is nearly identical to the previous shape.

```
[Range(0, 300)]
public int edgeVertexCount = 0;

[Range(1f, 100f)]
public float radius = 10f;

[SerializeField, HideInInspector]
MeshFilter[] meshFilters;
IcoFace[] icoFaces;

int maxFaces = 20;

private static float goldenRatio = ( 1f + Mathf.Sqrt(5f) ) / 2f;
private static Vector3 diraction = new Vector3(1, goldenRatio, 0).normalized;
```

Like before we also need the array of the initial vertices and indices.

```
Vector3[] vertices = new Vector3[] {
new Vector3( -diraction.x,  diraction.y, 0),
new Vector3(  diraction.x,  diraction.y, 0),
new Vector3( -diraction.x, -diraction.y, 0),
new Vector3(  diraction.x, -diraction.y, 0),

new Vector3(  0, -diraction.x, diraction.y),
new Vector3(  0,  diraction.x, diraction.y),
new Vector3(  0, -diraction.x,-diraction.y),
new Vector3(  0,  diraction.x,-diraction.y),

new Vector3(  diraction.y, 0, -diraction.x),
new Vector3(  diraction.y, 0,  diraction.x),
new Vector3( -diraction.y, 0, -diraction.x),
new Vector3( -diraction.y, 0,  diraction.x)
                                  };
```

```
int[] indices = new int[] {
                0, 11, 5,
                0, 5 , 1,
                0, 1 , 7,
                0, 7 , 10,
                0, 10, 11,

                1 , 5 , 9,
                5 , 11, 4,
                11, 10, 2,
                10, 7 , 6,
                7 , 1 , 8,

                3, 9, 4,
                3, 4, 2,
                3, 2, 6,
                3, 6, 8,
                3, 8, 9,

                4, 9, 5,
                2, 4, 11,
                6, 2, 10,
                8, 6, 7,
                9, 8, 1
                };
```

The Initialize method is quite similar as well, however we can omit passing in any indices by setting up the vertex list before passing it the IcoFace's constructor.

```
private void Initialize()
{
	if( meshFilters == null || meshFilters.Length == 0 )
	{
		meshFilters = new MeshFilter[maxFaces];
	}
	icoFaces = new IcoFace[maxFaces];

	for( int i = 0; i < maxFaces; ++i )
	{
		if( meshFilters[i] == null )
		{
			GameObject gameObject = new GameObject("TriFace");
			gameObject.transform.parent = transform;

			gameObject.AddComponent<MeshRenderer>();
			gameObject.GetComponent<MeshRenderer>().sharedMaterial = new Material(Shader.Find("Standard"));
			meshFilters[i] = gameObject.AddComponent<MeshFilter>();
			meshFilters[i].sharedMesh = new Mesh();
		}

		List<Vector3> verts = new List<Vector3>{ vertices[indices[i * 3]],
		                                         vertices[indices[(i * 3) + 1]],
		                                         vertices[indices[(i * 3) + 2]]};
		icoFaces[i] = new IcoFace(meshFilters[i].sharedMesh, verts, edgeVertexCount);
	}
}
```

In the same way as with the other shapes we make a CreateSphere that can be called when needed.

```
void CreateSphere()
{
	foreach( IcoFace face in icoFaces )
	{
		face.CreateFaces();
	}
}
```

For the IcoFace, the structure follows a similar pattern as the SideMesh and TriangleMesh. Store the variables that are passed in. Calculate the total amount of expected vertices, the calculation looks like this: $((\mathrm{N}+3)^2-(\mathrm{N}+3))/2$ where N is the number of vertices along one edge.

```
Mesh mesh;
int edgeVertexCount; // resolution
List<Vector3> vertices = new List<Vector3>();
List<Vector2> texCoords = new List<Vector2>();
List<int> indices = new List<int>();
int maxNumVerts = 0;

public IcoFace( Mesh mesh, List<Vector3> vertices, int edgeVertexCount )
{
	this.mesh = mesh;
	this.edgeVertexCount = edgeVertexCount;
	this.vertices = vertices;

	this.maxNumVerts = ((edgeVertexCount+3)*(edgeVertexCount+3)-(edgeVertexCount+3))/2;
	this.mesh.Clear();
}
```

The next step is to complete the CreateFace method. The IcoFace's CrateFace method consists of multiple steps. We will explain as we go this time instead of before. We start out at the very first by creating a list of integer arrays, this list is representative of the edges of the triangle, the arrays hold the index of the vertices that will be along that edge. Like the previous shape we must remember to add the original three vertices uv-coordinates before making any new vertices, this can be done at any point before the first new vertex is added to the vertex-container.

```
public void CreateFaces()
{
	List<int[]> edges = new List<int[]>{ new int[edgeVertexCount + 2],
	                                     new int[edgeVertexCount + 2],
	                                     new int[edgeVertexCount + 2] };
// the method continues
```

Next loop over each edge, sett the start and end point of the current edge and store the index of the starting edge. We add each vertex along the edge with a loop, we determine how far along the edge we are by linearly interpolating between the start and end vertex with an amount, starting from the 2nd vertex along that edge, skipping the already existing starting and end vertex. After calculating the new point, add it to the vertices buffer, calculate uvs and sett the index for the newly created vertex. Repeat this until the end of the edge, then finish the loop by setting the end point index.

```
for( int v = 0; v < 3; ++v )
{
	Vector3 start = vertices[v];
	Vector3 end = vertices[( v + 1 ) % 3];

	edges[v][0] = v;

	for( int divisionIndex = 0; divisionIndex < edgeVertexCount; divisionIndex++ )
	{
		float t = ( divisionIndex + 1f ) / ( edgeVertexCount + 1f );
		Vector3 newVec = Vector3.Lerp(start, end, t);

		vertices.Add(newVec);
		texCoords.Add(CalculateTexCoords(newUV));

		edges[v][divisionIndex + 1] = vertices.Count - 1;
	}
	edges[v][edgeVertexCount + 1] = ( v + 1 ) % 3;
}

// the method continues
```

With this we have all outer vertices along the triangle's edges. Now we have the information needed to create the rest of the interior vertices of the triangle. We create a container that can hold the indices for the vertices we will create. We set the first index to be "0" as that is the vertex we will start with, the "top" vertex of the triangle. As a side effect of calculating the edges this way, the last edge had been created backwards, we correct that now. Next, we calculate where to place the vertices going across the triangle, we start the loop by adding the current point along the edge to the index-container, store the position of the start and end. This will be the same point along an edge, but on opposing edges. We store how many points we expect to generate between these two points. We now create the vertices and store the necessary information in a similar manner as done for the edges. To finish off we add the bottom edge's indices to the index-container.

```
int[] verteMap = new int[maxNumVerts];
int indexer = 0;
verteMap[indexer++] = 0;
System.Array.Reverse(edges[2]);

for( int i = 1; i < edges[0].Length - 1; ++i )
{
	verteMap[indexer++] = edges[0][i];

	Vector3 sideAVertex = vertices[edges[0][i]];
	Vector3 sideBVertex = vertices[edges[2][i]];

	int numInnerPoints = i - 1;
	for( int j = 0; j < numInnerPoints; ++j )
	{
		float t = ( j + 1f ) / ( numInnerPoints + 1f );

		Vector3 newVec = Vector3.Lerp(sideAVertex, sideBVertex, t);
		vertices.Add(newVec);
		texCoords.Add(CalculateTexCoords(newUV));

		verteMap[indexer++] = vertices.Count - 1;
	}
	verteMap[indexer++] = edges[2][i];
}

for( int i = 0; i < edges[1].Length; ++i )
{
	verteMap[indexer++] = edges[1][i];
}

// the method continues
```

It is now possible to triangulate the mesh and afterwards store this information in the mesh.

```
int numRovs = edgeVertexCount + 1;
	for( int row = 0; row < numRovs; ++row )
	{
		int topVertex = ( ( row + 1 ) * ( row + 1 ) - row - 1 ) / 2;
		int bottomVertex = ( ( row + 2 ) * ( row + 2 ) - row - 2 ) / 2;

		int numTrianglesInRow = 1 + 2 * row;
		for( int column = 0; column < numTrianglesInRow; column++ )
		{
			int v0, v1, v2;

			if( column % 2 == 0 )
			{
				v0 = topVertex;
				v1 = bottomVertex;
				v2 = bottomVertex + 1;
				topVertex++;
				bottomVertex++;
			}
			else
			{
				v0 = topVertex - 1;
				v1 = bottomVertex;
				v2 = topVertex;
			}

			indices.Add(verteMap[v0]);
			indices.Add(verteMap[v1]);
			indices.Add(verteMap[v2]);
		}
	}

	mesh.vertices = vertices.ToArray();
	mesh.uv = texCoords.ToArray();
	mesh.triangles = indices.ToArray();
} // end of method
```

When this script is added to an empty game object and the necessary methods are called, we are presented with another icosphere. With this method one has way more control over how fast the vertex count of the sphere grows. We add a UV Grid-Texture to this mesh as well to inspect it.

