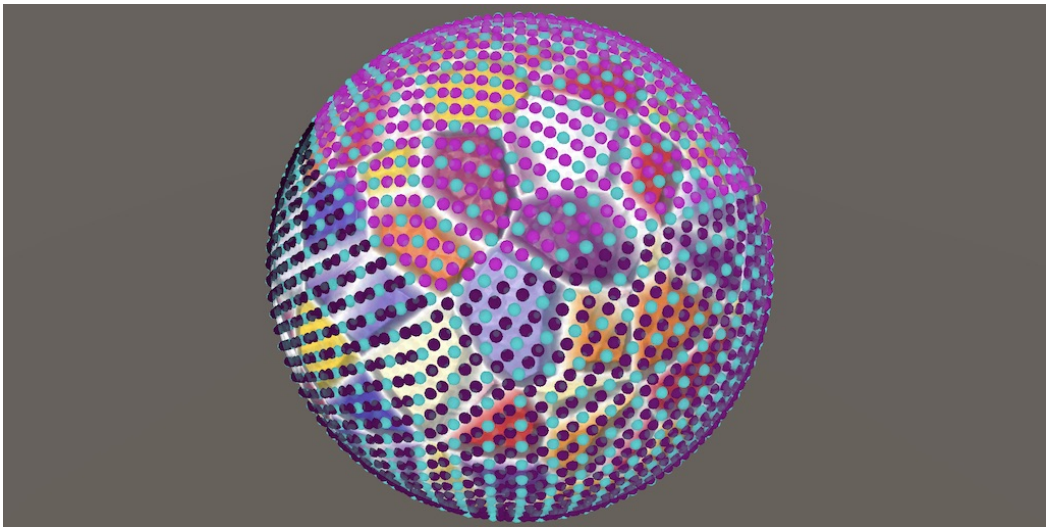
updated 2022-05-29 published 2022-03-31

Seamless Cube Sphere Stitching Sides Together

Support meshes with only position data.
Make all cube sphere vertices shared.
Stitch cube sides together with triangles.
Visualize triangle draw order.
Add mesh optimization options.

This is the seventh tutorial in a series about procedural meshes. This time we create a cube sphere with shared vertices, without any seams.

This tutorial is made with Unity 2020.3.23f1.



A seamless cube sphere showing triangle draw order.

1 Minimal Vertex Data

It is impossible to apply a 2D texture to a sphere without introducing seams, so when we create a cube sphere that lacks seams we cannot rely on per-vertex texture coordinates. So we'll have to use our *Cube Map* material, other materials won't work. This means that we only have to generate vertex positions and can omit normals, tangents, and texture coordinates.

1.1 Position-Only Stream

As we're only going to generate vertex positions, let's begin by creating an alternative `IMeshStreams` implementation for that. This significantly reduces the size of the generated mesh. Duplicate `MultiStream`, rename it to `PositionStream`, and remove all but its first stream field, keeping only `steam0`.

```
	public struct PositionStream : IMeshStreams {

		[NativeDisableContainerSafetyRestriction]
		NativeArray<float3> stream0; //, stream1;

		//[NativeDisableContainerSafetyRestriction]
		//NativeArray<float4> stream2;

		//[NativeDisableContainerSafetyRestriction]
		//NativeArray<float2> stream3;

		…
	}
```

Also remove the unneeded streams from `Setup`, reducing the descriptor array's length to 1.

```
		public void Setup (
			Mesh.MeshData meshData, Bounds bounds, int vertexCount, int indexCount
		) {
			var descriptor = new NativeArray<VertexAttributeDescriptor>(
				1, Allocator.Temp, NativeArrayOptions.UninitializedMemory
			);
			descriptor[0] = new VertexAttributeDescriptor(dimension: 3);
			//descriptor[1] = …;
			//descriptor[2] = …;
			//descriptor[3] = …;
			meshData.SetVertexBufferParams(vertexCount, descriptor);
			descriptor.Dispose();

			…

			stream0 = meshData.GetVertexData<float3>();
			//stream1 = meshData.GetVertexData<float3>(1);
			//stream2 = meshData.GetVertexData<float4>(2);
			//stream3 = meshData.GetVertexData<float2>(3);
			triangles = meshData.GetIndexData<ushort>().Reinterpret<TriangleUInt16>(2);
		}
```

Its `SetVertex` method only has to copy the vertex position, ignoring all other `Vertex` data.

```
		[MethodImpl(MethodImplOptions.AggressiveInlining)]
		public void SetVertex (int index, Vertex vertex) {
			stream0[index] = vertex.position;
			//stream1[index] = vertex.normal;
			//stream2[index] = vertex.tangent;
			//stream3[index] = vertex.texCoord0;
		}
```

1.2 Missing Gizmos

Before we continue we have to make sure that `ProceduralMesh.OnDrawGizmos` can deal with missing vertex data. If the mesh doesn't have normals or tangents then those properties will return an array with a length of zero, which would lead to an error because we assume that they have the same length as the vertex position array.

If normals are missing then it makes most sense to not draw them either, even when configured to do so. We can do this by checking whether normals exist directly before we would retrieve the normals array, by invoking `HasVertexAttribute` on the mesh with `VertexAttribute.Normal` as an argument. Then we set whether we should draw normals to the result of that, and if we still should draw them, only then do we retrieve the array.

```
		if (drawNormals && normals == null) {
			drawNormals = mesh.HasVertexAttribute(VertexAttribute.Normal);
			if (drawNormals) {
				normals = mesh.normals;
			}
		}
```

Use the same approach for the tangents.

```
		if (drawTangents && tangents == null) {
			drawTangents = mesh.HasVertexAttribute(VertexAttribute.Tangent);
			if (drawTangents) {
				tangents = mesh.tangents;
			}
		}
```

This will work in all but one case. It goes wrong after a hot reload while play mode is paused, in which case `Update` isn't invoked immediately. Array fields that are `null` are serialized and deserialized by the editor as arrays with length zero, which will lead to the trouble we tried to avoid. We can prevent this serialization side-effect by instructing the editor to never store our arrays, by attaching the `System.NonSerialized` attribute to them. This makes sure that the fields are always reset to their default value, which is `null`.

```
	[System.NonSerialized]
	Vector3[] vertices, normals;

	[System.NonSerialized]
	Vector4[] tangents;
```

1.3 Position-Only Cube Sphere

Duplicate `CubeSphere` and rename to `SharedCubeSphere`.

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

Add it to the options of `ProceduralMesh`, using the `PositionStream`.

```
	static MeshJobScheduleDelegate[] jobs = {
		…
		MeshJob<CubeSphere, SingleStream>.ScheduleParallel,
		MeshJob<SharedCubeSphere, PositionStream>.ScheduleParallel,
		MeshJob<UVSphere, SingleStream>.ScheduleParallel
	};

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

Without any other changes this creates a variant cube sphere mesh that only contains position data. You can verify this by inspecting the generated mesh in the editor and also by noticing that only the *Cube Map* material works with it. Also, if you inspect the code generated by *Burst* you'll see that all code exclusively used for normals, tangents, and texture coordinates has been stripped. But let's also remove this code manually from `SharedCubeSphere.Execute`.

