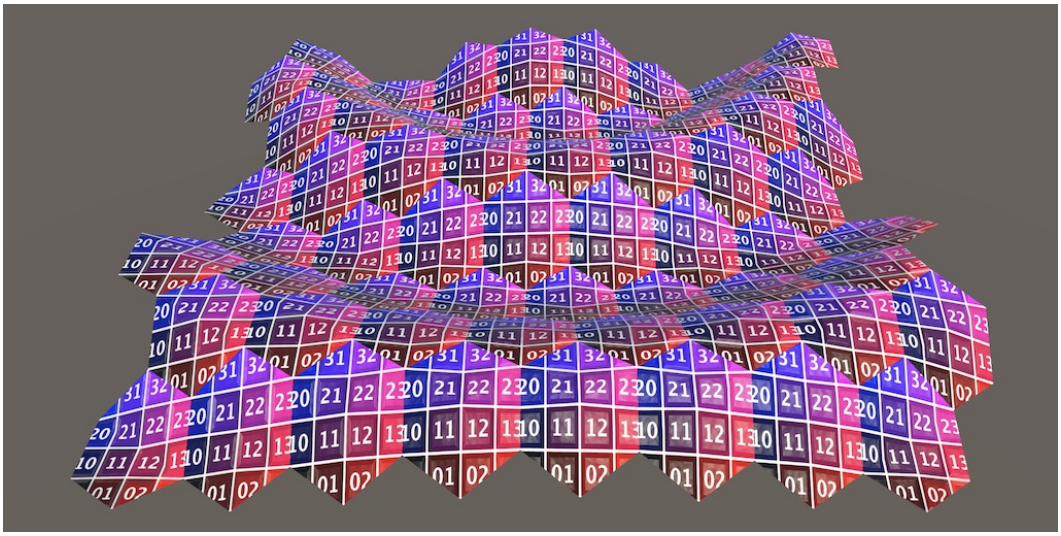
published 2021-12-18

Triangle Grid Triangles and Hexagons

Transition from right triangles to equilateral triangles.
Make a shared triangle grid.
Create two variants hexagon grids, with pointy and flat layout.

This is the fourth tutorial in a series about procedural meshes. It introduces a grid based on equilateral triangles, then moves on to hexagons.

This tutorial is made with Unity 2020.3.23f1.



A rippling pointy hexagon grid made with equilateral triangles.

1 Equilateral Triangles

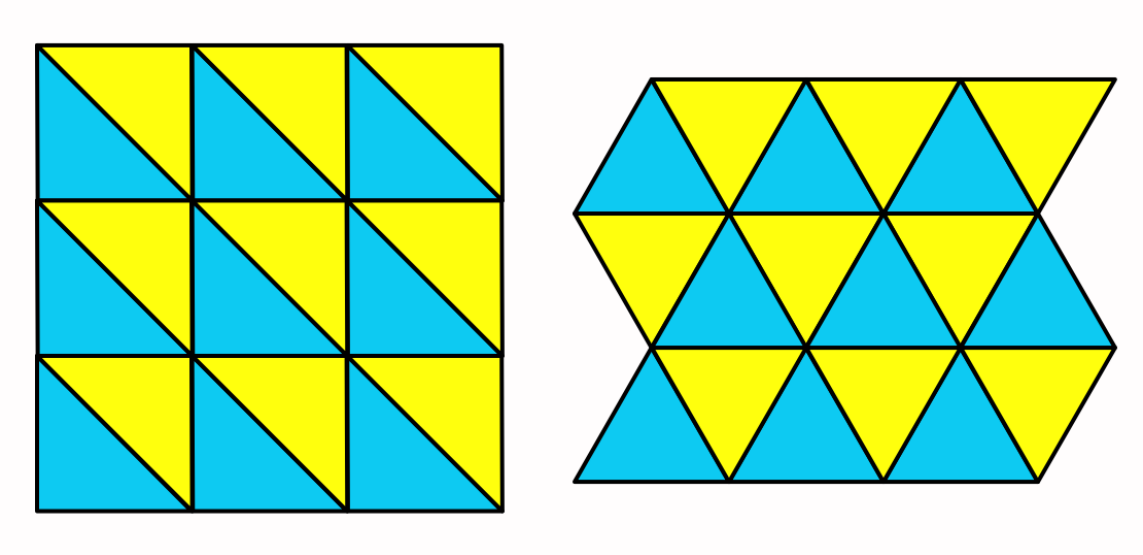
Although our square grids are conceptually made with squares, each quad consists of two right triangles. This becomes obvious when the grid is rippling. In case of a low grid resolution the ripples might look good or bad, depending on their orientation. For example, consider a resolution 10 grid with ripple period 3 and its origin set to 1, 0, 1.

Resolution 10 with ripple period 3 and origin 1, 0, 1.

This looks quite good considering the resolution. However, if the ripple origin's X coordinate is set to −1 it looks much worse.

Ripple origin −1, 0, 1.

The difference is caused by the shape of the triangles. In the first case the ripple's waves are mostly aligned with the hypotenuse of the right triangles, while in the second case they are mostly orthogonal to it. This stark directional difference could be eliminated by switching to a grid that consists of equilateral triangles.



Grids made with right and with equilateral triangles.

A triangle grid contains triangles with two orientations—pointing either up or down in our cause—so we'll always include the same amount of both, not preferring one over the other. We do this by treating a pair of both kind of triangle as a single unit. So a resolution 3 grid would contain 18 triangles, the same amount as a square grid.

1.1 Shared Triangle Grid

We're going to create a triangle grid with shared vertices, by duplicating and adjusting `SharedSquareGrid`, naming it `SharedTriangleGrid`.

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

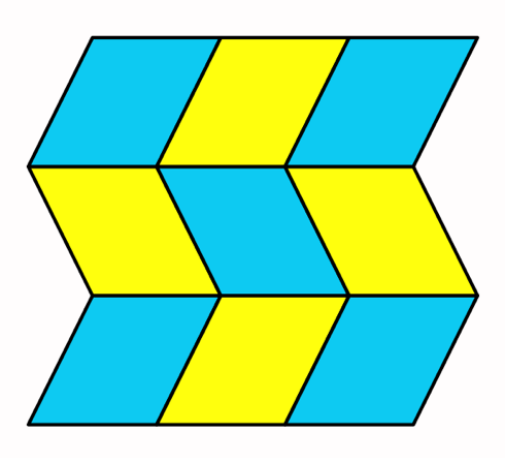
Add it to the jobs array and enum in `ProceduralMesh` so we can visualize it.

```
	static MeshJobScheduleDelegate[] jobs = {
		MeshJob<SquareGrid, SingleStream>.ScheduleParallel,
		MeshJob<SharedSquareGrid, SingleStream>.ScheduleParallel,
		MeshJob<SharedTriangleGrid, SingleStream>.ScheduleParallel
	};

	public enum MeshType {
		SquareGrid, SharedSquareGrid, SharedTriangleGrid
	};
```

I'm not going to include a triangle grid without shared vertices, because textures are almost never applied separately to individual triangles.

1.2 Rhombuses

By comparing the square and triangle grid layout it seems like we can convert one to the other by shifting the vertex rows sideways in alternating directions. If that is all that we did the result would be a rhombus grid.



Rhombus grid.

Let's start with this conversion from squares to rhombuses. `SharedTriangleGrid.Execute` currently uses a fixed offset of −0.5 for all X coordinates to keep the grid centered on the origin. As this offset will now differ per row turn it into a variable.

```
		int vi = (Resolution + 1) * z, ti = 2 * Resolution * (z - 1);

		float xOffset = -0.5f;

		…

		vertex.position.x = xOffset;
		…

		for (int x = 1; x <= Resolution; x++, vi++, ti += 2) {
			vertex.position.x = (float)x / Resolution + xOffset;
			…
		}
```

To ultimately form equilateral triangles we have to shift alternating rows half a triangle relative to each other. To keep the grid centered we'll shift even vertex rows by −0.25 and odd vertex rows by 0.25. Then we divide by the resolution and apply the −0.5 offset to keep the grid centered on the origin.

```
		float xOffset = -0.25f;

		if ((z & 1) == 1) {
			xOffset = 0.25f;
		}

		xOffset = xOffset / Resolution - 0.5f;
```

Resolution 3 rhombus grid.

The resulting rhombus grid also has slanted texture coordinates. We have to compensate for the shift to keep the U coordinates aligned, by adding an offset to them.

As the texture must stretch to cover the entire grid, in the case of even vertex rows the offset is zero. The U offset for odds rows must be half a triangle's width, but we also have to compensate for the fact that our grid has become a bit wider, by half a triangle. So for odd rows the 0.5 U offset has to be divided by the resolution plus 0.5. We also have to apply the same division when setting the U coordinates per vertex.

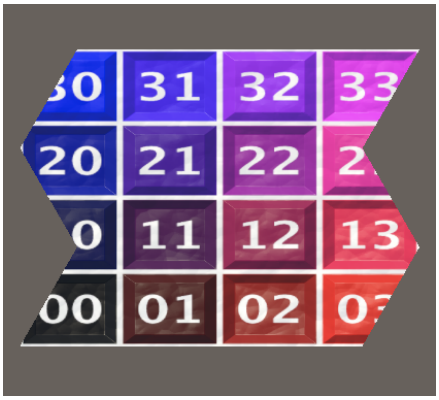
```
		float xOffset = -0.25f;
		float uOffset = 0f;

		if ((z & 1) == 1) {
			xOffset = 0.25f;
			uOffset = 0.5f / (Resolution + 0.5f);
		}

		…
		vertex.texCoord0.x = uOffset;
		vertex.texCoord0.y = (float)z / Resolution;
		…

		for (int x = 1; x <= Resolution; x++, vi++, ti += 2) {
			vertex.position.x = (float)x / Resolution + xOffset;
			vertex.texCoord0.x = x / (Resolution + 0.5f) + uOffset;
			…
		}
```

Aligned U texture coordinates.

1.3 Triangle Height

The next step is to make the triangles equilateral, which is done by reducing the height of the grid. The height of an equilateral triangle is equal to $\frac{\sqrt{3}}{2}$ times its edge length, so we have to multiply the Z coordinates with that.

```
vertex.position.z = ((float)z / Resolution - 0.5f) * sqrt(3f) / 2f;
```



Correct grid height.

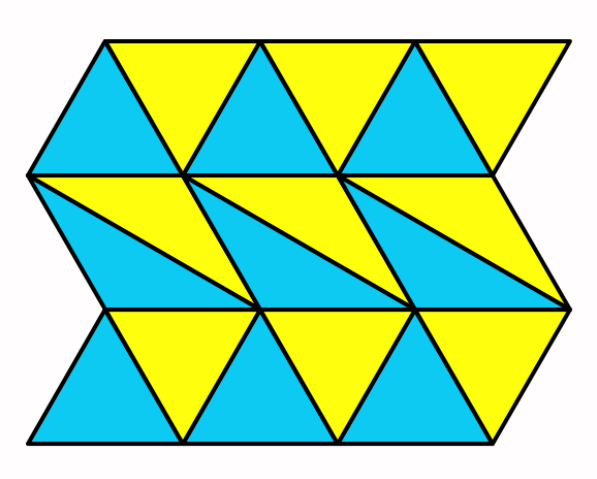
The triangles are now correct and the texture is stretched to exactly cover the grid. However, because the grid doesn't cover a square area the texture has been deformed. As the grid's height is smaller that its width we fix this by scaling down the texture vertically. We do this by making the V coordinate equal to the Z coordinate, divided by the grid's width, with a final 0.5 offset to keep the texture centered: $v = \frac{z}{1 + \frac{1}{2r}} + \frac{1}{2}$.

```
vertex.texCoord0.y = vertex.position.z / (1f + 0.5f / Resolution) + 0.5f;
```

Correct texture coordinates.

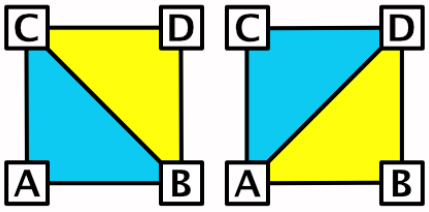
1.4 Alternating Triangle Orientation

Although the vertices and texture coordinates are correct, we aren't done yet. Currently the even triangle rows—those below even vertex rows—are malformed.



Incorrect triangles.

This happens because we use the same quad-based triangle layout everywhere: ACB and BCD. We have to use a different layout for triangles below even vertex rows: ACD and ADB.

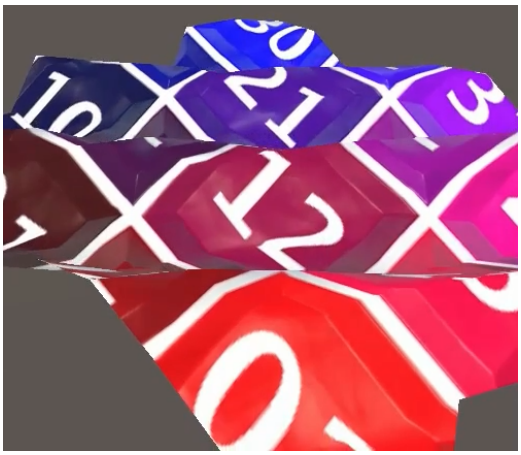


Two triangle orientations.

To make this possible introduce variables for the two triangle vertex offset triplets and set them appropriately. I also used variables for the individual index offsets to make this easier.

```
		int iA = -Resolution - 2, iB = -Resolution - 1, iC = -1, iD = 0;
		var tA = int3(iA, iC, iD);
		var tB = int3(iA, iD, iB);

		if ((z & 1) == 1) {
			xOffset = 0.25f;
			uOffset = 0.5f / (Resolution + 0.5f);
			tA = int3(iA, iC, iB);
			tB = int3(iB, iC, iD);
		}

		…

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

			if (z > 0) {
				streams.SetTriangle(ti + 0, vi + tA);
				streams.SetTriangle(ti + 1, vi + tB);
			}
		}
```

Our finished triangle grid has much more consistent ripple quality than our square grid.



Resolution 10 ripple origin −1, 0, 1.

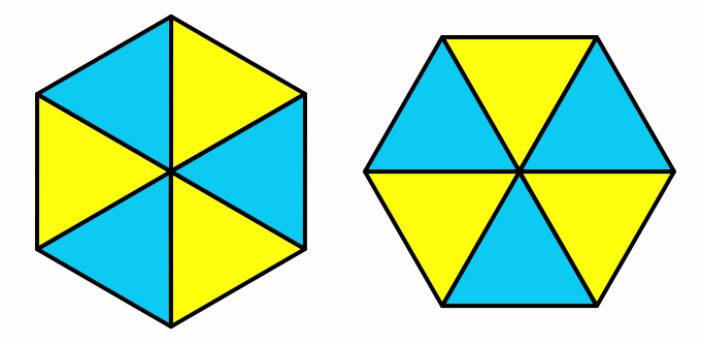
1.5 Bounds

Finally, because the aspect ratio of the area covered by our grid has changed, we also have to adjust its bounds to match. As we determined earlier, the grid's width is $1 + \frac{1}{2r}$ and its height is $\frac{\sqrt{3}}{2}$.

```
	public Bounds Bounds => new Bounds(
		Vector3.zero, new Vector3(1f + 0.5f / Resolution, 0f, sqrt(3f) / 2f)
	);
```

2 Hexagons

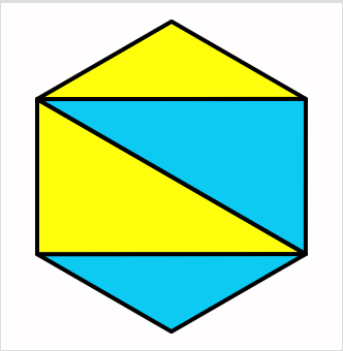
Individual triangles aren't often used as tiles, it is much more common to use hexagon tiles. So instead of a triangle grid without shared vertices we'll make a non-shared hexagon grid instead. Hexagons can be made with six equilateral triangles and are laid out in one of two orientations, usually described as either pointy or flat, matching the shape of each hexagon's top and bottom.



Pointy and flat hexagons.

Can't hexagons be made with fewer triangles?

Yes, they can be made with only four triangles each, in various layouts. However, if the grid isn't flat then six equilateral triangles provide the best quality, equivalent to that of a triangle grid.



A four-triangle hexagon.

2.1 Pointy Hexagon Grid

We start with the pointy hexagon grid variant, by duplicating `SquareGrid` and renaming it to `PointyHexagonGrid`. Adjust it to use seven vertices and six triangles per hexagon. Also remove all code from the loop in `Execute`.

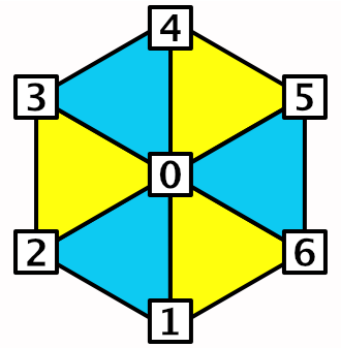
```
	public struct PointyHexagonGrid : IMeshGenerator {

		public Bounds Bounds => new Bounds(
			Vector3.zero, new Vector3(1f, 0f, 1f)
		);

		public int VertexCount => 7 * Resolution * Resolution;

		public int IndexCount => 18 * Resolution * Resolution;

		…

		public void Execute<S> (int z, S streams) where S : struct, IMeshStreams {
			int vi = 7 * Resolution * z, ti = 6 * Resolution * z;

			for (int x = 0; x < Resolution; x++, vi += 7, ti += 6) {
				// …
			}
		}
	}
```

Add it to `ProceduralMesh` so we can visualize it once the grid is finished.

```
	static MeshJobScheduleDelegate[] jobs = {
		MeshJob<SquareGrid, SingleStream>.ScheduleParallel,
		MeshJob<SharedSquareGrid, SingleStream>.ScheduleParallel,
		MeshJob<SharedTriangleGrid, SingleStream>.ScheduleParallel,
		MeshJob<PointyHexagonGrid, SingleStream>.ScheduleParallel
	};

	public enum MeshType {
		SquareGrid, SharedSquareGrid, SharedTriangleGrid, PointyHexagonGrid
	};
```

2.2 Seven Vertices

Each hexagon has seven vertices. We start with its central vertex, then its bottom vertex, and continue with its other vertices clockwise around the hexagon.



Seven vertices.

Start with creating a vertex and setting the central one in the loop inside `PointyHexagonGrid.Execute`. We ignore texture coordinates for now and also focus on a single hexagon initially. So the center vertex position is simply zero.

```
		for (int x = 0; x < Resolution; x++, vi += 7, ti += 6) {
			var vertex = new Vertex();
			vertex.normal.y = 1f;
			vertex.tangent.xw = float2(1f, -1f);

			streams.SetVertex(vi + 0, vertex);
		}
```

There are three different X coordinates per hexagon. Besides the central vertex 0, vertices 2 and 3 are on its left side and vertices 5 and 6 are on its right side. Their X coordinates are offset by the height of the equilateral triangle in both directions, which would be $\frac{\sqrt{3}}{2}$. However, to keep the hexagon grid about the same size as a square grid we'll halve the triangle size, so their height becomes $\frac{\sqrt{3}}{4}$ instead, divided by the resolution. Store the two extra X coordinates in a variable at the start of the loop.

```
		float h = sqrt(3f) / 4f;

		for (int x = 0; x < Resolution; x++, vi += 7, ti += 6) {
			var xCoordinates = float2(-h, h) / Resolution;

			…
		}
```

Do the same for the Z coordinates. In this case there are four extra besides the center: 0.5 and 0.25 offsets in both directions.

```
			var xCoordinates = float2(-h, h) / Resolution;
			var zCoordinates = float4(-0.5f, -0.25f, 0.25f, 0.5f) / Resolution;
```

Use these coordinates to add the other six vertices.

```
		streams.SetVertex(vi + 0, vertex);

		vertex.position.z = zCoordinates.x;
		streams.SetVertex(vi + 1, vertex);

		vertex.position.x = xCoordinates.x;
		vertex.position.z = zCoordinates.y;
		streams.SetVertex(vi + 2, vertex);

		vertex.position.z = zCoordinates.z;
		streams.SetVertex(vi + 3, vertex);

		vertex.position.x = 0f;
		vertex.position.z = zCoordinates.w;
		streams.SetVertex(vi + 4, vertex);

		vertex.position.x = xCoordinates.y;
		vertex.position.z = zCoordinates.z;
		streams.SetVertex(vi + 5, vertex);

		vertex.position.z = zCoordinates.y;
		streams.SetVertex(vi + 6, vertex);
```

Now let's set the texture coordinates. The hexagon is one unit high and two triangle heights wide. Use this to set the texture coordinates so it is centered on the hexagon and doesn't get deformed. Thus the V coordinates cover the entire 0–1 range while the U coordinates go from $\frac{1}{2} - h$ to $\frac{1}{2} + h$.

```
		vertex.texCoord0 = 0.5f;
		streams.SetVertex(vi + 0, vertex);

		vertex.position.z = zCoordinates.x;
		vertex.texCoord0.y = 0f;
		streams.SetVertex(vi + 1, vertex);

		vertex.position.x = xCoordinates.x;
		vertex.position.z = zCoordinates.y;
		vertex.texCoord0 = float2(0.5f - h, 0.25f);
		streams.SetVertex(vi + 2, vertex);

		vertex.position.z = zCoordinates.z;
		vertex.texCoord0.y = 0.75f;
		streams.SetVertex(vi + 3, vertex);

		vertex.position.x = 0f;
		vertex.position.z = zCoordinates.w;
		vertex.texCoord0 = float2(0.5f, 1f);
		streams.SetVertex(vi + 4, vertex);

		vertex.position.x = xCoordinates.y;
		vertex.position.z = zCoordinates.z;
		vertex.texCoord0 = float2(0.5f + h, 0.75f);
		streams.SetVertex(vi + 5, vertex);

		vertex.position.z = zCoordinates.y;
		vertex.texCoord0.y = 0.25f;
		streams.SetVertex(vi + 6, vertex);
```

2.3 Six Triangles

With the vertices set, add the six triangles. Like with the vertices, we start at the bottom and go in clockwise direction. So the offset triplets are 012, 023, 034, 045, 056, and 061.

```
		streams.SetVertex(vi + 6, vertex);

		streams.SetTriangle(ti + 0, vi + int3(0, 1, 2));
		streams.SetTriangle(ti + 1, vi + int3(0, 2, 3));
		streams.SetTriangle(ti + 2, vi + int3(0, 3, 4));
		streams.SetTriangle(ti + 3, vi + int3(0, 4, 5));
		streams.SetTriangle(ti + 4, vi + int3(0, 5, 6));
		streams.SetTriangle(ti + 5, vi + int3(0, 6, 1));
```

A single hexagon.

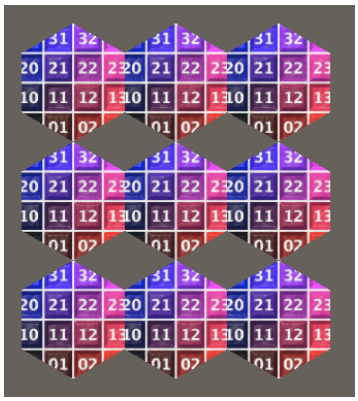
2.4 A Full Grid

To create an entire grid of hexagons we have to adjust the position of the center of each vertex. Each hexagon has a width of two triangle heights and a height of 1, both divided by the resolution. So begin with using that for the center, stored in a new variable at the start of the loop, then make the other X and Z coordinates relative to it.

```
		var center = float2(2f * h * x, z) / Resolution;
		var xCoordinates = center.x + float2(-h, h) / Resolution;
		var zCoordinates =
			center.y + float4(-0.5f, -0.25f, 0.25f, 0.5f) / Resolution;
```

Also use the center position for the first vertex and the center's X coordinate for the fifth vertex, instead of zero.

```
		vertex.position.xz = center;
		vertex.texCoord0 = 0.5f;
		streams.SetVertex(vi + 0, vertex);

		…

		vertex.position.x = center.x;
		vertex.position.z = zCoordinates.w;
		vertex.texCoord0 = float2(0.5f, 1f);
		streams.SetVertex(vi + 4, vertex);
```

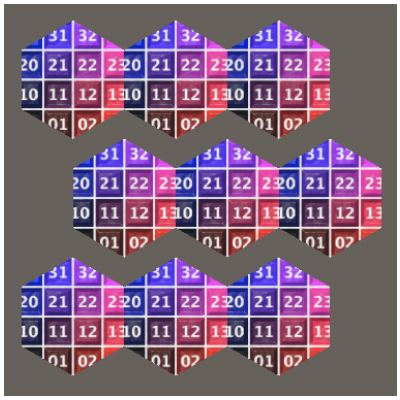


Resolution 3 grid with gaps.

This gives us a grid with gaps between rows that isn't centered yet. To close the gaps we have to first horizontally offset the rows in alternating directions. Once again we'll move even rows to the left and odd rows to the right.

In this case a resolution 1 grid needs no offset, because our hexagons are centered on zero. In all other cases the offset is equal to half the triangle height in both directions. And to center the grid we have to also subtract one less than the resolution. And all that is scaled by the triangles height. So that's $\pm\frac{1}{2}h - (r-1)h$ which can be simplified to $\left(\frac{1}{2} - r\right)h$ and $\left(\frac{3}{2} - r\right)h$. Apply this offset before dividing by the resolution in the loop.

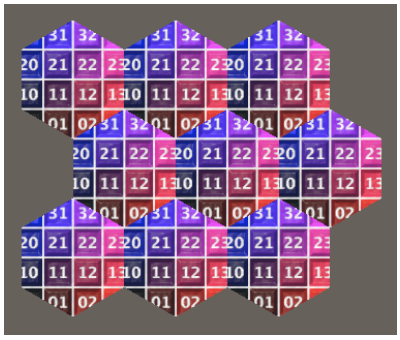
```
		float2 centerOffset = 0f;

		if (Resolution > 1) {
			centerOffset.x = (((z & 1) == 0 ? 0.5f : 1.5f) - Resolution) * h;
		}

		for (int x = 0; x < Resolution; x++, vi += 7, ti += 6) {
			var center = (float2(2f * h * x, z) + centerOffset) / Resolution;
			…
		}
```



Grid with X offsets.

Now we have make the rows fit vertically by reducing the distance between rows to 0.75, before the resolution division. And to keep the grid centered on the origin the center Z offset must become $-\frac{3}{8}(r-1)$ if the resolution is greater than 1.

```
		if (Resolution > 1) {
			centerOffset.x = (((z & 1) == 0 ? 0.5f : 1.5f) - Resolution) * h;
			centerOffset.y = -0.375f * (Resolution - 1);
		}

		for (int x = 0; x < Resolution; x++, vi += 7, ti += 6) {
			var center = (float2(2f * h * x, 0.75f * z) + centerOffset) / Resolution;
			…
		}
```



Grid without gaps.

2.5 Bounds

Finally, we have to adjust the bounds to match the area covered by the grid. Its width is equal to $(2r+1)\frac{\sqrt{3}}{4r} = \left(\frac{1}{2} + \frac{1}{4r}\right)\sqrt{3}$ if the resolution is greater than 1 and $\frac{1}{2}\sqrt{3}$ otherwise. And its height is equal to $\frac{3r+1}{4r} = \frac{3}{4} + \frac{1}{4r}$.

```
		public Bounds Bounds => new Bounds(Vector3.zero, new Vector3(
			(Resolution > 1 ? 0.5f + 0.25f / Resolution : 0.5f) * sqrt(3f),
			0f,
			0.75f + 0.25f / Resolution
		));
```

2.6 Flat Hexagon Grid

We wrap up by also including the flat variant of the hexagon grid. Duplicate `PointyHexagonGrid` and rename it to `FlatHexagonGrid`. Then add it to `ProceduralMesh` like before, for which I won't show the code changes.

The simplest way to create a flat grid is to swap the X and Z dimensions of a pointy grid. Begin by doing this for the bounds.

```
	public struct FlatHexagonGrid : IMeshGenerator {

		public Bounds Bounds => new Bounds(Vector3.zero, new Vector3(
			0.75f + 0.25f / Resolution,
			0f,
			(Resolution > 1 ? 0.5f + 0.25f / Resolution : 0.5f) * sqrt(3f)
		));

		…
	}
```

Then restructure `Execute` such that it processes X rows instead of Z rows, again by swapping the logic for X and Z. First change the offsets and coordinates variables. We also have to swap the signs of the now Z coordinate offsets, otherwise the winding order of the triangles would reverse and the grid would be visible from the wrong side.

```
	public void Execute<S> (int x, S streams) where S : struct, IMeshStreams {
		int vi = 7 * Resolution * x, ti = 6 * Resolution * x;

		float h = sqrt(3f) / 4f;

		float2 centerOffset = 0f;

		if (Resolution > 1) {
			centerOffset.x = -0.375f * (Resolution - 1);
			centerOffset.y = (((x & 1) == 0 ? 0.5f : 1.5f) - Resolution) * h;
		}

		for (int z = 0; z < Resolution; z++, vi += 7, ti += 6) {
			var center = (float2(0.75f * x, 2f * h * z) + centerOffset) / Resolution;
			var xCoordinates =
				center.x + float4(-0.5f, -0.25f, 0.25f, 0.5f) / Resolution;
			var zCoordinates = center.y + float2(h, -h) / Resolution;

			…
		}
	}
```

Vertex 1 now needs to have its X and U coordinates set, instead of Y and V.

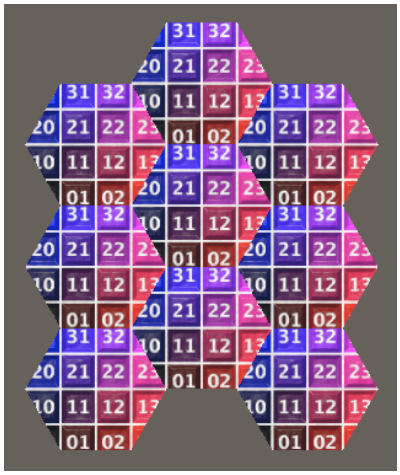
```
			vertex.position.x = xCoordinates.x;
			vertex.texCoord0.x = 0f;
			streams.SetVertex(vi + 1, vertex);
```

Vertex 2 also needs to get the appropriate position and its texture coordinates have to be adjusted to match its new position.

```
			vertex.position.x = xCoordinates.y;
			vertex.position.z = zCoordinates.x;
			vertex.texCoord0 = float2(0.25f, 0.5f + h);
			streams.SetVertex(vi + 2, vertex);
```

Adjust the other vertices accordingly.

```
			vertex.position.x = xCoordinates.z;
			vertex.texCoord0.x = 0.75f;
			streams.SetVertex(vi + 3, vertex);

			vertex.position.x = xCoordinates.w;
			vertex.position.z = center.y;
			vertex.texCoord0 = float2(1f, 0.5f);
			streams.SetVertex(vi + 4, vertex);

			vertex.position.x = xCoordinates.z;
			vertex.position.z = zCoordinates.y;
			vertex.texCoord0 = float2(0.75f, 0.5f - h);
			streams.SetVertex(vi + 5, vertex);

			vertex.position.x = xCoordinates.y;
			vertex.texCoord0.x = 0.25f;
			streams.SetVertex(vi + 6, vertex);
```



Flat hexagon grid.

The next tutorial is UV Sphere.

license

repository

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

BECOME A PATRON

Or make a direct donation!

made by Jasper Flick