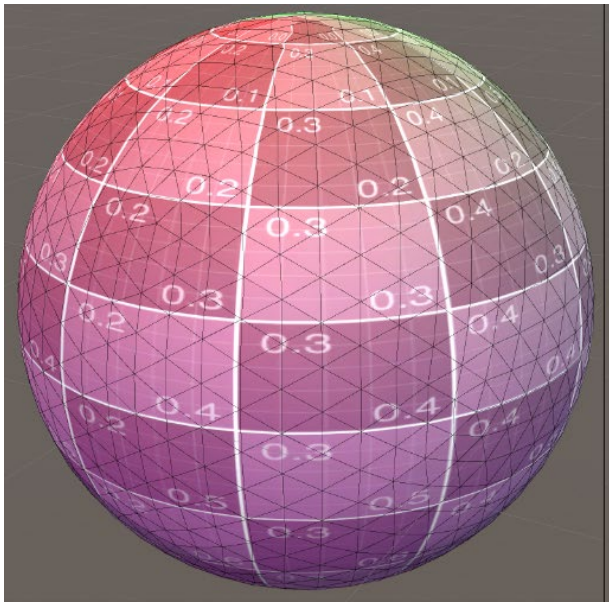


Figure 14. Icosphere subdivided 9 times. Radius 10. UV Grid-texture

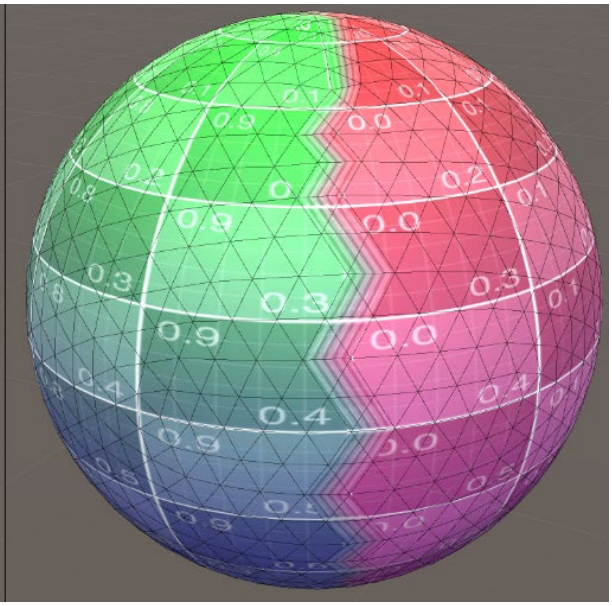


Figure 15. Issues with the uvs

Solving the issues with UVs being mapped incorrectly

Figure 14. shows us one side of the result, but on the opposite side as seen in Figure 15. there are some issues with the UV map. This is addressed by [9] Michael Thygesen in his article addressing this exact problem. The problem comes from parts of a triangle being mapped to one side of the UV space, and one or two of the vertices making up the triangle getting mapped to the entirely opposite side of the UV space. The result is a back-to-front triangle stretching across the entire UV space. This may be an issue for all the spheres that's been shown this far. This can be rectified right before assigning the information to the mesh. We create a method called `RectifyUVs` to do this.

First loop over every triangle UVs to see which are flipped, store the triangles with flipped UV-normals.

```
private void RectifyUVs()
{
	List<int> tempIndices = new List<int>();
	for( int i = 0; i < indices.Count; i += 3 )
	{
		int v1 = indices[i];
		int v2 = indices[i + 1];
		int v3 = indices[i + 2];

		Vector3 A = new Vector3(texCoords[v1].x, texCoords[v1].y, 0);
		Vector3 B = new Vector3(texCoords[v2].x, texCoords[v2].y, 0);
		Vector3 C = new Vector3(texCoords[v3].x, texCoords[v3].y, 0);

		Vector3 texNormal = Vector3.Cross(B - A, C - A);

		if( texNormal.z > 0 )
			tempIndices.Add(i);
	} // method continues
```

Now loop over all triangles that was collected. Check which vertex is out of position, duplicate it, move it back into position and finally add it back into the shapes vertices and indices container.

```
Dictionary<int, int> checkedVert = new Dictionary<int, int>();

	for( int i = 0; i < tempIndices.Count; ++i )
	{
		int a = indices[tempIndices[i]];
		int b = indices[tempIndices[i] + 1];
		int c = indices[tempIndices[i] + 2];

		Vector2 Atex = texCoords[a];
		Vector2 Btex = texCoords[b];
		Vector2 Ctex = texCoords[c];

		if( Atex.x < 0.25f )
		{
			int newI;
			if( !checkedVert.TryGetValue(a, out newI) )
			{
				Atex.x += 1;
				vertices.Add(vertices[a]);
				texCoords.Add(Atex);
				checkedVert[a] = vertices.Count - 1;
				newI = vertices.Count - 1;
			}
			indices[tempIndices[i]] = newI;
		}
		if( Btex.x < 0.25f )
		{
			int newI;
			if( !checkedVert.TryGetValue(b, out newI) )
			{
				Btex.x += 1;
				vertices.Add(vertices[b]);
				texCoords.Add(Btex);
				checkedVert[b] = vertices.Count - 1;
				newI = vertices.Count - 1;
			}
			indices[tempIndices[i] + 1] = newI;
		}
		if( Ctex.x < 0.25f )
		{
			int newI;
			if( !checkedVert.TryGetValue(c, out newI) )
			{
				Ctex.x += 1;
				vertices.Add(vertices[c]);
				texCoords.Add(Ctex);
				checkedVert[c] = vertices.Count - 1;
				newI = vertices.Count - 1;
			}
			indices[tempIndices[i] + 2] = newI;
		}
	}
}
```

Call RectifyUvs before finishing up CreateFaces methods.

```
RectifyUVs(); // before assigning anything to the mesh
mesh.vertices = vertices.ToArray();
mesh.uv = texCoords.ToArray();
mesh.triangles = indices.ToArray();
```

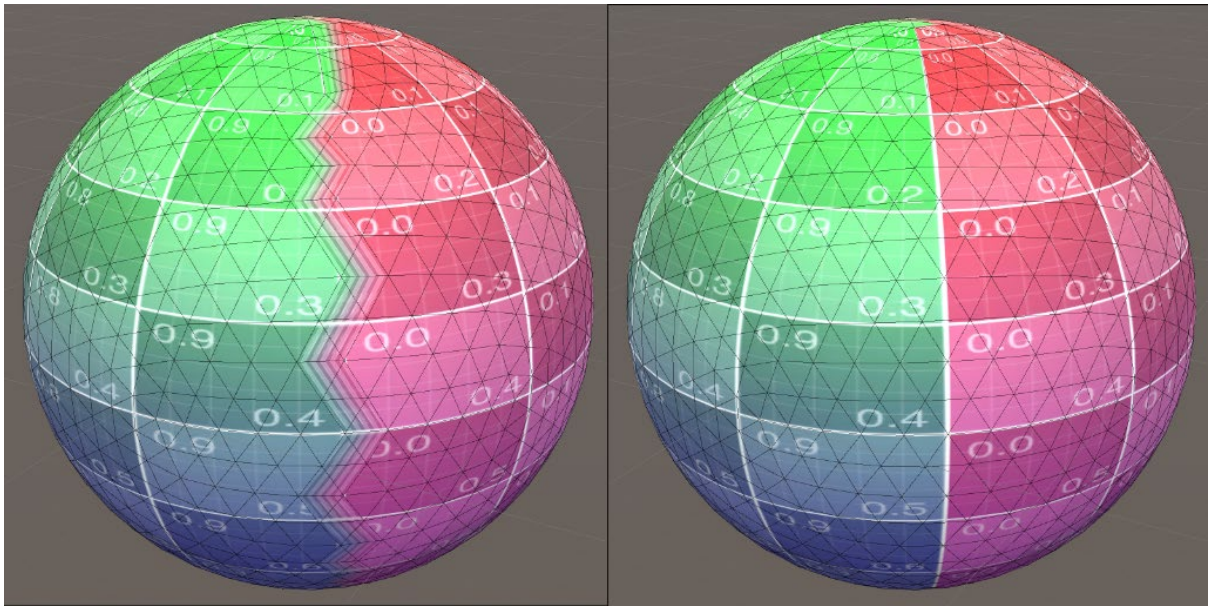


Figure 16. Sphere before fixing UVs (left), Sphere after having fixed UVs (right)

Making of a planet

At this point, different ways of generating a spherical mesh has been presented. With a little more work these spheres can become planets. In this next part we make a Compute Shader that will take in a heightmap and sample values to adjust the position of vertices on the sphere. With this script every sphere should have the possibility to look like Figure 17.