```
			var vertex = new Vertex();
			//vertex.tangent = float4(normalize(pB - pA), -1f);

			for (int v = 1; v <= Resolution; v++, vi += 4, ti += 2) {
				float3 pC = CubeToSphere(uA + side.vVector * v / Resolution);
				float3 pD = CubeToSphere(uB + side.vVector * v / Resolution);

				vertex.position = pA;
				//vertex.normal = normalize(cross(pC - pA, vertex.tangent.xyz));
				//vertex.texCoord0 = 0f;
				streams.SetVertex(vi + 0, vertex);

				vertex.position = pB;
				//vertex.normal = normalize(cross(pD - pB, vertex.tangent.xyz));
				//vertex.texCoord0 = float2(1f, 0f);
				streams.SetVertex(vi + 1, vertex);

				vertex.position = pC;
				//vertex.tangent.xyz = normalize(pD - pC);
				//vertex.normal = normalize(cross(pC - pA, vertex.tangent.xyz));
				//vertex.texCoord0 = float2(0f, 1f);
				streams.SetVertex(vi + 2, vertex);

				vertex.position = pD;
				//vertex.normal = pD;
				//vertex.normal = normalize(cross(pD - pB, vertex.tangent.xyz));
				//vertex.texCoord0 = 1f;
				streams.SetVertex(vi + 3, vertex);

				streams.SetTriangle(ti + 0, vi + int3(0, 2, 1));
				streams.SetTriangle(ti + 1, vi + int3(1, 2, 3));

				pA = pC;
				pB = pD;
			}
```

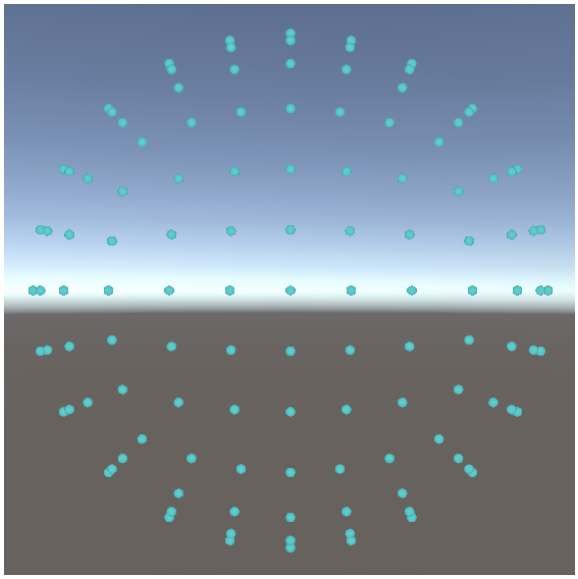
2 Vertices

Currently `SharedCubeSphere` still generates four separate vertices per quad. To share all vertices we have to get rid of a lot of them.

2.1 Eliminating Triangles

We'll initially exclusively focus on the vertices, leaving the triangles for later. But we still have to generate triangles, so we'll simply make them degenerate for now.

```
			streams.SetTriangle(ti + 0, 0);
			streams.SetTriangle(ti + 1, 0);
```



Only vertices of resolution 6 cube sphere.

2.2 Looping over Vertex Columns

When we consider a side of the cube sphere in isolation, what we need to do is similar to converting from `SquareGrid` to `SharedSquareGrid`. So instead of looping over a column of quads we'll change it so we loop over a column of vertices instead.

Change the loop so the position that is calculated first—quad vertex position C—is used for the single vertex position that we set. Then remove all other code that deals with positions and vertices A, B, and D.

```
		for (int v = 1; v <= Resolution; v++, vi += 4, ti += 2) {
			vertex.position = CubeToSphere(uA + side.vVector * v / Resolution);
			streams.SetVertex(vi, vertex);

			//float3 pD = CubeToSphere(uB + side.vVector * v / Resolution);

			//vertex.position = pA;
			//streams.SetVertex(vi + 0, vertex);

			//…

			streams.SetTriangle(ti + 0, 0);
			streams.SetTriangle(ti + 1, 0);

			//pA = pC;
			//pB = pD;
		}
```

Also remove the code that initializes `uB`, `pA`, and `pB` before the loop.

```
		float3 uA = side.uvOrigin + side.uVector * u / Resolution;
		//float3 uB = side.uvOrigin + side.uVector * (u + 1) / Resolution;
		//float3 pA = CubeToSphere(uA), pB = CubeToSphere(uB);
```

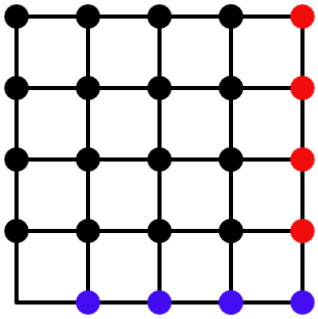
We need to keep `uA` because our vertex column is based on it, but let's rename it to `pStart` because it now represents the single starting position at the bottom of the vertex column.

```
		float3 pStart = side.uvOrigin + side.uVector * u / Resolution;

		var vertex = new Vertex();

		for (int v = 1; v <= Resolution; v++, vi += 4, ti += 2) {
			vertex.position = CubeToSphere(pStart + side.vVector * v / Resolution);
			streams.SetVertex(vi, vertex);

			streams.SetTriangle(ti + 0, 0);
			streams.SetTriangle(ti + 1, 0);
		}
```

At this point we already appear to generate an almost-complete cube sphere. Only the two minimum and maximum polar vertices are missing, leaving two small gaps. Besides that there could also be random vertices visible because we haven't set their positions.

2.3 Reducing the Vertex Count

Our current approach works because we're generating six grids of $r \times r$ vertices, with r being the resolution. These fit together such that the result is a cube with only two missing vertices: the polar vertices on the XYZ line.



First side, showing back (black), right (red), and bottom (blue) vertices.

This means that we can reduce the vertex count from $24r^2$ down to $6r^2 + 2$.

```
	public int VertexCount => 6 * Resolution * Resolution + 2;
```

Adjust the vertex offset at the start of `Execute` to match. We'll put the polar vertices at the beginning, so offset accordingly.

```
		int vi = Resolution * (Resolution * side.id + u) + 2;
```

Also, we should increment the vertex index only once per iteration, as we no longer generate separate quads.

