published 2022-05-31

Geodesic Octasphere Tracing Lines on a Sphere

Place vertices uniformly along meridians.
Interpolate horizontally across rhombuses.
Interpolate along geodesic lines.

This is the ninth tutorial in a series about procedural meshes. This time we use geodesic lines to position the vertices of an octasphere.

This tutorial is made with Unity 2020.3.29f1.

An octasphere with vertices placed on geodesic lines.

Fixes

The calculation for the triangle gizmo positions was incorrect. Missing parentheses caused the division to happen after transformation. See Seamless Cube Sphere section 4.1, where `TransformPoint` now opens with `((` and closes with `))`.

The formula for the bounds of the hexagon grids were wrong and have been corrected. See sections 2.5 and 2.6 of the Triangle Grid tutorial for the correct bounds.

1 Meridians

An octasphere has the same problem that a cube sphere has: the vertices bunch up near the corners of the base shape. In the case of the cube there are eight corners, while in the case of the octahedron there are six corners. So the octahedron has two more problems regions than the cube, but the distortion of the vertex distribution is less severe in each region.

Vertex distribution, looking at the top and back corners and a face; resolution 20.

The octasphere and the UV sphere have in common that both contain meridians along which vertices are placed. In the case of the UV sphere there are many meridians, while in the case of the octasphere there are two circles, matching the vertical edges of the octahedron. We distributed vertices evenly along those lines when generating the UV sphere, so it makes sense to try the same approach for the octasphere.

1.1 Geo Octasphere

Meridians are a special case of geodesic lines, which are the shortest lines that connect two points on the surface of a sphere. Thus these are always curved lines, which can be found by slicing a plane through the two points and the center of the sphere, or by rotating one of the points around the sphere center straight towards the other point, describing an orbit.

As we'll use more geodesic lines besides meridians later, we'll create a geodesic octasphere variant, shortened to geo octasphere. Duplicate `Octasphere` and name it `GeoOctasphere`.

```
	public struct GeoOctasphere : IMeshGenerator { … }
```

And add an option for it to `ProceduralMesh`.

```
	static MeshJobScheduleDelegate[] jobs = {
		…
		MeshJob<Octasphere, SingleStream>.ScheduleParallel,
		MeshJob<GeoOctasphere, SingleStream>.ScheduleParallel,
		MeshJob<UVSphere, SingleStream>.ScheduleParallel
	};

	public enum MeshType {
		SquareGrid, SharedSquareGrid, SharedTriangleGrid,
		FlatHexagonGrid, PointyHexagonGrid, CubeSphere, SharedCubeSphere,
		Octasphere, GeoOctasphere, UVSphere
	};
```

1.2 Seam

Let's begin by adjusting the seam vertex line first, turning it into a half meridian. We can do this by replacing the vertex code inside the seam loop of `GeoOctasphere.ExecutePolesAndSeam` with the seam code from `UVSphere`, which uses `sincos`.

```
			for (int v = 1; v < 2 * Resolution; v++) {
				//if (v < Resolution) { … }
				//else { … }
				sincos(
					PI + PI * v / (2 * Resolution),
					out vertex.position.z, out vertex.position.y
				);
				vertex.normal = vertex.position = normalize(vertex.position);
				vertex.texCoord0.y = GetTexCoord(vertex.position).y;
				streams.SetVertex(v + 7, vertex);
			}
```

This means that we no longer need to normalize those vertices and can also directly set the vertical texture coordinate, no longer relying on `GetTexCoord`.

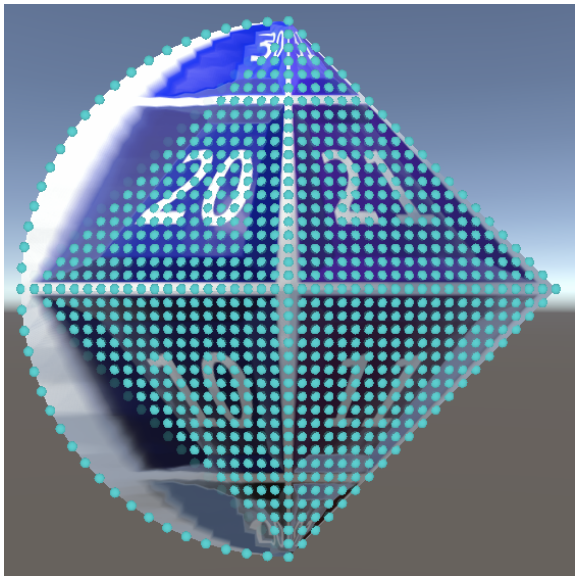
```
            vertex.normal = vertex.position; //= normalize(vertex.position);
            vertex.texCoord0.y = (float)v / (2 * Resolution);
```

If we generate the geo octasphere at this point then we will get two different vertex distributions along the seam. This allows us to see that they are indeed different and that the meridian version is superior.

