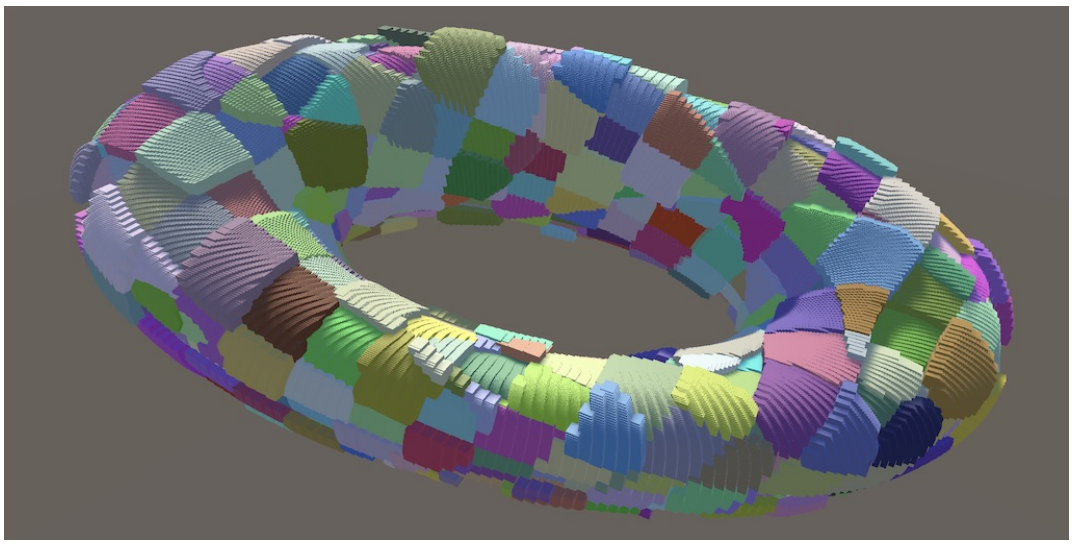
updated 2021-07-15 published 2021-04-30

Hashing Space Hashing in an Arbitrary Grid

Hash transformed 3D space.
Sample space with different shapes.
Manually vectorize jobs.
Create a shape job template.

This is the second tutorial in a series about pseudorandom noise. In it we'll adjust our hashing so it works with arbitrary grids and shapes.

This tutorial is made with Unity 2020.3.6f1.



Sampling rotated hashed space with a deformed torus.

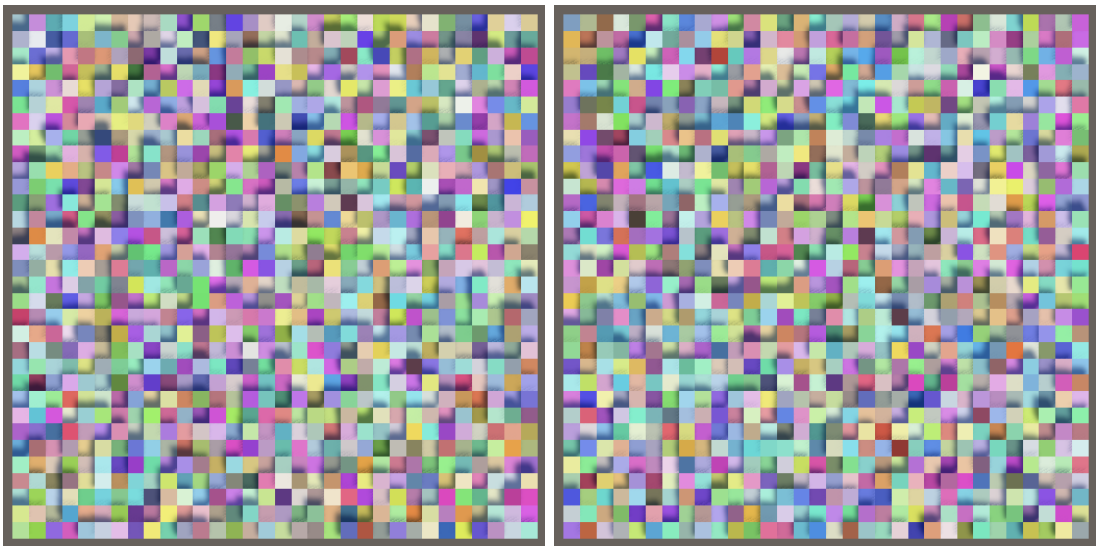
1 Grid Transformation

In the previous tutorial we colored and displaced sample points on a plane by hashing integer UV coordinates, so each point got its own hash value. In this tutorial we will instead hash space itself, then sample it. We'll do this based on the integer grid defined by the space. This decouples the hash from the sample points, which makes it independent of the shape and resolution of the visualization.

1.1 Changing Scale

To illustrate that the hash pattern and resolution can be decoupled, let's start by doubling the effective resolution of the hash pattern. We can do this by doubling the UV coordinates in `HashJob.Execute` before feeding then to the hash.

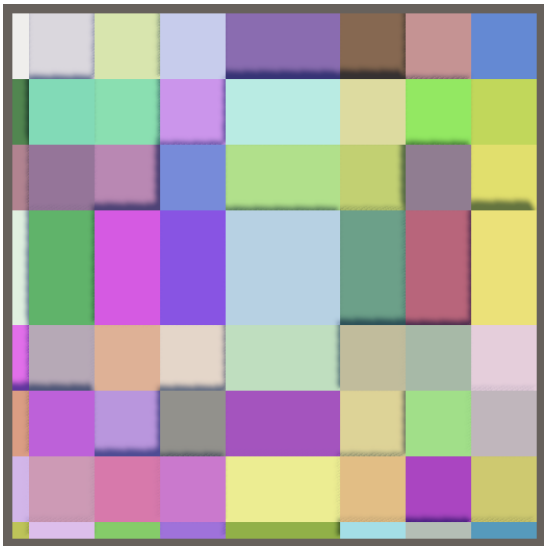
```
		public void Execute(int i) {
			int v = (int)floor(invResolution * i + 0.00001f);
			int u = i - resolution * v - resolution / 2;
			v -= resolution / 2;

			u *= 2;
			v *= 2;

			hashes[i] = hash.Eat(u).Eat(v);
		}
```



Normal and double scale; resolution 32 and seed 0.

Doubling the scale of the UV produces a different pattern, but doesn't appear fundamentally different. We've just scaled the domain of the hash function, moving through it twice as fast as before. We can also scale down instead, for example by dividing the UV by four.

```
			u /= 4;
			v /= 4;
```



Quarter scale, misaligned.

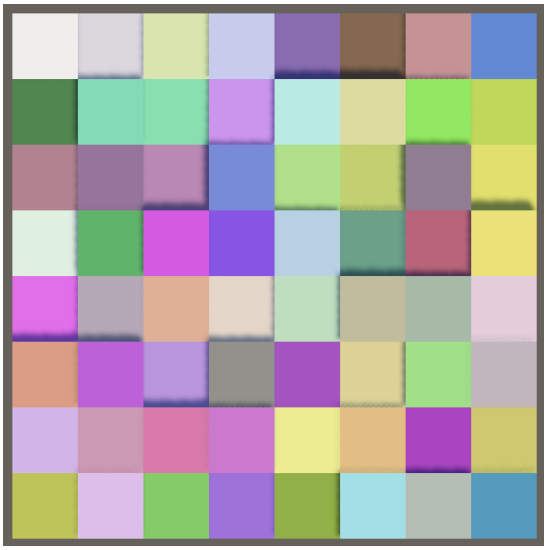
Now groups of 4×4 sample points end up with the same hash, so it appears as if we reduced the resolution of the visualization. However, we get a misalignment and repetition around zeros, because the integer divisions effectively round towards zero.

Both problems are solved by initially calculating the UV coordinates as floating-point values in the −0.5–0.5 range, instead of as integers. Then derive the integer coordinates from that by rounding down, via the `floor` method.

```
			float vf = floor(invResolution * i + 0.00001f);
			float uf = invResolution * (i - resolution * vf + 0.5f) - 0.5f;
			vf = invResolution * (vf + 0.5f) - 0.5f;

			int u = (int)floor(uf);
			int v = (int)floor(vf);

			hashes[i] = hash.Eat(u).Eat(v);
		}
```

To transform the hash pattern back to the previous scale, before rounding down multiply the UV with the visualization resolution that we used—which is 32—divided by four.

```
			int u = (int)floor(uf * 32f / 4f);
			int v = (int)floor(vf * 32f / 4f);
```



Quarter scale, aligned.

1.2 Domain Transformation

Instead of using a fixed domain scale we'll make it configurable. And we don't have to limit ourselves to just scaling the domain, we can treat it as a regular 3D space. We can apply any translation, rotation, and scale to it, just like we can transform game objects. However, we cannot use a `Transform` component in isolation, so let's define a custom `SpaceTRS` struct in a new file, with three public `float3` fields for the transformation.

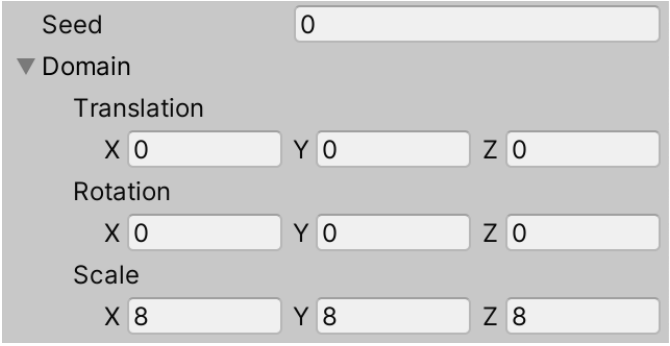
```
using Unity.Mathematics;

public struct SpaceTRS {

	public float3 translation, rotation, scale;
}
```

Add a field of this type to `HashVisualization` to support a configurable space transformation. Set its scale to 8 by default, matching our current hard-coded visualization.

```
	[SerializeField]
	SpaceTRS domain = new SpaceTRS {
		scale = 8f
	};
```

This only works if our custom `SpaceTRS` can be serialized. We make this possible by attaching the `System.Serializable` attribute to it.

```
[System.Serializable]
public struct SpaceTRS { … }
```



Domain configuration.

