Catlike Coding

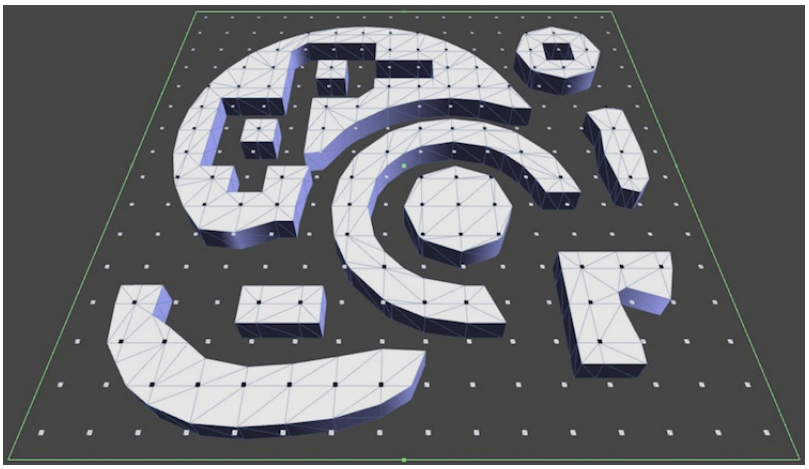
Unity C# Tutorials

Marching Squares 4 Erecting Walls

Refactor code.
Construct walls.
Add normals.

In this tutorial we'll add some depth to Marching Squares.

This tutorial comes after Marching Squares 3. Like the previous ones, it has been made with Unity 4.5.2 and might not work for older versions.

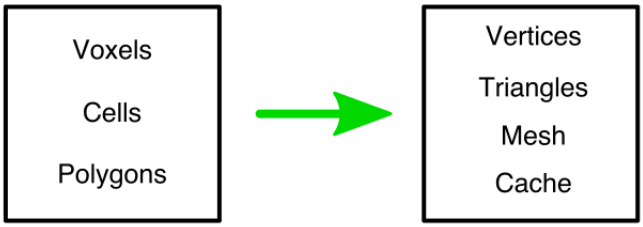


You'll add some depth to your shapes.

Refactoring Code

While Marching Squares is strictly 2D, we have no obligation to keep its visualization perfectly flat. For example, we could place walls at the borders between filled and empty space. However, that entails triangulating even more stuff, and `VoxelGrid` is already quite a large class. So let's see if we can split it up.

The main responsibility of `VoxelGrid` is to manage voxels and figure out how to triangulate them. To do so it also needs to manage a vertex cache and know how to convert polygons into mesh data. But if we were to change our meshing approach it wouldn't really affect the voxel management part. This indicates that we could isolate these chunks of functionality from each other.



Separation of concerns.

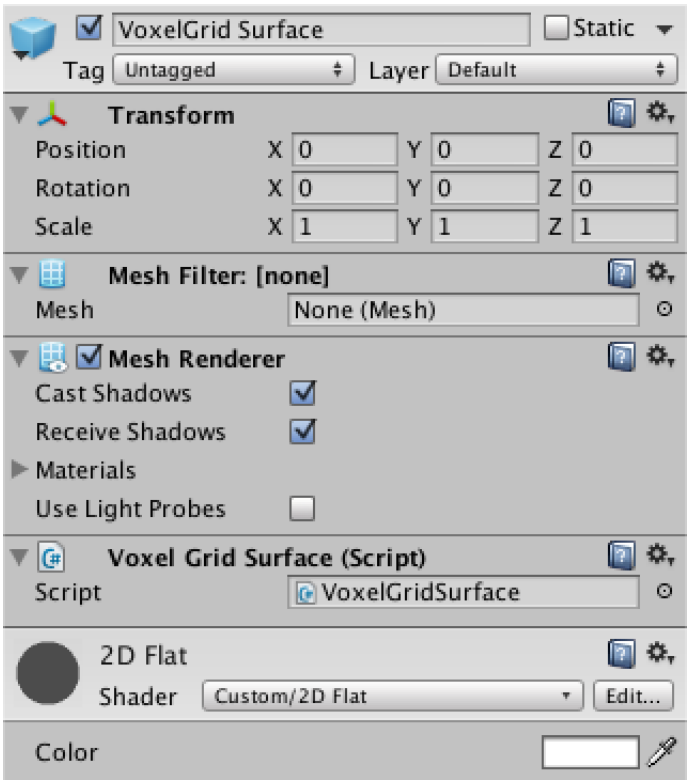
So let's create a `VoxelGridSurface` class to take care of the mesh, vertices, triangles, and caches. Just copy the existing fields from `VoxelGrid`.

```
using UnityEngine;
using System.Collections.Generic;

public class VoxelGridSurface : MonoBehaviour {

	private Mesh mesh;

	private List<Vector3> vertices;
	private List<int> triangles;

	private int[] rowCacheMax, rowCacheMin;
	private int edgeCacheMin, edgeCacheMax;

}
```

Also copy the initialization code into a new `Initialize` method. Adjust the mesh's name to indicate that it is now managed by a different class.

```
	public void Initialize (int resolution) {
		GetComponent<MeshFilter>().mesh = mesh = new Mesh();
		mesh.name = "VoxelGridSurface Mesh";
		vertices = new List<Vector3>();
		triangles = new List<int>();
		rowCacheMax = new int[resolution * 2 + 1];
		rowCacheMin = new int[resolution * 2 + 1];
	}
```