```
		for (int v = 1; v <= Resolution; v++, vi++, ti += 2) { … }
```

2.4 Adding the Polar Vertices

The two polar vertices are simple: their positions are $\pm\sqrt{\frac{1}{3}}$. Rather than put the code for that in a separate method we'll add the vertices directly after creating the `Vertex` value, first the minimum and then the maximum pole. This must be done only once, so we'll do it when the job index is zero.

```
		var vertex = new Vertex();
		if (i == 0) {
			vertex.position = -sqrt(1f / 3f);
			streams.SetVertex(0, vertex);
			vertex.position = sqrt(1f / 3f);
			streams.SetVertex(1, vertex);
		}
```

How do you find the position of the poles?

The poles lie on the XYZ line, meaning that their coordinates are all the same. They also lie on the surface of the unit sphere, so $x^2 + y^2 + z^2 = 1$. As $x = y = z$ we have $3x^2 = 1 \rightarrow x^2 = \frac{1}{3} \rightarrow x = \pm\sqrt{\frac{1}{3}}$.

2.5 Extracting First Vertex of Column

The approach that we used for `SharedSquareGrid` extracted the first vertex from the loop, because otherwise we couldn't generate the quads correctly. We'll have to do the same this time as well. So add the first vertex before the loop, directly using `pStart`. To compensate we have to reduce the loop length by one.

```
		vertex.position = CubeToSphere(pStart);
		streams.SetVertex(vi, vertex);
		vi += 1;

		for (int v = 1; v < Resolution; v++, vi++, ti += 2) { … }
```

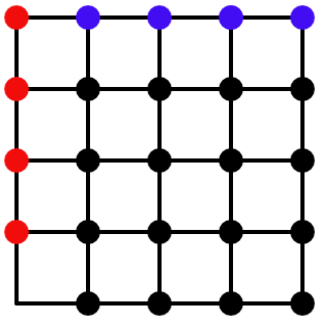
In this case this will result in missing quads, because the cube mesh vertex columns are one shorter than those of the flat grid. For now we'll simply compensate for this by adding an extra quad before the loop starts.

```
		vertex.position = CubeToSphere(pStart);
		streams.SetVertex(vi, vertex);

		streams.SetTriangle(ti + 0, 0);
		streams.SetTriangle(ti + 1, 0);
		vi += 1;
		ti += 2;

		for (int v = 1; v < Resolution; v++, vi++, ti += 2) { … }
```

These changes have shifted the V range of the vertex columns, as they now start at 0 instead of 1. This is easier to work with when stitching sides together later, but to keep the sides aligned properly we'll have to start U at 1 instead of 0. This is done by simply incrementing U after calculating the vertex and triangle indices and before calculating `pStart`.

```
		int u = i / 6;
		Side side = GetSide(i - 6 * u);
		int vi = Resolution * (Resolution * side.id + u) + 2;
		int ti = 2 * Resolution * (Resolution * side.id + u);

		u += 1;

		float3 pStart = side.uvOrigin + side.uVector * u / Resolution;
```

This changes the position of the vertex grid from the top left corner to the bottom right corner of each side.



First side, showing back (black), left (red), and top (blue) vertices.

3 Triangles

Our cube face has twelve edges where two sides meet and eight corners where three sides meet. When adding triangles we have to somehow stitch all these seams together. Thus we'll have to take care of multiple special cases, which we'll do one at a time.

3.1 First Triangle per Column

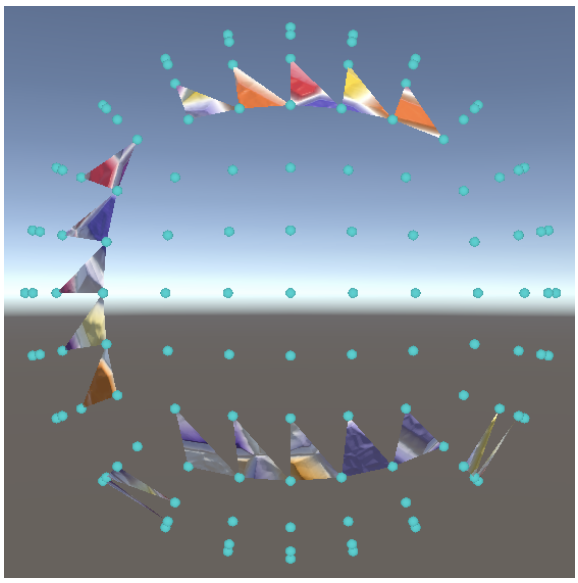
Let's begin with only showing the first triangle of each column, so only the first triangle of the bottoms quads of each side. As our vertex columns are aligned with the bottom of each side this is a simple case, except for the first quad.

When generating the first column of a side its first vertex is either borrowed from a different side or it is the minimum pole. So let's initially skip that first triangle, by checking whether we're working on the first column of a side, which is the case when U equals zero, before we increment it. Keep track of this fact via a boolean variable so we can easily check it later.

```
		int vi = Resolution * (Resolution * side.id + u) + 2;
		int ti = 2 * Resolution * (Resolution * side.id + u);
		bool firstColumn = u == 0;
		u += 1;
```

When it's the first column we keep the first triangle—the one added before the loop—degenerate. Otherwise we add the first triangle of the first quad, using the initial vertex index with offset $\begin{bmatrix} 0 \\ -r \\ -r+1 \end{bmatrix}$.

```
		if (firstColumn) {
			streams.SetTriangle(ti, 0);
		}
		else {
			streams.SetTriangle(ti, vi + int3(0, -Resolution, -Resolution + 1));
		}
		//streams.SetTriangle(ti + 0, 0);
		streams.SetTriangle(ti + 1, 0);
		vi += 1;
		ti += 2;
```



First triangles, with gaps.

3.2 Touching the Minimum Pole

To also include the first triangle of these rows we have to find the first two vertex indices of the side's column when U is zero. As we skip that column these vertices are found somewhere else in the mesh.

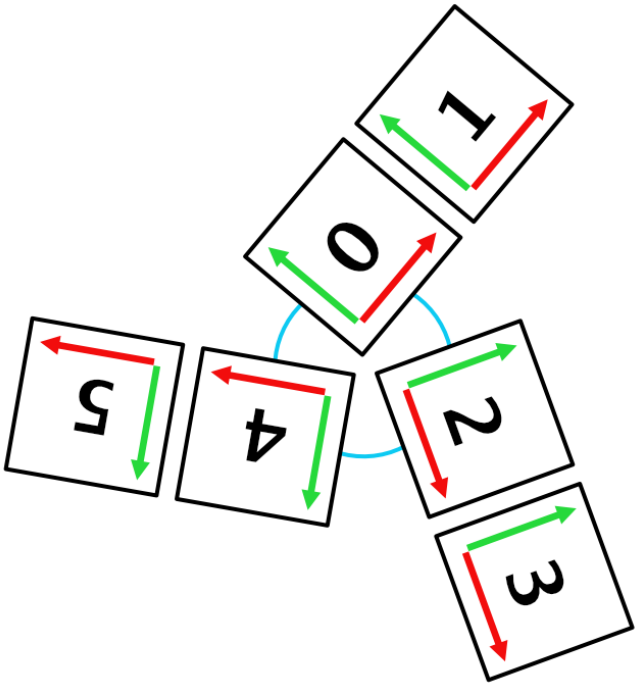
The easiest vertices to find are the minimum corner vertices of the sides that touch the minimum pole, as that's always the minimum pole vertex, so index zero. Those sides have identifiers 0, 2, and 4, so we can find out whether a sides touches the minimum pole by checking whether its identifier is even. Add a property for this to `Side`.