Two different vertex distributions along seam.

To better see the seam in isolation, let's remove the vertex normalization from `ExecuteRegular`.

```
            var vertex = new Vertex();
            vertex.normal = vertex.position = columnBottomStart;
            …

            for (int v = 1; v < Resolution; v++, vi++, ti += 2) {
                …
                vertex.normal = vertex.position; //= normalize(vertex.position);
                …
            }
```

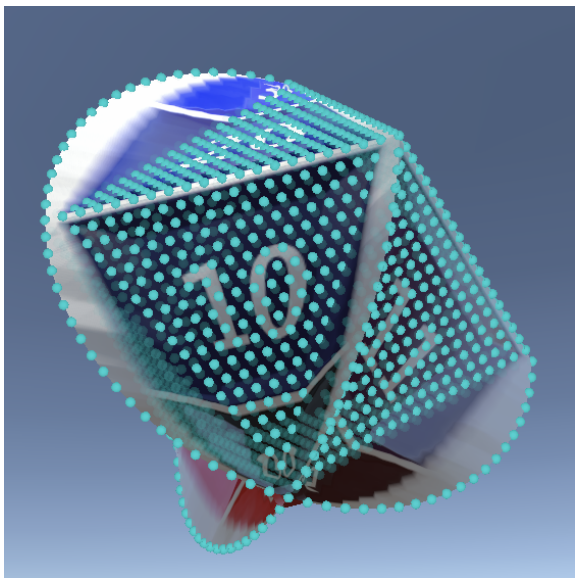


Octahedron with half-meridian seam.

1.3 Bottom Meridians

We can apply the same vertex distribution to the other vertical octahedron edges. Begin by doing this for the bottom edges only, by use the same distribution for the first vertex in `ExecuteRegular`. Here we're dealing with a variable edge direction, so we have to use the sine that we calculate for either the X or the Z coordinate, either positive or negative. We can do this by multiplying the sine with the right corner vector and subtract that from the position.

```
		var vertex = new Vertex();
		sincos(PI + PI * u / (2 * Resolution), out float sine, out vertex.position.y);
		vertex.position -= sine * rhombus.rightCorner;
		vertex.normal = vertex.position; //= columnBottomStart;
```



Four quarter meridians on bottom half.

Now we can also directly set the vertical texture coordinate for those meridians.

```
		vertex.texCoord0.x = GetTexCoord(vertex.position).x;
		vertex.texCoord0.y = (float)u / (2 * Resolution);
```

And we can completely avoid using `GetTexCoord` by noting that along these edges the horizontal texture coordinate is equal to the rhombus identifies plus one, divided by four.

```
		vertex.texCoord0.x = rhombus.id * 0.25f + 0.25f;
```