Figure 17. Compute shader used to adjust elevation, Earth Material (glossiness = 0.2).

Icosphere 2nd version, radius 10, subdivision 200, heightStrength 1.0

Setup

First a photo of earth is needed, [10] NASA is kind enough to let us use their photos. We also need a matching height map, a Topographical Map of Earth is good enough at this moment, one can be found on the same website. A Compute Shader script is also needed.

C# in the sphere scripts

For simplicity no other files are generated, changes and settings in this next part is added to already existing sphere scripts. Start by creating the settings we need. These are a ComputeShader, Texture2D for the HightMap and float to adjust the HeightStrenght.

```
public ComputeShader shapeComputeShader;
public Texture2D heightMap;

[Range(0f, 5f)]
public float heightStrenght = 0f;

public bool useHeightMap = false;
```

These settings can be added to all 4 spheres. Make the ComputeHeight method, this method passes and receives the necessary data to and from the ComputShader.

Start by making sure there is a ComputShader assigned to the public ComputeShader variable. Calculate the highest number of vertices across all meshes, this is done due to the vertex duplication, not all meshes has the same number of vertices anymore. This must be taken into account when creating ComputeBuffers. Three separate ComputeBuffers are needed, one to read the vertices from, one to write the new vertices' position to, and one for the UVs. These vertices consist of three floats, and UVs consist of two floats, the buffers strides are 12, 12 and 8 respectively as the size of a float is 4 bytes. Next, we set the variables that does not change in the Compute Shaders. The Heightmap texture, HeightStrenght, radius and a bool useHeightMap for convenience are now assigned.

```
void ComputHeight()
{
	if (shapeComputeShader == null)
		return;

	int maxLength = 0;
	for (int i = 0; i < maxFaces; ++i)
	{
		maxLength = Mathf.Max(maxLength, meshFilters[i].sharedMesh.vertices.Length);
	}

	ComputeBuffer computeVBuffer = new ComputeBuffer(maxLength, 12);
	ComputeBuffer computeHBuffer = new ComputeBuffer(maxLength, 12);
	ComputeBuffer computeUVBuffer = new ComputeBuffer(maxLength, 8);

	shapeComputeShader.SetTexture(0, "heightMap", heightMap);
	shapeComputeShader.SetFloat("heightIntensity", heightStrenght);
	shapeComputeShader.SetFloat("radius", radius);
	shapeComputeShader.SetBool("useHeightMap", useHeightMap);
	// method continues
```

Loop over every mesh, Set the data relevant to each ComputeBuffer, pass the buffers into the ComputeShader, Dispatch the shader. Now Collect the data and assign it to the mesh. After everything is done, we have to remember to release the ComputeBuffers.

```
        for (int i = 0; i < maxFaces; ++i)
        {
                Vector3[] allVerts = sphereFaces[i].GetVerts();
                Vector2[] alluvs = meshFilters[i].sharedMesh.uv;

                shapeComputeShader.SetInt("numVertices", allVerts.Length);

                computeVBuffer.SetData(allVerts);
                computeUVBuffer.SetData(alluvs);

                shapeComputeShader.SetBuffer(0, "vertices", computeVBuffer);
                shapeComputeShader.SetBuffer(0, "heights", computeHBuffer);
                shapeComputeShader.SetBuffer(0, "uvs", computeUVBuffer);

                shapeComputeShader.Dispatch(0, 512, 1, 1);

                computeHBuffer.GetData(allVerts);

                meshFilters[i].sharedMesh.vertices = allVerts;
                meshFilters[i].sharedMesh.RecalculateBounds();
                meshFilters[i].sharedMesh.RecalculateNormals();
                meshFilters[i].sharedMesh.RecalculateTangents();
        }
        computeHBuffer.Release();
        computeVBuffer.Release();
        computeUVBuffer.Release();

}
```

HLSL in for the Compute Shader

When a Compute Shader is made, it has some default code in it. It has a #kernel, a RWTexture2D<float4> Result, 8 by 8 by 1 thread groups, a CSMain that takes in the threadID as a uint3, the function itself writes to the Result-texture.

We reuse some of this code. We remove the already existing texture variable and replace it with all our variables and buffers.

```
// Each #kernel tells which function to compile; you can have many kernels
#pragma kernel CSMain

StructuredBuffer<float3> vertices;
RWStructuredBuffer<float3> heights;
StructuredBuffer<float2> uvs;

Texture2D<float4> heightmap;

float heightIntensity;
float radius;
bool useHeightMap;

uint numVertices;
```

Adjust the thread groups to be 512 by 1 by 1. Like it is in the C# script when the compute shader is dispatched. The uint3 can be changed to uint as we don't need any more. Now we can write the actual code to make this all work.

Start by making sure that the id passed in is not out of range, if it is greater than the number of vertices, return. To make sure that these calculations do not build on itself every time it is called, start by storing a normalized version of the current vertex. Now it is safe to multiply it with the given radius. We can write this into the current id of the height buffer. At this point we have a sphere again. Nothing has changed.

```
[numthreads(512, 1, 1)]
void CSMain( uint id : SV_DispatchThreadID)
{
	if( id > numVertices )
		return;

	const float3 normalizedPosition = normalize(vertices[id]);
	const float3 vertexPos = normalizedPosition * radius;

	heights[id] = vertexPos;
}
```

If the script is run at this point, we should expect nothing to have changed, and to be presented with a perfect-ball Earth. As seen in Figure 18. What we need now is to sample the heightmap and adjust the vertex height accordingly.

Figure 18. Current expected results from the Compute Shader after adding a Earth-texture to the sphere.

The hight calculation is done by sampling the hight-map. As the dimensions of the heightmap is unknown to us, we make a variable to hold the texture dimensions. Call GetDimentions on the height map, where the variable is passed. The texture stores its dimensions in the variable that's passed in. The Dimensions are used to calculate where to sample on the texture. The UV-Buffer that was passed into the compute shader is only in the range from 0 to 1. The pixel to be sampled from the heightmap can be calculated by multiplying the current UVs with the recently acquired texture dimensions. We must be mindful not to move along the x or y axis more than the dimensions of the texture, for safety we modulo the result with the texture dimensions. We can now safely store the correct UV-coordinates. These coordinates are used to directly index into the Heightmap texture, as it is a 2D array of float4 values. Store the sampled value in a float.

```
// --- HeightMap Calculations --- //
uint2 dim = (uint2)0;
heightMap.GetDimensions(dim.x, dim.y);

const float2 uv = float2(( uvs[id].x * dim.x ) % dim.x, ( uvs[id].y * dim.y ) % dim.y);

const float h1 = heightMap[uv].r;

const float3 height = normalizedPosition * h1 * heightIntensity * int(useHeightMap);
// --- HeightMap Calculations --- //

heights[id] = vertexPos + height;
```

This is almost what we are looking for to produce an acceptable result. The issue with leaving it here is that no matter what values we sample from the heightmap, the vertices will only move further away from the original sphere. In other words, the radius is lost, any value will "bump" the surface, there is no way to make an indentation. We correct this by remapping the sampled between a min and a max value. The Earth is currently the target, so we have remapped the values to be between the lowest and highest point on Earth. Mount Everest at 8849m and the Mariana Trench at -11034m.

```
const float h1 = heightMap[uv].r;
const float hMin = -1.1034; // depth of Mariana Trench 11034m
const float hMax = 0.8849; // height of Mount Everest 8849m
const float result = hMin + ( hMax - hMin ) * h1;

const float3 height = normalizedPosition * result * heightIntensity * int(useHeightMap);
// --- HeightMap Calculations --- //

heights[id] = vertexPos + height;
```

This should give a decent result. We can also download the Bathymetric textures from [11] GEBCO.

This can give a greatly exaggerated looking result of the planet's surface and sea bottom.

