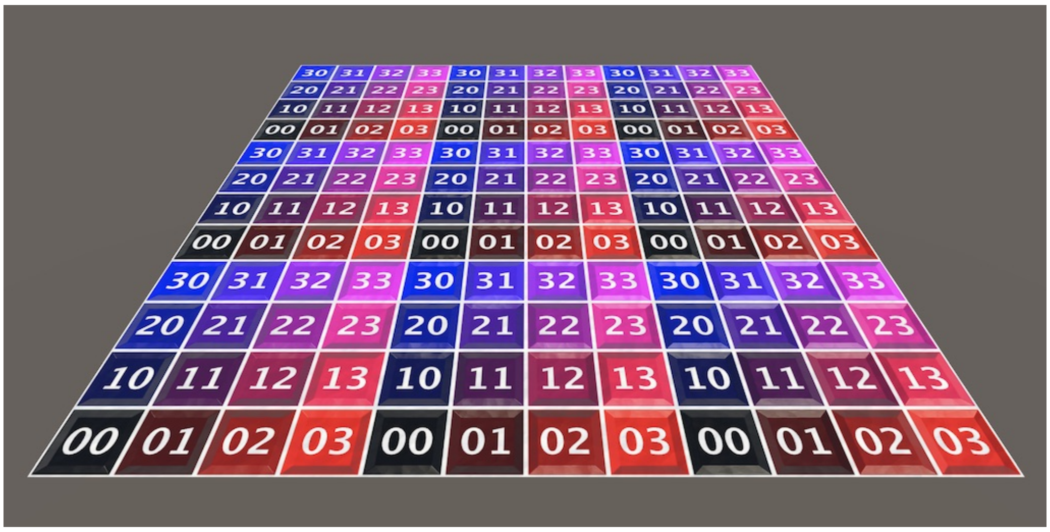
published 2021-11-23

Square Grid Mesh Jobs

Design a generic mesh job framework.
Define separate mesh streams and generators.
Disable restrictions on native container access.
Create a grid of quads on the XZ plane.
Generate rows of quads instead of individual quads.

This is the second tutorial in a series about procedural meshes. The previous tutorial introduced the advanced Mesh API. This time we'll use that API to make a Burst job that generates a square grid consisting of multiple quads.

This tutorial is made with Unity 2020.3.18f1.



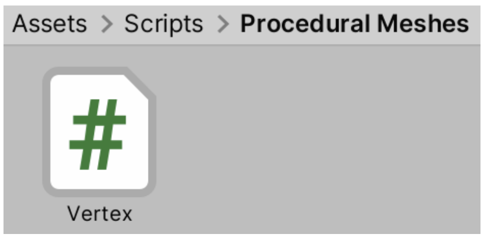
A 3×3 square grid.

1 Procedural Mesh Job Framework

A square grid is only one of many procedural meshes that we could generate. So instead of starting directly with a square grid job we'll first design a framework that supports a general approach. This will work somewhat similar to the generic approach that we used in the Pseudorandom Noise series, but with a few differences.

1.1 Generic Vertex

The first thing that we'll do is define a generic `Vertex` struct type to hold the vertex data. Let's put its asset file in a *Scripts / Procedural Meshes* subfolder.



Vertex asset in Procedural Meshes subfolder.

The contents of `Vertex` are the same as `AdvancedSingleStreamProceduralMesh.Vertex`, except that we won't bother with minimizing its size, so give everything the appropriate `float` type.

```
using Unity.Mathematics;

public struct Vertex {
    public float3 position, normal;
    public float4 tangent;
    public float2 texCoord0;
}
```

In the Pseudorandom Noise series we put all noise-related types in a single class and used partial classes to split the code into multiple files. This time we'll use a different approach: a custom namespace, which we'll named `ProceduralMeshes`.

To make a type part of a namespace it has to be defined inside a `namespace` block with the appropriate name, as if it were nested inside a `class` block. Do this for `Vertex`.

```
using Unity.Mathematics;

namespace ProceduralMeshes {

    public struct Vertex {
        public float3 position, normal;
        public float4 tangent;
        public float2 texCoord0;
    }
}
```

We'll put all other `ProceduralMeshes` type assets in the *Procedural Meshes* folder as well.

1.2 Mesh Streams

To store the mesh data we need to define the vertex and index buffers and copy the relevant data in the appropriate format. Rather than define this explicitly for each job we'll isolate this code by introducing a `ProceduralMeshes.IMeshStreams` interface. It will take care of setting up the vertex and index buffers, hiding the details of how many streams there are and what the exact data format is.

```
using Unity.Mathematics;
using UnityEngine;

namespace ProceduralMeshes {

    public interface IMeshStreams { }
}
```

Its first responsibility is to initialize the mesh data. We'll define a `Setup` method for this, with the mesh data as a paramater, along with the desired vertex count and index count.

```
        void Setup(Mesh.MeshData data, int vertexCount, int indexCount);
```

It also takes care of copying a vertex to the mesh's vertex buffer, regardless the amount of streams and the data format. We'll use a `SetVertex` method for this, with the vertex index and data to set as parameters.

```
        void SetVertex(int index, Vertex data);
```