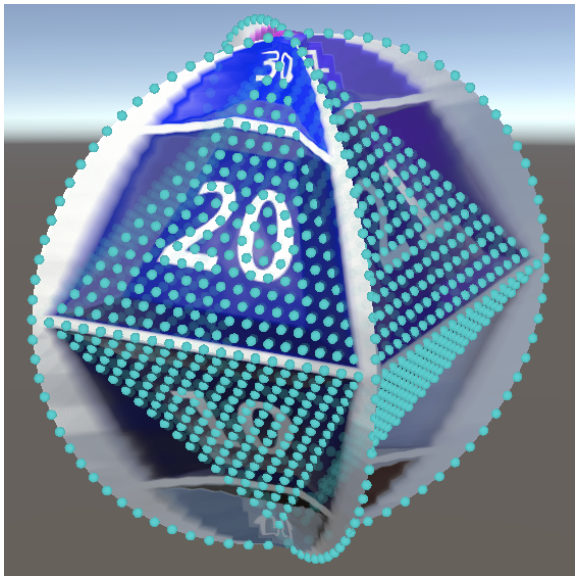
1.4 Top Meridians

Generating the top parts of the meridians is less straightforward, because they're formed by the last row of each rhombus, inside the loop, when U is equal to the resolution. We can add a special case for this inside the loop. We can use the same meridian code for that case, if we first reset the position to zero.

```
		if (u == Resolution) {
			vertex.position = 0f;
			sincos(
				PI + PI * u / (2 * Resolution),
				out sine, out vertex.position.y
			);
			vertex.position -= sine * rhombus.rightCorner;
		}
		else if (v <= Resolution - u) {
			vertex.position =
				lerp(columnBottomStart, columnBottomEnd, (float)v / Resolution);
		}
		else {
			vertex.position =
				lerp(columnTopStart, columnTopEnd, (float)v / Resolution);
		}
```

The other difference here is that the progression is not based on U but on V, plus the resolution because it's the top half of the octahedron.

```
			sincos(
				PI + PI * (v + Resolution) / (2 * Resolution),
				out sine, out vertex.position.y
			);
```



Complete meridians.

1.5 Meridian Interpolation

We can make all other vertices match the meridians by creating each horizontal line by interpolating from the right to the left meridian. We generate the rhombuses diagonally, so the current height from south to north pole is equal to the sum of U and V: $h = u + v$. Use this at the start of the loop to calculate the position on the right edge.

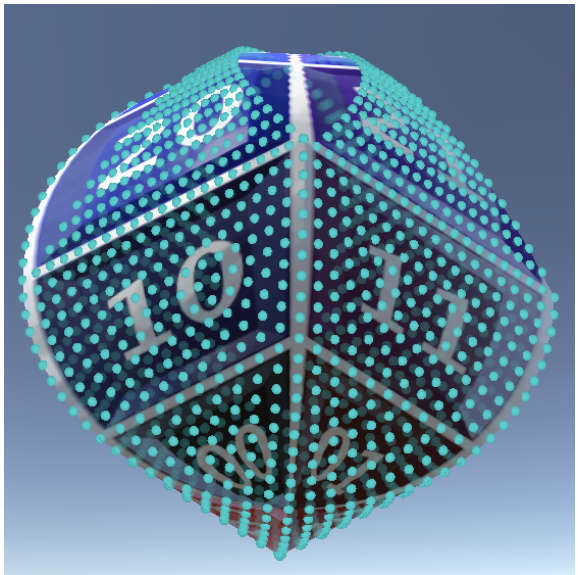
```
		for (int v = 1; v < Resolution; v++, vi++, ti += 2) {
			float h = u + v;
			float3 pRight = 0f;
			sincos(PI + PI * h / (2 * Resolution), out sine, out pRight.y);
			pRight -= sine * rhombus.rightCorner;

			…
		}
```

And also do it for the position on the left edge, by using the left instead of the right corner.

```
			float3 pRight = 0f;
			sincos(PI + PI * h / (2 * Resolution), out sine, out pRight.y);
			float3 pLeft = pRight - sine * rhombus.leftCorner;
			pRight -= sine * rhombus.rightCorner;
```

Now replace the code that sets the vertices with a single horizontal interpolation from the right to the left position. If we look at the bottom half of the octahedron it is clear that V matches the horizontal progression, while the height matches the amount of steps. So let's use $\frac{v}{h}$ as the interpolator.

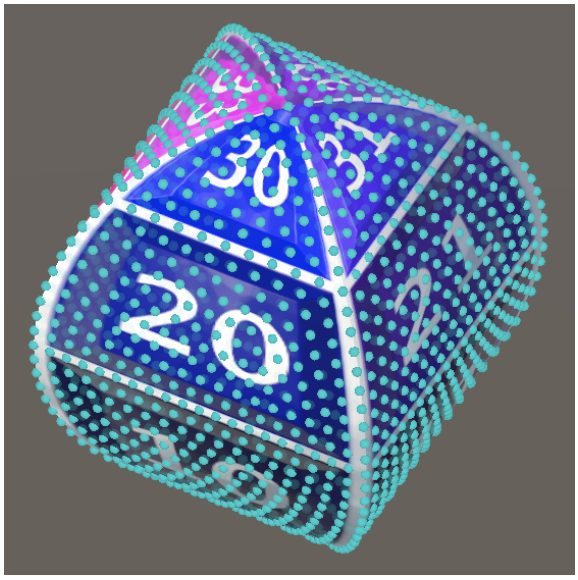
```
			//if (u == Resolution) { … }
			//else if (v <= Resolution - u) { … }
			//else { … }
			vertex.position = lerp(pRight, pLeft, v / h);
			vertex.normal = vertex.position;
```



Interpolation only correct for bottom half.