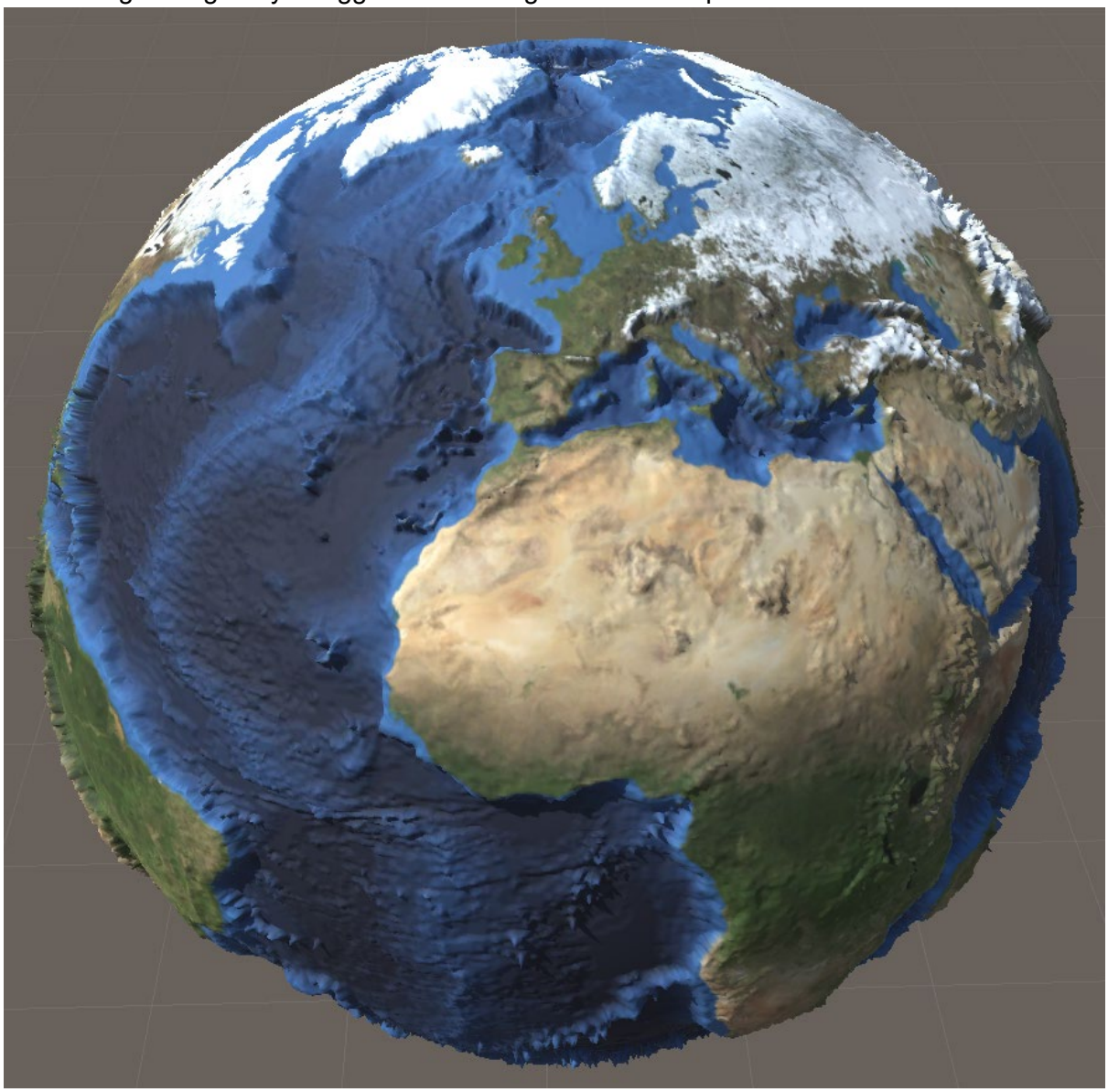


Figure 19. Current results from the Compute Shader by using a bathymetry map of earth and a NASA Earth map.

The Sphere used here is a Icosphere with 200 vertices per triangle edge, radius 10, height strength 1.5

Procedural Planet generation

At this point we have had a quick look at how compute shaders work and how we can utilize them. Now that we have some experience with compute shaders there is some other interesting things we can do. Using an openly accessible [12] Simplex Noise library we attempt to generate our own planet. Like the previous method, we integrate this into the C# sphere scripts, there will be some minor adjustments that will have to be done to make this work for all the spheres, but that will be highlighted when it is relevant.

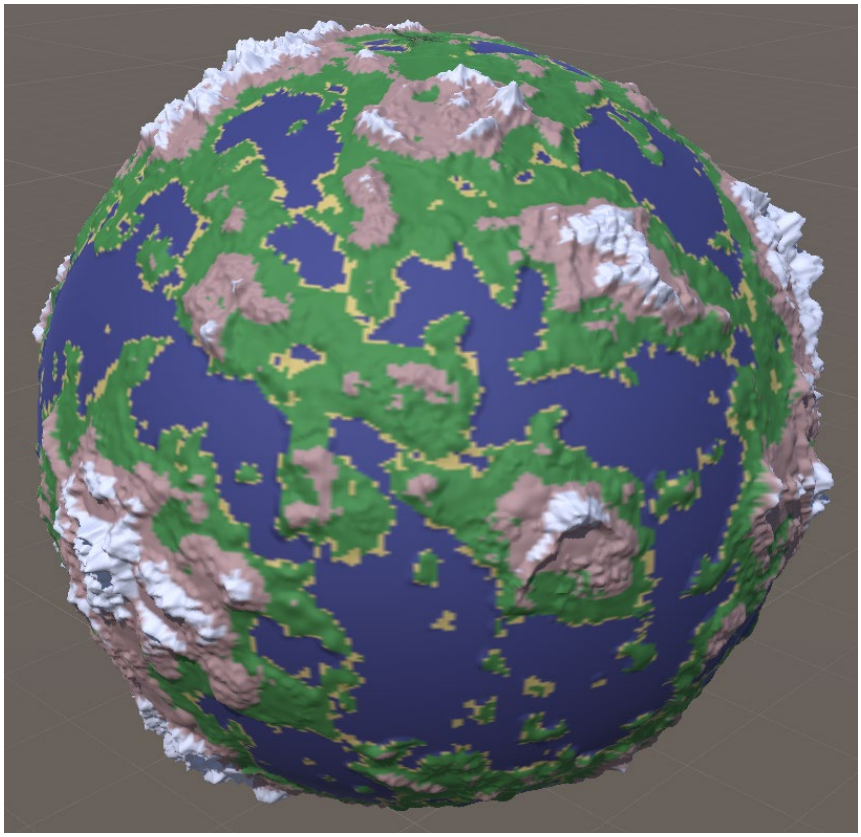


Figure 20. Compute shader used to adjust elevation based on simplex noise, then passed through a 2nd Compute shader to apply color based on elevation.

Setup

Create two new Compute Shader scripts, one for shape and one for colour. Download and add a noise library, make sure that the noise library file has the extension ".cginc" so it can be included in a compute shader file.

C# in the sphere scripts

Add a public ComputShader which will take in the shapeComputShader, also add a Boolean that can be used to toggle the effect of the noise one and off. Next is the main part of this shader, the noise settings, which consist of a layerCount – how many time the layer will accumulate. Scale – how strongly the noise should apply. Frequency - how irregular the simplex noise should start out. FrequencyMultiplyer – how rapid the frequency changes. Amplitude – how high the peal of the noise

should be. amplitueMultiplyer – how much the peaks should change per layerCount. Offset – where we start in the simplexNoise. MinimumValue – the smallest value the noise must be to have an effect. verticalShify – how far from the surface should the noise move. There are a few more settings to be added, but we won't consider them right now. All the Noise settings are in the form of floats. There should also be a container to hold the NoiseSettings.

```
// ------- Shader Settings ------- //
public ComputeShader shapeComputeShader;

// ------- Noise Settings ------- //
public bool useNoise = false; // for convenience

[SerializeField]
public NoiseType[] noiseLayerType;

[System.Serializable]
public struct NoiseSettings
{
	[Range(0, 10)]
	public float layerCount;
	public float scale;
	public float frequency;
	public float freqMultiplier;
	public float amplitude;
	public float ampMultiplier;
	public Vector3 offset;
	public float minValue;
	public float vShift;
}

[SerializeField]
public List<NoiseSettings> noiseSettings = new List<NoiseSettings>();
```