Applying 3D space transformations to a position or direction can be performed via multiplication with a 4×4 matrix and a vector. Add a public `float4x4 Matrix` getter property to `SpaceTRS` that returns such a matrix. It can be created by invoking `float4x4.TRS` with the translation, rotation, and scale vectors as parameters. The rotation must be a quaternion. We can create the rotation using Unity's convention by invoking `quaternion.EulerZXY`, passing it the rotation vector converted to radians via the `radians` method.

```
	public float4x4 Matrix {
		get {
			return float4x4.TRS(
				translation, quaternion.EulerZXY(math.radians(rotation)), scale
			);
		}
	}
```

However, because we limit ourselves to a translation-rotation-scale transformation the fourth row of the matrix will always be 0,0,0,1. So we can omit it to reduce the size of the data, returning a `float3x4` matrix instead. There is no direct conversion between those matrix types, so we have to invoke the `math.float3x4` method with the four column vectors of the matrix as arguments. These columns are named `c0`, `c1`, `c2`, and `c3`. We only need the first three components of each column, which we can extract by accessing their `xyz` properties.

```
	public float3x4 Matrix {
		get {
			float4x4 m = float4x4.TRS(
				translation, quaternion.EulerZXY(math.radians(rotation)), scale
			);
			return math.float3x4(m.c0.xyz, m.c1.xyz, m.c2.xyz, m.c3.xyz);
		}
	}
```

1.3 Applying the Matrix

To apply the matrix in `HashJob.Execute` we must first have a `float3` position. Use the −0.5–0.5 UV coordinates to create the point on the XZ plane, then use its XZ components for the integer UV coordinates.

```
		public void Execute(int i) {
			float vf = floor(invResolution * i + 0.00001f);
			float uf = invResolution * (i - resolution * vf + 0.5f) - 0.5f;
			vf = invResolution * (vf + 0.5f) - 0.5f;

			float3 p = float3(uf, 0f, vf);

			int u = (int)floor(p.x);
			int v = (int)floor(p.z);

			hashes[i] = hash.Eat(u).Eat(v);
		}
```

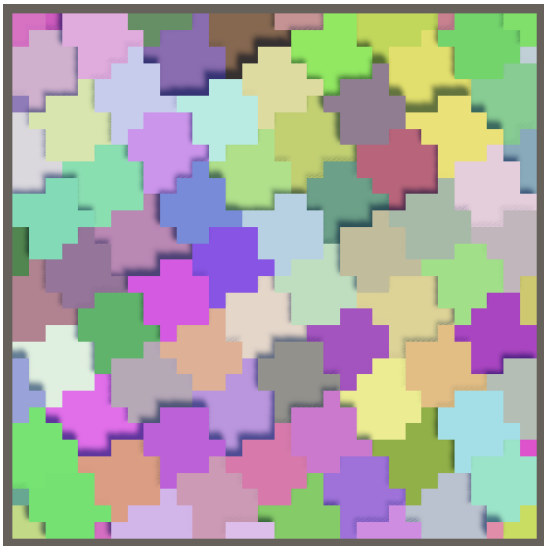
Then add a `float3x4 domainTRS` field to the job and multiply that matrix and the point via the `mul` method. This operation requires a `float4` instead of a `float3`, with its fourth component set to 1. This last component is multiplied with the fourth column of the matrix—the translation vector—so it ends up getting applied without change. Burst will optimize away the multiplication with 1.

```
		public float3x4 domainTRS;

		public void Execute(int i) {
			…

			float3 p = mul(domainTRS, float4(uf, 0f, vf, 1f));

			…
		}
```

If at this point you check the Burst inspector you'll notice that our job no longer gets vectorized. This happens because we're now working with vector types instead of individual values. We'll ignore this limitation for now, but will take care of it later.

To apply the domain transformation, pass the domain matrix to the job when scheduling it.

```
		new HashJob {
			hashes = hashes,
			resolution = resolution,
			invResolution = 1f / resolution,
			hash = SmallXXHash.Seed(seed),
			domainTRS = domain.Matrix
		}.ScheduleParallel(hashes.Length, resolution, default).Complete();
```

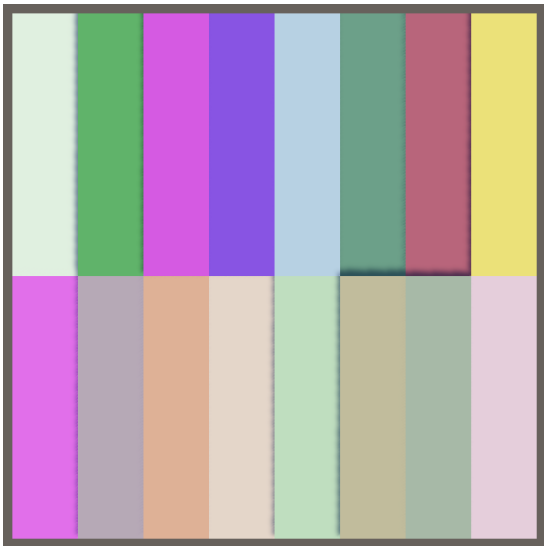


Domain rotated 30° around Y.

We can now move, rotate, and scale the domain used for generating the hashes. We see the results that we would get as if we had moved, rotated, or scaled the plane that we're using for sampling the hash function. Thus when moving the domain to the right it appears as if the hashes move to the left, because the plane itself remains stationary. Likewise, rotation appears to go in the opposite direction and scaling up makes the pattern appear smaller.

1.4 3D Hashing

The space transformation can be in 3D, for example we could move along Y, but this wouldn't make a difference because the hash currently only depends on X and Z. Likewise, if you'd for example rotate around the X axis the hash squares would elongate until they stretch all the way from the center to the edges of the plane.



Domain rotated 90° around X.

To partition the pattern into hash cubes all we have to do is feed all three coordinates to the hash in `Execute`.

```
		int u = (int)floor(p.x);
		int v = (int)floor(p.y);
		int w = (int)floor(p.z);

		hashes[i] = hash.Eat(u).Eat(v).Eat(w);
```

Now moving along Y also affects the hash and a 90° rotation around one axis will produce a square pattern, displaying a different slice of the hash volume.

2 Sample Shapes

Given that we have a 3D hash volume, we don't need to limit ourselves to sampling it with a planar shape. We could create other jobs that samples the hash with different shapes.

2.1 Shapes Job

Instead of creating different variants of `HashJob`, we'll separate the jobs of creating the shape and sampling the hash volume. Create a static `Shapes` class in a new file for this, initially containing a single `Job` struct that generates `float3` positions for our plane and stores them in a native array. The domain transformation applies to hashing and thus isn't part of the shapes job. So it only needs the resolution and inverse resolution and input.

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

using static Unity.Mathematics.math;

public static class Shapes {

	[BurstCompile(FloatPrecision.Standard, FloatMode.Fast, CompileSynchronously = true)]
	public struct Job : IJobFor {

		[WriteOnly]
		NativeArray<float3> positions;

		public float resolution, invResolution;

		public void Execute (int i) {
			float2 uv;
			uv.y = floor(invResolution * i + 0.00001f);
			uv.x = invResolution * (i - resolution * uv.y + 0.5f) - 0.5f;
			uv.y = invResolution * (uv.y + 0.5f) - 0.5f;

			positions[i] = float3(uv.x, 0f, uv.y);
		}
	}
}
```

Also give the job a static `ScheduleParallel` method that encapsulates the creation of the job and its scheduling, taking and returning a job handle. This method doesn't require the inverse resolution as a parameter because it can calculate the inverse itself.

```
	public struct Job : IJobFor {

		…

		public static JobHandle ScheduleParallel (
			NativeArray<float3> positions, int resolution, JobHandle dependency
		) {
			return new Job {
				positions = positions,
				resolution = resolution,
				invResolution = 1f / resolution
			}.ScheduleParallel(positions.Length, resolution, dependency);
		}
	}
```

2.2 Generating Positions

We'll feed the positions to `HashJob` and will also use them when drawing the instances. So add a shader identifier, native array, and buffer for the positions to `HashVisualization`. Create and dispose them when needed.

```
	static int
		hashesId = Shader.PropertyToID("_Hashes"),
		positionsId = Shader.PropertyToID("_Positions"),
		configId = Shader.PropertyToID("_Config");

	…

	NativeArray<uint> hashes;

	NativeArray<float3> positions;

	ComputeBuffer hashesBuffer, positionsBuffer;

	MaterialPropertyBlock propertyBlock;

	void OnEnable () {
		int length = resolution * resolution;
		hashes = new NativeArray<uint>(length, Allocator.Persistent);
		positions = new NativeArray<float3>(length, Allocator.Persistent);
		hashesBuffer = new ComputeBuffer(length, 4);
		positionsBuffer = new ComputeBuffer(length, 3 * 4);

		…
	}

	void OnDisable () {
		hashes.Dispose();
		positions.Dispose();
		hashesBuffer.Release();
		positionsBuffer.Release();
		hashesBuffer = null;
		positionsBuffer = null;
	}
```

In `OnEnable`, schedule the shape job before the hash job, passing along the handle to the latter because the positions must be generated before hashing can take place. Also send the positions to the GPU afterwards.

```
		JobHandle handle = Shapes.Job.ScheduleParallel(positions, resolution, default);

		new HashJob {
			…
		}.ScheduleParallel(hashes.Length, resolution, handle).Complete();

		hashesBuffer.SetData(hashes);
		positionsBuffer.SetData(positions);

		propertyBlock ??= new MaterialPropertyBlock();
		propertyBlock.SetBuffer(hashesId, hashesBuffer);
		propertyBlock.SetBuffer(positionsId, positionsBuffer);
```

Adjust *HashGPU* so it uses the provided positions instead of generating a plane itself. Apply the vertical offset on top of that.

```
#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
	StructuredBuffer<uint> _Hashes;
	StructuredBuffer<float3> _Positions;
#endif

float4 _Config;

void ConfigureProcedural () {
	#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
		//float v = floor(_Config.y * unity_InstanceID + 0.00001);
		//float u = unity_InstanceID - _Config.x * v;

		unity_ObjectToWorld = 0.0;
		unity_ObjectToWorld._m03_m13_m23_m33 = float4(
			_Positions[unity_InstanceID],
			1.0
		);
		unity_ObjectToWorld._m13 +=
			_Config.z * ((1.0 / 255.0) * (_Hashes[unity_InstanceID] >> 24) - 0.5);
		unity_ObjectToWorld._m00_m11_m22 = _Config.y;
	#endif
}
```

2.3 Using Positions as Input

Add the positions as input to `HashJob`, then have it retrieve the pre-generated position instead of calculating one itself. Then apply the domain transformation to it. From now on hashing no longer relies on the sample resolution, so those inputs can be removed.

```
	[ReadOnly]
	public NativeArray<float3> positions;

	[WriteOnly]
	public NativeArray<uint> hashes;

	//public int resolution;

	//public float invResolution;

	public SmallXXHash hash;

	public float3x4 domainTRS;

	public void Execute(int i) {
		//float vf = floor(invResolution * i + 0.00001f);
		//float uf = invResolution * (i - resolution * vf + 0.5f) - 0.5f;
		//vf = invResolution * (vf + 0.5f) - 0.5f;

		float3 p = mul(domainTRS, float4(positions[i], 1f));

		…
	}
```

Adjust the creation of `HashJob` in `OnEnable` to match.

```
	new HashJob {
		positions = positions,
		hashes = hashes,
		//resolution = resolution,
		//invResolution = 1f / resolution,
		hash = SmallXXHash.Seed(seed),
		domainTRS = domain.Matrix
	}.ScheduleParallel(hashes.Length, resolution, handle).Complete();
```

At this point we can visualize the hash as before, but now with two jobs running in sequence instead of a single job.

2.4 Shape Transformation

We apply a transformation to the domain when sampling and we can also apply a transformation when generating the shape. Add a `float3x4` position transformation to `Shapes.Job` for this and use it to modify the final position in `Execute`.

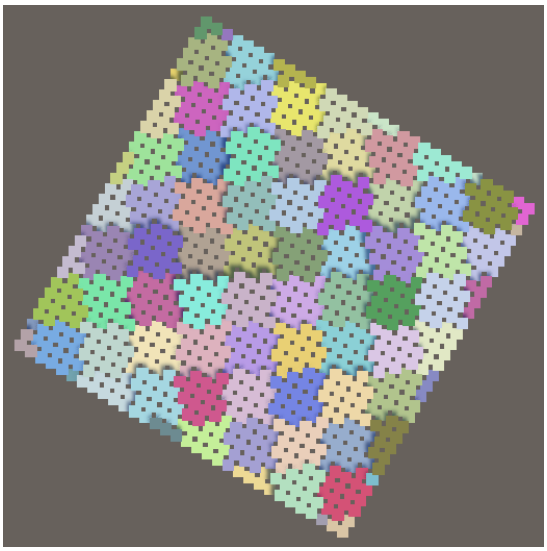
```
	public float3x4 positionTRS;

	public void Execute (int i) {
		…

		positions[i] = mul(positionTRS, float4(uv.x, 0f, uv.y, 1f));
	}
```

In this case we can directly use the matrix from the `Transform` component of the visualization game object, so the shape would transform as expected instead of remaining fixed at the origin. To make this easy, add a `float4x4` parameter to `ScheduleParallel` and pass the relevant 3×4 portion of it to the job.

```
	public static JobHandle ScheduleParallel (
		NativeArray<float3> positions, int resolution,
		float4x4 trs, JobHandle dependency
	) {
		return new Job {
			positions = positions,
			resolution = resolution,
			invResolution = 1f / resolution,
			positionTRS = float3x4(trs.c0.xyz, trs.c1.xyz, trs.c2.xyz, trs.c3.xyz)
		}.ScheduleParallel(positions.Length, resolution, dependency);
	}
```

When scheduling the job in `HashVisualization.OnEnable`, pass the local-to-world transformation matrix to it.

```
		JobHandle handle = Shapes.Job.ScheduleParallel(
			positions, resolution, transform.localToWorldMatrix, default
		);
```



Shape rotated 30° around Y.

Now the shape responds appropriately to the transformation of the game object, at least the initial one when play mode is entered. Note that the individual instances remain axis-aligned and unscaled.

> **Could we also rotate and scale the instance meshes?**
>
> Yes, by passing an appropriate matrix to the GPU and having *HashGPU* incorporate it in the final matrices. However, this can cause issues when nonuniform scales are used, because we assume that the scale is uniform. I leave the instances as they are.

2.5 Updating Positions

To respond to changes of the transformation while in play mode we'll have to run the jobs in `Update` instead of `OnEnable`. But we only need to run them after something changed, not all the time. We'll indicate this by adding a `bool isDirty` field to `HashVisualization`. Our visualization is considered dirty and in need of a refresh when something relevant has changed. This is always the case when `OnEnable` is invoked, so set `isDirty` to `true` there.

```
	bool isDirty;

	void OnEnable () {
		isDirty = true;

		…
	}
```

Check whether the visualization is dirty before drawing in `Update`. If so, set `isDirty` to `false`, run the jobs, and fill the buffers with fresh data.

```
    void Update () {
        if (isDirty) {
            isDirty = false;

            JobHandle handle = Shapes.Job.ScheduleParallel(
                positions, resolution, transform.localToWorldMatrix, default
            );

            new HashJob {
                positions = positions,
                hashes = hashes,
                hash = SmallXXHash.Seed(seed),
                domainTRS = domain.Matrix
            }.ScheduleParallel(hashes.Length, resolution, handle).Complete();

            hashesBuffer.SetData(hashes);
            positionsBuffer.SetData(positions);
        }

        Graphics.DrawMeshInstancedProcedural(…);
    }
```

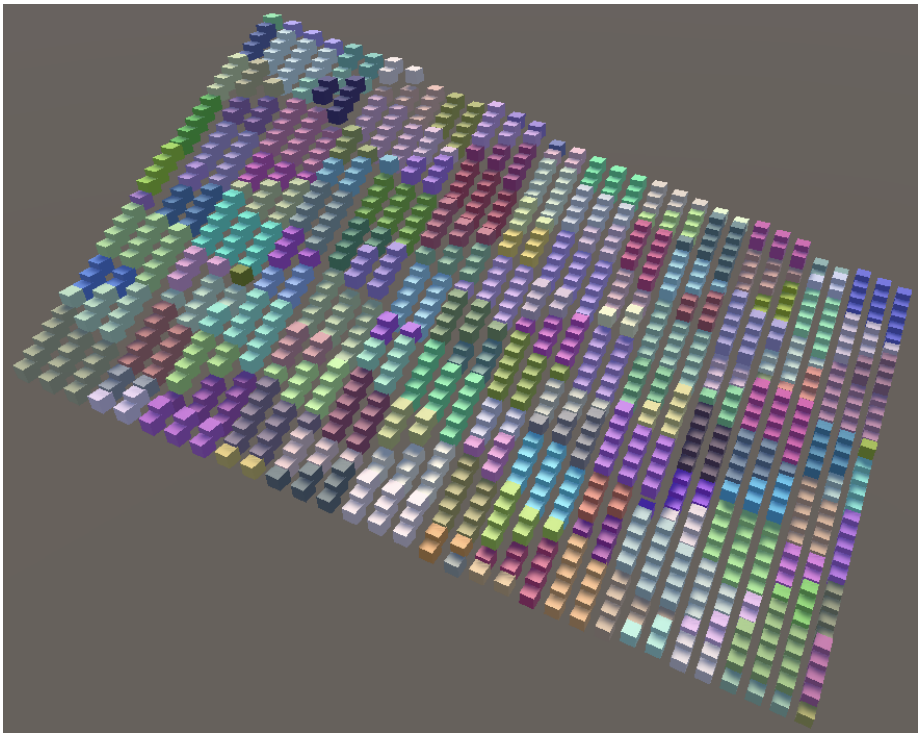
This means that we no longer need to perform that work in `OnEnable`.

```
    void OnEnable () {
        …

        //JobHandle handle = Shapes.Job.ScheduleParallel(
        //    positions, resolution, transform.localToWorldMatrix, default
        //);

        //new HashJob {
        //    …
        //}.ScheduleParallel(hashes.Length, resolution, handle).Complete();

        //hashesBuffer.SetData(hashes);
        //positionsBuffer.SetData(positions);

        propertyBlock ??= new MaterialPropertyBlock();
        …
    }
```

Finally, also refresh the visualization in `Update` after a transformation change happened. This is indicated by the `Transform.hasChanged` property, which is set to `true` after each change. It doesn't get set back to `false`, we have to do that manually after detecting a change.

```
        if (isDirty || transform.hasChanged) {
            isDirty = false;
            transform.hasChanged = false;

            …
        }
```

From now on the shape immediately responds to transformation adjustments while in play mode. This makes it possible to explore the hash volume by moving the shape through it.

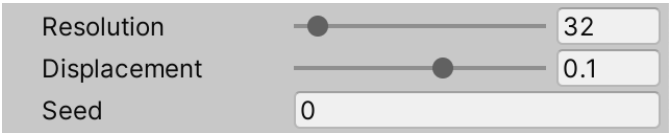


Rotated shape with nonuniform scale.

2.6 Displacement

If the plane can have an arbitrary orientation then it makes sense that the offset that we apply based on the hash should be along the plane's normal vector instead of always along the world Y axis. In general the direction of displacement depends on the shape, which might not be a flat plane at all, in which case each sample point would have its own displacement direction. To indicate this replace the `verticalOffset` field with a `displacement` field, which from now on will be expressed as units in world space instead of relative to the resolution.

```
	//[SerializeField, Range(-2f, 2f)]
	//float verticalOffset = 1f;

	[SerializeField, Range(-0.5f, 0.5f)]
	float displacement = 0.1f;

	…

	void OnEnable () {
		…

		propertyBlock ??= new MaterialPropertyBlock();
		propertyBlock.SetVector(configId, new Vector4(
			resolution, 1f / resolution, displacement
		));
	}
```



Displacement instead of vertical offset.

Displacement along the surface normal vector—away from the shape's surface—requires that `Shapes.Job` also outputs normal vectors. In case of our XZ plane the normal vector always points straight up, though the transformation should also be applied to it. Adjust it and the `ScheduleParallel` method as needed.

```
	public struct Job : IJobFor {

		[WriteOnly]
		NativeArray<float3> positions, normals;

		public float resolution, invResolution;

		public float3x4 positionTRS;

		public void Execute (int i) {
			…

			positions[i] = mul(positionTRS, float4(uv.x, 0f, uv.y, 1f));
			normals[i] = normalize(mul(positionTRS, float4(0f, 1f, 0f, 1f)));
		}

		public static JobHandle ScheduleParallel (
			NativeArray<float3> positions, NativeArray<float3> normals, int resolution,
			float4x4 trs, JobHandle dependency
		) {
			return new Job {
				positions = positions,
				normals = normals,
				…
			}.ScheduleParallel(positions.Length, resolution, dependency);
		}
	}
```

As displacement is done on the GPU `HashVisualization` has to send the normal vectors to the GPU, just like the positions. Add a shader identifier, array, and buffer for this purpose.

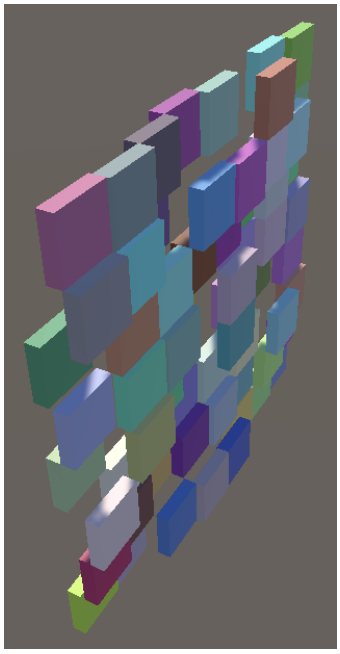
```
	static int
		hashesId = Shader.PropertyToID("_Hashes"),
		positionsId = Shader.PropertyToID("_Positions"),
		normalsId = Shader.PropertyToID("_Normals"),
		configId = Shader.PropertyToID("_Config");

	…

	NativeArray<float3> positions, normals;

	ComputeBuffer hashesBuffer, positionsBuffer, normalsBuffer;

	…

	void OnEnable () {
		isDirty = true;

		int length = resolution * resolution;
		hashes = new NativeArray<uint>(length, Allocator.Persistent);
		positions = new NativeArray<float3>(length, Allocator.Persistent);
		normals = new NativeArray<float3>(length, Allocator.Persistent);
		hashesBuffer = new ComputeBuffer(length, 4);
		positionsBuffer = new ComputeBuffer(length, 3 * 4);
		normalsBuffer = new ComputeBuffer(length, 3 * 4);

		propertyBlock ??= new MaterialPropertyBlock();
		propertyBlock.SetBuffer(hashesId, hashesBuffer);
		propertyBlock.SetBuffer(positionsId, positionsBuffer);
		propertyBlock.SetBuffer(normalsId, normalsBuffer);
		…
	}

	void OnDisable () {
		hashes.Dispose();
		positions.Dispose();
		normals.Dispose();
		hashesBuffer.Release();
		positionsBuffer.Release();
		normalsBuffer.Release();
		hashesBuffer = null;
		positionsBuffer = null;
		normalsBuffer = null;
	}
```

Pass the normal array to the job and copy it to the buffer in `Update`.

```
		JobHandle handle = Shapes.Job.ScheduleParallel(
			positions, normals, resolution, transform.localToWorldMatrix, default
		);

		new HashJob {
			…
		}.ScheduleParallel(hashes.Length, resolution, handle).Complete();

		hashesBuffer.SetData(hashes);
		positionsBuffer.SetData(positions);
		normalsBuffer.SetData(normals);
```

Then adjust *HashGPU* so it displaces along the normal instead of always vertically.

```
#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
	StructuredBuffer<uint> _Hashes;
	StructuredBuffer<float3> _Positions, _Normals;
#endif

float4 _Config;

void ConfigureProcedural () {
	#if defined(UNITY_PROCEDURAL_INSTANCING_ENABLED)
		unity_ObjectToWorld = 0.0;
		unity_ObjectToWorld._m03_m13_m23_m33 = float4(
			_Positions[unity_InstanceID],
			1.0
		);
		unity_ObjectToWorld._m03_m13_m23 +=
			(_Config.z * ((1.0 / 255.0) * (_Hashes[unity_InstanceID] >> 24) - 0.5)) *
			_Normals[unity_InstanceID];
		unity_ObjectToWorld._m00_m11_m22 = _Config.y;
	#endif
}
```



Rotated displacement.

2.7 Bounds

The transformation of the game object should also affect the bounds that we specify when drawing the visualization. We have to use the transformed position for the bound's center. Its size is a bit more complicated, because the bounds define an axis-aligned box while our shape can both rotate and scale, which can get even more complicated if the visualization is a child of another game object. So we'll take the transform's `lossyScale`, grab its largest absolute component via the `cmax` method, double that, add the displacement, and use that for the final scale in all three dimensions. This isn't the tightest possible fit, but it's easy to calculate and good enough for our purposes.

```
		Graphics.DrawMeshInstancedProcedural(
			instanceMesh, 0, material,
			new Bounds(
				transform.position,
				float3(2f * cmax(abs(transform.lossyScale)) + displacement)
			),
			hashes.Length, propertyBlock
		);
```

We don't have to recalculate the bounds every frame, only when the visualization is dirty. So we can store it in a field and reuse it.

```
	Bounds bounds;

	…

	void Update () {
		if (isDirty || transform.hasChanged) {
			…

			bounds = new Bounds(
				transform.position,
				float3(2f * cmax(abs(transform.lossyScale)) + displacement)
			);
		}

		Graphics.DrawMeshInstancedProcedural(
			instanceMesh, 0, material, bounds, hashes.Length, propertyBlock
		);
	}
```

3 Manual Vectorization

As noted earlier, automatic vectorization of our jobs failed after we introduced vector types. Typical automatic vectorization works by replacing operations performed on `float`, `int` and other such primitive values with the same operations performed on `float4`, `int4` and such. The job then executes four iterations in parallel, taking advantage of SIMD instructions. Unfortunately this is no longer possible for our jobs because we use `float3` values for positions and normal vectors. But with some work manual vectorization is possible.

3.1 Vectorized Hash

Our `SmallXXHash` struct is designed with automatic vectorization in mind. To support manual vectorization we need to create a new `SmallXXHash4` variant that operates on vectors of four values in parallel. Copy and rename the struct, keeping both in the same file. We also need to use the Mathematics library here from now on.

```
using Unity.Mathematics;

public readonly struct SmallXXHash { … }

public readonly struct SmallXXHash4 { … }
```

Make sure that all references to its own type inside `SmallXXHash4` refer to the vector version and not to the single-value version. Also, as there is no vector type for `byte` so remove that method from it, along with the `primeA` constant as no other method uses that value.

```
public readonly struct SmallXXHash4 {

    //const uint primeA = 0b10011110001101110111100110110001;
    const uint primeB = 0b10000101111010111100101001110111;
    const uint primeC = 0b11000010101100101010111000111101;
    const uint primeD = 0b00100111111010100111010110010111;
    const uint primeE = 0b00010110010101100110011110110001;

    readonly uint accumulator;

    public SmallXXHash4 (uint accumulator) {
        this.accumulator = accumulator;
    }

    public static implicit operator SmallXXHash4 (uint accumulator) =>
        new SmallXXHash4(accumulator);

    public static SmallXXHash4 Seed (int seed) => (uint)seed + primeE;

    static uint RotateLeft (uint data, int steps) =>
        (data << steps) | (data >> 32 - steps);

    public SmallXXHash4 Eat (int data) =>
        RotateLeft(accumulator + (uint)data * primeC, 17) * primeD;

    //public SmallXXHash Eat (byte data) =>
    //    RotateLeft(accumulator + data * primeE, 11) * primeA;

    public static implicit operator uint (SmallXXHash4 hash) { … }
}
```

Next, replace all occurrences of the `int` and `uint` types with `int4` and `uint4`. The only exception is the `steps` parameter of `RotateLeft`, which must remain a single `int`.

```
    readonly uint4 accumulator;

    public SmallXXHash4 (uint4 accumulator) {
        this.accumulator = accumulator;
    }

    public static implicit operator SmallXXHash4 (uint4 accumulator) =>
        new SmallXXHash4(accumulator);

    public static SmallXXHash4 Seed (int4 seed) => (uint4)seed + primeE;

    static uint4 RotateLeft (uint4 data, int steps) =>
        (data << steps) | (data >> 32 - steps);

    public SmallXXHash4 Eat (int4 data) =>
        RotateLeft(accumulator + (uint4)data * primeC, 17) * primeD;

    public static implicit operator uint4 (SmallXXHash4 hash) {
        uint4 avalanche = hash.accumulator;
        …
    }
```

This is enough to vectorize `SmallXXHash4`. Let's also make conversion from the single-value version to the vectorized version implicit by adding an operator for it to `SmallXXHash`.

```
public readonly struct SmallXXHash {

	…

	public static implicit operator SmallXXHash4 (SmallXXHash hash) =>
		new SmallXXHash4(hash.accumulator);
}
```

Can we also make a conversion in the other direction?

Conversion from one hash to a vector hash is simply a matter of duplicating the single value. But when going the other way we'd have to somehow reduce four independent hashes to a single hash. There is no straightforward way to do this, so we won't add a conversion for it.

3.2 Vectorized Hash Job

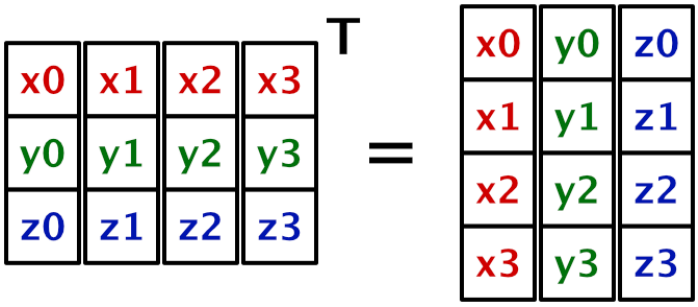
Next, we'll vectorize `HashJob`. Begin by replacing its `hashes` output and `hash` input with their four-component variants.

```
		[WriteOnly]
		public NativeArray<uint4> hashes;

		public SmallXXHash4 hash;
```

We'll also need to work with four positions in parallel. A sequence of four positions can be stored in a single `float3x4` matrix value, each of its columns containing a position.

```
		[ReadOnly]
		public NativeArray<float3x4> positions;
```

However, to vectorize our calculations `Execute` needs separate vectors for the x, y, and z components, not one vector per position. We can get this by transposing the positions matrix—via the `transpose` method—which gives us a `float4x3` matrix with the desired layout. Then we can vectorize the u, v, and w extraction as well, omitting the domain transformation for now.



Transposing 3×4 to 4×3.

```
	public void Execute(int i) {
		float4x3 p = transpose(positions[i]);

		int4 u = (int4)floor(p.c0);
		int4 v = (int4)floor(p.c1);
		int4 w = (int4)floor(p.c2);

		hashes[i] = hash.Eat(u).Eat(v).Eat(w);
	}
```

Now we need to pass vectorized arrays to the job in **`Update`**. We already have the required arrays, all we have to do is reinterpret them as if each element contained four **`float3`** values. **`NativeArray`** allows us to do this, by invoking the generic `Reinterpret` method for the desired type on it, with the original element size as an argument. As this reduces the effective array length for the job to a quarter, we have to divide the scheduled job length by four as well.

```
		new HashJob {
			positions = positions.Reinterpret<float3x4>(3 * 4),
			hashes = hashes.Reinterpret<uint4>(4),
			hash = SmallXXHash.Seed(seed),
			domainTRS = domain.Matrix
		}.ScheduleParallel(hashes.Length / 4, resolution, handle).Complete();
```

Note that this requires that the array lengths are divisible by four, otherwise reinterpretation will fail. So make sure that the resolution is even.

3.3 Vectorized Transformation

There is no existing `math` method that can multiply a 3×4 TRS matrix and a 4×3 XYZ-column matrix. Add a `TransformPositions` method to `HashJob` that does this for us, with the mentioned matrices as parameters, returning a transformed 4×3 XYZ-column matrix. Initially it can return the untransformed positions. Then use it in `Execute`.

```
		float4x3 TransformPositions (float3x4 trs, float4x3 p) => p;

		public void Execute(int i) {
			float4x3 p = TransformPositions(domainTRS, transpose(positions[i]));

			…
		}
```

The transformation is a regular matrix multiplication, except that the fourth TRS column—containing the translation—is simply added. In the previous matrix multiplication the last part was multiplied with the constant 1, but we can omit that.

Because the position components are vectorized the multiplication steps are performed on entire columns of the position matrix, instead of on single components.

```
		float4x3 TransformPositions (float3x4 trs, float4x3 p) => float4x3(
			trs.c0.x * p.c0 + trs.c1.x * p.c1 + trs.c2.x * p.c2 + trs.c3.x,
			trs.c0.y * p.c0 + trs.c1.y * p.c1 + trs.c2.y * p.c2 + trs.c3.y,
			trs.c0.z * p.c0 + trs.c1.z * p.c1 + trs.c2.z * p.c2 + trs.c3.z
		);
```

At this point our visualization works as before, except that `HashJob` is now vectorized. The Burst inspector's diagnostics view still indicates that the job is not vectorized, because Burst doesn't know that we have done this manually. Inspecting the assembly will show that the job does use SIMD instructions and generates four hashes in parallel.

3.4 Vectorized Shapes Job

Finally, we vectorize `Shapes.Job` using the same approach. Begin by changing the element type of the position and normal arrays to `float3x4`.

```
		[WriteOnly]
		NativeArray<float3x4> positions, normals;

		…

		public static JobHandle ScheduleParallel (
			NativeArray<float3x4> positions, NativeArray<float3x4> normals,
			int resolution, float4x4 trs, JobHandle dependency
		) { … }
```

In `Execute`, vectorize the calculation of the UV coordinates by using a 4×2 UV-column matrix.

```
		float4x2 uv;
		uv.c1 = floor(invResolution * i + 0.00001f);
		uv.c0 = invResolution * (i - resolution * uv.c1 + 0.5f) - 0.5f;
		uv.c1 = invResolution * (uv.c1 + 0.5f) - 0.5f;
```

We have to replace the single index parameter with the corresponding four vectorized indices, which is done by multiplying the original by four and then adding zero, 1, 2, and 3 to the individual indices.

```
		float4 i4 = 4f * i + float4(0f, 1f, 2f, 3f);
		uv.c1 = floor(invResolution * i4 + 0.00001f);
		uv.c0 = invResolution * (i4 - resolution * uv.c1 + 0.5f) - 0.5f;
```

Next, we also need to apply a TRS transformation, but now for both positions and for normal vectors. To support both with a single method copy `TransformPositions` from **`HashJob`** to **`Shapes.Job`**, rename it to `TransformVectors`, and add a **`float`** `w` parameter set to 1 by default. Multiply the translation portion by this value.

```
	float4x3 TransformVectors (float3x4 trs, float4x3 p, float w = 1f) => float4x3(
		trs.c0.x * p.c0 + trs.c1.x * p.c1 + trs.c2.x * p.c2 + trs.c3.x * w,
		trs.c0.y * p.c0 + trs.c1.y * p.c1 + trs.c2.y * p.c2 + trs.c3.y * w,
		trs.c0.z * p.c0 + trs.c1.z * p.c1 + trs.c2.z * p.c2 + trs.c3.z * w
	);
```

In `Execute`, generate the 4×3 XYZ-column matrix with planar positions, apply the position TRS, then transpose it so it can be assigned to the positions output element.

```
		positions[i] =
			transpose(TransformVectors(positionTRS, float4x3(uv.c0, 0f, uv.c1)));
```

Do the same for the normal vectors, but this time with zero passed as the third argument to `TransformVectors` so the translation is ignored. Burst will eliminate anything multiplied with constant zero. Then normalize the vectors stored in the columns of the 3×4 matrix.

```
		float3x4 n =
			transpose(TransformVectors(positionTRS, float4x3(0f, 1f, 0f), 0f));
		normals[i] = float3x4(
			normalize(n.c0), normalize(n.c1), normalize(n.c2), normalize(n.c3)
		);
```

3.5 Vectorized Arrays

Because both jobs now require vectorized arrays, let's directly define the arrays as such in `HashVisualization` instead of reinterpreting them.

```
	NativeArray<uint4> hashes;

	NativeArray<float3x4> positions, normals;
```

In `OnEnable`, divide the calculated length by four before creating the arrays and buffers.

```
	void OnEnable () {
		isDirty = true;

		int length = resolution * resolution;
		length /= 4;
		hashes = new NativeArray<uint4>(length, Allocator.Persistent);
		positions = new NativeArray<float3x4>(length, Allocator.Persistent);
		normals = new NativeArray<float3x4>(length, Allocator.Persistent);
		…
	}
```

To keep the compute buffers at the same size we now have to quadruple the length for them.

```
		hashesBuffer = new ComputeBuffer(length * 4, 4);
		positionsBuffer = new ComputeBuffer(length * 4, 3 * 4);
		normalsBuffer = new ComputeBuffer(length * 4, 3 * 4);
```

This requires us to still reinterpret once, when copying the data to the buffers in `Update`.

```
			hashesBuffer.SetData(hashes.Reinterpret<uint>(4 * 4));
			positionsBuffer.SetData(positions.Reinterpret<float3>(3 * 4 * 4));
			normalsBuffer.SetData(normals.Reinterpret<float3>(3 * 4 * 4));
```

Can't we directly copy the vectorized data to the compute buffers?

Although the CPU and GPU both interpret the compute buffer data in their own way, it turns out that in some situations data misalignment can happen. So we have to make sure that the compute buffers explicitly use the un-vectorized data.

At this point we can also support odd resolutions again. For example, if the resolution is 3 then the initial length is 9 and the vectorized length would become 2, which only support 8 elements. We can fit the ninth element by adding 1 to the final length. This means that we'll add four values, three of which are superfluous, but that's insignificant overhead. We can make this work for all resolutions by adding the least-significant bit of the initial length to the vectorized length in `OnEnable`, because it's zero for even and 1 for odd values.

```
		int length = resolution * resolution;
		length = length / 4 + (length & 1);
```

We no longer need to reinterpret the arrays for `HashJob` in `Update`. We also have to again use the array's length while scheduling, as it has already been reduced by vectorization.

```
			new HashJob {
				positions = positions,
				hashes = hashes,
				hash = SmallXXHash.Seed(seed),
				domainTRS = domain.Matrix
			}.ScheduleParallel(hashes.Length, resolution, handle).Complete();
```

Finally, to draw the correct amount of instances pass the square resolution to `Graphics.DrawMeshInstancedProcedural`.

```
		Graphics.DrawMeshInstancedProcedural(
			instanceMesh, 0, material, bounds, resolution * resolution, propertyBlock
		);
```

4 More Shapes

Now that both our jobs are vectorized we're going to add two alternative sample shapes. We could do this by creating extra jobs, but most of the code of those jobs would be identical. So we'll instead go for a template-based approach.

4.1 Plane Struct

To support multiple shapes we'll hoist the code for generating positions and normals out of `Shapes.Job` up into the `Shapes` class. Each shape could be generated in its own unique way, but we'll base them all on a UV grid. To avoid code repetition copy the UV code from `Job.Execute` to a new static `IndexTo4UV` method inside `Shapes`. Adjust it so the UV range becomes 0–1 instead of −0.5–0.5, by omitting the subtractions of 0.5.

```
public static class Shapes {

	public static float4x2 IndexTo4UV (int i, float resolution, float invResolution) {
		float4x2 uv;
		float4 i4 = 4f * i + float4(0f, 1f, 2f, 3f);
		uv.c1 = floor(invResolution * i4 + 0.00001f);
		uv.c0 = invResolution * (i4 - resolution * uv.c1 + 0.5f);
		uv.c1 = invResolution * (uv.c1 + 0.5f);
		return uv;
	}

	…
}
```

The only unique portion of each shape job will be how it sets the positions and normals. To pass this data around introduce a `Point4` struct into `Shapes` that is a simple container for vectorized positions and normals.

```
public static class Shapes {

	public struct Point4 {
		public float4x3 positions, normals;
	}

	…
}
```

To extract the code for generating a plane from `Job.Execute`, add a `Plane` struct type to `Shapes` that contains a `GetPoint4` method that uses an index, resolution, and inverse resolution to produce a `Point4` value. Because we changed the range of the UV we now have to subtract 0.5 from the UV here to keep the plane centered on the origin.

```
	public struct Plane {

		public Point4 GetPoint4 (int i, float resolution, float invResolution) {
			float4x2 uv = IndexTo4UV(i, resolution, invResolution);
			return new Point4 {
				positions = float4x3(uv.c0 - 0.5f, 0f, uv.c1 - 0.5f),
				normals = float4x3(0f, 1f, 0f)
			};
		}
	}
```

The `Plane` struct doesn't contain any fields, its only purpose it to provide the `GetPoint4` method. We can access this method in `Job.Execute` by invoking it on the default `Plane` value, via `default(Plane)`. This replaces the explicit plane-related code.

```
		public void Execute (int i) {
			//float4x2 uv;
			//float4 i4 = 4f * i + float4(0f, 1f, 2f, 3f);
			//uv.c1 = floor(invResolution * i4 + 0.00001f);
			//uv.c0 = invResolution * (i4 - resolution * uv.c1 + 0.5f) - 0.5f;
			//uv.c1 = invResolution * (uv.c1 + 0.5f) - 0.5f;

			Point4 p = default(Plane).GetPoint4(i, resolution, invResolution);

			positions[i] = transpose(TransformVectors(positionTRS, p.positions));

			float3x4 n = transpose(TransformVectors(positionTRS, p.normals, 0f));
			normals[i] = float3x4(
				normalize(n.c0), normalize(n.c1), normalize(n.c2), normalize(n.c3)
			);
		}
```

4.2 Shape Interface

The idea is that we should be able to replace `Plane` with a different struct type to generate different shapes. To make this work for generic shapes we'll introduce an interface type, which acts like a contract for classes or structs, mandating what public methods or properties they must have.

An interface is declared with the `interface` keyword and the convention is to prefix its type name with an I, so we'll name it `IShape`. Define the GetPoint4 method signature inside it, so without a code body but terminated with a semicolon. Interface members are public by definition, so it has no explicit access modifier. `Plane` can then implement the interface, by extending it.

```
public interface IShape {
    Point4 GetPoint4 (int i, float resolution, float invResolution);
}

public struct Plane : IShape {

    public Point4 GetPoint4 (int i, float resolution, float invResolution) { … }
}
```

4.3 Generic Job

The next step is to make our job generic, turning it into a template for jobs. We do this by appending a generic type parameter within angle brackets to the type declaration of `Job`. Type parameter names are a single letter by conventions. As this parameter will represent a shape type let's name it `S`.

```
	public struct Job<S> : IJobFor { … }
```

We want to limit what `S` can be in two ways. First, we expect it to be a struct type. Second, it must implement the `IShape` interface. We can declare this by appending `where S : struct, IShape` to the type declaration of `Job`.

```
	public struct Job<S> : IJobFor where S : struct, IShape { … }
```

Now we can use the generic type in `Execute` instead of an explicit type.

```
		public void Execute (int i) {
			Point4 p = default(S).GetPoint4(i, resolution, invResolution);

			…
		}
```

We also have to specify what kind of job we're creating in our `ScheduleParallel` method, just like when we create a `NativeArray` or other generic type value. It should be the same job type, so we pass `S` as the generic parameter.

```
			return new Job<S> {
				…
			}.ScheduleParallel(positions.Length, resolution, dependency);
```

Finally, to make this work we have to be explicit about what shape job we're scheduling in `HashVisualization.Update`.

```
			JobHandle handle = Shapes.Job<Shapes.Plane>.ScheduleParallel(
				positions, normals, resolution, transform.localToWorldMatrix, default
			);
```

Note that our plane job is now listed as *Shapes.Job`1[Shapes.Plane]* in the Burst inspector instead of just *Shapes.Job*. Besides that the generated assembly code is the same as before.

4.4 Sphere and Torus

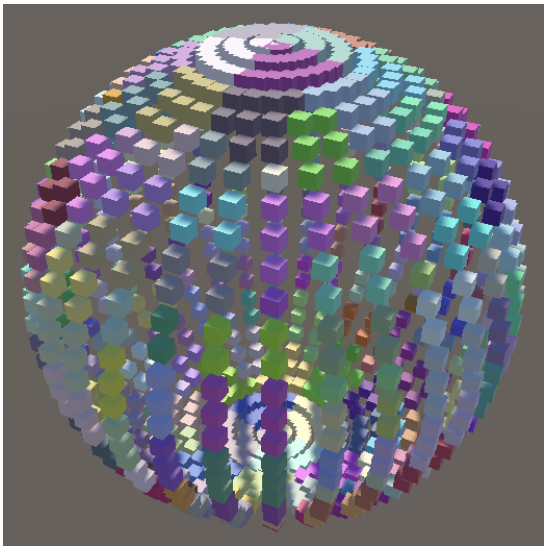
It is now possible to easily add more shape jobs. We'll add two more, beginning with a sphere. We can copy and adapt the code from the Mathematical Surfaces tutorial for this. The only differences are that due to the different UV range we have to double all arguments of the `sin` and `cos` methods and have to swap `sin` and `cos` for the calculation of `s` and `c1`. We set its radius to 0.5 so it fits inside a unit cube.

Because it's a sphere we can directly use the positions for the normal vectors. The length of these vectors is 0.5 but that's not a problem because the job normalizes them later, after applying the space transformation.

```
	public struct Plane : IShape { … }

	public struct Sphere : IShape {

		public Point4 GetPoint4 (int i, float resolution, float invResolution) {
			float4x2 uv = IndexTo4UV(i, resolution, invResolution);

			float r = 0.5f;
			float4 s = r * sin(PI * uv.c1);

			Point4 p;
			p.positions.c0 = s * sin(2f * PI * uv.c0);
			p.positions.c1 = r * cos(PI * uv.c1);
			p.positions.c2 = s * cos(2f * PI * uv.c0);
			p.normals = p.positions;
			return p;
		}
	}
```

To use the sphere shape all we have to do is change the generic type argument of `Shapes.Job` in `HashVisualization.Update`.

```
			JobHandle handle = Shapes.Job<Shapes.Sphere>.ScheduleParallel(
				positions, normals, resolution, transform.localToWorldMatrix, default
			);
```



Sphere shape, zero displacement.

The second shape that we add is the torus, also copied from Mathematical Surfaces, once again doubling the arguments for the `sin` and `cos` methods. Use 0.375 for `r1` and 0.125 for `r2`.

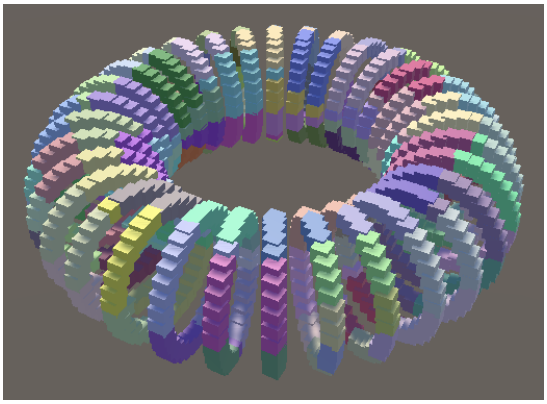
```
	public struct Torus : IShape {

		public Point4 GetPoint4 (int i, float resolution, float invResolution) {
			float4x2 uv = IndexTo4UV(i, resolution, invResolution);

			float r1 = 0.375f;
			float r2 = 0.125f;
			float4 s = r1 + r2 * cos(2f * PI * uv.c1);

			Point4 p;
			p.positions.c0 = s * sin(2f * PI * uv.c0);
			p.positions.c1 = r2 * sin(2f * PI * uv.c1);
			p.positions.c2 = s * cos(2f * PI * uv.c0);
			p.normals = p.positions;
			return p;
		}
	}
```

The surface normals for a torus are a bit more complicated than for a sphere. Instead of all pointing away from the center they have to point away from the ring inside the torus. We don't need to worry about avoiding duplicate calculations here because Burst eliminates those.

```
			p.normals = p.positions;
			p.normals.c0 -= r1 * sin(2f * PI * uv.c0);
			p.normals.c2 -= r1 * cos(2f * PI * uv.c0);
```

Now we can switch to sampling with a torus shape.

```
			JobHandle handle = Shapes.Job<Shapes.Torus>.ScheduleParallel(
				positions, normals, resolution, transform.localToWorldMatrix, default
			);
```



Torus shape, zero displacement.

4.5 Selecting Shapes

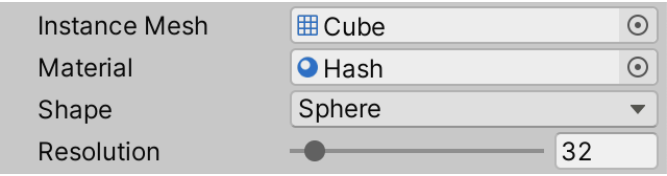
To make it possible to switch shapes via the inspector we'll have to add code to invoke `ScheduleParallel` on the correct job type. Again we'll base our approach on Mathematical Surfaces, using a selection enum and a static delegate array.

Add a `ScheduleDelegate` delegate type to `Shapes` that matches the signature of `ScheduleParallel`.

```
	public delegate JobHandle ScheduleDelegate (
		NativeArray<float3x4> positions, NativeArray<float3x4> normals,
		int resolution, float4x4 trs, JobHandle dependency
	);
```

Which shapes are available is up to `HashVisualization`, so we'll add the selection enum and delegate array there, along with a shape configuration field.

```
	public enum Shape { Plane, Sphere, Torus }

	static Shapes.ScheduleDelegate[] shapeJobs = {
		Shapes.Job<Shapes.Plane>.ScheduleParallel,
		Shapes.Job<Shapes.Sphere>.ScheduleParallel,
		Shapes.Job<Shapes.Torus>.ScheduleParallel
	};

	…

	[SerializeField]
	Shape shape;
```



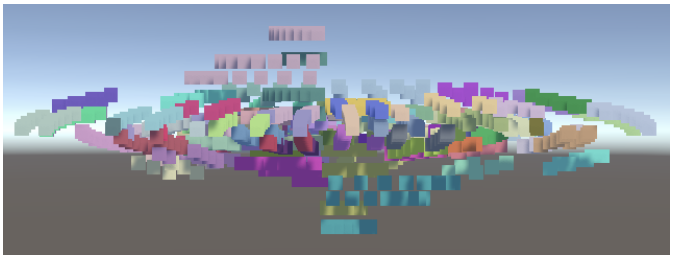
Shape dropdown selection menu.

Then adjust `Update` so the selected shape is generated.

```
			JobHandle handle = shapeJobs[(int)shape](
				positions, normals, resolution, transform.localToWorldMatrix, default
			);
```

4.6 Transformed Normals

Everything appears to work as expected at this point, except when using a nonuniform scale for the sphere or torus. For example, when almost flattening the sphere the displacement is still away from its center, instead of directly away from its implied surface. The result is that the displacement is too flat.



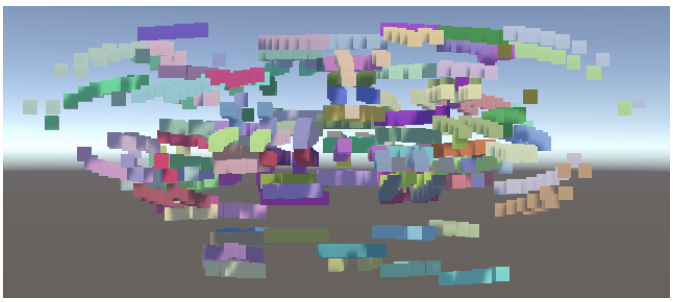
Sphere with Y scale 0.01 and 0.5 displacement.

This happens because nonuniform scaling messes up normal vectors. They have to be multiplied with a different transformation matrix. To fix this, add input for a separate normal transformation to `Shapes.Job` and use it to transform the normals.

```
		public float3x4 positionTRS, normalTRS;

		…

		public void Execute (int i) {
			…

			float3x4 n = transpose(TransformVectors(normalTRS, p.normals, 0f));
			…
		}
```

To produce correct surface normal vectors we have to transform them with the transpose of the inverse 4×4 TRS matrix, which we can get via `transpose(inverse(trs))` in `ScheduleParallel`.

```
		public static JobHandle ScheduleParallel (
			NativeArray<float3x4> positions, NativeArray<float3x4> normals,
			int resolution, float4x4 trs, JobHandle dependency
		) {
			float4x4 tim = transpose(inverse(trs));
			return new Job<S> {
				…
				positionTRS = float3x4(trs.c0.xyz, trs.c1.xyz, trs.c2.xyz, trs.c3.xyz),
				normalTRS = float3x4(tim.c0.xyz, tim.c1.xyz, tim.c2.xyz, tim.c3.xyz)
			}.ScheduleParallel(positions.Length, resolution, dependency);
		}
```



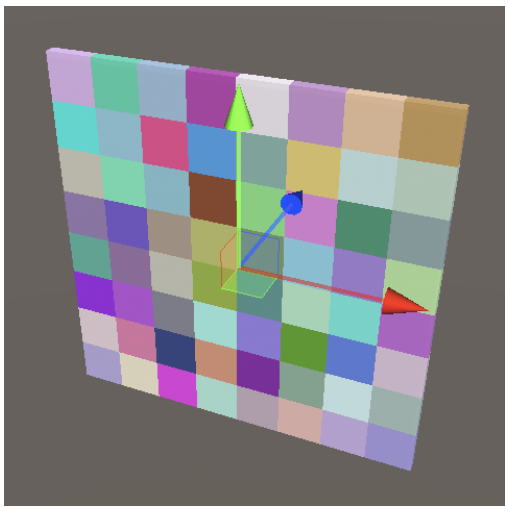
Correct displacement.

4.7 Octahedron Sphere

The sphere that we currently generate is known as a UV sphere. It consists of rings that degenerate into points at its two poles. The distribution of points is obviously not uniform. Near the poles the points are too close together and near the equator the points are too far apart. So let's switch to a different approach, generating an octahedron sphere instead.

An octahedron sphere is made by first generating an octahedron at the origin and then normalizing all its position vectors. It is possible to generate an octahedron from 0–1 UV coordinates in a few steps. We begin with an XY plane centered at the origin.

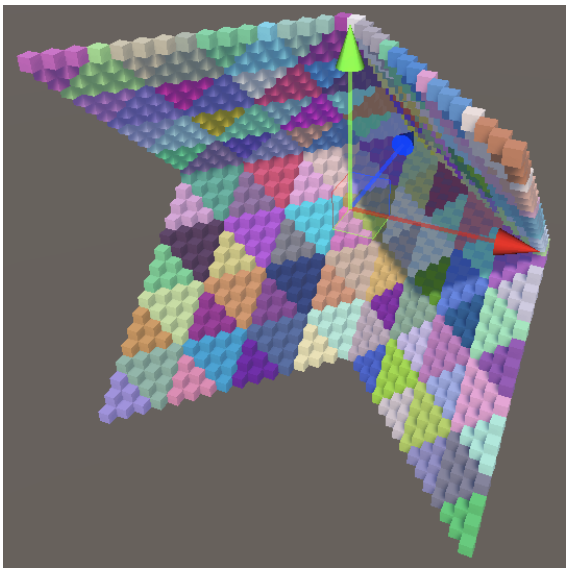
```
	public struct Sphere : IShape {

		public Point4 GetPoint4 (int i, float resolution, float invResolution) {
			float4x2 uv = IndexTo4UV(i, resolution, invResolution);

			//float r = 0.5f;
			//float4 s = r * sin(PI * uv.c1);

			Point4 p;
			p.positions.c0 = uv.c0 - 0.5f;
			p.positions.c1 = uv.c1 - 0.5f;
			p.positions.c2 = 0f;
			p.normals = p.positions;
			return p;
		}
	}
```



XY plane, zero displacement.

The second step is to displace the plane along Z to create facets, by making Z equal to 0.5 minus absolute X and absolute Z. This produces something that looks like an octahedron that's folded open on one side.

```
		p.positions.c2 = 0.5f - abs(p.positions.c0) - abs(p.positions.c1);
```



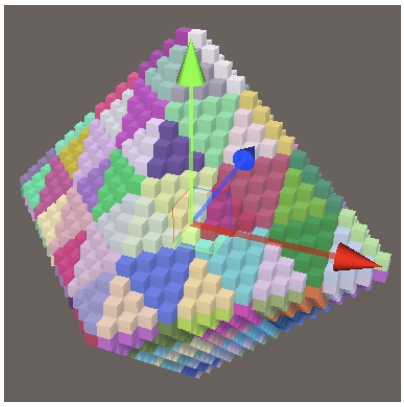
Octahedron folded open.

On the positive Z side the octahedron is already complete. To close the octahedron on the negative Z side, we need an offset equal to negative Z, with a minimum of zero to leave the positive side unaffected.

```
		p.positions.c2 = 0.5f - abs(p.positions.c0) - abs(p.positions.c1);
		float4 offset = max(-p.positions.c2, 0f);
```

We have to either add or subtract this offset to or from X and Y independently. If X is negative then add, otherwise subtract. The same goes for Y. To make this choice for vector data we have to use the `select` method. Its signature is `select(valueIfFalse, valueIfTrue, condition)`.

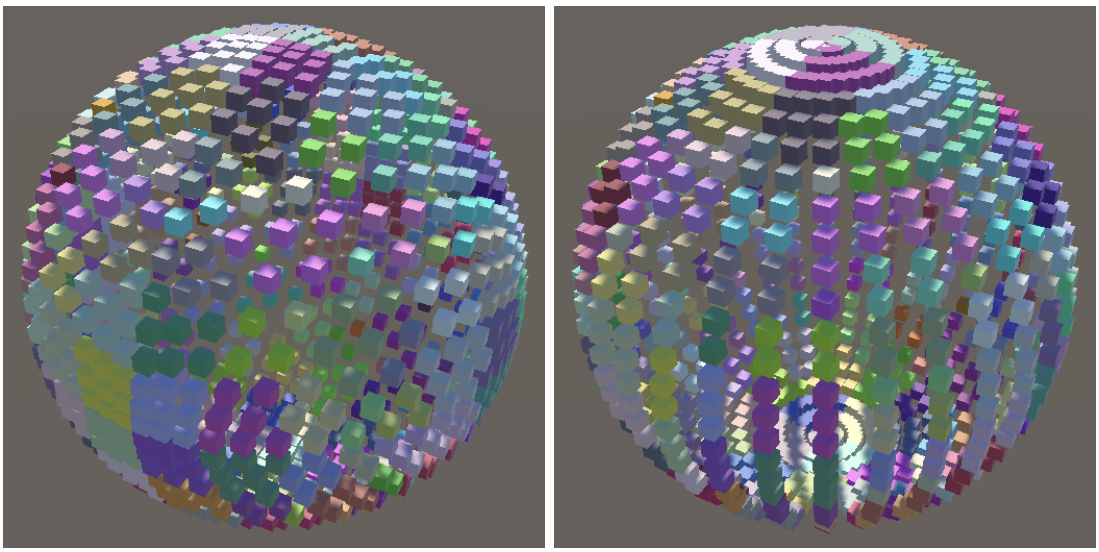
```
		float4 offset = max(-p.positions.c2, 0f);
		p.positions.c0 += select(-offset, offset, p.positions.c0 < 0f);
		p.positions.c1 += select(-offset, offset, p.positions.c1 < 0f);
```



Octahedron closed.

Finally, to turn the octahedron into a sphere with radius 0.5 we have to scale by 0.5 divided by the vector length, for which we can use the Pythagorean theorem and the `rsqrt` method.

```
		p.positions.c1 += select(-offset, offset, p.positions.c1 < 0f);

		float4 scale = 0.5f * rsqrt(
			p.positions.c0 * p.positions.c0 +
			p.positions.c1 * p.positions.c1 +
			p.positions.c2 * p.positions.c2
		);
		p.positions.c0 *= scale;
		p.positions.c1 *= scale;
		p.positions.c2 *= scale;
		p.normals = p.positions;
```

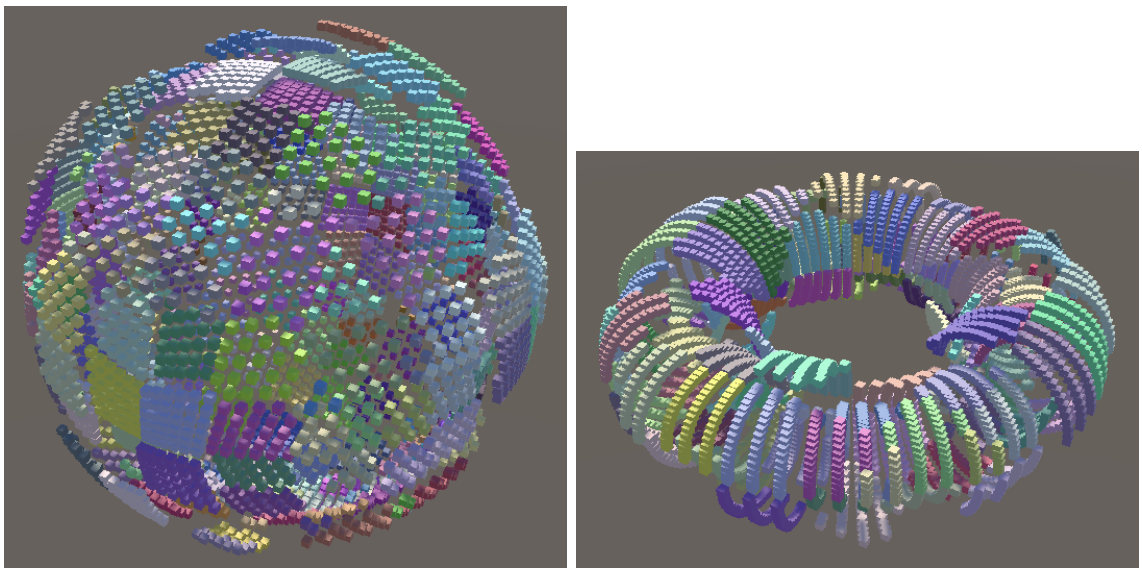


Octahedron sphere and UV sphere.

Compared to the UV sphere, the octahedron sphere has six instead of two regions where points clump together, but its point distribution is more uniform.

4.8 Instance Scale

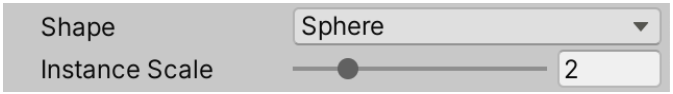
The sample points of the sphere and torus are further apart than those of the plane, which makes it harder to see their surfaces due to the empty space between the instances.



Resolution 64 with 0.1 displacement.

We wrap up this tutorial by adding a configurable instance scale to `HashVisualization`, which could be used to make the visualization more solid, or even more sparse.

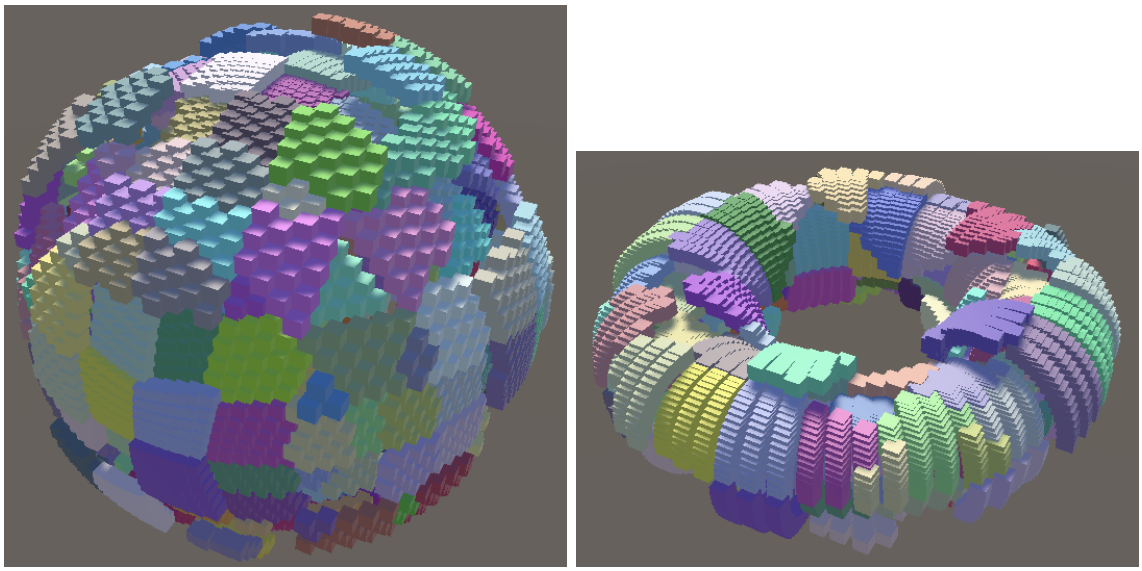
```
	[SerializeField, Range(0.1f, 10f)]
	float instanceScale = 2f;
```



Instance scale slider.

The instance scale is applied by dividing it by the resolution in `OnEnable` and sending that to the GPU, instead of the inverse resolution.

```
		propertyBlock.SetVector(configId, new Vector4(
			resolution, instanceScale / resolution, displacement
		));
```



Instance scale 2.

The next tutorial is Value Noise.

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