This produces correct interpolation for the bottom half of the octahedron only. For the top half the horizontal progression is based on U and both it and the amount of steps decreases. So we need to use $\frac{r - u}{2r - h}$ for the interpolator instead.

```
		vertex.position = lerp(
			pRight, pLeft,
			v <= Resolution - u ? v / h : (Resolution - u) / (2f * Resolution - h)
		);
```



Complete interpolation.

1.6 Sphere

Finally, to produce a valid sphere we normalize the vertex positions again.

```
		vertex.position = lerp(
			pRight, pLeft,
			v <= Resolution - u ? v / h : (Resolution - u) / (2f * Resolution - h)
		);
		vertex.normal = vertex.position = normalize(vertex.position);
```

Complete meridian-based octahedron sphere.

Now the vertices no longer bunch up as much near the octahedron corners. The vertex distribution has drastically changed near the poles, because those regions are dominated by the meridians. Near the equator the linearly interpolated lines dominate, so vertices are still pulled towards the corners there, horizontally. Note that this means that the vertex distribution is not as symmetrical as the distribution of a normalized octasphere.

At this point we can get rid of the code that calculates the old interpolation data.

```
		//float3 columnBottomDir = rhombus.rightCorner - down();
		//…
		//float3 columnTopEnd = rhombus.leftCorner + columnTopDir * u / Resolution;
```

2 Geodesic Interpolation

We can improve our geo octasphere further by eliminating all straight lines and instead interpolate along geodesic lines only.

2.1 Quaternions

As described earlier, a geodesic line between two arbitrary points can be made by rotating one point straight towards the other, orbiting around the sphere center. We can use quaternions to generate an arbitrary rotation, via the static `quaternion.AxisAngle` method.

```
using static Unity.Mathematics.math;

using quaternion = Unity.Mathematics.quaternion;
```

Why do we explicitly use the `quaternion` type?

To avoid a name clash with `math.quaternion`. See Basics / Jobs / 4.8 Mathematics Library.

2.2 Axis

A geodesic line can also be found by slicing a plane through the sphere, which goes through both points and the center of the sphere. The normal vector of this plane matches the rotation axis that we need. We can find it by taking the cross product of the vectors pointing to the right and left meridian points, inside the loop in `ExecuteRegular`.

```
			float3 pLeft = pRight - sine * rhombus.leftCorner;
			pRight -= sine * rhombus.rightCorner;

			float3 axis = cross(pRight, pLeft);
```

Because the angle between these vectors isn't fixed the length of the cross product varies, so we have to normalize it.

```
			float3 axis = normalize(cross(pRight, pLeft));
```

2.3 Angle

We also need to know the angle by which we have to rotate. We begin by determining the angle between the two meridian points, by calculating their dot product. This gives us the cosine of the angle so we have use the `acos` method to convert it to radians.

```
float3 axis = normalize(cross(pRight, pLeft));
float angle = acos(dot(pRight, pLeft));
```