Now we can make the Method for the compute shader, parts of this method should look familiar. Start by checking if the shapeComputeShader exists. Find and store the max amount of vertices a mesh holds. Make the ComputBuffer for vertices and a ComputeBuffer to write new vertex data to. Just like it was done in the previous shader. Create a ComputeBuffer for the NoiseSettings. The stride for this is 44. Assign the buffers to the compute shader, also assign other settings. Note that at this point if a Single-Mesh Cube-Sphere is used, also set the cubes resolution, this is needed for normalizing.

```
void ComputeProceduralPlanet()
{
	if (shapeComputeShader == null)
		return;

	int maxLength = 0;
	for (int i = 0; i < maxFaces; ++i)
	{
		maxLength = Mathf.Max(maxLength, meshFilters[i].sharedMesh.vertices.Length);
	}

	ComputeBuffer computeVBuffer = new ComputeBuffer(maxLength, 12);
	ComputeBuffer computeHBuffer = new ComputeBuffer(maxLength, 12);

	ComputeBuffer computeNoiceSettingsBuffer = new ComputeBuffer(noiseSettings.Count, sizeof(float) * 11);

	computeNoiceSettingsBuffer.SetData(noiseSettings);

	shapeComputeShader.SetInt("noiseLayers", noiseSettings.Count);
	shapeComputeShader.SetFloat("radius", radius);
	shapeComputeShader.SetBool("useNoise", useNoise);
	// for single-mesh cube-Sphere
	// shapeComputeShader.SetFloat("resolution", resolution);
	shapeComputeShader.SetBuffer(0, "noiseSettings", computeNoiceSettingsBuffer);
	// method continues
```

Looping over all meshes is identical to how it was done previously, except this time we do not need UVs to be submitted to the compute shader.

```
        for( int i = 0; i < maxFaces; ++i)
        {
                Vector3[] allVerts = icoFaces[i].GetVerts();

                shapeComputeShader.SetFloat("numVertices", allVerts.Length);

                computeVBuffer.SetData(allVerts);

                shapeComputeShader.SetBuffer(0, "vertices", computeVBuffer);
                shapeComputeShader.SetBuffer(0, "heights", computeHBuffer);

                shapeComputeShader.Dispatch(0, 512, 1, 1);

                computeHBuffer.GetData(allVerts);

                meshFilters[i].sharedMesh.vertices = allVerts;
                meshFilters[i].sharedMesh.RecalculateBounds();
                meshFilters[i].sharedMesh.RecalculateNormals();
                meshFilters[i].sharedMesh.RecalculateTangents();
        }

        computeHBuffer.Release();
        computeNoiceSettingsBuffer.Release();
        computeNLTBuffer.Release();
        computeVBuffer.Release();
}
```

That is all that needs to be done in the sphere script for now.

HLSL for the Compute Shader

Like how it was done previously, all the data that needs to be received by the compute shader needs to be declared. Another important thing is to now include the noise library.

```
#include "Noise.cginc"
#pragma kernel CSMain

StructuredBuffer<float3> vertices;
RWStructuredBuffer<float3> heights;

float radius;
float resolution; // unly used for the single-mesh cube-sphere
uint numVertices;

bool useNoise;

struct NoiseSettings
{
	float numLayers;
	float scale;
	float frequency;
	float frequencyMultiplier;
	float amplitude;
	float amplitudeMultiplier;
	float3 offset;
	float minValue;
	float vShift;
	float ridgeWeight;
	float ridgeWeightMultiplier;
	float useFirstLayerAsMask;
};

StructuredBuffer<NoiseSettings> noiseSettings;
int noiseLayers;
```

In the Main function start by making sure that the current thread ID is not out of range. Next normalize the current vertex, this is done differently depending on what shape is in use. But this only needs to be done if the shapes are not already normalized, the change here is to show that it is possible to filter out certain things that are done on the CPU, and pass it to the compute shader, where it can be done much quicker. Also, at this point in the shader, it is not yet important for the shape to be sphere.

```
[numthreads(512, 1, 1)]
void CSMain( uint id : SV_DispatchThreadID)
{
	// TODO: insert actual code here!
	if( id >= numVertices )
	{
		return;
	}

	// --- Spherification Calculations --- //
	// for IcoSpheres
	const float3 normalizedPosition = normalize(vertices[id]);
	// for multi-mesh cube-Sphere
	// const float3 normalizedPosition = GetBetterPoint(vertices[id]);
	// for single-mesh cube-Sphere
	// const float3 normalizedPosition = GetBetterPoint(vertices[id], resolution);

	const float3 vertPos = normalizedPosition * radius;

	heights[id] = vertPos;
}
```

```
// --- Noise Calculations --- //
const float3 noiseResult = normalizedPosition * CalculateNoise(vertPos) * int(useNoise);

heights[id] = vertPos + noiseResult;
```

CalculateNoise is used as somewhere to calculate the rest, so it does not get as cluttered in the main method as it did when the heightMap was used. For the moment CalculateNoise accumulates the resulting height value from the noise functions before returning the result.

```
float CalculateNoise( float3 p )
{
	float height = 0;
	for( int ns = 0; ns < noiseLayers; ++ns )
	{
		height += SimpleNoise(noiseSettings[ns], p);
	}
	return height;
}
```

The SimpleNoise function starts by extracting and storing all the relevant information from the NoiseSettings that was passed in. Create a variable to store the accumulated results. Next loop according to how many times the noise should be layered on top of itself. The loop uses the SimplexNoise Library. We pass in our point p multiplied with the current frequency plus an offset, store the result and remap it to a value between 0 and 1. This in done because the SimplexNoise library returns a value that can be in the range between -1 to 1. We multiply the result with our amplitude. When the noise is done accumulating the value gets measured against the preferred minimumValue then it is scaled and vertically shifted. Before the value is returned.

```
float SimpleNoise( NoiseSettings ns, float3 p )
{
	const int numLayers		= ns.numLayers;
	const float scale		= ns.scale;
	float frequency			= ns.frequency;
	const float freqMultiplier	= ns.frequencyMultiplier;
	float amplitude			= ns.amplitude;
	const float ampMultiplier	= ns.amplitudeMultiplier;
	const float3 offset		= ns.offset;
	const float minValue		= ns.minValue;
	const float vShift		= ns.vShift;

	float accumulatedNoice = 0;

	for( int i = 0; i < numLayers; ++i )
	{
		float noise = snoise(p * frequency + offset);
		accumulatedNoice += ( noise + 1 ) * 0.5 * amplitude;

		frequency *= freqMultiplier;
		amplitude *= ampMultiplier;
	}

	accumulatedNoice = max(0, accumulatedNoice - minValue);

	return ( accumulatedNoice * scale ) + vShift;
}
```

With just some simple values passed in at this point we end up with a sphere with details resembling continents.

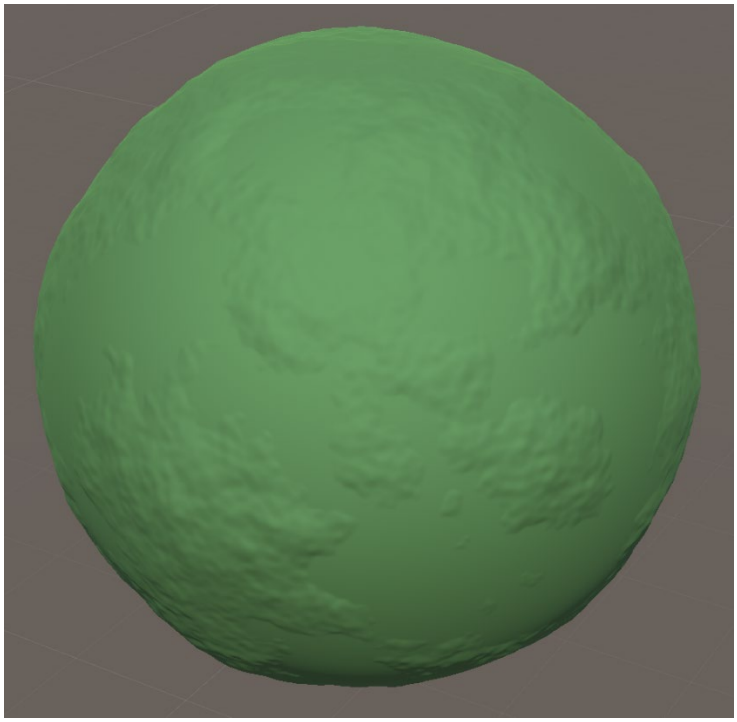


Figure 21. Noise Settings used: layerCount = 4, scale = 0.65, frequency = 0.125, freqMultiplier = 2.45, amplitude = 0.63, ampMultiplier = 0.44, offset = (5, -9, -25), minValue = 0.45, vShift = 0

This can continue to be layered on top of itself to make even more interesting looking surfaces. But they quickly lose the form from an earlier layer and become just noise. We solve this by letting layer

```
float calcNoise( float3 p )
{
	float firstLayerResult = 0;
	float height = 0;

	if( noiseLayers > 0 )
	{
		firstLayerResult += SimpleNoise(noiseSettings[0], p);
		height = firstLayerResult;
	}

	for( int ns = 1; ns < noiseLayers; ++ns )
	{
		const float mask = ( noiseSettings[ns].useFirstLayerAsMask > 0 ) ? firstLayerResult : 1;

		height += SimpleNoise(noiseSettings[ns], p) * mask;
	}

	return height;
}
```

The result from this is already looking like a planet.