We have to do this for the index buffer as well. As it's more convenient to work with triangles instead of individual indices, we'll define a `SetTriangle` method with the triangle index and an `int3` vertex index triplet as parameters.

```
        void SetTriangle(int index, int3 triangle);
```

The most straightforward implementation of this interface would be a single-stream approach. We'll name this type `SingleStream` and it has to be a struct to work with Burst jobs. We'll also group the stream implementations in the `ProceduralMeshes.Streams` nested namespace. I'll also put their assets in the *Scripts / Procedural Meshes / Streams* subfolder.

```
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using Unity.Collections;
using Unity.Mathematics;
using UnityEngine;
using UnityEngine.Rendering;

namespace ProceduralMeshes.Streams {

    public struct SingleStream : IMeshStreams {}
}
```

Add the `Setup` method and use it to define the mesh's buffers like we did in `AdvancedSingleStreamProceduralMesh`, except that we'll use 32-bit floats everywhere. Also immediately set the single submesh, not yet worrying about its bounds.

```
        public void Setup (Mesh.MeshData meshData, int vertexCount, int indexCount) {
            var descriptor = new NativeArray<VertexAttributeDescriptor>(
                4, Allocator.Temp, NativeArrayOptions.UninitializedMemory
            );
            descriptor[0] = new VertexAttributeDescriptor(dimension: 3);
            descriptor[1] = new VertexAttributeDescriptor(
                VertexAttribute.Normal, dimension: 3
            );
            descriptor[2] = new VertexAttributeDescriptor(
                VertexAttribute.Tangent, dimension: 4
            );
            descriptor[3] = new VertexAttributeDescriptor(
                VertexAttribute.TexCoord0, dimension: 2
            );
            meshData.SetVertexBufferParams(vertexCount, descriptor);
            descriptor.Dispose();

            meshData.SetIndexBufferParams(indexCount, IndexFormat.UInt32);

            meshData.subMeshCount = 1;
            meshData.SetSubMesh(0, new SubMeshDescriptor(0, indexCount));
        }
```

To store the vertex data in the single stream, introduce a private nested `Stream0` type. It exactly matches `Vertex`, except that here we should make sure that the field order is fixed, by attaching the `StructLayout(LayoutKind.Sequential)` attribute to it. Use it to define a native array field for this stream and retrieve it at the end of `Setup`.

```
        [StructLayout(LayoutKind.Sequential)]
        struct Stream0 {
            public float3 position, normal;
            public float4 tangent;
            public float2 texCoord0;
        }

        NativeArray<Stream0> stream0;

        public void Setup (Mesh.MeshData meshData, int vertexCount, int indexCount) {
            …

            stream0 = meshData.GetVertexData<Stream0>();
        }
```

The implementation of SetVertex then consists of copying the vertex data to a Stream0 value and storing it at the appropriate index in the stream.

```
	public void SetVertex (int index, Vertex vertex) => stream0[index] = new Stream0 {
		position = vertex.position,
		normal = vertex.normal,
		tangent = vertex.tangent,
		texCoord0 = vertex.texCoord0
	};
```

Can't we just use Vertex for the stream type?

Yes, if we gave Vertex an explicit sequential layout. However, this approach allows us more flexibility to add data to Vertex—for example a vertex color—without having to immediately adjust SingleStream. Keep in mind that Burst will optimize away intermediate steps like copying from Vertex to Stream0.

Our implementation of SetVertex is trivial, but it could be a lot more complex, for example if we decided to store part of the data as 16-bit values, requiring conversions. In such cases Burst might decide to include the SetVertex code only once and insert a call instruction—a method invocation—each time a vertex gets set. This approach is slow and prevents aggressive code optimizations. So we'll instruct Burst to always insert the entire code inline instead of going for a call. This is done by attaching the MethodImpl attribute to the method, with MethodImplOptions.AggressiveInlining as its argument. These types are part of the System.Runtime.CompilerServices namespace.

```
	[MethodImpl(MethodImplOptions.AggressiveInlining)]
	public void SetVertex (int index, Vertex vertex) => stream0[index] = new Stream0 {
		…
	};
```

Shouldn't we always suggest aggressive inlining for Burst code?

As a rule of thumb you could indeed always do this, but it mostly isn't necessary. Small methods are inlined automatically and those that are used only one are also inlined. To be sure, inspect the code generated by Burst to see whether undesired call instructions are present.

In our specific case we'll invoke SetVertex four times per quad. If we'd include code to convert from float to half in SetVertex then Burst likely won't inline the method. At this point SetVertex doesn't require the attribute, but I include it as a demonstration.