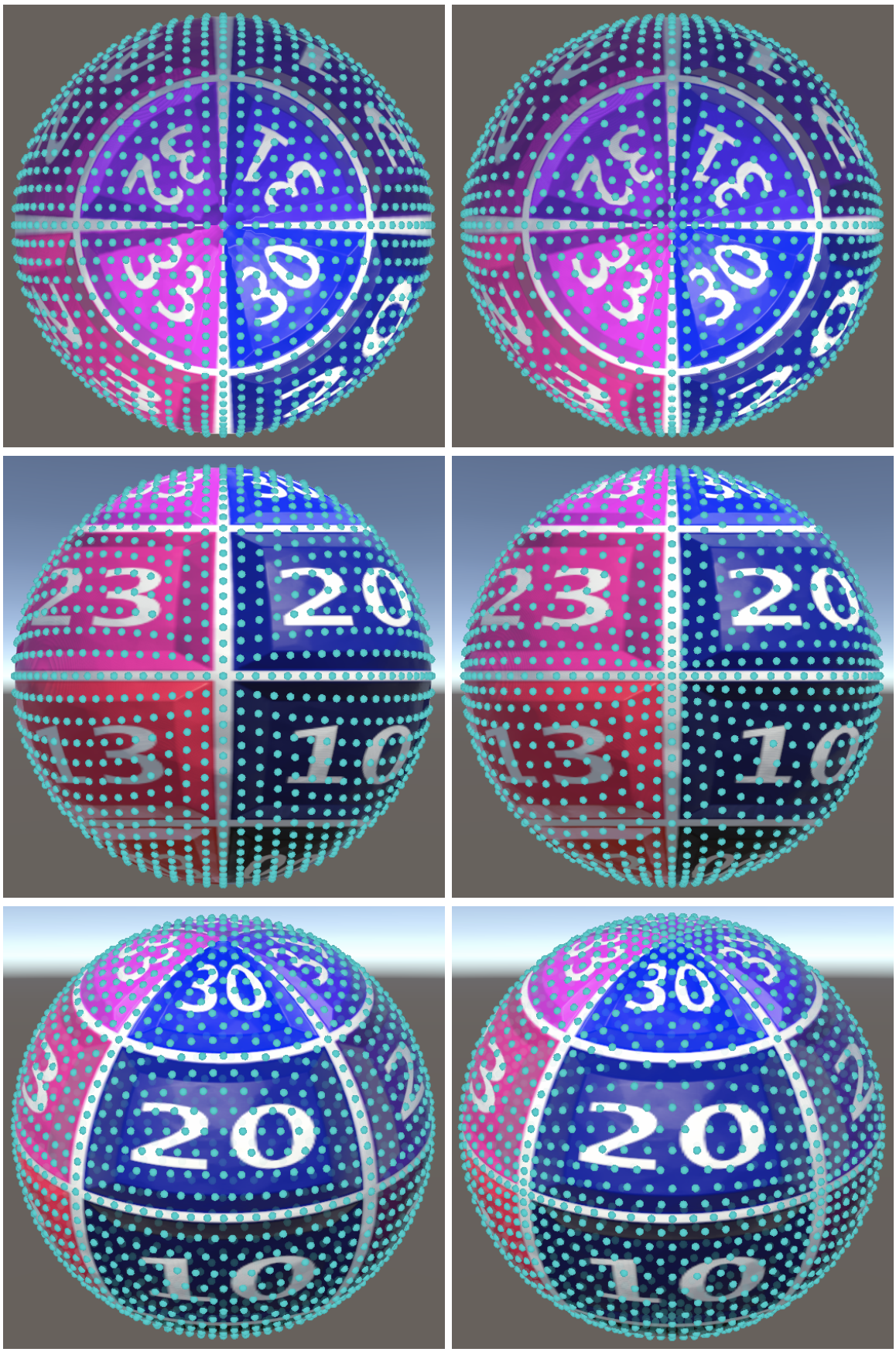
Then, to interpolate along the geodesic line, we have to apply the interpolator as a factor to this angle, which gives us the amount of rotation that we need.

```
float angle = acos(dot(pRight, pLeft)) * (
    v <= Resolution - u ? v / h : (Resolution - u) / (2f * Resolution - h)
);
```

2.4 Interpolation

We apply the rotation by invoking `quaternion.AxisAngle` with the axis and angle, then multiply the resulting quaternion with the right point to apply the rotation.

```
//vertex.position = lerp(
//    pRight, pLeft,
//    v <= Resolution - u ? v / h : (Resolution - u) / (2f * Resolution - h)
//);
//vertex.normal = vertex.position = normalize(vertex.position);
vertex.normal = vertex.position = mul(
    quaternion.AxisAngle(axis, angle), pRight
);
```



Interpolation along geodesic lines, compared with normalized octahedron.

The polar regions haven't changed much, but the vertices no longer get pulled as much toward the octahedron corners on the equator. Note that this approach does vertically stretch the triangle rows near the equator a bit.

Isn't there an alternative mapping, like for the cube sphere?

It it possible to apply an alternative mapping from octahedron to sphere, based on the barycentric coordinates per octahedron face. However, these mappings are inferior to simple vertex normalization because they affect vertices near the middle of edges more than those near corners, which is the opposite of what we need.

The next tutorial will cover one more way to generate a sphere. Want to know when it gets released? Keep tabs on my Patreon page!

made by Jasper Flick