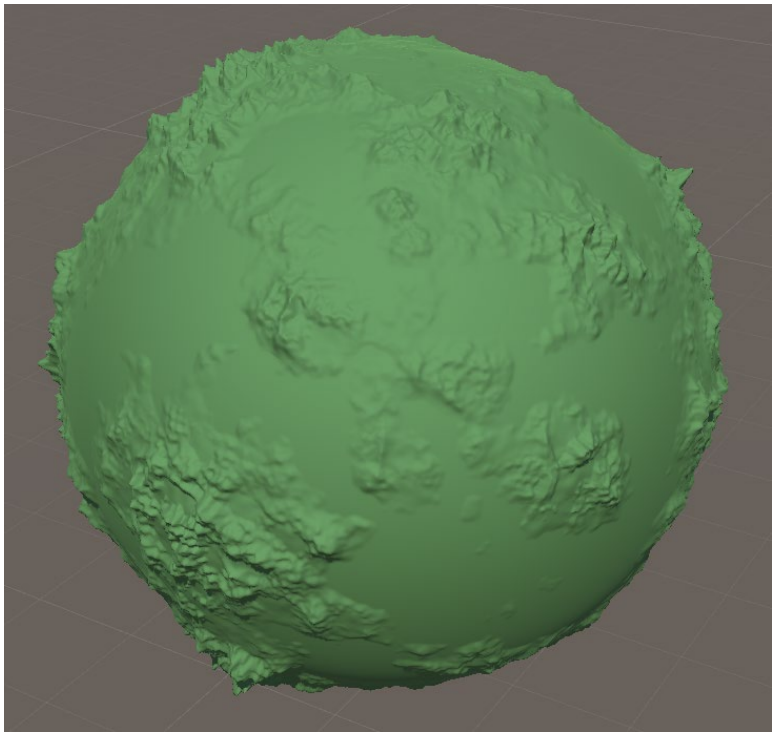


Figure 22. Noise Settings used 2nd layer: layerCount = 5, scale = 1.78, frequency = 0.68, freqMultiplier = 5, amplitude = 2.11, ampMultiplier = 0.29, offset = (-2, -29, 9), minValue = 1.3, vShift = 0, effetFirstLayer = 1

There is now one last adjustment needed, another noise. A noise that can produce mountain ranges, or rivers, depending on how it's used. For this we need to further adjust the C# script. We define an enum for different types of Noise. Add two new variables to the NoiseSettings: rangeWeight and rangeWeightMultiplier. This needs to be adjusted in the stride and in the compute shader. We store this in a separate container as we have hade issues when attempting to fit the enum inside the NoiceSettings.

```
public enum NoiseType : int
{
	None = 0,
	SimplexNoise = 1,
	RangeNoise = 2,
}

[SerializeField]
public NoiseType[] noiseLayerType;
```

This array gets passed to the compute shader. The compute shader expects an array of ints. The ComputBuffer is created alongside the other compute buffers and is submitted to the compute shader before the loop.

```
ComputeBuffer computeNLTBuffer = new ComputeBuffer(noiseLayerType.Length, sizeof(int));
computeNLTBuffer.SetData(noiseLayerType);
shapeComputeShader.SetBuffer(0, "noiseLayerType", computeNLTBuffer);
```

In the compute shader the CalculateNoise is updated to take the new int array into account by using a switch statement to select which Noise method to call. This adjustment is done in both the loop and the first layer. In the loop the result is added to height instead of firstLayerResult.

```
switch( noiseLayerType[0] )
{
	case 1:
		firstLayerResult += SimpleNoise(noiseSettings[0], p);
		break;
	case 2:
		firstLayerResult += RangeNoise(noiseSettings[0], p);
		break;
	default:
		break;
}
height = firstLayerResult;
```

RangeNoise is almost identical to SimpleNoise, there is a change to how the result from the NoiseLibrary is handled, and then we also take into use the new variables. The result from snoise() is set to be an absolute value, the value is subtracted from 1 so to make the peak of the range to be sharp, instead of having rounded hills with sharp steep rivers. Next, we multiply the result with itself and the rangeWeight which is one of the new variables to amplify the effect. The rangeWeightMultiplyer is multiplied with the current noise result, and then assigned to rangeWeight. The rest from here is the same as SimpleNoise.

```
const float rangeWeightMultiplier = ns.rangeWeightMultiplier;
float rangeWeight = ns.rangeWeight;

float accumulatedNoise = 0;

for( int i = 0; i < numLayers; ++i )
{
	float noise = 1 - abs(snoise(( p * frequency ) + offset));
	noise *= noise;
	noise *= rangeWeight;
	rangeWeight = noise * rangeWeightMultiplier;

	accumulatedNoise += noise * amplitude;

	frequency *= freqMultiplier;
	amplitude *= ampMultiplier;
}
```

This is the part that is different from the SimpleNoise Method, it has been adjusted accordingly.

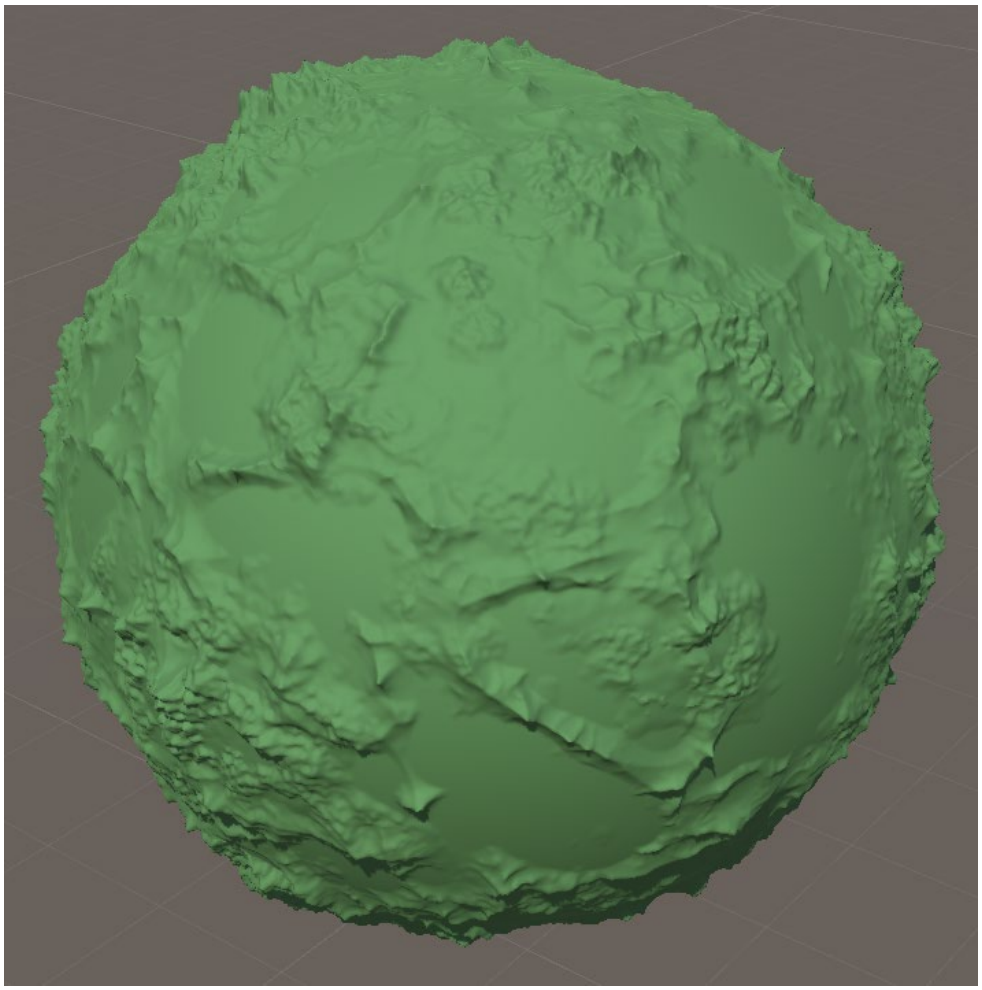


Figure 23. Noise Settings used by 3nd layer: layerCount = 5, scale = 0.75, frequency = 0.15, freqMultiplier = 2, amplitude = 0.25, ampMultiplier = 1, offset = (6, 4, 2), minValue = 0.25, vShift = 0, rangeWeight = 1.125, rangeMultiplyer = 0.9, effetFirstLayer = 1

That is all for this Compute Shader.