Finally, we can directly copy the triangle data to the index buffer by reinterpreting the index data to `int3` triangle data. Store the native array in a field at the end of `Setup` and perform the copy in `SetTriangle`.

```
		NativeArray<int3> triangles;

		public void Setup (Mesh.MeshData meshData, int vertexCount, int indexCount) {
			…

			stream0 = meshData.GetVertexData<Stream0>();
			triangles = meshData.GetIndexData<int>().Reinterpret<int3>(4);
		}

		…

		public void SetTriangle (int index, int3 triangle) => triangles[index] = triangle;
```

1.3 Mesh Generators

We'll also introduce an interface for the part of the code that takes care of generating the mesh, naming it `ProceduralMeshes.IMeshGenerator`. It defines the code that gets executed by the job, so it needs an `Execute` method with an index parameter. We also give it a second parameter for the streams used for storage. This has to be a generic parameter, constrained to be a struct than implements `IMeshStreams`. We don't need to make the entire interface generic, we can limit this to the `Execute` method only.

```
using UnityEngine;

namespace ProceduralMeshes {

	public interface IMeshGenerator {

		void Execute<S> (int i, S streams) where S : struct, IMeshStreams;
	}
}
```

We'll need to know the vertex count for the mesh that gets generated, and the generator can provide it via a `VertexCount` getter property. We can add it to the interface by writing `int VertexCount { get; }`. Also include a getter property for the index count.

```
		int VertexCount { get; }

		int IndexCount { get; }
```

Besides that, the length of the job must also be known when scheduling it. Add a `JobLength` getter property to provide this information.

```
		int JobLength { get; }
```

To generate our square grid we have to implement this interface, by defining the `ProceduralMeshes.Generators.SquareGrid` struct type, once again in a nested namespace and separate subfolder.

```
using Unity.Mathematics;
using UnityEngine;

using static Unity.Mathematics.math;

namespace ProceduralMeshes.Generators {

    public struct SquareGrid : IMeshGenerator {}
}
```

We won't generate the grid just yet, focusing on completing the framework first. So for now only provide a minimal implementation that generates an empty mesh and does nothing.

```
        public int VertexCount => 0;

        public int IndexCount => 0;

        public int JobLength => 0;

        public void Execute<S> (int i, S streams) where S : struct, IMeshStreams {}
```

1.4 Mesh Job

The next step is to define a Burst job to generate meshes, for which we introduce the `ProceduralMeshes.MeshJob` type. This is a generic `IJobFor` struct with type parameters for `IMeshGenerator` and `IMeshStreams`.

```
using Unity.Burst;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

namespace ProceduralMeshes {

    [BurstCompile(FloatPrecision.Standard, FloatMode.Fast, CompileSynchronously = true)]
    public struct MeshJob<G, S> : IJobFor
        where G : struct, IMeshGenerator
        where S : struct, IMeshStreams {}
}
```

Give it private fields for its generator and streams. Its `Execute` method simply forwards the invocation to the generator, passing it both the index and streams.

```
		G generator;

		S streams;

		public void Execute (int i) => generator.Execute(i, streams);
```

Because we're only writing to the streams when generating the mesh and don't read from them we can attach the `WriteOnly` attribute to the streams. This will indirectly apply the write-only state to the native arrays contained by the `IMeshStreams` implementation.

```
		[WriteOnly]
		S streams;
```

Like we did in the Pseudorandom Noise series, we also give this job its own public static `ScheduleParallel` method that creates and schedules the job, returning its job handle. It needs mesh data and a job depency as parameters. In this case we have to invoke `Setup` on the job's streams before scheduling, passing it the mesh data along with the vertex and index counts that we retrieve from the job's generator.

```
		public static JobHandle ScheduleParallel (
			Mesh.MeshData meshData, JobHandle dependency
		) {
			var job = new MeshJob<G, S>();
			job.streams.Setup(
				meshData, job.generator.VertexCount, job.generator.IndexCount
			);
			return job.ScheduleParallel(job.generator.JobLength, 1, dependency);
		}
```

1.5 Procedural Mesh Component

To try out our framework we'll create a `ProceduralMesh` component type that will set the mesh of its `MeshFilter` component, like the components of the previous tutorial. This type isn't part of the framework itself, so we'll put its asset in the *Scripts* folder. Also, as it isn't part of our namespaces we'll have to import them all.

```
using ProceduralMeshes;
using ProceduralMeshes.Generators;
using ProceduralMeshes.Streams;
using UnityEngine;
using UnityEngine.Rendering;

[RequireComponent(typeof(MeshFilter), typeof(MeshRenderer))]
public class ProceduralMesh : MonoBehaviour {}
```

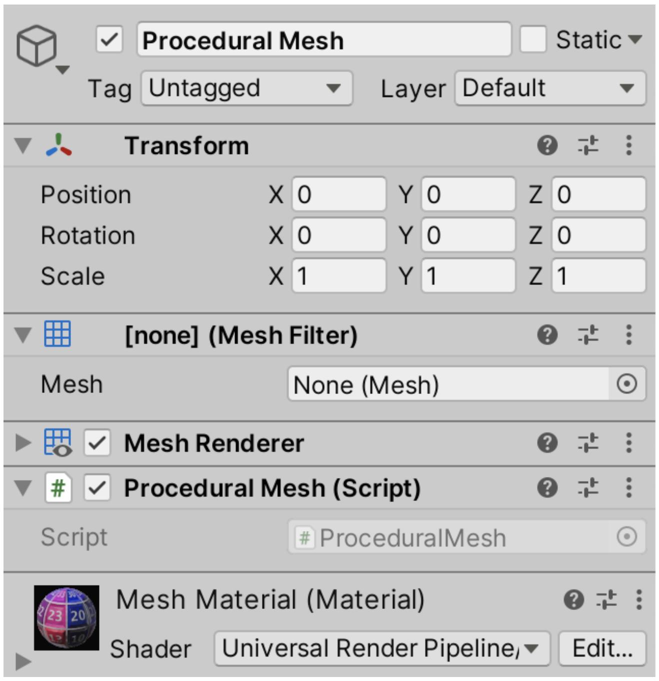
This time we'll create the mesh object in the `Awake` method, generate the mesh, and assign it to the `MeshFilter`. We put the mesh-generating code in a separate `GenerateMesh` method and keep track of the mesh via a field.

```
	Mesh mesh;

	void Awake () {
		mesh = new Mesh {
			name = "Procedural Mesh"
		};
		GenerateMesh();
		GetComponent<MeshFilter>().mesh = mesh;
	}

	void GenerateMesh () {}
```