```
		struct Side {
			public int id;
			public float3 uvOrigin, uVector, vVector;

			public bool TouchesMinimumPole => (id & 1) == 0;
		}
```

Back in `Execute`, if we're in the first column and the side touches the minimum pole, then we know the first two indices of the first triangle: the vertex index and zero, with the third yet unknown. Otherwise we keep using a degenerate triangle for the first column.

```
			if (firstColumn) {
				if (side.TouchesMinimumPole) {
					streams.SetTriangle(ti, int3(vi, 0, 0));
				}
				else {
					streams.SetTriangle(ti, 0);
				}
			}
			else {
				streams.SetTriangle(ti, vi + int3(0, -Resolution, -Resolution + 1));
			}
```

To find the final vertex we have to go one step counterclockwise through the polar cube sphere side layout.



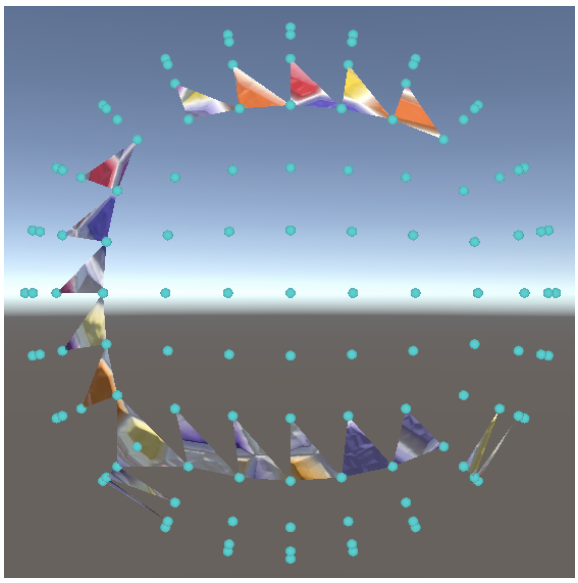
Cube sphere side layout.

Analyzing the layout tells use that for sides 2 and 4 we find the vertex with a relative offset of $-2r^2$ while for side 0 we need to wrap around and use $4r^2$ instead.

How are those offsets determined?

The required vertex for side 2 is the first vertex of side 0, so a negative offset of two sides. Each side has r^2 vertices. So the offset for side 2 is $-2r^2$ and likewise for side 4. In the case of side 0 we have to offset in the other direct to reach side 4, hence $4r^2$.

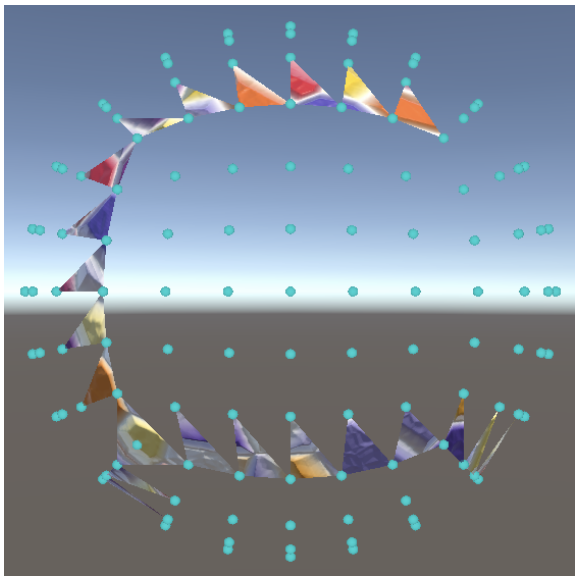
```
		if (side.TouchesMinimumPole) {
			streams.SetTriangle(ti, int3(
				vi,
				0,
				vi + (side.id == 0 ? 4 : -2) * Resolution * Resolution
			));
		}
```



With triangles touching the minimum pole.

We can also see that for the other sides we can simple jump back a single column as that wraps to the previous side.

```
			if (side.TouchesMinimumPole) {
				streams.SetTriangle(ti, int3(
					vi,
					0,
					vi + (side.id == 0 ? 4 : -2) * Resolution * Resolution
				));
			}
			else {
				streams.SetTriangle(ti, vi + int3(0, -Resolution, -Resolution + 1));
			}
```



All first triangles.

However, this goes wrong for the special case of using a resolution 1 cube sphere. In that case the third index offset has to be 4 for side 1 and −2 for sides 3 and 5.

```
			if (side.TouchesMinimumPole) {
				streams.SetTriangle(ti, int3(
					vi,
					0,
					vi + (side.id == 0 ? 4 : -2) * Resolution * Resolution
				));
			}
			else {
				streams.SetTriangle(ti, vi + int3(
					0,
					-Resolution,
					Resolution == 1 ? (side.id == 1 ? 4 : -2) : -Resolution + 1
				));
			}
```

3.3 Seam Step

Note that we're using the value 4 for sides 0 and 1 and the value −2 for all other sides when stepping across an edge seam. Let's define this as the seam-step value and add it to `Side`.

```
	struct Side {
		public int id;
		public float3 uvOrigin, uVector, vVector;
		public int seamStep;

		public bool TouchesMinimumPole => (id & 1) == 0;
	}
```

Initialize it in `GetSide`.

```
	static Side GetSide (int id) => id switch {
		0 => new Side {
			…,
			seamStep = 4
		},
		1 => new Side {
			…,
			seamStep = 4
		},
		2 => new Side {
			…,
			seamStep = -2
		},
		3 => new Side {
			…,
			seamStep = -2
		},
		4 => new Side {
			…,
			seamStep = -2
		},
		_ => new Side {
			…,
			seamStep = -2
		}
	};
```

Then use it in `Execute`.