The Colouring shader

Additionally, one can add another compute shader that can draw to a texture, which then is applied to the sphere. This is a quick and dirty way of doing it. It draws a 3x3 pixel colour to a texture based on the vertex elevation. The settings needed is a new ComputeShader. 4 floats to determine where the colour should change from one to another, a renderTexture, and a Texture2D.

```
public ComputeShader colorComputeShader;
public int textureSize = 256;
public RenderTexture renderTexture;
public Texture2D dest;

public float waterMax;
public float grassLine;
public float mountainLine;
public float snowLinef;
public bool freeUpdateColor;
```

The new PlanetColoring Method begins with checking if there is a computShader assigned, just like the other methods have started. Then a new renderTexture is made and enabled for randomWritin. Next, we bind all the settings to the compute shader.

```
void ComputeProceduralPlanetColor()
{
	if (colorComputeShader == null || useMaterial || !freeUpdateColor)
	{
		return;
	}

	if (renderTexture == null || (textureSize != renderTexture.width) || freeUpdateColor)
	{
		renderTexture = new RenderTexture(textureSize, textureSize, 24);
		renderTexture.enableRandomWrite = true;
		renderTexture.Create();
	}

	int maxLength = 0;
	for (int i = 0; i < maxFaces; ++i)
	{
		maxLength = Mathf.Max(maxLength, meshFilters[i].sharedMesh.vertices.Length);
	}

	ComputeBuffer computeVBuffer = new ComputeBuffer(maxLength, 12);
	ComputeBuffer computeUVBuffer = new ComputeBuffer(maxLength, 8);

	colorComputeShader.SetTexture(0, "Result", renderTexture);

	colorComputeShader.SetFloat("pxOffset", pixelOffset);
	colorComputeShader.SetFloat("waterMax", waterMax);
	colorComputeShader.SetFloat("grassLine", grassLine);
	colorComputeShader.SetFloat("mountainLine", mountainLine);
	colorComputeShader.SetFloat("snowLine", snowLine);
	colorComputeShader.SetFloat("radius", radius);
	colorComputeShader.SetFloat("resolution", renderTexture.width);
	colorComputeShader.SetFloat("lerpValue", lerpValue);
	// method continues
```

```
for (int i = 0; i < maxFaces; ++i)
{
	Vector3[] allVerts = meshFilters[i].sharedMesh.vertices;
	Vector2[] alluvs = meshFilters[i].sharedMesh.uv;

	colorComputeShader.SetInt("numVertices", meshFilters[i].sharedMesh.vertices.Length);

	computeVBuffer.SetData(allVerts);
	computeUVBuffer.SetData(alluvs);

	colorComputeShader.SetBuffer(0, "vertices", computeVBuffer);
	colorComputeShader.SetBuffer(0, "uvs", computeUVBuffer);

	colorComputeShader.Dispatch(0, 512, 256, 1);
}

dest = new Texture2D(renderTexture.width, renderTexture.height, TextureFormat.RGBA32, false);

dest.filterMode = FilterMode.Bilinear;
dest.Apply(false);
Graphics.ConvertTexture(renderTexture, dest);

for (int i = 0; i < maxFaces; ++i)
{
	Material material = new Material(meshFilters[i].GetComponent<MeshRenderer>().sharedMaterial);
	material.SetTexture("_MainTex", dest);
	material.SetFloat("_Glossiness", 0.2f);
	material.color = new Color(1f, 1f, 1f);
	meshFilters[i].GetComponent<MeshRenderer>().sharedMaterial = material;
}

computeVBuffer.Release();
renderTexture.Release();
computeUVBuffer.Release();
}
```

HLSL for the Compute Shader

The variables are set to be received in the shader. A new aspect this time is that we use a uint2 as a thread id, this is to easily draw to the correct place on the texture. We combine the x and y component to get the current thread index, to index into the correct UV coordinate. But before that the magnitude of the current vertex is calculated, a colour is created, and a waterline is defined. The vertex magnitude gets compared with the defined altitudes, if the magnitude is less than any of them, the colour gets assigned. Calculate the current pixel and draw a 3x3 grid around it.

```
RWTexture2D<float4> Result;
StructuredBuffer<float3> vertices;
StructuredBuffer<float2> uvs;
StructuredBuffer<float> minMax;

uint numVertices;
float pxOffset;
float resolution;
float radius;
float lerpValue;
float waterMax;
float grassLine;
float mountainLine;
float snowLine;
```

```
[numthreads(16, 16, 1)]
void CSMain( uint2 id : SV_DispatchThreadID)
{
	const uint idx = id.x + id.y;
	if( idx >= numVertices )
		return;

	const float mag = magnitude(vertices[idx]) * 10;
	float4 col = (float4)0;

	const float waterline = radius * radius * 10 + waterMax;

	if( mag > snowLine )
	{
		col.r = 0.60; col.g = 0.60; col.b = 0.7; // snow
	}
	else if( mag > mountainLine )
	{
		col.r = 0.45; col.g = 0.32; col.b = 0.34; // mountain
	}
	else if( mag > grassLine )
	{
		col.r = 0.07; col.g = 0.3; col.b = 0.09; // grass
	}
	else if( mag > waterline )
	{
		col.r = 0.56; col.g = 0.47; col.b = 0.24; // sand/beach
	}
	else
	{
		col.r = 0.06; col.g = 0.07; col.b = 0.30; // water
	}
		const float res = resolution;
	const float pixelSize = pxOffset;

	const float2 coord = ( uvs[idx] * res ) % res;

	const float2 c1 = coord - float2(pixelSize, pixelSize);
	const float2 c2 = coord - float2(0, pixelSize);
	const float2 c3 = coord + float2(pixelSize, -pixelSize);
	const float2 c4 = coord - float2(pixelSize, 0);
	const float2 c5 = coord;
	const float2 c6 = coord + float2(pixelSize, 0);
	const float2 c7 = coord + float2(-pixelSize, pixelSize);
	const float2 c8 = coord + float2(0, pixelSize);
	const float2 c9 = coord + float2(pixelSize, pixelSize);


	Result[c1] = lerp(Result[c1], col, lerpValue);
	Result[c2] = lerp(Result[c2], col, lerpValue);
	Result[c3] = lerp(Result[c3], col, lerpValue);

	Result[c4] = lerp(Result[c4], col, lerpValue);
	Result[c5] = lerp(Result[c5], col, lerpValue);
	Result[c6] = lerp(Result[c6], col, lerpValue);

	Result[c7] = lerp(Result[c7], col, lerpValue);
	Result[c8] = lerp(Result[c8], col, lerpValue);
```

The result of applying this to the previously generated sphere is a planet-like object.

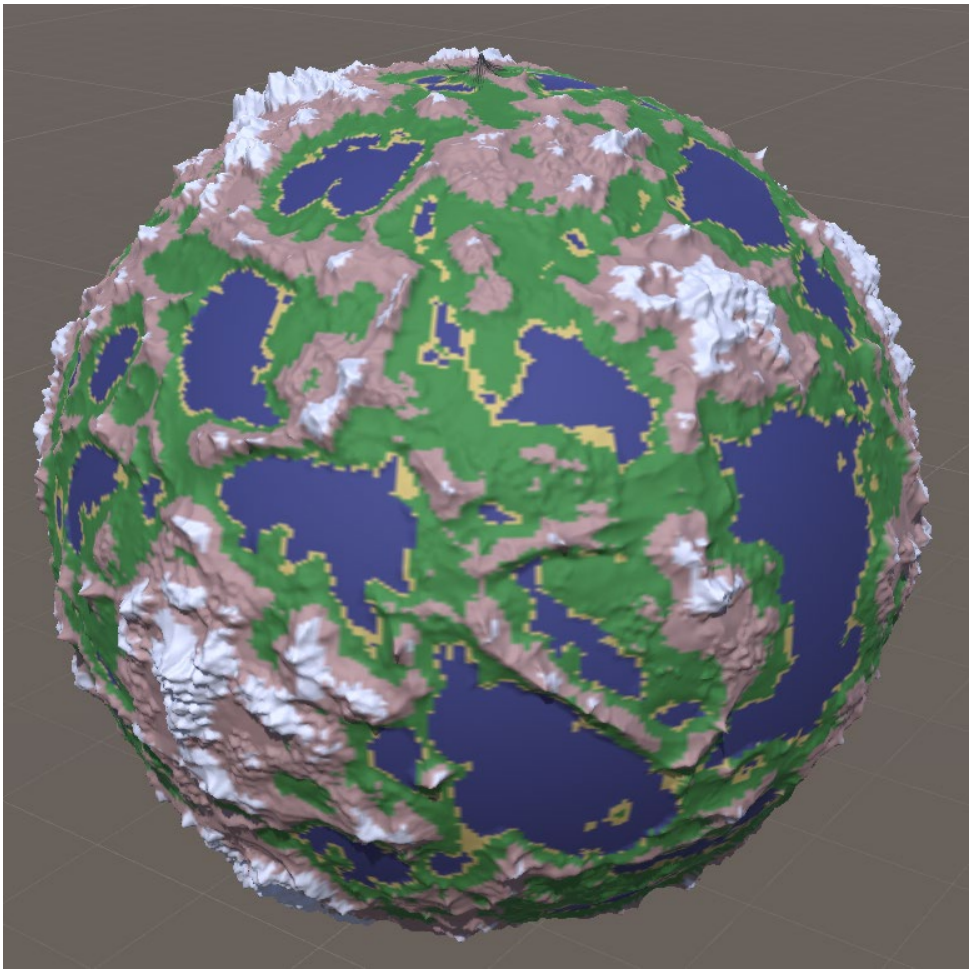


Figure 24. Colored planet. Water-levels end at`= 0.1, sand/beach level end at = 1003, grass levels end at = 1026, mountain levels end at = 1065, everything above 1065 is snow.