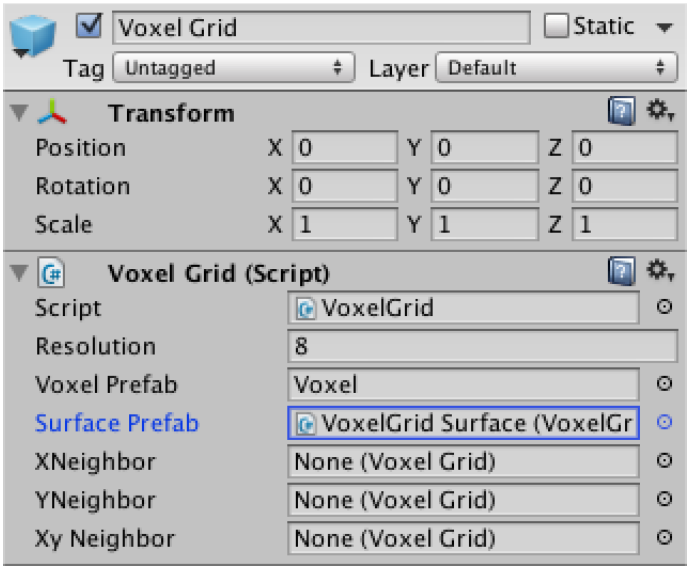
Now create a prefab with our new `VoxelGridSurface` component. It also needs a `MeshFilter` and a `MeshRenderer` with our flat material, just like the *Voxel Grid* prefab.



The surface as a prefab.

Add a prefab variable to `VoxelGrid`, so we can connect the prefabs. As it will no longer create a mesh for itself, the *Voxel Grid* prefab no longer needs its mesh components.

```
	public VoxelGridSurface surfacePrefab;
```



Trimmed-down grid prefab.

> **Why a separate prefab?**
>
> We could also have `VoxelGrid` construct a surface instance through code, or bake the surface instance into the grid prefab. The choice of a separate prefab means that all the details of construction and mesh components are completely removed from the voxel grid itself.
>
> Also, it will make it easier to attach multiple surfaces to a single voxel grid. While this is not something we're interested in right now, we might in a future tutorial.

`VoxelGrid` now has to instantiate the surface prefab when it is initialized, instead of creating its own mesh and cache data. To position it correctly, make it a child of the grid.

```
	private VoxelGridSurface surface;

	public void Initialize (int resolution, float size, float maxFeatureAngle) {
		…

		for (int i = 0, y = 0; y < resolution; y++) {
			for (int x = 0; x < resolution; x++, i++) {
				CreateVoxel(i, x, y);
			}
		}

		surface = Instantiate(surfacePrefab) as VoxelGridSurface;
		surface.transform.parent = transform;
		surface.transform.localPosition = Vector3.zero;
		surface.Initialize(resolution);

		Refresh();
	}
```

Now transplant the `AddTriangle`, `AddQuad`, `AddPentagon`, and `AddHexagon` methods from `VoxelGrid` to `VoxelGridSurface`. Do the same with the other `Add…` methods – `AddTriangleA` and so on – and make them public.

We now get lots of compile errors in `VoxelGrid`, as its methods are gone. To fix this, prepend `surface.` to all invocations of those methods, redirecting them to the surface instance. This fixes the errors.

After that, delete the mesh and cache fields from `VoxelGrid`. You can also remove the `using System.Collections.Generic` statement, as we no longer create any generic lists here.

Of course we again get compile errors. Let's start by fixing `Triangulate`. It tries to clear and apply changes to the mesh, which is now gone. Instead, delegate these tasks to the surface.

```
	private void Triangulate () {
		surface.Clear();
		FillFirstRowCache();
		TriangulateCellRows();
		if (yNeighbor != null) {
			TriangulateGapRow();
		}
		surface.Apply();
	}
```

The old code gets a new home in `VoxelGridSurface`.

```
	public void Clear () {
		vertices.Clear();
		triangles.Clear();
		mesh.Clear();
	}

	public void Apply () {
		mesh.vertices = vertices.ToArray();
		mesh.triangles = triangles.ToArray();
	}
```

Now we have to deal with the caching code. Let's extract the minimum code that accesses the caches and vertices and put it in new methods for `VoxelGridSurface`. While we are at it, we can simplify the caching of edge intersections by using the edge point properties that we added in the previous tutorial.

```
	public void CacheFirstCorner (Voxel voxel) {
		rowCacheMax[0] = vertices.Count;
		vertices.Add(voxel.position);
	}

	public void CacheNextCorner (int i, Voxel voxel) {
		rowCacheMax[i + 2] = vertices.Count;
		vertices.Add(voxel.position);
	}

	public void CacheXEdge (int i, Voxel voxel) {
		rowCacheMax[i + 1] = vertices.Count;
		vertices.Add(voxel.XEdgePoint);
	}

	public void CacheYEdge (Voxel voxel) {
		edgeCacheMax = vertices.Count;
		vertices.Add(voxel.YEdgePoint);
	}

	public void PrepareCacheForNextCell () {
		yEdgeMin = yEdgeMax;
	}

	public void PrepareCacheForNextRow () {
		int[] rowSwap = rowCacheMin;
		rowCacheMin = rowCacheMax;
		rowCacheMax = rowSwap;
	}
```

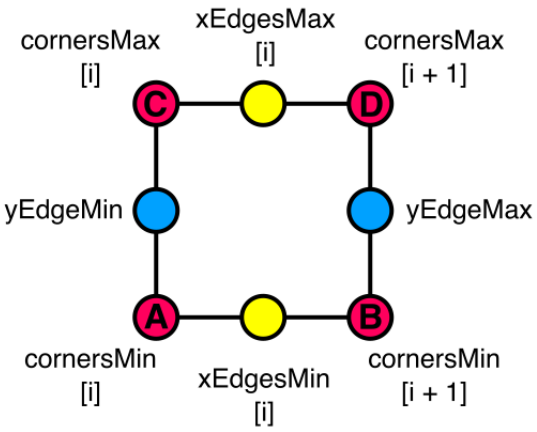
From now on `VoxelGrid` will have to use these methods.

```
	private void CacheFirstCorner (Voxel voxel) {
		if (voxel.state) {
			surface.CacheFirstCorner(voxel);
		}
	}

	private void CacheNextEdgeAndCorner (int i, Voxel xMin, Voxel xMax) {
		if (xMin.state != xMax.state) {
			surface.CacheXEdge(i, xMin);
		}
		if (xMax.state) {
			surface.CacheNextCorner(i, xMax);
		}
	}

	private void CacheNextMiddleEdge (Voxel yMin, Voxel yMax) {
		surface.PrepareCacheForNextCell();
		if (yMin.state != yMax.state) {
			surface.CacheYEdge(yMin);
		}
	}

	private void SwapRowCaches () {
		surface.PrepareCacheForNextRow();
	}
```

It works again, and `VoxelGrid` no longer needs to worry about the details of the cache arrays. Or does it? It still has to double the cell's row index to provide a cache index. It would make more sense to simply pass along the current cell's row index unmodified. So let's do that instead!

```
	private void FillFirstRowCache () {
		CacheFirstCorner(voxels[0]);
		int i;
		for (i = 0; i < resolution - 1; i++) {
			CacheNextEdgeAndCorner(i, voxels[i], voxels[i + 1]);
		}
		if (xNeighbor != null) {
			dummyX.BecomeXDummyOf(xNeighbor.voxels[0], gridSize);
			CacheNextEdgeAndCorner(i, voxels[i], dummyX);
		}
	}

	private void TriangulateCellRows () {
		…
			for (int x = 0; x < cells; x++, i++) {
				Voxel
					a = voxels[i],
					b = voxels[i + 1],
					c = voxels[i + resolution],
					d = voxels[i + resolution + 1];
				CacheNextEdgeAndCorner(x, c, d);
				CacheNextMiddleEdge(b, d);
				TriangulateCell(x, a, b, c, d);
			}
		…
	}

	private void TriangulateGapCell (int i) {
		…
		int cacheIndex = resolution - 1;
		…
	}

	private void TriangulateGapRow () {
		…
		for (int x = 0; x < cells; x++) {
			Voxel dummySwap = dummyT;
			dummySwap.BecomeYDummyOf(yNeighbor.voxels[x + 1], gridSize);
			dummyT = dummyY;
			dummyY = dummySwap;
			CacheNextEdgeAndCorner(x, dummyT, dummyY);
			CacheNextMiddleEdge(voxels[x + offset + 1], dummyY);
			TriangulateCell(
				x, voxels[x + offset], voxels[x + offset + 1], dummyT, dummyY);
		}

		if (xNeighbor != null) {
			dummyT.BecomeXYDummyOf(xyNeighbor.voxels[0], gridSize);
			CacheNextEdgeAndCorner(cells, dummyY, dummyT);
			CacheNextMiddleEdge(dummyX, dummyT);
			TriangulateCell(
				cells, voxels[voxels.Length - 1], dummyX, dummyY, dummyT);
		}
	}
```

Too keep the cache functional, we'll now have to double the index in `VoxelGridSurface`. But let's change things so we don't have to, by splitting the cache rows into separate arrays for corners and edges. Let's do some renaming too.



The new cache structure.

First rename `edgeCacheMin` to `yEdgeMin` and `edgeCacheMax` to `yEdgeMax`. Then rename `rowCacheMin` to `cornersMin` and `rowCacheMax` to `cornersMax`. Then add two new arrays for the X edges.

```
	private int[] cornersMin, cornersMax;
	private int[] xEdgesMax, xEdgesMin;
	private int yEdgeMin, yEdgeMax;
```

Next, include these arrays in `Initialize` and `PrepareCacheForNextRow`.

```
	public void Initialize (int resolution) {
		…
		cornersMax = new int[resolution + 1];
		cornersMin = new int[resolution + 1];
		xEdgesMax = new int[resolution];
		xEdgesMin = new int[resolution];
	}

	public void PrepareCacheForNextRow () {
		…
		rowSwap = xEdgesMax;
		xEdgesMax = xEdgesMin;
		xEdgesMin = rowSwap;
	}
```

Now we have to rewrite some code fragments. Replace `cornersMin[i + 1]` with `xEdgesMax[i]` and `cornersMax[i + 1]` with `xEdgesMin[i]`. That way every access to the X edges will work with the new arrays.

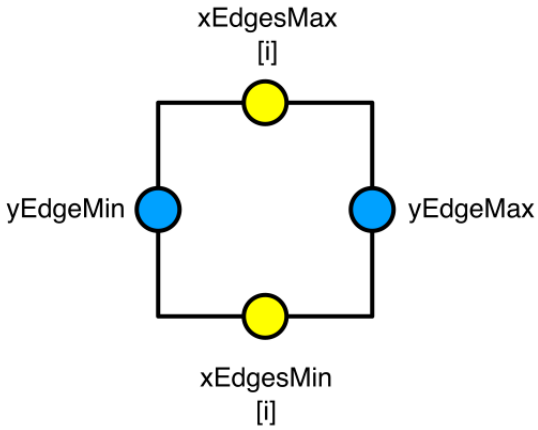
Finally, correct the corner cache indices by replacing `[i + 2]` with `[i + 1]`. It is important to do this last, otherwise we would mix up corners and edges.

Does this require tedious manual work?

While you could do it by hand, it is a really good idea to use the capabilities of your code editor. The options vary between IDEs, but all of them should at least have basic replace functionality. If used properly, it both makes you life easier and leads to less errors.

Adding Depth

With that out of the way, it's time to add some walls! And to keep from overburdening `VoxelGrid` again, let's create a new `VoxelGridWall` class to take care of the required meshing and caching. The basic structure is the same as `VoxelGridSurface`, except that walls always go through cell edges and never go through corners. So we don't need to cache corner vertices.



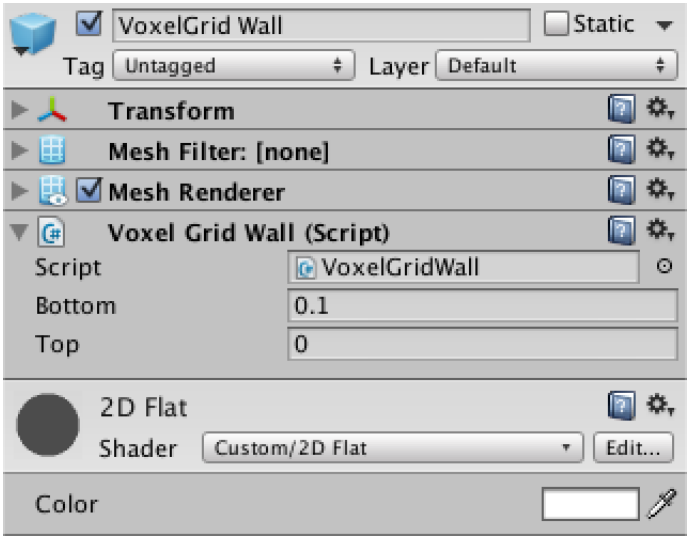
Walls need no corners.

```
using UnityEngine;
using System.Collections.Generic;

public class VoxelGridWall : MonoBehaviour {

	private Mesh mesh;

	private List<Vector3> vertices;
	private List<int> triangles;

	private int[] xEdgesMin, xEdgesMax;
	private int yEdgeMin, yEdgeMax;

	public void Initialize (int resolution) {
		GetComponent<MeshFilter>().mesh = mesh = new Mesh();
		mesh.name = "VoxelGridWall Mesh";
		vertices = new List<Vector3>();
		triangles = new List<int>();
		xEdgesMin = new int[resolution];
		xEdgesMax = new int[resolution];
	}

	public void Clear () {
		vertices.Clear();
		triangles.Clear();
		mesh.Clear();
	}

	public void Apply () {
		mesh.vertices = vertices.ToArray();
		mesh.triangles = triangles.ToArray();
	}
}
```

As the voxel data doesn't include height, we have to tell the wall how high it should be. We do that by adding a bottom and top offset to control the wall's dimensions placement.

```
	public float bottom, top;
```

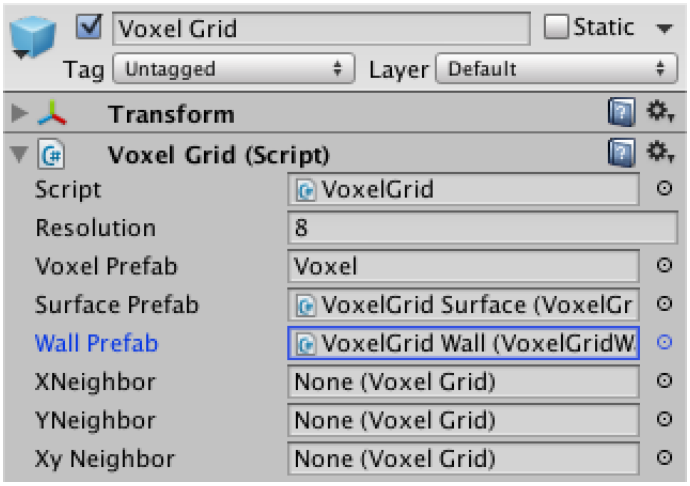
Now create a prefab for it, just like we did for `VoxelGridSurface`. Let's use the voxel surface to define where the top of the wall is. Because the surface has no elevation, set the wall's top to **0** and bottom somewhere below that, say at **−0.1**.



Wall prefab.

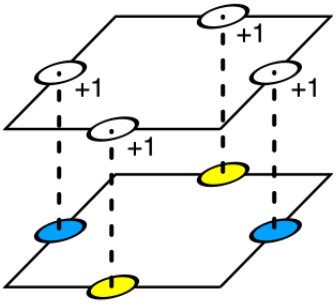
Again add prefab and instance variables to `VoxelGrid` and connect the prefabs.

```
public VoxelGridWall wallPrefab;

private VoxelGridWall wall;
```



Now with both surface and wall prefabs.

Caching is different for `VoxelGridWall`, because it has to store two vertices per edge intersection. One for the bottom, and one for the top of the wall. Because these two vertices will aways be put into the vertex array together, we can suffice with caching only a single index per edge intersection, which would be the bottom one. Note that this means that the caches will only contain odd indices, never even indices.



Only bottom indices need to be cached.

```
public void CacheXEdge (int i, Voxel voxel) {
	xEdgesMax[i] = vertices.Count;
	Vector3 v = voxel.XEdgePoint;
	v.z = bottom;
	vertices.Add(v);
	v.z = top;
	vertices.Add(v);
}

public void CacheYEdge (Voxel voxel) {
	yEdgeMax = vertices.Count;
	Vector3 v = voxel.YEdgePoint;
	v.z = bottom;
	vertices.Add(v);
	v.z = top;
	vertices.Add(v);
}

public void PrepareCacheForNextCell () {
	yEdgeMin = yEdgeMax;
}

public void PrepareCacheForNextRow () {
	int[] swap = xEdgesMin;
	xEdgesMin = xEdgesMax;
	xEdgesMax = swap;
}
```

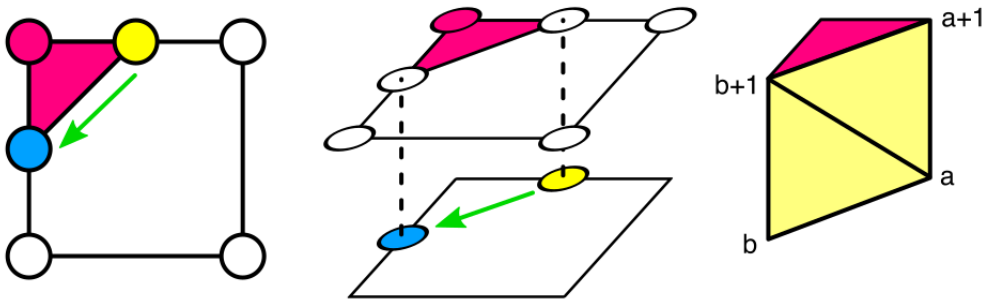
Now we can include the wall in the initialization and caching code of `VoxelGrid`.

```
public void Initialize (int resolution, float size, float maxFeatureAngle) {
	…

	wall = Instantiate(wallPrefab) as VoxelGridWall;
	wall.transform.parent = transform;
	wall.transform.localPosition = Vector3.zero;
	wall.Initialize(resolution);

	Refresh();
}

private void Triangulate () {
	surface.Clear();
	wall.Clear();
	…
	surface.Apply();
	wall.Apply();
}

private void CacheNextEdgeAndCorner (int i, Voxel xMin, Voxel xMax) {
	if (xMin.state != xMax.state) {
		surface.CacheXEdge(i, xMin);
		wall.CacheXEdge(i, xMin);
	}
	if (xMax.state) {
		surface.CacheNextCorner(i, xMax);
	}
}

private void CacheNextMiddleEdge (Voxel yMin, Voxel yMax) {
	surface.PrepareCacheForNextCell();
	wall.PrepareCacheForNextCell();
	if (yMin.state != yMax.state) {
		surface.CacheYEdge(yMin);
		wall.CacheYEdge(yMin);
	}
}

private void SwapRowCaches () {
	surface.PrepareCacheForNextRow();
	wall.PrepareCacheForNextRow();
}
```

Building Walls

We have the vertices, now to actually triangulate the walls. A single straight wall section is simply a quad. Because the cached bottom and top vertices are paired, we need just two vertex indices to construct one.

Besides that, we also need to know which side of the wall is facing outward so we can use the correct triangle winding order. Let's say that placing wall segments in a clockwise order will indicate that the walls should face outward.



Constructing a wall section.

```
private void AddSection (int a, int b) {
	triangles.Add(a);
	triangles.Add(b);
	triangles.Add(b + 1);
	triangles.Add(a);
	triangles.Add(b + 1);
	triangles.Add(a + 1);
}
```

If a cell has a sharp feature, we can simply use two wall sections to correctly triangulate it. Just like with the surface, we can support this via another method that accepts an extra point.

```
private void AddSection (int a, int b, Vector3 extraPoint) {
	int p = vertices.Count;
	extraPoint.z = bottom;
	vertices.Add(extraPoint);
	extraPoint.z = top;
	vertices.Add(extraPoint);
	AddSection(a, p);
	AddSection(p, b);
}
```

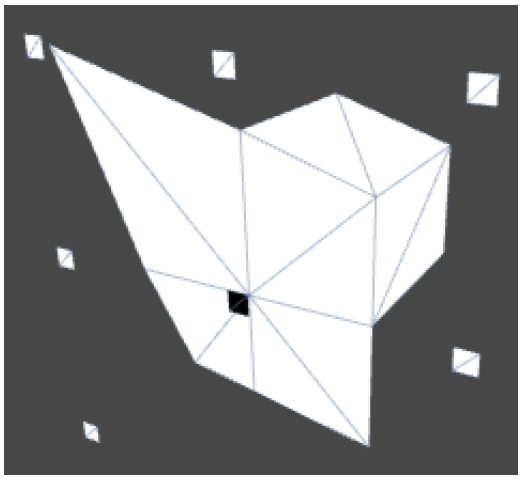
But these methods are for private use. `VoxelGrid` will simply want a wall connection between two edges of a cell. It's up to `VoxelGridWall` to figure out which cache entries that maps to.

Consider cell case 1. It should have a wall connecting the AC edge with the AB edge, either directly or via a sharp feature point. `VoxelGridWall` can facilitate this by adding two public methods.

```
	public void AddACAB (int i) {
		AddSection(yEdgeMin, xEdgesMin[i]);
	}

	public void AddACAB (int i, Vector2 extraVertex) {
		AddSection(yEdgeMin, xEdgesMin[i], extraVertex);
	}
```

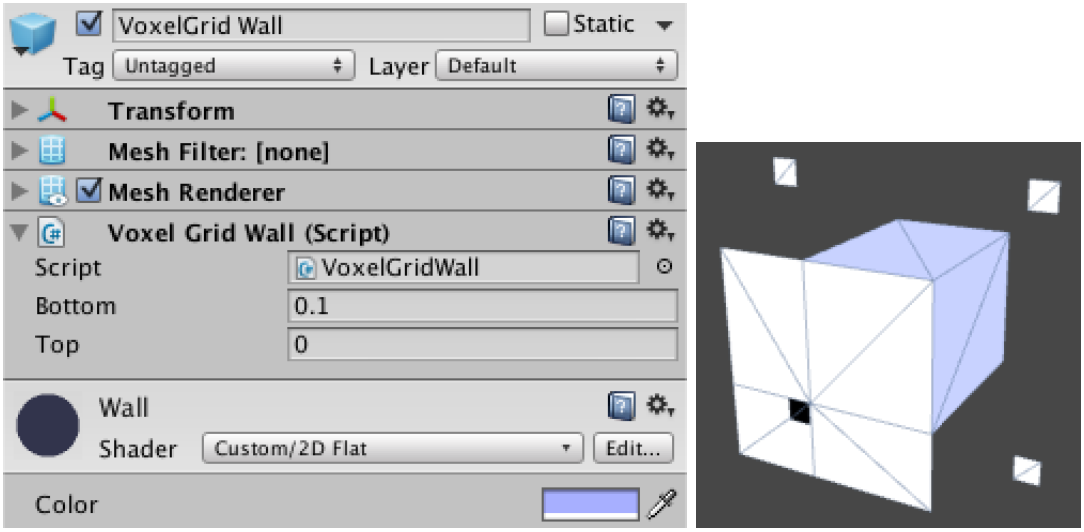
Now `VoxelGrid` can request walls for all case 1 cells.

```
	private void TriangulateCase1 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = GetIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			if (ClampToCellMaxMax(ref point, a, d)) {
				surface.AddQuadA(i, point);
				wall.AddACAB(i, point);
				return;
			}
		}
		surface.AddTriangleA(i);
		wall.AddACAB(i);
	}
```



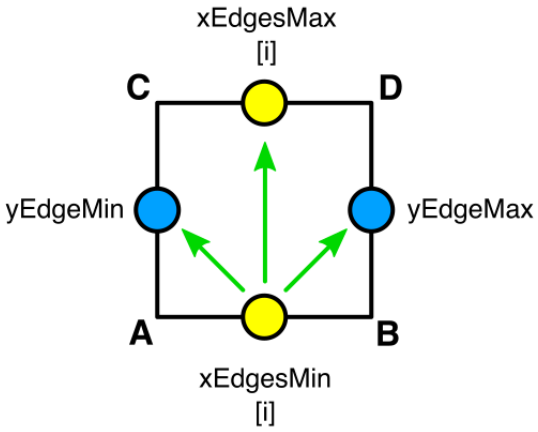
The first glimpse of a wall.

To better distinguish between wall and surface, give the prefab another material. Just duplicating the 2D material and changing its color will do for now.



Color-coded walls.

As you can start a wall from each edge and then go to one of the three other edges, there are twelve different ways to place a wall. This can be either without of with an extra feature point, so we end up with 24 public `Add…` methods for `VoxelGridWall`. It's simply a matter of picking the correct cache entries. Here are those starting at AB.



Walls starting at AB.

```
public void AddABAC (int i) {
    AddSection(xEdgesMin[i], yEdgeMin);
}

public void AddABAC (int i, Vector2 extraVertex) {
    AddSection(xEdgesMin[i], yEdgeMin, extraVertex);
}

public void AddABBD (int i) {
    AddSection(xEdgesMin[i], yEdgeMax);
}

public void AddABBD (int i, Vector2 extraVertex) {
    AddSection(xEdgesMin[i], yEdgeMax, extraVertex);
}

public void AddABCD (int i) {
    AddSection(xEdgesMin[i], xEdgesMax[i]);
}

public void AddABCD (int i, Vector2 extraVertex) {
    AddSection(xEdgesMin[i], xEdgesMax[i], extraVertex);
}
```

And those that start at AC. We already did ACAB.

```
public void AddACAB (int i) {
    AddSection(yEdgeMin, xEdgesMin[i]);
}

public void AddACAB (int i, Vector2 extraVertex) {
    AddSection(yEdgeMin, xEdgesMin[i], extraVertex);
}

public void AddACBD (int i) {
    AddSection(yEdgeMin, yEdgeMax);
}

public void AddACBD (int i, Vector2 extraVertex) {
    AddSection(yEdgeMin, yEdgeMax, extraVertex);
}

public void AddACCD (int i) {
    AddSection(yEdgeMin, xEdgesMax[i]);
}

public void AddACCD (int i, Vector2 extraVertex) {
    AddSection(yEdgeMin, xEdgesMax[i], extraVertex);
}
```

Next up are those starting at BD.