Generating the mesh consists of allocating writable mesh data, followed by scheduling and immediately completing a `MeshJob` for it—using our `SquareGrid` and `SingleStream` types —and then applying it to the mesh.

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

		MeshJob<SquareGrid, SingleStream>.ScheduleParallel(
			meshData, default
		).Complete();

		Mesh.ApplyAndDisposeWritableMeshData(meshDataArray, mesh);
	}
```

Now create a procedural mesh game object, either in a new scene or replacing the existing quad-generating game objects of the previous tutorial


Procedural Mesh ✓ — Static
Tag: Untagged — Layer: Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

[none] (Mesh Filter)
Mesh: None (Mesh)

Mesh Renderer ✓

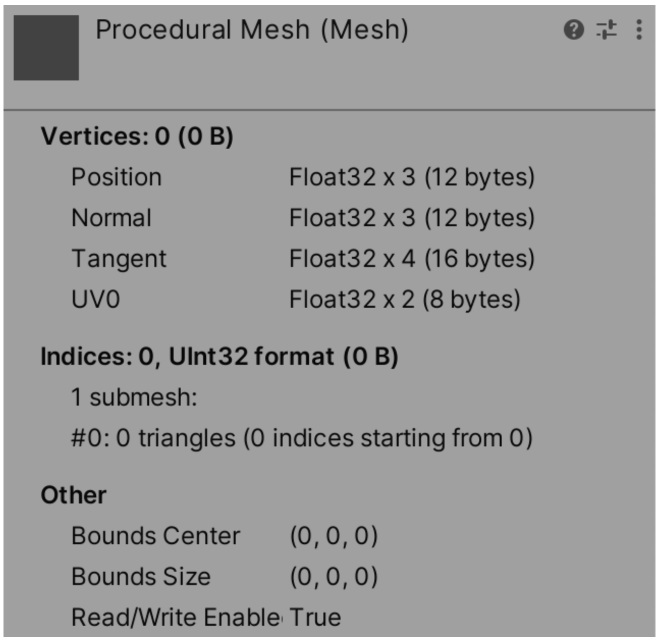
Procedural Mesh (Script) ✓
Script: ProceduralMesh

Mesh Material (Material)
Shader: Universal Render Pipeline/ — Edit...

Procedural mesh game object.

1.6 Generating a Quad

At this point an empty mesh is generated when entering play mode.



Procedural Mesh (Mesh)

Vertices: 0 (0 B)

Position	Float32 x 3 (12 bytes)
Normal	Float32 x 3 (12 bytes)
Tangent	Float32 x 4 (16 bytes)
UV0	Float32 x 2 (8 bytes)

Indices: 0, UInt32 format (0 B)

1 submesh:

#0: 0 triangles (0 indices starting from 0)

Other

Bounds Center	(0, 0, 0)
Bounds Size	(0, 0, 0)
Read/Write Enable	True

Empty procedural mesh.

We get an empty mesh because our job doesn't do anything yet. Currently the job isn't scheduled at all because its length is zero. We activate the job by making `SquareGrid.JobLength` return 1.

```
	public int JobLength => 1;
```

This causes our job to get scheduled, but when entering play mode we now get an invalid operation exception complaining that two containers might be the same thing. This refers to the two native arrays of `SingleStream`. Unity complains that they might be aliasing, which means that the native arrays might represent overlapping data. The reason for this is that all the mesh data is a single unmanaged block of memory. Our job tries to access two subsections of this data—the vertex part and the triangle index part—at the same time and Unity disallows this because it might produce faulty results.

In general Unity's safety checks are valid and should be heeded, but in this case we are certain that the vertex and index data never overlap. So we'll disable the safety, by attaching the `NativeDisableContainerSafetyRestriction` attribute from the `Unity.Collections.LowLevel.Unsafe` namespace to both native array fields.

```
using Unity.Collections;
using Unity.Collections.LowLevel.Unsafe;
using Unity.Mathematics;
…

namespace ProceduralMeshes.Streams {

	public struct SingleStream : IMeshStreams {

		…

		[NativeDisableContainerSafetyRestriction]
		NativeArray<Stream0> stream0;

		[NativeDisableContainerSafetyRestriction]
		NativeArray<int3> triangles;

		…
	}
}
```

To test our framework we'll have `SquareGrid` generate only a single quad for now, exactly like the one that we generated in the previous tutorial. So its vertex count has to become four.