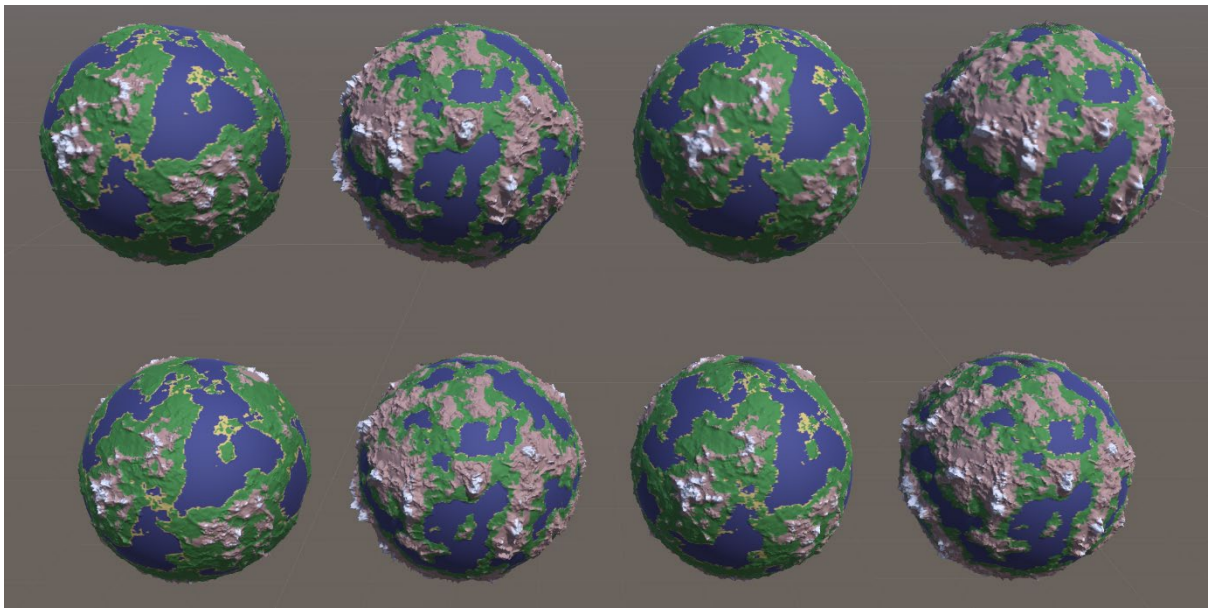


Figure 25. Two spheres next to each other same type different scale and radius.
Left scale 1, radius 10. Right scale 3.7, radius 2.7.
Two Top left: Multi-Mesh Cube Sphere. Two Top right: Single-Mesh Cube Spheres.
Two Bottom left: Icospheres alternative method, Two bottom Right: Icospheres.

Conclusion

The worst performing shape out of these has been the Single-Mesh Cube-Sphere. As its subdivisions peaks out at 52, the total amount of vertices is no more than 65,105. With any further subdivisions the containers holding the information overflows. This was an unexpected discovery. The limiting factor for this is the use of only one mesh. The mesh and its containers can only hold so much data. The shape starts to deteriorate when the number of vertices goes over the max value of a 16-bit datatype: 65,535.

The next shape that performed worse than expected was the first implementation of the IcoSphere, as it also quickly hit the upper limits of what a Mesh could hold. At 8 subdivisions, the mesh already contains 65,538 vertices, and others as many as 65,984. This limited the shape to stay within only 7 subdivisions, though the shape does not start looking like a sphere until subdivision level 4. On the other hand, the second icosphere implementation with exact control over how many vertices should be along each edge of a base triangle performed very well. With the control of up to 358 vertices per edge, we could easily get up to the limit without stepping over it. This on the other hand started showing in the implementation of the colouring of the planet, as we quickly ran out of thread groups after 250 vertices per edge, and parts of the planet stopped getting coloured. This however is just a shortcoming on the implementation and can easily be fixed by painting a texture in a different manner.

Lastly the Multi-Mesh Cube-Sphere was similar to the 2nd Icosphere in behaviour and performance, however it was easy to notice how each of the sides of the cube could have been further split into multiple meshes and handled accordingly.

In the end the second Icosphere performed the best and seems like the best candidate for larger scale planets, as one can remove the triangles that are not seen, and only keep a few. Where with the cube sphere we would need to always keep rendering at least one sixth of the shape at any given time. The cube must spread all 65k vertices along one entire side, it is not the best choice for a larger planet when a player may walk around on the surface. Not without a lot of additional splitting of the meshes. The Icosphere has this already in order from the start, where it has more meshes to choose from.

The Single-Mesh Cube-Spheres best use case can be considered as a distant prop.

Future Work

Future iteration of this project can include further examination of the Multi-Mesh Cube-Sphere and how we could subdivide it into multiple new meshes to achieve more detail. Ways to circumvent the limitation of the 65k vertices per mesh limit can be explored. It would also be interesting to add a Level of Detail system to adjust the granularity on the fly. Additionally, more Noise-Types and ways to shape the planets is a point that can be expanded on. A proper system to colour the planets can be added to make the planets look more distinct. Moving the Mesh Generation code to the GPU and a proper water and atmosphere shader can be added.

Bibliography

[1] *OpenGL Sphere*. (n.d.). Retrieved January 18, 2022, from http://www.songho.ca/opengl/gl_sphere.html#example_icosphere

[2] González, Á. (2010). Measurement of Areas on a Sphere Using Fibonacci and Latitude-Longitude Lattices. *Mathematical Geosciences*, *42*(1), 49–64. https://doi.org/10.1007/s11004-009-9257-x

[3] Kogan, J. (2017). A New Computationally Efficient Method for Spacing n Points on a Sphere. In *Undergraduate Mathematics Journal Rose-Hulman Undergraduate Mathematics Journal* (Vol. 18, Issue 2).

[4] Ronchi, C., Iacono, R., & Paolucci, P. S. (1996). The "'Cubed Sphere'": A New Method for the Solution of Partial Differential Equations in Spherical Geometry. In *JOURNAL OF COMPUTATIONAL PHYSICS* (Vol. 124).

[5] Mapping a Cube to a Sphere | Math Proofs. (n.d.). Retrieved November 2, 2021, from http://mathproofs.blogspot.com/2005/07/mapping-cube-to-sphere.html

[6] Baumgardner, J. R., & Frederickson, P. O. (1985). ICOSAHEDRAL DISCRETIZATION OF THE TWO-SPHERE*. In *SIAM J. NUMER ANAL* (Vol. 22, Issue 6). http://www.siam.org/journals/ojsa.php

[7] Rounded Cube, a Unity C# Tutorial. (n.d.). Retrieved December 27, 2021, from https://catlikecoding.com/unity/tutorials/rounded-cube/

[8] catch 22 - Andreas Kahler's blog: Creating an icosphere mesh in code. (n.d.). Retrieved January 27, 2022, from http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html

[9] *UV mapping a sphere triangle mesh - MFT Development*. (n.d.). Retrieved December 12, 2021, from https://mft-dev.dk/uv-mapping-sphere/

[10] *NASA Visible Earth - Home*. (n.d.). Retrieved December 8, 2021, from https://visibleearth.nasa.gov/collection/1484/blue-marble?page=1

[11] *GEBCO - The General Bathymetric Chart of the Oceans*. (n.d.). Retrieved January 28, 2022, from https://www.gebco.net/

[12] *2D / 3D / 4D optimised Perlin Noise Cg/HLSL library (cginc) - Unity Forum*. (n.d.). Retrieved December 28, 2022, from https://forum.unity.com/threads/2d-3d-4d-optimised-perlin-noise-cg-hlsl-library-cginc.218372/