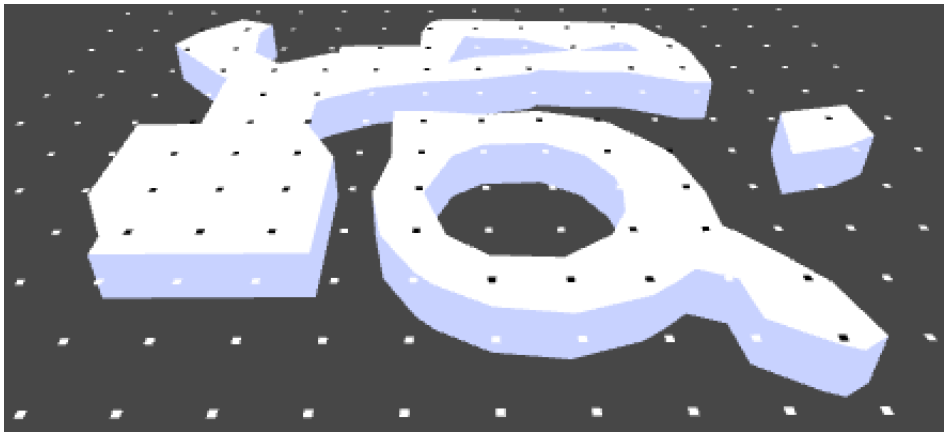
```
	public void AddBDAB (int i) {
		AddSection(yEdgeMax, xEdgesMin[i]);
	}

	public void AddBDAB (int i, Vector2 extraVertex) {
		AddSection(yEdgeMax, xEdgesMin[i], extraVertex);
	}

	public void AddBDAC (int i) {
		AddSection(yEdgeMax, yEdgeMin);
	}

	public void AddBDAC (int i, Vector2 extraVertex) {
		AddSection(yEdgeMax, yEdgeMin, extraVertex);
	}

	public void AddBDCD (int i) {
		AddSection(yEdgeMax, xEdgesMax[i]);
	}

	public void AddBDCD (int i, Vector2 extraVertex) {
		AddSection(yEdgeMax, xEdgesMax[i], extraVertex);
	}
```

And finally those going from CD to the other edges.

```
	public void AddCDAB (int i) {
		AddSection(xEdgesMax[i], xEdgesMin[i]);
	}

	public void AddCDAB (int i, Vector2 extraVertex) {
		AddSection(xEdgesMax[i], xEdgesMin[i], extraVertex);
	}

	public void AddCDAC (int i) {
		AddSection(xEdgesMax[i], yEdgeMin);
	}

	public void AddCDAC (int i, Vector2 extraVertex) {
		AddSection(xEdgesMax[i], yEdgeMin, extraVertex);
	}

	public void AddCDBD (int i) {
		AddSection(xEdgesMax[i], yEdgeMax);
	}

	public void AddCDBD (int i, Vector2 extraVertex) {
		AddSection(xEdgesMax[i], yEdgeMax, extraVertex);
	}
```

Now to use them in the triangulation methods of `VoxelGrid`. It's simply a matter of adding `wall.Add…` each time `surface.Add…` is invoked, like we already did for case 1. You just have to add the correct walls. I won't bother showing all the code, instead here's a table which shows which surface and wall methods belong together, and which cell cases use them.

Surface	Wall	Cases
A	ACAB	1, 9
B	ABBD	2, 6
C	CDAC	4, 6
D	BDCD	8, 9
ABC	CDBD	7
ABD	ACCD	11
ACD	BDAB	13
BCD	ABAC	14
AB	ACBD	3
AC	CDAB	5
BD	ABCD	10
CD	BDAC	12
BCToA	ABAC	6
BCToD	CDBD	6
ADToB	BDAB	9
ADToC	ACCD	9



Everything walled up.

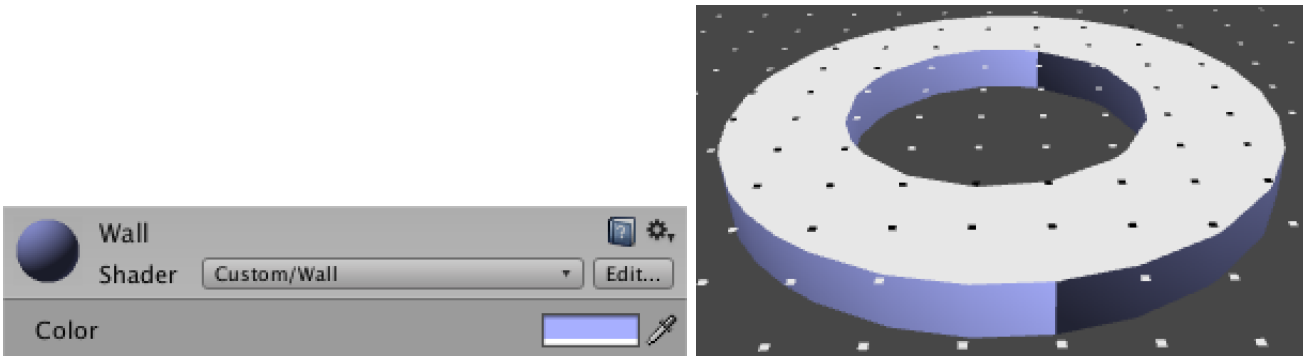
Lighting the Walls

While we have complete walls now, they're all the same solid color without any lighting. Currently our directional light is shining straight at the surface. To get any variety in the lighting, give it some rotation, for example **(0, 315, 0)**.

The result is still flat. This is because we're using our *Flat 2D* shader, which has fixed normals. We could use the standard diffuse shader instead, but that one wants texture coordinates and we don't have those. Fortunately, we can create a new wall shader by simply duplicating the *2D Flat* shader and removing the vertex program that inserts normals.

```
Shader "Custom/Wall" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert

        fixed4 _Color;

        struct Input {
            float dummy;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            o.Albedo = _Color.rgb;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Then adjust the *Wall* material so it uses this new shader instead of the flat one.



Using normals, without providing any.

We get some shading now, but it doesn't make any sense because our wall mesh doesn't have normals yet. So `VoxelGridWall` also has to add and cache normals.

```
	private List<Vector3> vertices, normals;

	public void Initialize (int resolution) {
		GetComponent<MeshFilter>().mesh = mesh = new Mesh();
		mesh.name = "VoxelGridWall Mesh";
		vertices = new List<Vector3>();
		normals = new List<Vector3>();
		triangles = new List<int>();
		xEdgesMax = new int[resolution];
		xEdgesMin = new int[resolution];
	}

	public void Clear () {
		vertices.Clear();
		normals.Clear();
		triangles.Clear();
		mesh.Clear();
	}

	public void Apply () {
		mesh.vertices = vertices.ToArray();
		mesh.normals = normals.ToArray();
		mesh.triangles = triangles.ToArray();
	}
```

When an edge intersection is cached, we can simply grab the intersection's normal and directly use it as our wall's normal. This will result in smooth shading along cell boundaries. We have to add them twice, for both the bottom and the top vertex.

```
	public void CacheXEdge (int i, Voxel voxel) {
		…
		Vector3 n = voxel.xNormal;
		normals.Add(n);
		normals.Add(n);
	}

	public void CacheYEdge (Voxel voxel) {
		…
		Vector3 n = voxel.yNormal;
		normals.Add(n);
		normals.Add(n);
	}
```

We need to add normals for the sharp feature point as well. The most straightforward approach is to simply reuse the normals of the starting point.

```
	private void AddSection (int a, int b, Vector3 extraPoint) {
		int p = vertices.Count;
		extraPoint.z = bottom;
		vertices.Add(extraPoint);
		extraPoint.z = top;
		vertices.Add(extraPoint);
		Vector3 n = normals[a];
		normals.Add(n);
		normals.Add(n);
		AddSection(a, p);
		AddSection(p, b);
	}
```

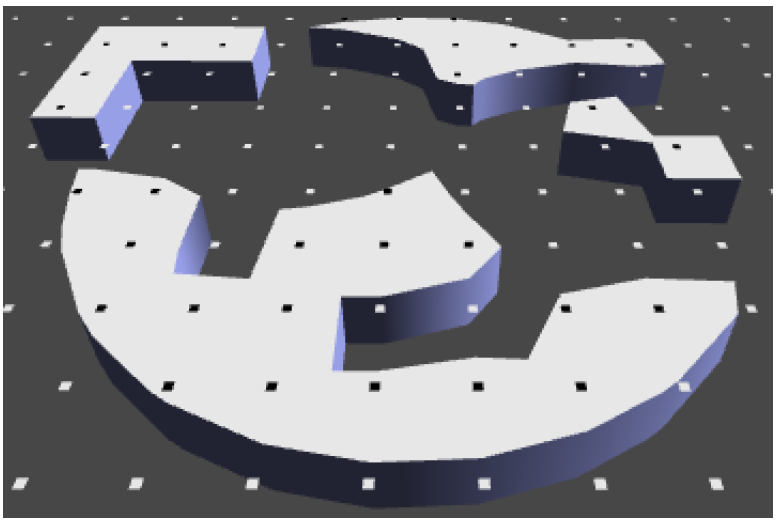
However, that only makes sense for the first section. As it is a sharp feature, we should not have any smooth shading here. To make it a proper crease, we have to add the feature point again, now with the normals of the end point.

```
	private void AddSection (int a, int b, Vector3 extraPoint) {
		…
		AddSection(a, p);

		p = vertices.Count;
		extraPoint.z = bottom;
		vertices.Add(extraPoint);
		extraPoint.z = top;
		vertices.Add(extraPoint);
		n = normals[b];
		normals.Add(n);
		normals.Add(n);
		AddSection(p, b);
	}
```

Do we really need to duplicate the point?

Yes, as each vertex can have only one normal associated with it. This vertex duplication trick is used in every mesh that has creases or smoothing groups.

While technically you could store more data per vertex, which you could interpret as an additional normal, you'd have no way of knowing which normal to use when. This is because triangles do not have an identity.



Shaded walls.

And we have nicely shaded walls! Only sharp features have hard transitions, the rest is smooth. However, you could end up with wobbly lighting when painting loosely with the small brushes. This is because small deviations in edge intersection directions show up in the shading, even though they don't produce sharp features. It can also look strange when a potential sharp feature was discarded, which could produce sharp bends at cell boundaries in combination with smooth shading. Overall though, it's reasonably accurate.

It is also possible to use flat shading, which produces a faceted look. That would require every wall section to use the same normal for its four vertices. Like with sharp feature points, you'd have to duplicate every point, which means that you couldn't reuse any vertices.

made by Jasper Flick