```
		if (side.TouchesMinimumPole) {
			streams.SetTriangle(ti, int3(
				vi,
				0,
				vi + side.seamStep * Resolution * Resolution
			));
		}
		else {
			streams.SetTriangle(ti, vi + int3(
				0,
				-Resolution,
				Resolution == 1 ? side.seamStep : -Resolution + 1
			));
		}
```

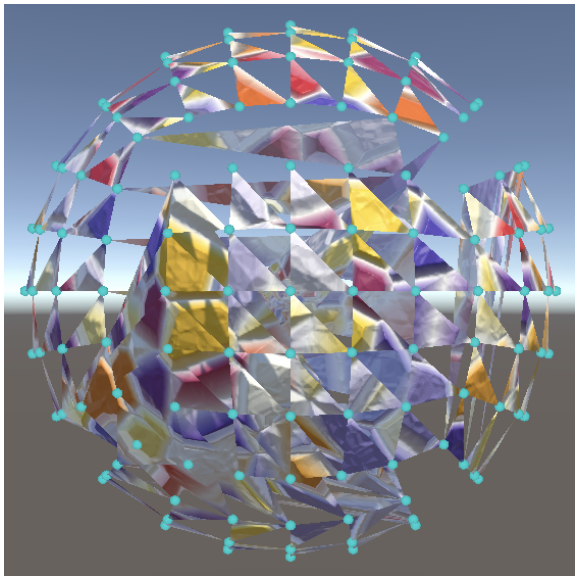
Let's also rewrite the triangle index code entirely, reducing it to a single expression that creates an `int3` triangle, using conditional operators instead of nested conditional blocks. Then we need to invoke `SetTriangle` only in one place.

```
		var triangle = int3(
			vi,
			firstColumn && side.TouchesMinimumPole ? 0 : vi - Resolution,
			vi + (firstColumn ?
				side.TouchesMinimumPole ?
					side.seamStep * Resolution * Resolution :
					Resolution == 1 ? side.seamStep : -Resolution + 1 :
				-Resolution + 1
			)
		);

		//if (firstColumn) { … }
		//else {
		//	streams.SetTriangle(ti, vi + int3(0, -Resolution, -Resolution + 1));
		//}
		streams.SetTriangle(ti, triangle);
		streams.SetTriangle(ti + 1, 0);
```

3.4 Closing the Seams

Now that we have the first triangle of each column, let's fill the entire column by simply incrementing that triangle each iteration and using that for the first part of the quad.

```
		for (int v = 1; v < Resolution; v++, vi++, ti += 2) {
			vertex.position = CubeToSphere(pStart + side.vVector * v / Resolution);
			streams.SetVertex(vi, vertex);

			triangle += 1;
			streams.SetTriangle(ti + 0, triangle);
			streams.SetTriangle(ti + 1, 0);
		}
```



Incrementing triangles, partially correct.

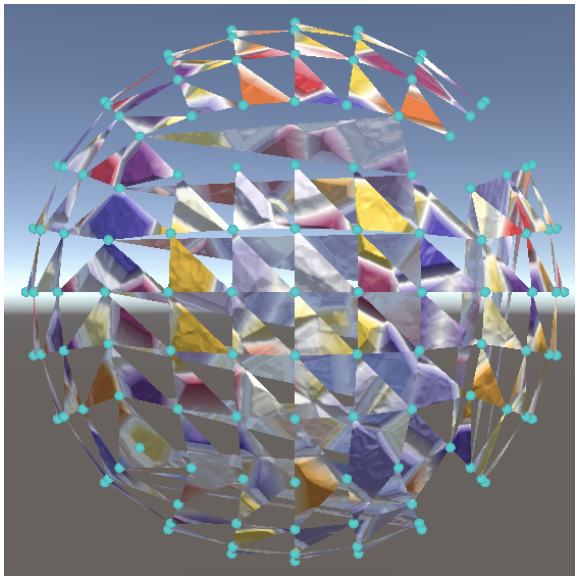
This works for most of the sphere but fails for multiple seams in between sides, so let's take care of those one by one.

Let's begin by looking at the first column of each side. It is always correct to increment the triangle's X index, because it corresponds to the vertex column that we're working through. But incrementing the Y index won't work when we start at the minimum pole. However, Y should always become equal to Z in the next iteration, so we can just assign Z to Y and it will always work.

```
			//triangle += 1;
			triangle.x += 1;
			triangle.y = triangle.z;
```

That leaves the Z index. Incrementing it works fine for the sides that don't touch the minimum pole. Those that do end up connected to sides with a different orientation. This means that for the first column when touching the minimum pole, Z should be incremented by r instead.

```
			triangle.x += 1;
			triangle.y = triangle.z;
			triangle.z += firstColumn && side.TouchesMinimumPole ? Resolution : 1;
```



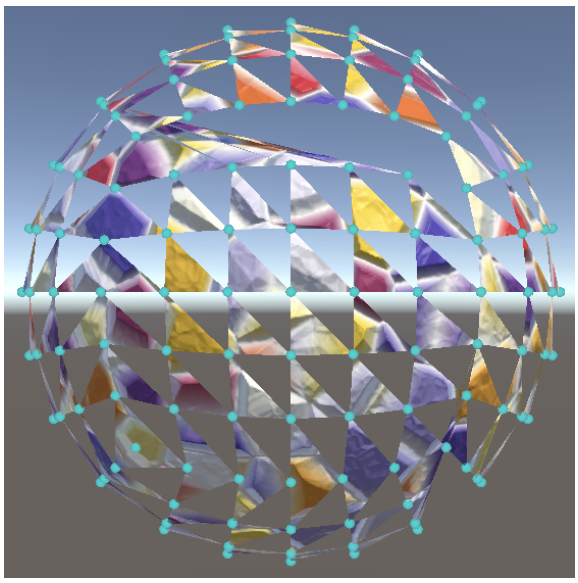
First columns correct.

Let's now look at the last rows—the last iteration of each column—when V equals $r - 1$. We'll again start by using conditional blocks, condensing it later. It already appears to work correctly for the first columns touching the minimum pole, so we begin with copying the current logic.

```
		if (v == Resolution - 1) {
			if (firstColumn && side.TouchesMinimumPole) {
				triangle.z += Resolution;
			}
			else {
				triangle.z += 1;
			}
		}
		else {
			triangle.z += firstColumn && side.TouchesMinimumPole ? Resolution : 1;
		}
```

Looking at the sides not touching the minimum pole and analyzing the layout of their relevant adjacent sides, it becomes clear that the correct vertex is found by adding $(s + 1)r^2 - ur + u$ to Z, where s is the seam step.

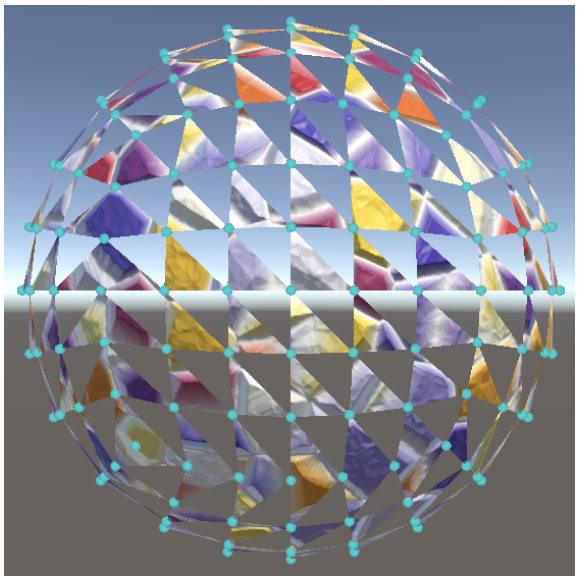
```
		if (firstColumn && side.TouchesMinimumPole) {
			triangle.z += Resolution;
		}
		else {
			triangle.z +=
				Resolution * ((side.seamStep + 1) * Resolution - u) + u;
		}
```



Half of the last rows correct.

But this is only correct when working neither on the first column nor on a side touching the minimum pole. In all remaining cases U isn't factored into the equation so we add $(s+1)r^2 - r + 1$ instead.