```
		public int VertexCount => 4;
```

In `Execute`, begin by creating a generic vertex value and settings its normal and tangent vector, which are the same for all vertices. As all values are initialized to zero we can suffice with setting only the nonzero components. Thus the normal Z component becomes −1 and the tangent XW components become 1 and −1.

```
		public void Execute<S> (int i, S streams) where S : struct, IMeshStreams {
			var vertex = new Vertex();
			vertex.normal.z = -1f;
			vertex.tangent.xw = float2(1f, -1f);
		}
```

How does the assignment to `xw` work?

This is a swizzle operation, which allows us to assign to a subset of the vector's components in the order that we want. A swizzle operation can also be used to extract a subset of the components, in whatever order we like and even with repetition. For example, instead of `.xy` we could also access `.yx`, `.zy`, `.xx`, `.xxy`, `.zxxy`, etc. The Mathematics types implement these via properties.

We can then set the first vertex, with index zero. We ignore the index passed to `Execute` method, because we'll generate the entire quad at once. We can do this because we disabled the safety restrictions for the native arrays.

```
            var vertex = new Vertex();
            vertex.normal.z = -1f;
            vertex.tangent.xw = float2(1f, -1f);

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

Don't we need the `NativeDisableParallelForRestriction` attribute to write to any index?

Yes, but the `NativeDisableContainerSafetyRestriction` attribute disables all restrictions, so we don't need to also apply `NativeDisableParallelForRestriction`.

Complete the quad by adjusting the positions and texture coordinates and setting the other three vertices.

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

            vertex.position = right();
            vertex.texCoord0 = float2(1f, 0f);
            streams.SetVertex(1, vertex);

            vertex.position = up();
            vertex.texCoord0 = float2(0f, 1f);
            streams.SetVertex(2, vertex);

            vertex.position = float3(1f, 1f, 0f);
            vertex.texCoord0 = 1f;
            streams.SetVertex(3, vertex);
```

We also need two triangles, so set the index count to six.

```
        public int IndexCount => 6;
```

Then set the two triangles at the end of `Execute`.

```
        public void Execute<S> (int i, S streams) where S : struct, IMeshStreams {
            …

            streams.SetTriangle(0, int3(0, 2, 1));
            streams.SetTriangle(1, int3(1, 2, 3));
        }
```

This should produce a quad, but instead we get an argument exception even before the jobs are scheduled. It happens when the submesh is set in `SingleStream.Setup`. When we invoke `SetSubMesh` it immediately validates the triangle indices and recalculates the bounds. This is virtually guaranteed to fail, because at this point the job hasn't run yet so the index buffer contains arbitrary data. We must supply `MeshUpdateFlags` to indicate that `SetSubMesh` should not do anything with the data. We already used `DontRecalculateBounds` in the previous tutorial. This time we also have to use `DontValidateIndices`. We apply both by merging the flags with the binary OR `|` operator.

```
        meshData.SetSubMesh(
            0, new SubMeshDescriptor(0, indexCount),
            MeshUpdateFlags.DontRecalculateBounds |
            MeshUpdateFlags.DontValidateIndices
        );
```

30
31
32
33
20
21
22
23
10
11
12
13
00
01
02
03

Quad generated via job.

1.7 Bounds

The only thing that our mesh still lacks is valid bounds. The generator should provide these bounds, so add a property to get them to the `IMeshGenerator` interface.

```
    Bounds Bounds { get; }
```

Then add the implementation to `SquareGrid`.

```
    public Bounds Bounds => new Bounds(new Vector3(0.5f, 0.5f), new Vector3(1f, 1f));
```

To set the bounds, add a parameter for the mesh to `MeshJob.ScheduleParallel`. I make it the first parameter. We can then set the mesh bounds immediately after creating the job.

```
    public static JobHandle ScheduleParallel (
        Mesh mesh, Mesh.MeshData meshData, JobHandle dependency
    ) {
        var job = new MeshJob<G, S>();
        mesh.bounds = job.generator.Bounds;
        …
    }
```

Pass along the mesh in `ProceduralMesh.GenerateMesh`.

```
    MeshJob<SquareGrid, SingleStream>.ScheduleParallel(
        mesh, meshData, default
    ).Complete();
```

We should also set the bounds of the submesh. To make this possible we'll add the bounds as a second parameter to `IMeshStreams.Setup`.

```
		void Setup(
			Mesh.MeshData meshData, Bounds bounds, int vertexCount, int indexCount
		);
```

Adjust `SingleStream.Setup` so it sets the bounds and vertex count of the submesh.

```
		public void Setup (
			Mesh.MeshData meshData, Bounds bounds, int vertexCount, int indexCount
		) {
			…
			meshData.SetSubMesh(
				0, new SubMeshDescriptor(0, indexCount) {
					bounds = bounds,
					vertexCount = vertexCount
				},
				MeshUpdateFlags.DontRecalculateBounds |
				MeshUpdateFlags.DontValidateIndices
			);

			…
		}
```

Finally, include the bounds when setting up the streams in `MeshJob.ScheduleParallel`. We could either store the bounds in a variable or directly use the result of the mesh bounds assignment expression as an argument for `Setup`. I do the latter to demonstrate this usage.