```
if (firstColumn && side.TouchesMinimumPole) {
	triangle.z += Resolution;
}
else if (!firstColumn && !side.TouchesMinimumPole) {
	triangle.z +=
		Resolution * ((side.seamStep + 1) * Resolution - u) + u;
}
else {
	triangle.z +=
		(side.seamStep + 1) * Resolution * Resolution -
		Resolution + 1;
}
```



All last rows correct.

At this point we have closed all seams and covered the entire sphere with half-quads. Let's now also condense the logic for incrementing Z, only keeping the last-iteration check a conditional block as it's the only portion that isn't constant for the entire loop.

```
			if (v == Resolution - 1) {
				triangle.z += firstColumn && side.TouchesMinimumPole ?
					Resolution :
					!firstColumn && !side.TouchesMinimumPole ?
						Resolution * ((side.seamStep + 1) * Resolution - u) + u :
						(side.seamStep + 1) * Resolution * Resolution -
						Resolution + 1;
			}
			else {
				triangle.z += firstColumn && side.TouchesMinimumPole ? Resolution : 1;
			}
```

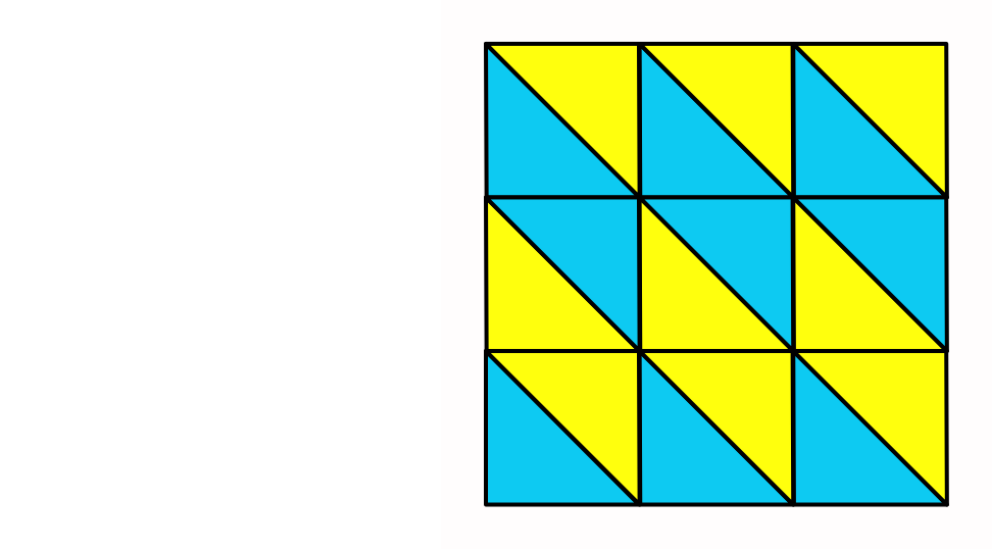
Although a big portion of this code is invariant and can be hoisted out of the loop, inspecting the compiled code shows that *Burst* has some trouble with this, due to the complexity and the combination of invariant and variable evaluation. We can help *Burst* by pulling the logic out of the loop ourselves.

```
		int zAdd = firstColumn && side.TouchesMinimumPole ? Resolution : 1;
		int zAddLast = firstColumn && side.TouchesMinimumPole ?
			Resolution :
			!firstColumn && !side.TouchesMinimumPole ?
				Resolution * ((side.seamStep + 1) * Resolution - u) + u :
				(side.seamStep + 1) * Resolution * Resolution - Resolution + 1;

		for (int v = 1; v < Resolution; v++, vi++, ti += 2) {
			vertex.position = CubeToSphere(pStart + side.vVector * v / Resolution);
			streams.SetVertex(vi, vertex);

			triangle.x += 1;
			triangle.y = triangle.z;
			triangle.z += v == Resolution - 1 ? zAddLast : zAdd;
			//if (v == Resolution - 1) { … }
			//else { … }
			streams.SetTriangle(ti + 0, triangle);
			streams.SetTriangle(ti + 1, 0);
		}
```

3.5 Filling the Gaps

All that's left at this point is to complete the quads, filling the gaps in the mesh. However, because of the nature of the seams we cannot simply create square quads. This is easiest to see for a resolution 1 cube sphere, as in that case all we have is seams.

The solution is to keep the fist and the last triangle of each column outside the loop, instead of the entire fist quad. The quads in between are then vertically sheared.



Sheared quad layout.

Begin by removing the second triangle that is added before the loop and instead adding one after it. Thus the triangle index is only incremented by 1 before the loop starts.

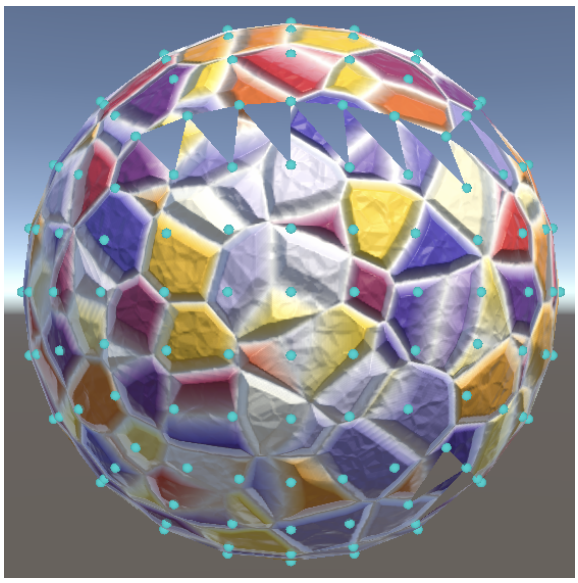
```
streams.SetTriangle(ti, triangle);
//streams.SetTriangle(ti + 1, 0);
vi += 1;
ti += 1;

int zAdd = …;
int …;

for (int v = 1; v < Resolution; v++, vi++, ti += 2) { … }

streams.SetTriangle(ti, 0);
```

Second, inside the loop it now makes sense to swap the order of the triangles, as what we're currently generating are the top triangles of each sheared quad.

```
triangle.z += v == Resolution - 1 ? zAddLast : zAdd;
streams.SetTriangle(ti + 0, 0);
streams.SetTriangle(ti + 1, triangle);
```

The bottom triangle of each quad can then be found relative to its top triangle.

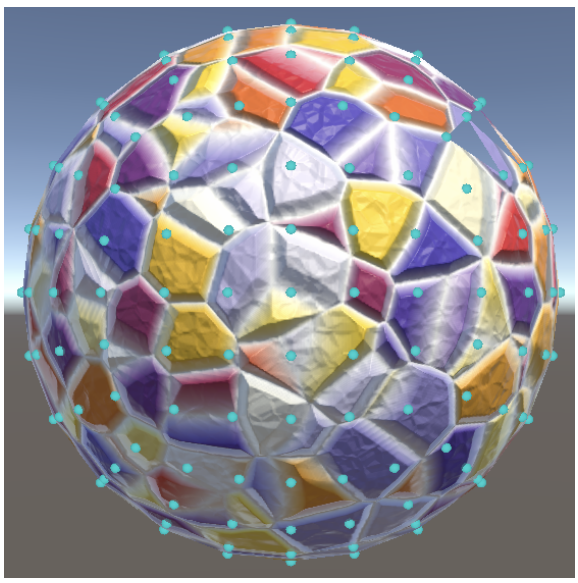
```
streams.SetTriangle(ti + 0, int3(triangle.x - 1, triangle.y, triangle.x));
streams.SetTriangle(ti + 1, triangle);
```



Filled all but the last rows.

The final step is to fill in the last triangle of each column. The first two triangle indices come from the previous triangle. The last index again depends on the side we're working on. If the side touches the minimum pole then it's the triangle Z index plus r, otherwise we'll initially use zero.

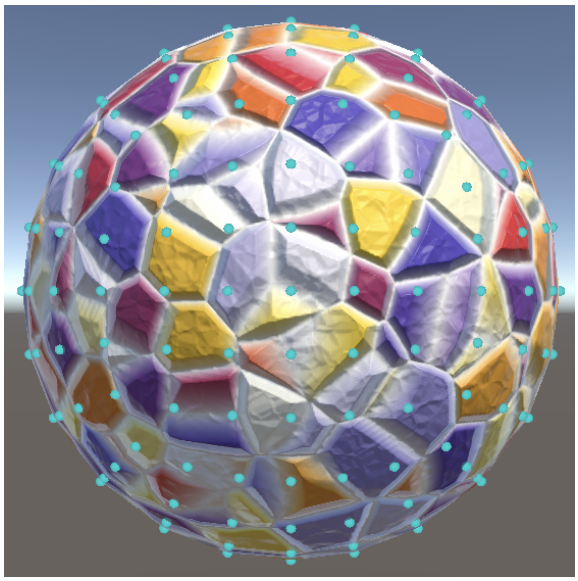
```
		for (int v = 1; v < Resolution; v++, vi++, ti += 2) { … }

		streams.SetTriangle(ti, int3(
			triangle.x,
			triangle.z,
			side.TouchesMinimumPole ? triangle.z + Resolution : 0
		));
```



Sides touching minimum pole are complete.

And for the remaining sides we have to add 1 instead of r, except when the resolution is 1, when we have to use the index of the maximum pole, which is 1.

```
		streams.SetTriangle(ti, int3(
			triangle.x,
			triangle.z,
			side.TouchesMinimumPole ?
				triangle.z + Resolution :
				u == Resolution ? 1 : triangle.z + 1
		));
```



Complete sphere.

4 Mesh Optimization

With our seamless cube sphere complete, let's take a short look at the way the mesh is constructed and whether it's possible to optimize this.

4.1 Showing Triangles

An interesting question related to performance is how the triangles of our mesh are organized, because that determines the order in which they are drawn. If meshes that share the same vertices are drawn quickly after each other the GPU can reuse cached vertices, which might improve performance. So let's use gizmos to visualize our triangles. Add an option for this to `ProceduralMesh.GizmoMode`.

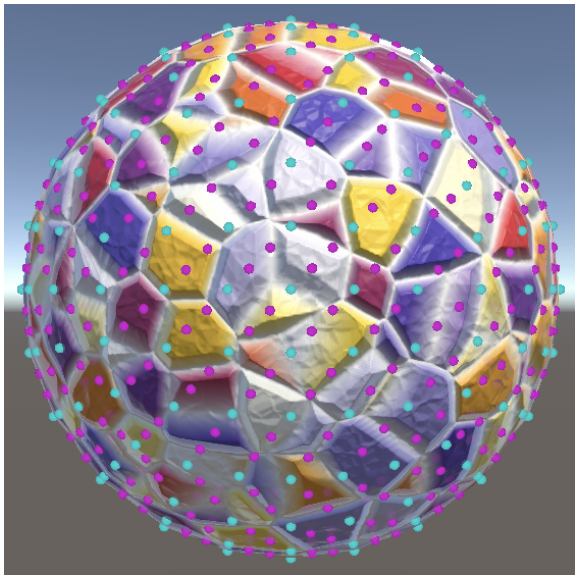
```
	public enum GizmoMode {
		Nothing = 0, Vertices = 1, Normals = 0b10, Tangents = 0b100, Triangles = 0b1000
	}
```

Add a field for the triangles and code to set and clear it, just like drawing vertices, except it requires an integer array.

```
	[System.NonSerialized]
	int[] triangles;

	…

	void OnDrawGizmos () {
		…
		bool drawTangents = (gizmos & GizmoMode.Tangents) != 0;
		bool drawTriangles = (gizmos & GizmoMode.Triangles) != 0;

		…
		if (drawTangents && tangents == null) {
			drawTangents = mesh.HasVertexAttribute(VertexAttribute.Tangent);
			if (drawTangents) {
				tangents = mesh.tangents;
			}
		}
		if (drawTriangles && triangles == null) {
			triangles = mesh.triangles;
		}

		…
	}

	…

	void Update () {
		…
		tangents = null;
		triangles = null;

		GetComponent<MeshRenderer>().material = materials[(int)material];
	}
```

The drawing of triangles is a little different because they are defined by three vertex indices. The triangles array contains these indices sequentially, so we'll work through these triplets in a separate loop. For visualization we use a magenta sphere placed at the average of the triangle's vertex positions.