```
		public static JobHandle ScheduleParallel (
			Mesh mesh, Mesh.MeshData meshData, JobHandle dependency
		) {
			var job = new MeshJob<G, S>();
			~~//mesh.bounds = job.generator.Bounds;~~
			job.streams.Setup(
				meshData,
				mesh.bounds = job.generator.Bounds,
				job.generator.VertexCount,
				job.generator.IndexCount
			);
			return job.ScheduleParallel(job.generator.JobLength, 1, dependency);
		}
```

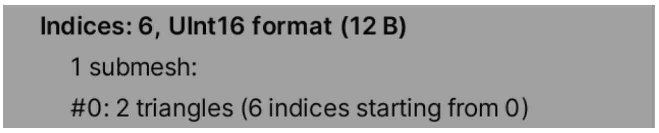
1.8 16-Bit Indices

In the previous tutorial we reduced the triangle indices from 32-bit to 16-bit, because that halves the size of the index buffer. Let's do the same for our framework as well. A convenient way to do this is by defining a `TriangleUInt16` type in the `ProceduralMeshes.Streams` namespace. It's a sequential struct containing three `ushort` values. Give it an implicit conversion operator from `int3` to `TriangleUInt16`.

```
using System.Runtime.InteropServices;
using Unity.Mathematics;

namespace ProceduralMeshes.Streams {

    [StructLayout(LayoutKind.Sequential)]
    public struct TriangleUInt16 {

        public ushort a, b, c;

        public static implicit operator TriangleUInt16 (int3 t) => new TriangleUInt16 {
            a = (ushort)t.x,
            b = (ushort)t.y,
            c = (ushort)t.z
        };
    }
}
```

Now we can switch `SingleStream` to 16-bit indices simply by changing the triangle index element types and the index buffer format.

```
        [NativeDisableContainerSafetyRestriction]
        NativeArray<TriangleUInt16> triangles;

        public void Setup (
            Mesh.MeshData meshData, Bounds bounds, int vertexCount, int indexCount
        ) {
            …

            meshData.SetIndexBufferParams(indexCount, IndexFormat.UInt16);

            …
            triangles = meshData.GetIndexData<ushort>().Reinterpret<TriangleUInt16>(2);
        }
```



```
Indices: 6, UInt16 format (12 B)
    1 submesh:
    #0: 2 triangles (6 indices starting from 0)
```

16-Bit indices.

1.9 Multiple Vertex Streams

As an example of a different `IMeshStreams` implementation, let's include a multi-stream approach, like `AdvancedMultiStreamProceduralMesh`. Duplicate `SingleStream` and rename it to `MultiStream`. Replace its single stream with four streams for the individual vertex attributes.

```
	public struct MultiStream : IMeshStreams {

		~~//[StructLayout(LayoutKind.Sequential)]~~
		~~//struct Stream0 {~~
		~~//	…~~
		~~//}~~

		~~//[NativeDisableContainerSafetyRestriction]~~
		~~//NativeArray<Stream0> stream0;~~

		[NativeDisableContainerSafetyRestriction]
		NativeArray<float3> stream0, stream1;

		[NativeDisableContainerSafetyRestriction]
		NativeArray<float4> stream2;

		[NativeDisableContainerSafetyRestriction]
		NativeArray<float2> stream3;

		…

		public void Setup (
			Mesh.MeshData meshData, Bounds bounds, int vertexCount, int indexCount
		) {
			…
			descriptor[1] = new VertexAttributeDescriptor(
				VertexAttribute.Normal, dimension: 3, stream: 1
			);
			descriptor[2] = new VertexAttributeDescriptor(
				VertexAttribute.Tangent, dimension: 4, stream: 2
			);
			descriptor[3] = new VertexAttributeDescriptor(
				VertexAttribute.TexCoord0, dimension: 2, stream: 3
			);
			…

			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);
		}

		[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;
		}

		…
	}
```

It is now possible to switch to a multi-stream approach by replacing `SingleStream` with `MultiStream` in `ProceduralMesh.GenerateMesh`.

```
		MeshJob<SquareGrid, MultiStream>.ScheduleParallel(
			mesh, meshData, default
		).Complete();
```

Note that the generator code only knows about the generic `Vertex`. It is completely oblivious of how the vertex data gets stored. It's even possible that only part of the data gets stored, for example omitting the normal and tangent. Burst will optimize away the unneeded code.

2 A Grid of Quads

Now that we have a functional framework we move on to generating a mesh that consists of multiple quads, placed so that they form a regular square grid. Such a grid itself doesn't provide any benefit compared to a single quad, but it can be used as the basis of more complex meshes that aren't entirely flat. In this tutorial we'll limit ourselves to the simple grid.

2.1 Mesh Resolution

We'll adapt our code so it can produce a grid of R×R squares, where R stands for the resolution of the grid. The resolution of the mesh is a general concept, for which we can add a property to `IMeshGenerator`. In this case the property should be settable, which we enforce by also including `set;` in its block.

```
		int Resolution { get; set; }
```

We can implement this property in `SquareGrid` by including the same line of code, only adding the `public` access modifier. This generates a trivial automatic property, which implicitly includes a field used by the property.

```
		public int Resolution { get; set; }
```

The vertex count, index count, and job length now depend on the resolution. The amount of quads is equal to the resolution squared, so have to multiplied all by that.

```
		public int VertexCount => 4 * Resolution * Resolution ;

		public int IndexCount => 6 * Resolution * Resolution;

		public int JobLength => Resolution * Resolution;
```

Add a resolution parameter to `MeshJob.ScheduleParallel` and use it to set the generator's resolution immediately after creating the job.

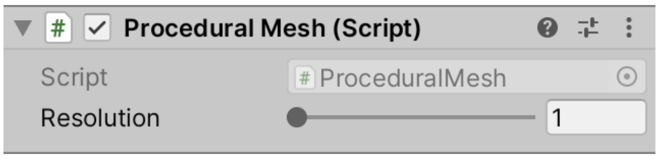
```
		public static JobHandle ScheduleParallel (
			Mesh mesh, Mesh.MeshData meshData, int resolution, JobHandle dependency
		) {
			var job = new MeshJob<G, S>();
			job.generator.Resolution = resolution;
			…
		}
```

Then add a resolution slider to `ProceduralMesh` and use it when generating the mesh. The minimum should be 1 and I'll use 10 for the maximum.

```
	[SerializeField, Range(1, 10)]
	int resolution = 1;

	…

	void GenerateMesh () {
		…

		MeshJob<SquareGrid, MultiStream>.ScheduleParallel(
			mesh, meshData, resolution, default
		).Complete();

		…
	}
```

To support regenerating the mesh when we change the resolution while in play mode we have to make some more changes. Our approach this time is to include an `Update` method that generates the mesh and then disables the component. This way `Update` isn't needlessly invoked every frame. We enable the component in a new `OnValidate` method. This means that we no longer need to generate the mesh in `Awake`.

```
	void Awake () {
		…
		//GenerateMesh();
		GetComponent<MeshFilter>().mesh = mesh;
	}

	void OnValidate () => enabled = true;

	void Update () {
		GenerateMesh();
		enabled = false;
	}
```



Resolution slider.

2.2 Generating All Quads

To make sure that we set the data for all quads we'll determine the correct vertex and triangle indices at the start of `Execute`. The job index passed to `Execute` represents the quad index. So its first vertex index is the quadruple that and its first triangle index is double that.

```
		public void Execute<S> (int i, S streams) where S : struct, IMeshStreams {
			int vi = 4 * i, ti = 2 * i;

			…
		}
```

We find the other vertex indices by adding an offset to the first one. I include the zero offset for clarity, even though it does not affect the code.

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

			vertex.position = right();
			vertex.texCoord0 = float2(1f, 0f);
			streams.SetVertex(vi + 1, vertex);

			vertex.position = up();
			vertex.texCoord0 = float2(0f, 1f);
			streams.SetVertex(vi + 2, vertex);

			vertex.position = float3(1f, 1f, 0f);
			vertex.texCoord0 = 1f;
			streams.SetVertex(vi + 3, vertex);
```

The same goes for the triangles. In this case we also have to add the first vertex index to the relative vertex indices that define the triangles, to keep them relative.

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

We also have to determine the position offsets for the quads, relative to their bottom-left corners. We find the Y offset via an integer division of the quad index by the resolution. The X offset is then found by subtracting Y times the resolution from the quad index.

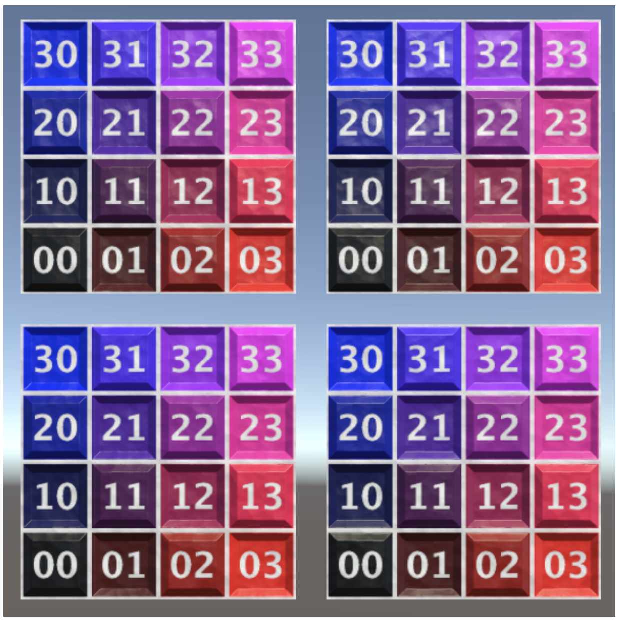
```
			int vi = 4 * i, ti = 2 * i;

			int y = i / Resolution;
			int x = i - Resolution * y;
```

We can define all four coordinates that we need for the quad in a single `float4` value, containing X, X + 1, Y, and Y + 1. But we'll initially only add 0.9 to leave a visible gap between the quads.

```
			int y = i / Resolution;
			int x = i - Resolution * y;

			var coordinates = float4(x, x + 0.9f, y, y + 0.9f);
```

We can set the positions correctly via swizzle operations on the coordinates, selecting the appropriate two coordinates per position.

```
			vertex.position.xy = coordinates.xz;
			streams.SetVertex(vi + 0, vertex);

			vertex.position.xy = coordinates.yz;
			vertex.texCoord0 = float2(1f, 0f);
			streams.SetVertex(vi + 1, vertex);

			vertex.position.xy = coordinates.xw;
			vertex.texCoord0 = float2(0f, 1f);
			streams.SetVertex(vi + 2, vertex);

			vertex.position.xy = coordinates.yw;
			vertex.texCoord0 = 1f;
			streams.SetVertex(vi + 3, vertex);
```



Resolution 2 grid with gaps.

2.3 A Plane

Grids are typically used for flat planes, so let's adjust ours so it lies in the XZ plane. Begin by renaming `y` to `z` and also close the gaps between the quads.

```
			int z = i / Resolution;
			int x = i - Resolution * z;

			var coordinates = float4(x, x + 1f, z, z + 1f);
```

We change the orientation of the grid by assigning to the XZ components of the vertex position instead of to XY.

```
			vertex.position.xz = coordinates.xz;
			streams.SetVertex(vi + 0, vertex);

			vertex.position.xz = coordinates.yz;
			vertex.texCoord0 = float2(1f, 0f);
			streams.SetVertex(vi + 1, vertex);

			vertex.position.xz = coordinates.xw;
			vertex.texCoord0 = float2(0f, 1f);
			streams.SetVertex(vi + 2, vertex);

			vertex.position.xz = coordinates.yw;
```

We also have to change the normal vector so it points up.

```
			vertex.normal.y = 1f;
```

Resolution 3 plane.

It's also convenient if the plane is centered on the origin and has a fixed size, regardless of its resolution. We can achieve this by dividing all coordinates by the resolution and then subtracting ½.

```
			var coordinates = float4(x, x + 1f, z, z + 1f) / Resolution - 0.5f;
```

Adjust the bounds to match.

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

2.4 Generating Rows of Quads

Our job currently generates each quad of the grid in isolation. Creating a single quad isn't much work, but the vertex data cannot be vectorized. So everything has to be calculated per quad and Unity's job framework adds additional overhead. We can improve efficiency by combining the generation of multiple quads in a single invocation of `Execute`. It makes the most sense to generates all quads of a single row together. That will make the job length equal to the resolution, no longer squaring it.

```
		public int JobLength => Resolution;
```

We'll let each invocation of `Execute` take care of a whole row of quads along the X axis. The job index will thus represent the Z offset of the row instead of the quad index. Let's rename it accordingly. Also, the first quad index of the row is thus equal to the resolution times Z.

```
		public void Execute<S> (int z, S streams) where S : struct, IMeshStreams {
			int vi = 4 * Resolution * z, ti = 2 * Resolution * z;

			~~//int z = i / Resolution;~~
			…
		}
```

Now instead of using a fixed X offset we introduce a loop for the entire row, which encloses the code that fills the streams.

```
			~~//int x = i - Resolution * z;~~

			for (int x = 0; x < Resolution; x++) {
				var coordinates = float4(x, x + 1f, z, z + 1f) / Resolution - 0.5f;

				…

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

To set the correct quads, after each iteration of the loop we have to increment the vertex index by four and the triangle index by two.

```
			for (int x = 0; x < Resolution; x++, vi += 4, ti += 2) { … }
```

Finally, Burst can detect code inside the loop that never changes and automatically pull it out of the loop. However, it won't split vectors so we can optimize a little bit by manually splitting the coordinates vector in separate X and Z pairs. The Z coordinate calculation is constant and will thus be hoisted out of the loop.

```
~~//var coordinates = float4(x, x + 1f, z, z + 1f) / Resolution - 0.5f;~~
var xCoordinates = float2(x, x + 1f) / Resolution - 0.5f;
var zCoordinates = float2(z, z + 1f) / Resolution - 0.5f;

~~//vertex.position.xz = coordinates.xz;~~
vertex.position.x = xCoordinates.x;
vertex.position.z = zCoordinates.x;
streams.SetVertex(vi + 0, vertex);

~~//vertex.position.xz = coordinates.yz;~~
vertex.position.x = xCoordinates.y;
vertex.texCoord0 = float2(1f, 0f);
streams.SetVertex(vi + 1, vertex);

~~//vertex.position.xz = coordinates.xw;~~
vertex.position.x = xCoordinates.x;
vertex.position.z = zCoordinates.y;
vertex.texCoord0 = float2(0f, 1f);
streams.SetVertex(vi + 2, vertex);

~~//vertex.position.xz = coordinates.yw;~~
vertex.position.x = xCoordinates.y;
vertex.texCoord0 = 1f;
streams.SetVertex(vi + 3, vertex);
```

Shouldn't we also introduce an `invResolution` variable to avoid multiple divisions?

Because we use `FloatMode.Fast` Burst will do this automatically when it detects repeated division by the same value. So a division is only calculated once per invocation of `Execute` and all divisions that we wrote become multiplications.

The next tutorial is Modified Grid.