```
	void OnDrawGizmos () {
		…

		if (drawTriangles) {
			Gizmos.color = Color.magenta;
			for (int i = 0; i < triangles.Length; i += 3) {
				Gizmos.DrawSphere(
					t.TransformPoint((
						vertices[triangles[i]] +
						vertices[triangles[i + 1]] +
						vertices[triangles[i + 2]]
					) * (1f / 3f)),
					0.02f
				);
			}
		}
	}
```

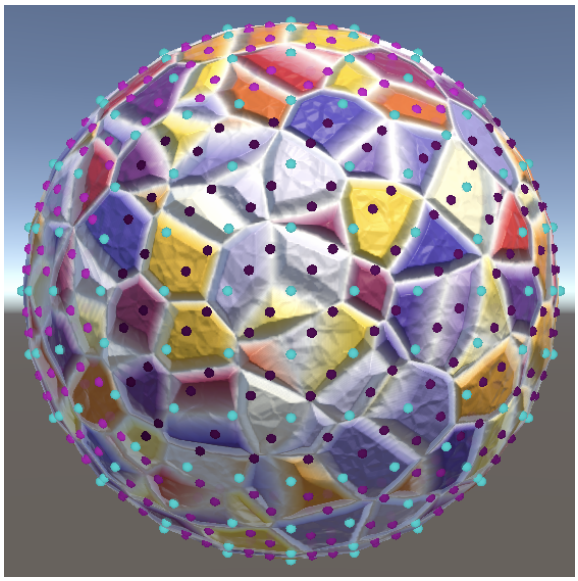


Showing average position of triangle vertices.

4.2 Triangle Draw Order

To visualize the draw order of the triangles we'll use a gradient going from black to magenta as we progress through the array.

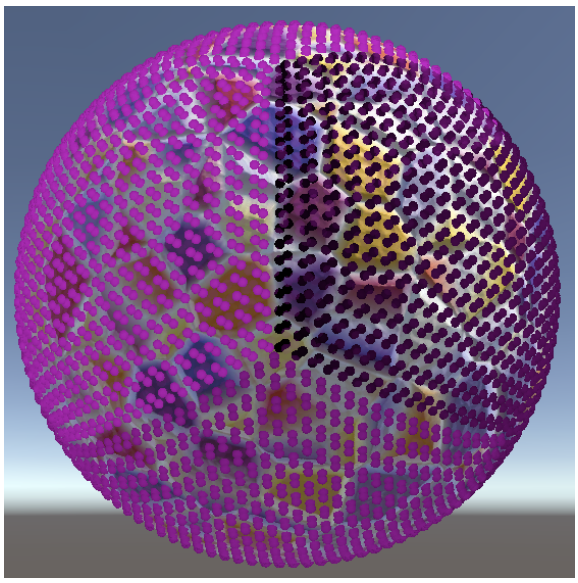
```
		if (drawTriangles) {
			//Gizmos.color = Color.magenta;
			float colorStep = 1f / (triangles.Length - 3);
			for (int i = 0; i < triangles.Length; i += 3) {
				float c = i * colorStep;
				Gizmos.color = new Color(c, 0f, c);
				Gizmos.DrawSphere(…);
			}
		}
```



Colored base on draw order.

4.3 Optimizing Draw Order

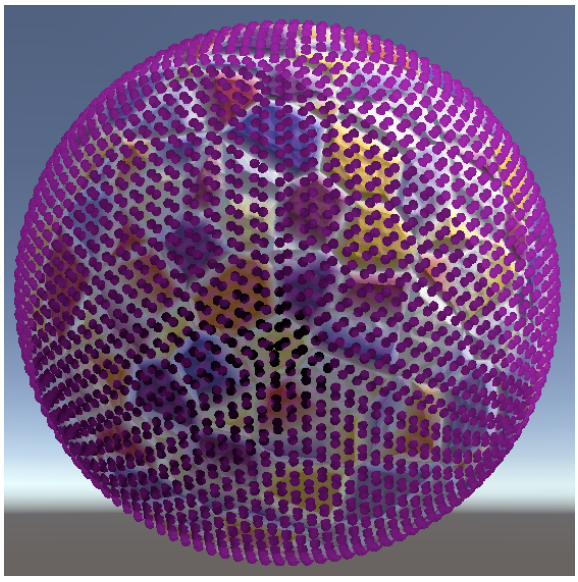
It is now possible to get a visual impression of the triangle draw order, which is easiest to see when only showing triangles with a fairly high resolution. In the case of our cube sphere it reveals how we generated the triangles in columns per side. Clear seams are visible especially near the minimum pole.



Triangle order around minimum pole.

Unity offers a way to automatically optimize a mesh, by invoking `Optimize` on it. Let's do so after generating it.

```
	void GenerateMesh () {
		Mesh.MeshDataArray meshDataArray = Mesh.AllocateWritableMeshData(1);
		Mesh.MeshData meshData = meshDataArray[0];

		jobs[(int)meshType](mesh, meshData, resolution, default).Complete();

		Mesh.ApplyAndDisposeWritableMeshData(meshDataArray, mesh);

		mesh.Optimize();
	}
```



Optimized triangle order.

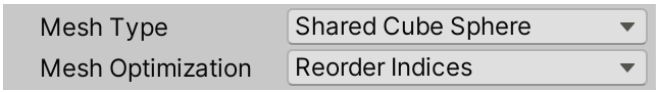
This makes a big difference for our seamless cube sphere, at least visually. Unity's optimization algorithm decides to smooth out the region near the minimum pole a lot. Whether this has a significant impact on rendering performance is something that has to be tested though. As our seamless cube sphere has very little vertex data and the *Cube Map* material has a simple vertex shader stage the impact of optimization will likely be small, but it's a potentially useful option to have available.

If you also analyze the other meshes you'll find that optimization won't make a difference, because we already generate triangles in an efficient order.

4.4 Configurable Optimization

Invoking `Optimize` doesn't guarantee better performance and it cannot be done as part of a job, so it's not something that should always be done by default. Also, there are a few different ways in which Unity can optimize a mesh. Invoking `Optimize` allows Unity to both reorder vertex data and triangle indices, but it's also possible to do only one of those. Let's make this configurable by adding a new `MeshOptimizationMode` enum type and configuration field that allows enabling reordering of indices and vertices independently.

```
	[System.Flags]
	public enum MeshOptimizationMode {
		Nothing = 0, ReorderIndices = 1, ReorderVertices = 0b10
	}

	[SerializeField]
	MeshOptimizationMode meshOptimization;
```



Optimization set to reorder indices only.

In `GenerateMesh`, if only indices should be reordered invoke `OptimizeIndexBuffers`. Otherwise, if only vertices should be reordered invoke `OptimizeReorderVertexBuffer`. Otherwise, unless optimization is disabled invoke `Optimize` as before to do both.

```
		if (meshOptimization == MeshOptimizationMode.ReorderIndices) {
			mesh.OptimizeIndexBuffers();
		}
		else if (meshOptimization == MeshOptimizationMode.ReorderVertices) {
			mesh.OptimizeReorderVertexBuffer();
		}
		else if (meshOptimization != MeshOptimizationMode.Nothing) {
			mesh.Optimize();
		}
```

The next tutorial is Octasphere.

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick