Marching Squares 3, staying sharp

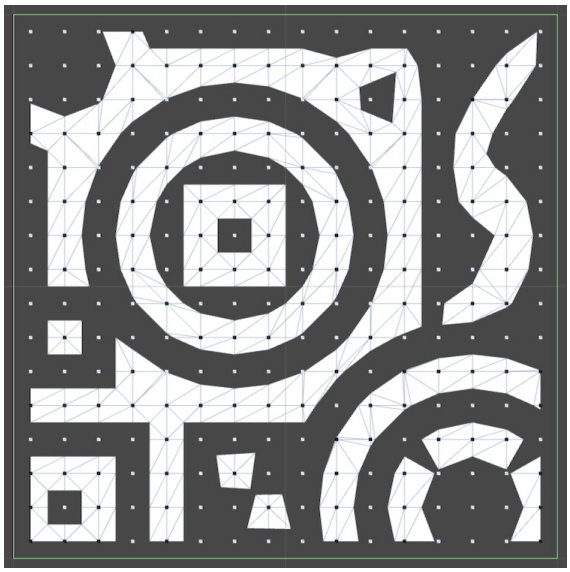
Marching squares with exact edge intersections allows us to triangulate a wide variety of shapes. However, it does not preserve any sharp angles that we draw. The corners of a square will be cut off. The edge intersections remain, but the sharp feature inside the cell is lost. This time we'll make sure that those features are maintained.

You'll learn to

- Calculate normals of edge intersections;
- Store Hermite data;
- Discover sharp features;
- Decide which features to add;
- Resolve ambiguous cases.

This tutorial is the third in a series about Marching Squares.

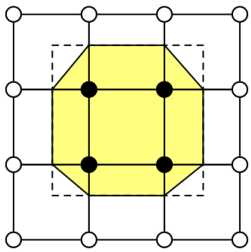
This tutorial has been made with Unity 4.5.2. It might not work for older versions.



Painting with sharp features.

Hermite Data

At this point we know where the edges or our square cells are crossed. This allows us to create a better approximation than always placing the edge vertices at the midpoint. Unfortunately, this is not enough to preserve any sharp features inside the cells, like the corners of a square. To reconstruct – or at least reasonably approximate – those features we also need to know at what angles a cell's edges are crossed.



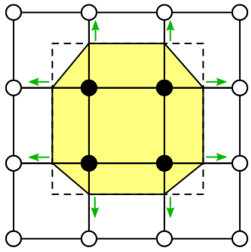
Features inside cells are lost.

We could represent the direction of an edge crossing with a normal vector, which points away from the filled space. So we store both the intersection point and normal, which means that we're working with Hermite data.

That means we have to store two additional 2D vectors per `Voxel`. That's quite a bit of extra data, so we better make use of it!

What's Hermite data?

```
public Vector2 xNormal, yNormal;
```



Surface normals of the actual square.

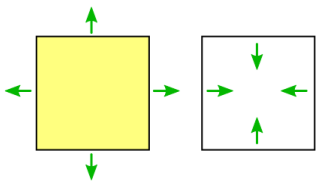
We also have to copy the normals when creating dummies, though it's only needed for edges that we'll end up using. So only the Y edge's normal for X dummies and only the X edge normal for Y dummies.

```
public void BecomeXDummyOf (Voxel voxel, float offset) {
	state = voxel.state;
	position = voxel.position;
	position.x += offset;
	xEdge = voxel.xEdge + offset;
	yEdge = voxel.yEdge;
	yNormal = voxel.yNormal;
}

public void BecomeYDummyOf (Voxel voxel, float offset) {
	state = voxel.state;
	position = voxel.position;
	position.y += offset;
	xEdge = voxel.xEdge;
	yEdge = voxel.yEdge + offset;
	xNormal = voxel.xNormal;
}
```

The next step is to actually store the normals, which is the responsibility of the stencils.

Square Stencil Normals

Normals for the square `VoxelStencil` are straightforward. When we find that our stencil's edge is crossed on the right, we add a normal that points to the right. The same goes for the other three directions. However, that assumes that we're filling the voxels inside the stencil's area. If we're actually emptying the voxels, then the normals should point in the opposite direction. So the final direction depends on the fill type.



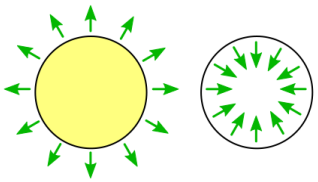
Square normals.

```
protected virtual void FindHorizontalCrossing (Voxel xMin, Voxel xMax) {
	if (xMin.position.y < YStart || xMin.position.y > YEnd) {
		return;
	}
	if (xMin.state == fillType) {
		if (xMin.position.x <= XEnd && xMax.position.x >= XEnd) {
			if (xMin.xEdge == float.MinValue || xMin.xEdge < XEnd) {
				xMin.xEdge = XEnd;
				xMin.xNormal = new Vector2(fillType ? 1f : -1f, 0f);
```

```
				}
			}
		}
		else if (xMax.state == fillType) {
			if (xMin.position.x <= XStart && xMax.position.x >= XStart) {
				if (xMin.xEdge == float.MinValue || xMin.xEdge > XStart) {
					xMin.xEdge = XStart;
					xMin.xNormal = new Vector2(fillType ? -1f : 1f, 0f);
				}
			}
		}
	}

	protected virtual void FindVerticalCrossing (Voxel yMin, Voxel yMax) {
		if (yMin.position.x < XStart || yMin.position.x > XEnd) {
			return;
		}
		if (yMin.state == fillType) {
			if (yMin.position.y <= YEnd && yMax.position.y >= YEnd) {
				if (yMin.yEdge == float.MinValue || yMin.yEdge < YEnd) {
					yMin.yEdge = YEnd;
					yMin.yNormal = new Vector2(0f, fillType ? 1f : -1f);
				}
			}
		}
		else if (yMax.state == fillType) {
			if (yMin.position.y <= YStart && yMax.position.y >= YStart) {
				if (yMin.yEdge == float.MinValue || yMin.yEdge > YStart) {
					yMin.yEdge = YStart;
					yMin.yNormal = new Vector2(0f, fillType ? -1f : 1f);
				}
			}
		}
	}
```

Circle Stencil Normals

The same goes for `VoxelStencilCircle`, but normals are a bit more complex for circles. When filling, normals are equal to the vector pointing from the stencil center to the intersection point, normalized. When emptying, it's the other way around. Let's add a helper method for that.



Circle normals.

```
	private Vector3 ComputeNormal (float x, float y) {
		if (fillType) {
			return new Vector2(x - centerX, y - centerY).normalized;
		}
		else {
			return new Vector2(centerX - x, centerY - y).normalized;
		}
	}
```

Then Invoke it with four all four edge cases.

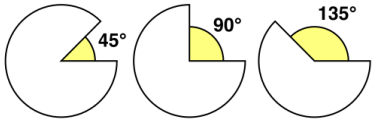
```
	protected override void FindHorizontalCrossing (Voxel xMin, Voxel xMax) {
		float y2 = xMin.position.y - centerY;
		y2 *= y2;
		if (xMin.state == fillType) {
			float x = xMin.position.x - centerX;
			if (x * x + y2 <= sqrRadius) {
				x = centerX + Mathf.Sqrt(sqrRadius - y2);
				if (xMin.xEdge == float.MinValue || xMin.xEdge < x) {
					xMin.xEdge = x;
					xMin.xNormal = ComputeNormal(x, xMin.position.y);
				}
			}
		}
		else if (xMax.state == fillType) {
			float x = xMax.position.x - centerX;
			if (x * x + y2 <= sqrRadius) {
				x = centerX - Mathf.Sqrt(sqrRadius - y2);
				if (xMin.xEdge == float.MinValue || xMin.xEdge > x) {
					xMin.xEdge = x;
					xMin.xNormal = ComputeNormal(x, xMin.position.y);
				}
			}
		}
```

```
	}

	protected override void FindVerticalCrossing (Voxel yMin, Voxel yMax) {
		float x2 = yMin.position.x - centerX;
		x2 *= x2;
		if (yMin.state == fillType) {
			float y = yMin.position.y - centerY;
			if (y * y + x2 <= sqrRadius) {
				y = centerY + Mathf.Sqrt(sqrRadius - x2);
				if (yMin.yEdge == float.MinValue || yMin.yEdge < y) {
					yMin.yEdge = y;
					yMin.yNormal = ComputeNormal(yMin.position.x, y);
				}
			}
		}
		else if (yMax.state == fillType) {
			float y = yMax.position.y - centerY;
			if (y * y + x2 <= sqrRadius) {
				y = centerY - Mathf.Sqrt(sqrRadius - x2);
				if (yMin.yEdge == float.MinValue || yMin.yEdge > y) {
					yMin.yEdge = y;
					yMin.yNormal = ComputeNormal(yMin.position.x, y);
				}
			}
		}
	}
```

Although we're not seeing any of it, we're now storing Hermite data. The next step is to put it to good use.

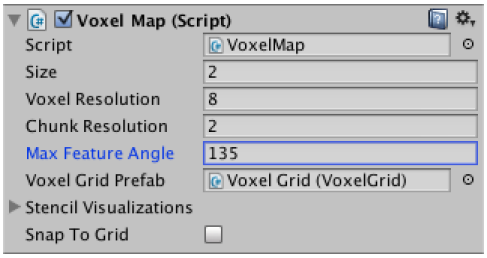
Showing Sharp Features

The first question we should ask ourselves is what counts as a sharp feature. A 90° angle and anything below that seems obvious, but what is the upper limit? 100°? 120°? Different maximum angles will produce different visual results. There isn't a single correct answer. So let's add a configuration option to `VoxelMap` and let's use a default of 135 degrees.



What is still a sharp feature?

```
	public float maxFeatureAngle = 135f;
```



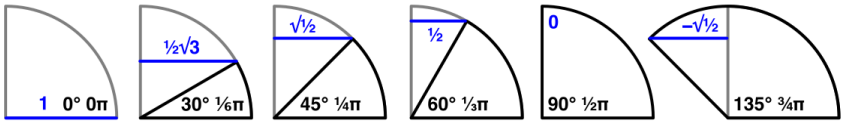
Configuring maximum sharp feature angle.

Of course the voxel map doesn't deal with cells directly, so it passes this data to its voxel grids.

```
	private void CreateChunk (int i, int x, int y) {
		VoxelGrid chunk = Instantiate(voxelGridPrefab) as VoxelGrid;
		chunk.Initialize(voxelResolution, chunkSize, maxFeatureAngle);
		…
	}
```

Now `VoxelGrid` needs to remember it as well. But instead of storing the angle in degrees, we store its cosine.

Why the cosine?



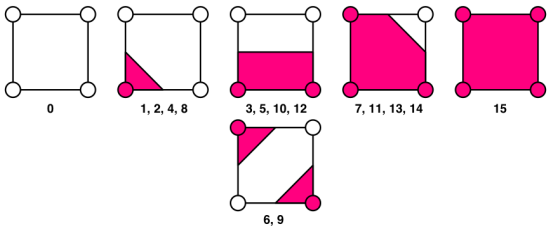
Degrees, radians, and cosines.

```
	private float sharpFeatureLimit;

	public void Initialize (int resolution, float size, float maxFeatureAngle) {
		sharpFeatureLimit = Mathf.Cos(maxFeatureAngle * Mathf.Deg2Rad);
		…
	}
```

Preparing for Sharp Features

The detection of a sharp feature will have to happen when triangulating individual cells. Right now `TriangulateCell` does all the work directly in its large switch statement. However, it is about to become quite a bit more complicated. So let's introduce separate methods for each case, using the same method signature and just copy the code from the case blocks into their new methods.

Why add a method for case 0?

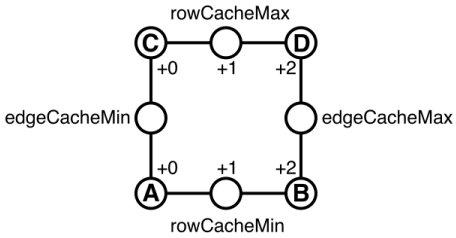


The sixteen cell configurations, grouped by type.

```
	private void TriangulateCell (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		…
		switch (cellType) {
		case 0: TriangulateCase0(i, a, b, c, d); break;
		case 1: TriangulateCase1(i, a, b, c, d); break;
		case 2: TriangulateCase2(i, a, b, c, d); break;
		case 3: TriangulateCase3(i, a, b, c, d); break;
		case 4: TriangulateCase4(i, a, b, c, d); break;
		case 5: TriangulateCase5(i, a, b, c, d); break;
		case 6: TriangulateCase6(i, a, b, c, d); break;
		case 7: TriangulateCase7(i, a, b, c, d); break;
		case 8: TriangulateCase8(i, a, b, c, d); break;
		case 9: TriangulateCase9(i, a, b, c, d); break;
		case 10: TriangulateCase10(i, a, b, c, d); break;
		case 11: TriangulateCase11(i, a, b, c, d); break;
		case 12: TriangulateCase12(i, a, b, c, d); break;
		case 13: TriangulateCase13(i, a, b, c, d); break;
		case 14: TriangulateCase14(i, a, b, c, d); break;
		case 15: TriangulateCase15(i, a, b, c, d); break;
		}
	}

	private void TriangulateCase0 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
	}

	private void TriangulateCase15 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuad(rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 2], rowCacheMin[i + 2]);
	}

	private void TriangulateCase1 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangle(rowCacheMin[i], edgeCacheMin, rowCacheMin[i + 1]);
	}

	…
```

Actually, let's go a step further and add additional methods that add triangles, quads, and pentagons given a cache index. Then the new case methods can use those and won't have to deal with the complexity of the vertex cache directly.

Isn't this bad for performance?



Anatomy of a cell and its vertex cache.

```
	private void TriangulateCase15 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuadABCD(i);
	}

	private void TriangulateCase1 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleA(i);
	}

	private void TriangulateCase2 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleB(i);
	}

	private void TriangulateCase4 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleC(i);
	}

	private void TriangulateCase8 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleD(i);
	}

	private void TriangulateCase7 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddPentagonABC(i);
	}

	private void TriangulateCase11 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddPentagonABD(i);
	}

	private void TriangulateCase13 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddPentagonACD(i);
	}

	private void TriangulateCase14 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddPentagonBCD(i);
	}

	private void TriangulateCase3 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuadAB(i);
	}

	private void TriangulateCase5 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuadAC(i);
	}

	private void TriangulateCase10 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuadBD(i);
	}

	private void TriangulateCase12 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddQuadCD(i);
	}

	private void TriangulateCase6 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleB(i);
		AddTriangleC(i);
	}

	private void TriangulateCase9 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		AddTriangleA(i);
		AddTriangleD(i);
	}
```

And so the code that started inside the `switch` statement ends up in their own methods.

```
	private void AddQuadABCD (int i) {
		AddQuad(rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 2], rowCacheMin[i + 2]);
	}

	private void AddTriangleA (int i) {
		AddTriangle(rowCacheMin[i], edgeCacheMin, rowCacheMin[i + 1]);
	}

	private void AddTriangleB (int i) {
		AddTriangle(rowCacheMin[i + 2], rowCacheMin[i + 1], edgeCacheMax);
	}
```

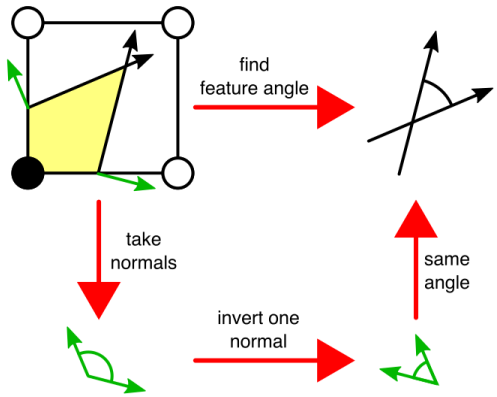
```
	private void AddTriangleC (int i) {
		AddTriangle(rowCacheMax[i], rowCacheMax[i + 1], edgeCacheMin);
	}

	private void AddTriangleD (int i) {
		AddTriangle(rowCacheMax[i + 2], edgeCacheMax, rowCacheMax[i + 1]);
	}

	private void AddPentagonABC (int i) {
		AddPentagon(rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 1], edgeCacheMax, rowCacheMin[i + 2]);
	}

	private void AddPentagonABD (int i) {
		AddPentagon(rowCacheMin[i + 2], rowCacheMin[i], edgeCacheMin, rowCacheMax[i + 1], rowCacheMax[i + 2]);
	}

	private void AddPentagonACD (int i) {
		AddPentagon(rowCacheMax[i], rowCacheMax[i + 2], edgeCacheMax, rowCacheMin[i + 1], rowCacheMin[i]);
	}

	private void AddPentagonBCD (int i) {
		AddPentagon(rowCacheMax[i + 2], rowCacheMin[i + 2], rowCacheMin[i + 1], edgeCacheMin, rowCacheMax[i]);
	}

	private void AddQuadAB (int i) {
		AddQuad(rowCacheMin[i], edgeCacheMin, edgeCacheMax, rowCacheMin[i + 2]);
	}

	private void AddQuadAC (int i) {
		AddQuad(rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 1], rowCacheMin[i + 1]);
	}

	private void AddQuadBD (int i) {
		AddQuad(rowCacheMin[i + 1], rowCacheMax[i + 1], rowCacheMax[i + 2], rowCacheMin[i + 2]);
	}

	private void AddQuadCD (int i) {
		AddQuad(edgeCacheMin, rowCacheMax[i], rowCacheMax[i + 2], edgeCacheMax);
	}
```

Detecting Sharp Features

Cases 0 and 15 have no edge crossings so we don't need to change them. Case 1 is the first that might have a sharp feature. How to find out?

We have two lines crossing adjacent edges of the cell. If those lines cross at a sharp enough angle, we treat it as a sharp feature.

If you have two vectors of unit length, then their dot product is equal to the cosine of the angle between then. Our two lines don't have lengths, but we could use their normals. If we invert one of them, then we end up with the same angle. So we can use the normals to determine whether we have a sharp feature.



Finding the angle of a feature.

Because the normals have unit length, their dot product is equal to the cosine of the absolute angle between them. If this ends up larger than the limit that we set, then we have a sharp feature. Let's create a method for that.

Why does dot lead to cosine?

```
	private bool IsSharpFeature (Vector2 n1, Vector2 n2) {
		float dot = Vector2.Dot(n1, -n2);
		return dot >= sharpFeatureLimit;
	}
```

However, we have to make sure that we exclude parallel lines. As the angle between such lines is 0° – which cosine is 1 – we would count them as sharp. But we will run into trouble later when trying to find their intersection point. So disqualify line crossings that are close to 0°.

```
		return dot >= sharpFeatureLimit && dot < 0.9999f;
```

Now we can test whether case 1 has a sharp feature. Initially don't do anything yet if we find one, and add the usual triangle if there isn't a feature.

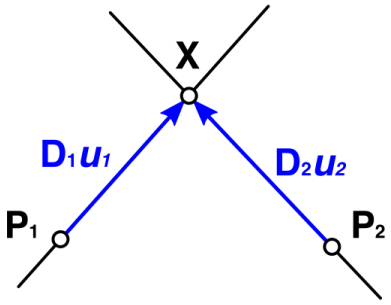
```
	private void TriangulateCase1 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = a.yNormal;
		if (IsSharpFeature(n1, n2)) {

		}
		else {
			AddTriangleA(i);
		}
	}
```

You can sculpt a sharp feature to try this out. Simply use the square stencil, or use the circle stencil to control the angle at which you cross edges. Fill a voxel and then cut away until you get the shape you want. It might take a while to get the hang of this, as you always have to include at least one voxel in your stencil.

An invisible sharp feature.

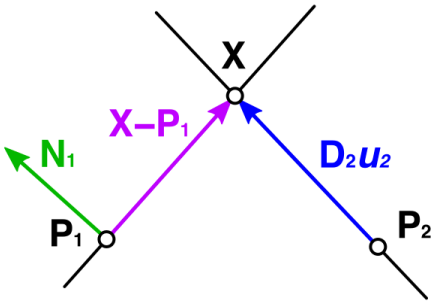
The next step is to find the intersection point of two lines. How can we do that?

A line can be represented by a point and a direction. Any point on the line can be represented with the formula $\mathbf{P} + \mathbf{D}\boldsymbol{u}$, where $\boldsymbol{u}$ can be any number for an infinite line, both positive and negative. We have two such lines and they cross at some point $\mathbf{X}$.



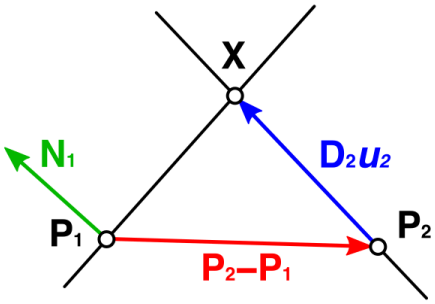
Two lines crossing.

Looking at the first line, we know that the vectors $\mathbf{D_1}$ and $\mathbf{X - P_1}$ are parallel. Of course both are perpendicular to the line's normal vector $\mathbf{N_1}$. This means that the dot product between them and the normal is zero. So we have $\mathbf{N_1 \cdot (X - P_1) = 0}$.



Going from P_1 to X, perpendicular to its normal.

An alternative way to go from $\mathbf{P_1}$ to $\mathbf{X}$ is to take a detour and go to $\mathbf{P_2}$ first and then follow $\mathbf{D_2}\boldsymbol{u_2}$. So we can replace $\mathbf{X - P_1}$ with $\mathbf{(P_2 - P_1) + D_2}\boldsymbol{u_2}$, which again ends up being parallel to $\mathbf{D_1}$ and thus is perpendicular to $\mathbf{N_1}$.



Taking a detour.

So now we have $\mathbf{N_1 \cdot ((P_2 - P_1) + D_2}\boldsymbol{u_2}\mathbf{) = 0}$, which we can rewrite to extract the variable that we need, $\boldsymbol{u_2} = \mathbf{-N_1 \cdot (P_2 - P_1) / (N_1 \cdot D_2)}$.

This tells us how far along $\mathbf{D_2}$ we should walk, starting at $\mathbf{P_2}$, so that the line between us a $\mathbf{P_1}$ becomes perpendicular to $\mathbf{N_1}$. So we finally find $\mathbf{X}$ by calculating $\mathbf{P_2 + D_2}\boldsymbol{u_2}$.

You can get $\mathbf{D_2}$ by rotating $\mathbf{N_2}$ by 90° using the perp operator, which is the vector $\mathbf{(-N_2.y, N_2.x)}$.

How to rewrite the dot product?

Why a static method?

```
private static Vector2 ComputeIntersection (Vector2 p1, Vector2 n1, Vector2 p2, Vector2 n2) {
	Vector2 d2 = new Vector2(n2.y, -n2.x);
	float u2 = -Vector2.Dot(n1, p2 - p1) / Vector2.Dot(n1, d2);
	return p2 + d2 * u2;
}
```

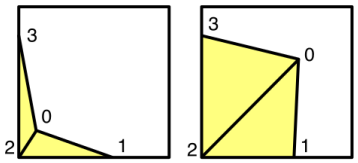
Now we just have to invoke this method to find our intersection point. To get the edge crossing points that we need for this, add two convenient properties to `Voxel`.

```
public Vector2 XEdgePoint {
	get {
		return new Vector2(xEdge, position.y);
	}
}

public Vector2 YEdgePoint {
	get {
		return new Vector2(position.x, yEdge);
	}
}
```

Of course once we have the new point, we have to triangulate it. Instead of a triangle, we'll need to add a quad. And because the sharp feature could end up pointing inwards – forming a valley instead of a peak – we should add it as the first vertex, effectively creating a triangle fan centered

on it. That way the triangulation should always be valid. Let's add a new method to `VoxelGrid` to take care of all this.

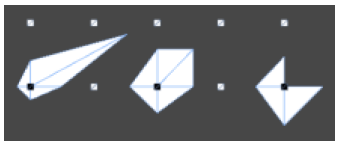
```
	private void AddQuadA (int i, Vector2 extraVertex) {
		AddQuad(vertices.Count, rowCacheMin[i + 1], rowCacheMin[i], edgeCacheMin);
		vertices.Add(extraVertex);
	}
```



Triangulating sharp features, valleys and peaks.

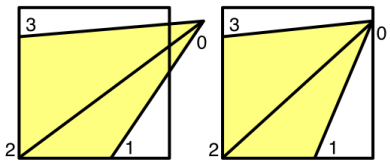
Now we can finally go back to `VoxelGrid.TriangulateCase1` to find the point and add the quad.

```
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			AddQuadA(i, point);
		}
```



Unconstrained sharp features.

The result looks good for the square stencil, but while sculpting with circle stencil the feature point can end up outside the cell. We want to keep features local to the cell, which we can do by clamping to cell bounds. Create another handy method for that.

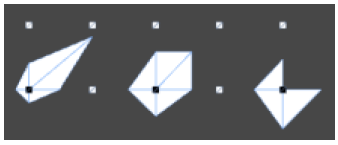


Clamping a peak.

```
	private static Vector2 ClampToCell (Vector2 point, Voxel min, Voxel max) {
		if (point.x < min.position.x) {
			point.x = min.position.x;
		}
		else if (point.x > max.position.x) {
			point.x = max.position.x;
		}
		if (point.y < min.position.y) {
			point.y = min.position.y;
		}
		else if (point.y > max.position.y) {
			point.y = max.position.y;
		}
		return point;
	}
```

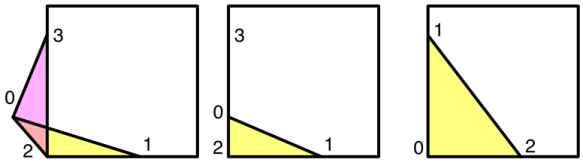
And use it in `TriangulateCase1`.

```
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			point = ClampToCell(point, a, d);
			AddQuadA(i, point);
		}
```



Clamped sharp features.

Clamping works when the sharp feature is a peak, but not when it's a valley, as that can create degenerate triangles, which makes the grid structure painfully obvious. Better not have a sharp feature at all in that case.



Better discard valleys that are out of bounds.

To support this we could adjust our clamp method to only clamp on the maximum sides an indicate failure if a minimum side is passed. We now want to return two results, which is not possible. We solve this by turning the point into a reference parameter. That way we don't get a copy but a reference to the actual variable from the invoker's context. That means that we directly alter that variable, which allows us to return a boolean indicating success or failure.

Shouldn't we avoid using `ref`?

```
private static bool ClampToCellMaxMax (ref Vector2 point, Voxel min, Voxel max) {
	if (point.x < min.position.x || point.y < min.position.y) {
		return false;
	}
	if (point.x > max.position.x) {
		point.x = max.position.x;
	}
	if (point.y > max.position.y) {
		point.y = max.position.y;
	}
	return true;
}
```

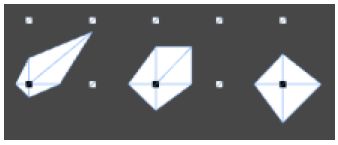
Now we can adjust `TriangulateCase1` so it only adds the feature if the clamp succeeds, otherwise add the normal triangle.

```
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 = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
		if (ClampToCellMaxMax(ref point, a, d)) {
			AddQuadA(i, point);
		}
		else {
			AddTriangleA(i);
		}
	}
	else {
		AddTriangleA(i);
	}
}
```

Alternatively, by returning after adding the sharp feature we only need to write `AddTriangleA` once and can eliminate the `else` blocks.

```
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 = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
		if (ClampToCellMaxMax(ref point, a, d)) {
			AddQuadA(i, point);
			return;
		}
```

```
		}
		AddTriangleA(i);
	}
```



Valid triangulations, though not always sharp.

Checking the Other Three Corners

Case 2 goes similar, but uses different edge points and other clamp criteria. And of course it needs to add different polygons. I marked the differences with case 1.

```
	private void TriangulateCase2 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, b.YEdgePoint, n2);
			if (ClampToCellMinMax(ref point, a, d)) {
				AddQuadB(i, point);
				return;
			}
		}
		AddTriangleB(i);
	}
```

Also add the required clamp and quad methods, and let's immediately do so for cases 4 and 8 as well.

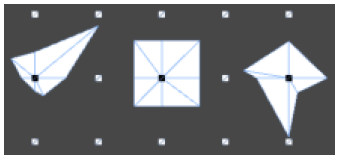
```
	private static bool ClampToCellMinMin (ref Vector2 point, Voxel min, Voxel max) {
		if (point.x > max.position.x || point.y > max.position.y) {
			return false;
		}
		if (point.x < min.position.x) {
			point.x = min.position.x;
		}
		if (point.y < min.position.y) {
			point.y = min.position.y;
		}
		return true;
	}

	private static bool ClampToCellMinMax (ref Vector2 point, Voxel min, Voxel max) {
		if (point.x > max.position.x || point.y < min.position.y) {
			return false;
		}
		if (point.x < min.position.x) {
			point.x = min.position.x;
		}
		if (point.y > max.position.y) {
			point.y = max.position.y;
		}
		return true;
	}

	private static bool ClampToCellMaxMin (ref Vector2 point, Voxel min, Voxel max) {
		if (point.x < min.position.x || point.y > max.position.y) {
			return false;
		}
		if (point.x > max.position.x) {
			point.x = max.position.x;
		}
		if (point.y < min.position.y) {
			point.y = min.position.y;
		}
		return true;
	}

	private void AddQuadB (int i, Vector2 extraVertex) {
		AddQuad(vertices.Count, edgeCacheMax, rowCacheMin[i + 2], rowCacheMin[i + 1]);
		vertices.Add(extraVertex);
	}

	private void AddQuadC (int i, Vector2 extraVertex) {
		AddQuad(vertices.Count, edgeCacheMin, rowCacheMax[i], rowCacheMax[i + 1]);
		vertices.Add(extraVertex);
	}

	private void AddQuadD (int i, Vector2 extraVertex) {
		AddQuad(vertices.Count, rowCacheMax[i + 1], rowCacheMax[i + 2], edgeCacheMax);
```

```
		vertices.Add(extraVertex);
	}
```

The triangulation methods for case 4 and case 8 require similar simple adjustments.

Can we generalize this code?

```
	private void TriangulateCase4 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = c.xNormal;
		Vector2 n2 = a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, a.YEdgePoint, n2);
			if (ClampToCellMaxMin(ref point, a, d)) {
				AddQuadC(i, point);
				return;
			}
		}
		AddTriangleC(i);
	}

	private void TriangulateCase8 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = c.xNormal;
		Vector2 n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, b.YEdgePoint, n2);
			if (ClampToCellMinMin(ref point, a, d)) {
				AddQuadD(i, point);
				return;
			}
		}
		AddTriangleD(i);
	}
```



Looking sharp.

Sharpening the Missing Corners

Next up are cases 7, 11, 13, and 14. These are the opposites of the first four cases, having three filled corners. Without a sharp feature they require a pentagon, so we need to add a hexagon for the sharp features.

```
	private void AddHexagon (int a, int b, int c, int d, int e, int f) {
		triangles.Add(a);
		triangles.Add(b);
		triangles.Add(c);
		triangles.Add(a);
		triangles.Add(c);
		triangles.Add(d);
		triangles.Add(a);
		triangles.Add(d);
		triangles.Add(e);
		triangles.Add(a);
		triangles.Add(e);
		triangles.Add(f);
	}
```

For these cases we don't want to clamp at all, because both peaks and valleys could produce degenerate triangles and ugly results. So instead of clamping, we simply check whether the point lies inside the cell.

```
	private static bool IsInsideCell (Vector2 point, Voxel min, Voxel max) {
		return
			point.x > min.position.x && point.y > min.position.y &&
			point.x < max.position.x && point.y < max.position.y;
	}
```

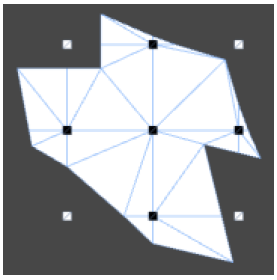
Here's case 7, a modified case 8.

```
	private void TriangulateCase7 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = c.xNormal;
		Vector2 n2 = b.yNormal;
```

```
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, b.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddHexagonABC(i, point);
				return;
			}
		}
		AddPentagonABC(i);
	}

	private void AddHexagonABC (int i, Vector2 extraVertex) {
		AddHexagon(
			vertices.Count, edgeCacheMax, rowCacheMin[i + 2],
			rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 1]);
		vertices.Add(extraVertex);
	}
```

Likewise, cases 11, 13, and 14 are modifications of cases 4, 2, and 1.

```
	private void TriangulateCase11 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = c.xNormal;
		Vector2 n2 = a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, a.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddHexagonABD(, point);
				return;
			}
		}
		AddPentagonABD(i);
	}

	private void TriangulateCase13 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, b.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddHexagonACD(, point);
				return;
			}
		}
		AddPentagonACD(i);
	}

	private void TriangulateCase14 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddHexagonBCD(, point);
				return;
			}
		}
		AddPentagonBCD(i);
	}

	private void AddHexagonABD (int i, Vector2 extraVertex) {
		AddHexagon(
			vertices.Count, rowCacheMax[i + 1], rowCacheMax[i + 2],
			rowCacheMin[i + 2], rowCacheMin[i], edgeCacheMin);
		vertices.Add(extraVertex);
	}

	private void AddHexagonACD (int i, Vector2 extraVertex) {
		AddHexagon(
			vertices.Count, rowCacheMin[i + 1], rowCacheMin[i],
			rowCacheMax[i], rowCacheMax[i + 2], edgeCacheMax);
		vertices.Add(extraVertex);
	}

	private void AddHexagonBCD (int i, Vector2 extraVertex) {
		AddHexagon(
			vertices.Count, edgeCacheMin, rowCacheMax[i],
			rowCacheMax[i + 2], rowCacheMin[i + 2], rowCacheMin[i + 1]);
		vertices.Add(extraVertex);
	}
```



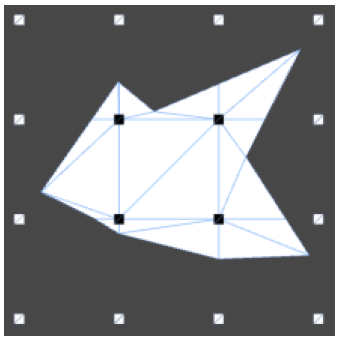
Cutting sharp corners.

Adding Features to Straight Edges

Cases 3, 5, 10, and 12 connect opposite sides of a cell instead of adjacent ones, but we can still use the same approach. This time it's either quads without feature, and pentagons with feature. Once again clamping could produce bad results, so we won't.

```
	private void TriangulateCase3 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.yNormal;
		Vector2 n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.YEdgePoint, n1, b.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddPentagonAB(i, point);
				return;
			}
		}
		AddQuadAB(i);
	}

	private void TriangulateCase5 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = c.xNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, c.XEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddPentagonAC(i, point);
				return;
			}
		}
		AddQuadAC(i);
	}

	private void TriangulateCase10 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = c.xNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, c.XEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddPentagonBD(i, point);
				return;
			}
		}
		AddQuadBD(i);
	}

	private void TriangulateCase12 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.yNormal;
		Vector2 n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.YEdgePoint, n1, b.YEdgePoint, n2);
			if (IsInsideCell(point, a, d)) {
				AddPentagonCD(i, point);
				return;
			}
		}
		AddQuadCD(i);
	}

	private void AddPentagonAB (int i, Vector2 extraVertex) {
		AddPentagon(vertices.Count, edgeCacheMax, rowCacheMin[i + 2], rowCacheMin[i], edgeCacheMin);
		vertices.Add(extraVertex);
	}

	private void AddPentagonAC (int i, Vector2 extraVertex) {
		AddPentagon(vertices.Count, rowCacheMin[i + 1], rowCacheMin[i], rowCacheMax[i], rowCacheMax[i + 1]);
		vertices.Add(extraVertex);
	}

	private void AddPentagonBD (int i, Vector2 extraVertex) {
		AddPentagon(
			vertices.Count, rowCacheMax[i + 1], rowCacheMax[i + 2], rowCacheMin[i + 2], rowCacheMin[i + 1]);
		vertices.Add(extraVertex);
	}
```

```
	private void AddPentagonCD (int i, Vector2 extraVertex) {
		AddPentagon(vertices.Count, edgeCacheMin, rowCacheMax[i], rowCacheMax[i + 2], edgeCacheMax);
		vertices.Add(extraVertex);
	}
```



More than straight edges.

Resolving Ambiguities

The remaining two cases are those with ambiguities. Do they have a diagonal connection, or not? So far we decided to always keep both corners separate, but that is about to change. Let's start simple, by copying the code of cases 2 and 4 into the method of case 6.

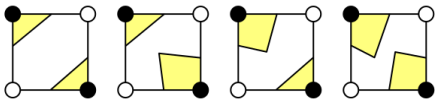
```
	private void TriangulateCase6 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = -b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, b.YEdgePoint, n2);
			if (ClampToCellMinMax(ref point, a, d)) {
				AddQuadB(i, point);
				return;
			}
		}
		AddTriangleB(i);

		n1 = c.xNormal;
		n2 = -a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, a.YEdgePoint, n2);
			if (ClampToCellMaxMin(ref point, a, d)) {
				AddQuadC(i, point);
				return;
			}
		}
		AddTriangleC(i);
	}
```

What we want to do depends of which sharp features exist. Instead of directly triangulating we should track whether we found features and where they are.

```
	private void TriangulateCase6 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		bool sharp1, sharp2;
		Vector2 point1, point2;

		Vector2 n1 = a.xNormal;
		Vector2 n2 = -b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			point1 = ComputeIntersection(a.XEdgePoint, n1, b.YEdgePoint, n2);
			sharp1 = ClampToCellMinMax(ref point1, a, d);
		}
		else {
			point1.x = point1.y = 0f;
			sharp1 = false;
		}

		n1 = c.xNormal;
		n2 = -a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			point2 = ComputeIntersection(c.XEdgePoint, n1, a.YEdgePoint, n2);
			sharp2 = ClampToCellMaxMin(ref point2, a, d);
		}
		else {
			point2.x = point2.y = 0f;
			sharp2 = false;
		}
	}
```

There are four possible outcomes. Either both corners have sharp features, only the first or the second corner has a sharp feature, or neither have a sharp feature.

```
	private void TriangulateCase6 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		…

		if (sharp1) {
			if (sharp2) {
				// Both sharp.
			}
			else {
				// First sharp.
			}
		} else if (sharp2) {
			// Second sharp.
		}
		else {
			// Neither sharp.
		}
	}
```



Four possible situations.

As each possibility is an end point of our method, let's return as soon as we're done.

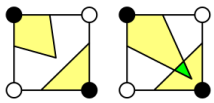
```
		if (sharp1) {
			if (sharp2) {
				// Both sharp.
				return;
			}
			// First sharp.
			return;
		}
		if (sharp2) {
			// Second sharp.
			return;
		}
		// Neither sharp.
```

The simplest situation is when there are no sharp features. There can be no diagonal connection in this case, which is what we assumed until now.

```
		if (sharp1) {
			if (sharp2) {
				// Both sharp.
				return;
			}
			// First sharp.
			return;
		}
		if (sharp2) {
			// Second sharp.
			return;
		}
		AddTriangleB(i);
		AddTriangleC(i);
```

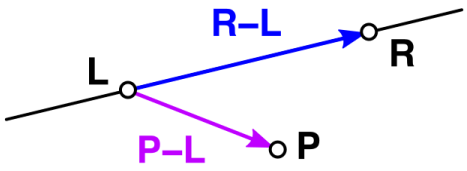
Taking Care of One Sharp Feature

Let's first deal with a single sharp feature. That means one side is a quad while the other is a triangle. If the sharp feature point of the quad ends up inside the triangle, then they overlap and both sides are connected. This boils down to checking whether a point lies below a line. How can we check this?



Disconnected vs. connected.

Suppose we have a horizontal line going through two points **L** and **R**, from left to right. We also have a point **P** somewhere. Then we can define two vectors, **R - L** and **P - L**, which describe how you can go from **L** to the two other points.

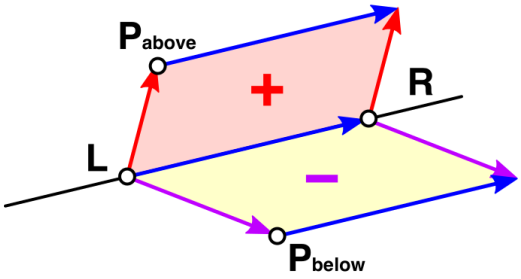


Is P below the line through L and R?

We can use the determinant of these two vectors to figure out the relationship between the line and the point. If their determinant is negative, **P** lies below the line. If it is positive, **P** lies above the line. And if it is zero **P** lies exactly on the line.

Let's put this check in a helper method.

What's the determinant?



Positive determinant above, negative determinant below.

```
private static bool IsBelowLine (Vector2 p, Vector2 start, Vector2 end) {
	float determinant = (end.x - start.x) * (p.y - start.y) - (end.y - start.y) * (p.x - start.x);
	return determinant < 0f;
}
```

Back to `TriangulateCase16`, lets deal with the second sharp feature first. The bottom and right edge points define our line. If the sharp feature ends up below it, we have a connection. If so, triangulate that connection and return, otherwise add a disconnected triangle and quad.

```
		if (sharp2) {
			if (IsBelowLine(point2, a.XEdgePoint, b.YEdgePoint)) {
				TriangulateCase6Connected(i, a, b, c, d);
				return;
			}
			AddTriangleB(i);
			AddQuadC(i, point2);
			return;
		}
```

We add a separate method for the connected case, because we'll be using it in multiple places. Leave it empty for now.

```
private void TriangulateCase6Connected (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
}
```

When checking whether the first feature is below the other line, keep in mind that the situation is upside down from our point of view. That means the line should go from the top edge point to the left edge point, not the other way.

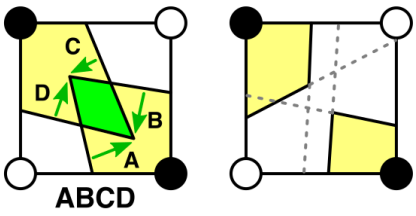
```
		if (sharp1) {
			if (sharp2) {
				// Both sharp.
				return;
			}
			if (IsBelowLine(point1, c.XEdgePoint, a.YEdgePoint)) {
				TriangulateCase6Connected(i, a, b, c, d);
```

```
				return;
			}
			AddQuadB(i, point1);
			AddTriangleC(i);
			return;
		}
```

Investigating Two Sharp Features

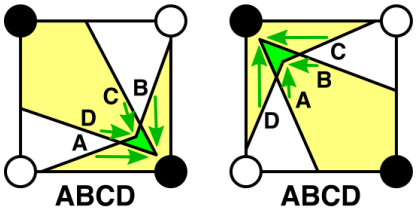
Lastly, we have two sharp features. Once again we can determine whether the two elements overlap by looking at points and lines, but this time it is more complicated. We're dealing with two points that can be above or below two lines each, so there are sixteen possible configurations.

Let's start considering two peaks that both end inside each other. There is clearly an overlap here. For this to be possible, both feature points must lie below the two lines formed by the other feature. Let's use the letters A through D to mark these point-below-line relationships. Then what we just described can be labeled with ABCD. Its complement would be the empty label, which cannot have an overlap.



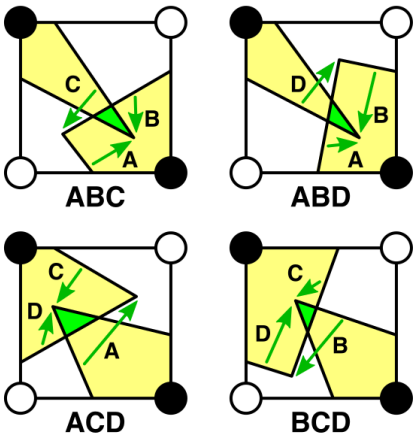
Definitely connected, and definitely not.

ABCD also covers the case when a peak pierces straight through the middle of a valley, in which also have a connection.



ABCD is always connected, shape doesn't matter.

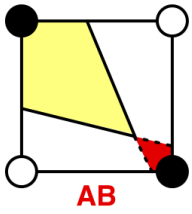
What if only three of these relationships exist, which would be ABC, ABD, ACD, and BCD? Then one of the feature points has to lie inside the other shape, so there is still a guaranteed overlap.



Still clearly overlapping.

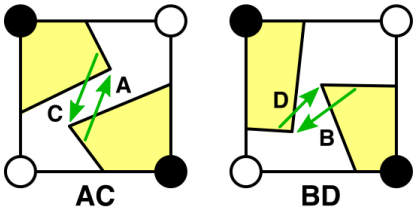
What if we only have two of these relationships? AB and CD means that one feature's point lies below both lines of the other. But then the other point must lie above both lines of the first one, which means it must lie in front of the first point. But then the first point cannot lie inside the other

shape at all. This contradiction means that these two configurations are not possible, so we can ignore them.



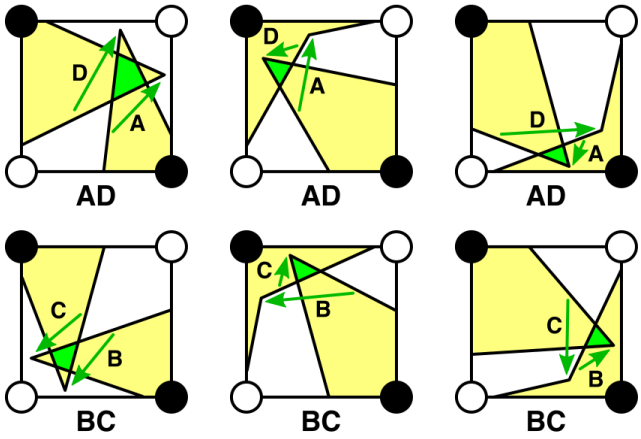
No way to satisfy AB while avoiding C and D.

AC and BD describe configurations for which both points are below one of the other's lines, in a symmetrical way. This means that the features aren't overlapping.



It's a miss.

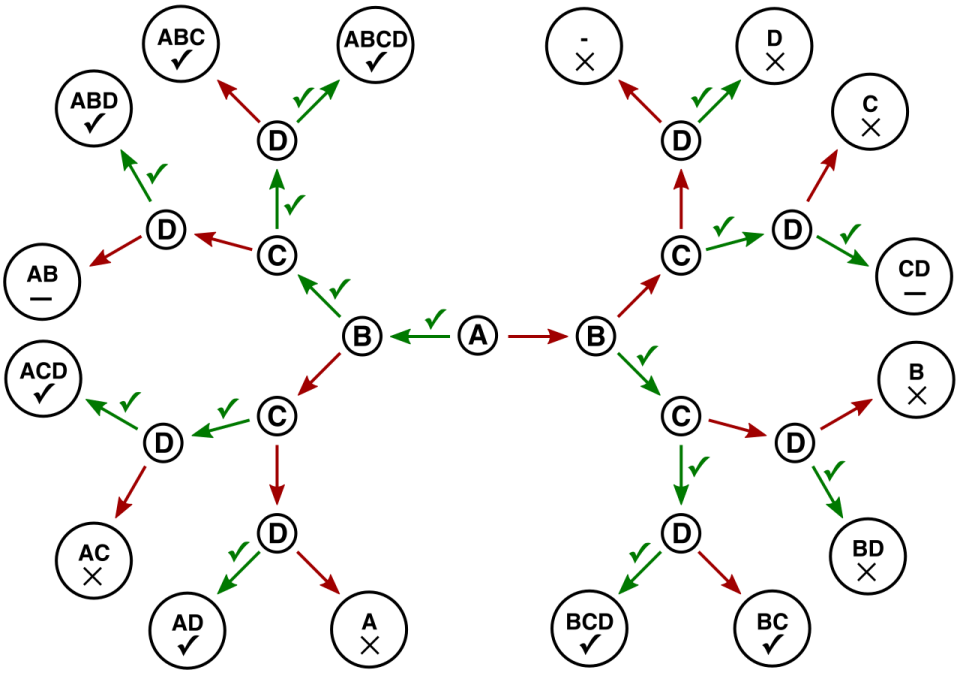
AD and BC are less straightforward. They can represent configurations in which two peaks go through each other, or a peak pierces one side of a valley. Of course this means that there is always a diagonal connection.



Peak or valley, there is a connection.

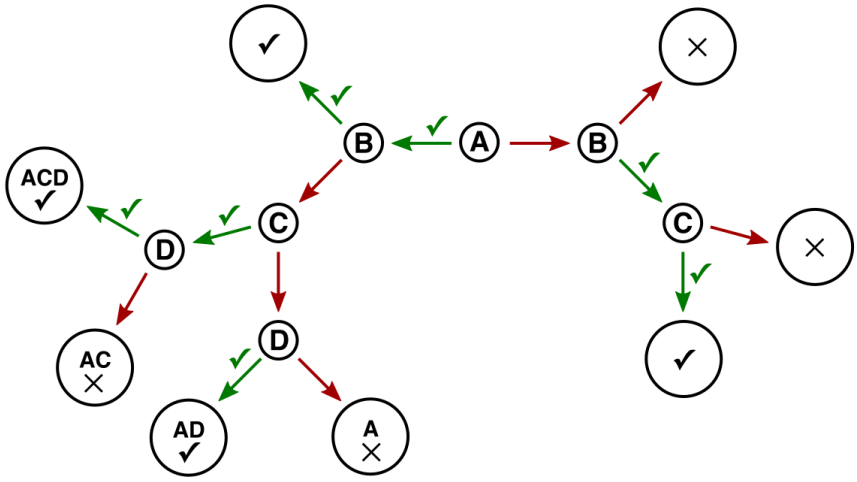
The remaining options are A, B, C, and D in isolation. They are similar to AC and BD, never forming a connection.

So we must form a connection when we have ABCD, ABC, ABD, ACD, BCD, AD, or BC. We must not when we have AC, A, BD, B, C, D, or nothing. And we don't need to worry about AB or CD. Below is a graph that visualizes this. A check mark at an end points represents a connection, while a cross represents no connection. A check mark next to an arrow indicates that the relationship if comes out of exists.



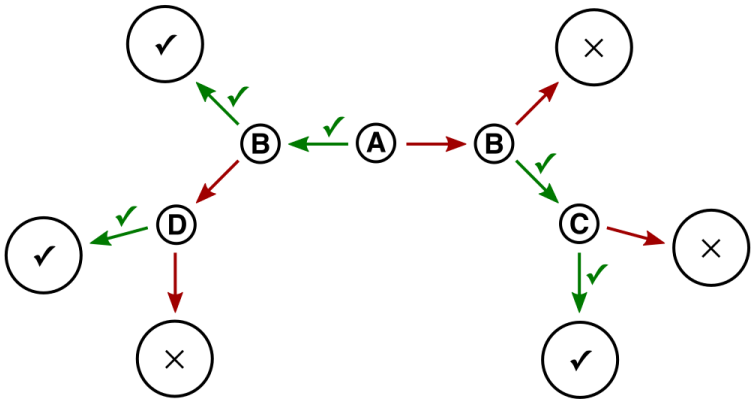
Drawing all possibilities.

All we really care about is whether a path along the graph ends with a check mark or a cross. We can stop early if we already know what we'll end up with.



Collapsing nodes.

If we swap the C and D nodes in the bottom left of the graph, we can collapse that part also, leading to a minimal graph.



Maximum simplification.

This means that when we have A and either B or D, then there is a connection. Alternatively, when we do not have A but do have both B and C, then there is also a connection.

Now we finally know what to test for in `TriangulateCase16` when we have two sharp features.

```
		if (sharp1) {
			if (sharp2) {
				if (A) {
					if (B || C) {
						TriangulateCase6Connected(i, a, b, c, d);
						return;
					}
				}
				else if (B && D) {
					TriangulateCase6Connected(i, a, b, c, d);
					return;
				}
				AddQuadB(i, point1);
				AddQuadC(i, point2);
				return;
			}
			if (IsBelowLine(point1, c.XEdgePoint, a.YEdgePoint)) {
				TriangulateCase6Connected(i, a, b, c, d);
				return;
			}
			AddQuadB(i, point1);
			AddTriangleC(i);
			return;
		}
```

Of course you have to replace the labels with their corresponding point-below-line checks.

```
			if (sharp2) {
				if (IsBelowLine(point2, a.XEdgePoint, point1)) {
					if (
						IsBelowLine(point2, point1, b.YEdgePoint) ||
						IsBelowLine(point1, point2, a.YEdgePoint)) {
						TriangulateCase6Connected(i, a, b, c, d);
						return;
					}
				}
				else if (
					IsBelowLine(point2, point1, b.YEdgePoint) &&
					IsBelowLine(point1, c.XEdgePoint, point2)) {
					TriangulateCase6Connected(i, a, b, c, d);
					return;
				}
				AddQuadB(i, point1);
				AddQuadC(i, point2);
				return;
			}
```

Triangulating the Connection

When there is a diagonal connection, we could just connect both corners of the cell with a hexagon. However, there could be sharp features here as well, which can form shapes that are hard to triangulate. If we only allow sharp features inside their own triangular half of the cell, we eliminate hard cases and can triangulate each side independently.

First consider the case that's like case 14, except that we only triangulate half of the cell. This time we cannot directly return from the method, because we have other half of the cell to take care of. Also note that both halves triangulate the BC diagonal, so we need some way to distinguish the polygon methods. Let's add the cell corner towards which they point.

```
	private void TriangulateCase6Connected (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		Vector2 n1 = a.xNormal;
		Vector2 n2 = -a.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			if (IsInsideCell(point, a, d) && IsBelowLine(point, c.position, b.position)) {
				AddPentagonBCToA(i, point);
			}
			else {
				AddQuadBCToA(i);
			}
		}
		else {
```

```
			AddQuadBCToA(i);
		}
	}

	private void AddQuadBCToA (int i) {
		AddQuad(edgeCacheMin, rowCacheMax[i], rowCacheMin[i + 2], rowCacheMin[i + 1]);
	}

	private void AddPentagonBCToA (int i, Vector2 extraVertex) {
		AddPentagon(vertices.Count, edgeCacheMin, rowCacheMax[i], rowCacheMin[i + 2], rowCacheMin[i + 1]);
		vertices.Add(extraVertex);
	}
```

Then follow up with the other half of the cell, which is based on case 7. As it's the last part, this time we can return early.

```
	private void TriangulateCase6Connected (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		…

		n1 = c.xNormal;
		n2 = -b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			Vector2 point = ComputeIntersection(c.XEdgePoint, n1, b.YEdgePoint, n2);
			if (IsInsideCell(point, a, d) && IsBelowLine(point, b.position, c.position)) {
				AddPentagonBCToD(i, point);
				return;
			}
		}
		AddQuadBCToD(i);
	}

	private void AddQuadBCToD (int i) {
		AddQuad(edgeCacheMax, rowCacheMin[i + 2], rowCacheMax[i], rowCacheMax[i + 1]);
	}

	private void AddPentagonBCToD (int i, Vector2 extraVertex) {
		AddPentagon(vertices.Count, edgeCacheMax, rowCacheMin[i + 2], rowCacheMax[i], rowCacheMax[i + 1]);
		vertices.Add(extraVertex);
	}
```

Doing the Other Diagonal

Case 9 is done in a similar fashion. First detect features similar to cases 1 and 8. I marked the differences with case 6.

```
	private void TriangulateCase9 (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
		bool sharp1, sharp2;
		Vector2 point1, point2;
		Vector2 n1 = a.xNormal;
		Vector2 n2 = a.yNormal;

		if (IsSharpFeature(n1, n2)) {
			point1 = ComputeIntersection(a.XEdgePoint, n1, a.YEdgePoint, n2);
			sharp1 = ClampToCellMaxMax(ref point1, a, d);
		}
		else {
			point1.x = point1.y = 0f;
			sharp1 = false;
		}

		n1 = c.xNormal;
		n2 = b.yNormal;
		if (IsSharpFeature(n1, n2)) {
			point2 = ComputeIntersection(c.XEdgePoint, n1, b.YEdgePoint, n2);
			sharp2 = ClampToCellMinMin(ref point2, a, d);
		}
		else {
			point2.x = point2.y = 0f;
			sharp2 = false;
		}
	}
```

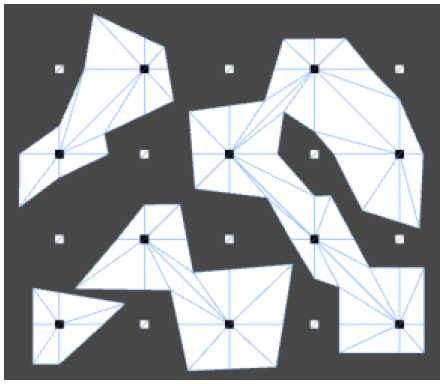
Then again act on which features are present. The double sharp feature overlap check is the same as for case 16, but rotated.

```
		if (sharp1) {
			if (sharp2) {
				if (IsBelowLine(point1, b.YEdgePoint, point2)) {
					if (
						IsBelowLine(point1, point2, c.XEdgePoint) ||
						IsBelowLine(point2, point1, a.XEdgePoint)) {
						TriangulateCase9Connected(i, a, b, c, d);
```

```
					return;
				}
			}
			else if (
				IsBelowLine(point1, point2, c.XEdgePoint) &&
				IsBelowLine(point2, a.YEdgePoint, point1)) {
				TriangulateCase9Connected(i, a, b, c, d);
				return;
			}
			AddQuadA(i, point1);
			AddQuadD(i, point2);
			return;
		}
		if (IsBelowLine(point1, b.YEdgePoint, c.XEdgePoint)) {
			TriangulateCase9Connected(i, a, b, c, d);
			return;
		}
		AddQuadA(i, point1);
		AddTriangleD(i);
		return;
	}
	if (sharp2) {
		if (IsBelowLine(point2, a.YEdgePoint, a.XEdgePoint)) {
			TriangulateCase9Connected(i, a, b, c, d);
			return;
		}
		AddTriangleA(i);
		AddQuadD(i, point2);
		return;
	}
	AddTriangleA(i);
	AddTriangleD(i);
```

And we end with triangulating the connected case.

```
private void TriangulateCase9Connected (int i, Voxel a, Voxel b, Voxel c, Voxel d) {
	Vector2 n1 = a.xNormal;
	Vector2 n2 = b.yNormal;
	if (IsSharpFeature(n1, n2)) {
		Vector2 point = ComputeIntersection(a.XEdgePoint, n1, b.YEdgePoint, n2);
		if (IsInsideCell(point, a, d) && IsBelowLine(point, a.position, d.position)) {
			AddPentagonADToB(i, point);
		}
		else {
			AddQuadADToB(i);
		}
	}
	else {
		AddQuadADToB(i);
	}

	n1 = c.xNormal;
	n2 = a.yNormal;
	if (IsSharpFeature(n1, n2)) {
		Vector2 point = ComputeIntersection(c.XEdgePoint, n1, a.YEdgePoint, n2);
		if (IsInsideCell(point, a, d) && IsBelowLine(point, d.position, a.position)) {
			AddPentagonADToC(i, point);
			return;
		}
	}
	AddQuadADToC(i);
}

private void AddQuadADToB (int i) {
	AddQuad(rowCacheMin[i + 1], rowCacheMin[i], rowCacheMax[i + 2], edgeCacheMax);
}

private void AddPentagonADToB (int i, Vector2 extraVertex) {
	AddPentagon(vertices.Count, rowCacheMin[i + 1], rowCacheMin[i], rowCacheMax[i + 2], edgeCacheMax);
	vertices.Add(extraVertex);
}

private void AddQuadADToC (int i) {
	AddQuad(rowCacheMax[i + 1], rowCacheMax[i + 2], rowCacheMin[i], edgeCacheMin);
}

private void AddPentagonADToC (int i, Vector2 extraVertex) {
	AddPentagon(vertices.Count, rowCacheMax[i + 1], rowCacheMax[i + 2], rowCacheMin[i], edgeCacheMin);
	vertices.Add(extraVertex);
}
```



Diagonal connections, or not.

Finally we can keep the sharp angles intact! Of course it is not perfect, but for most cases we can produce a reasonable approximation. If you find yourself in a situation where the results are not good enough, you can always increase the grid resolution.

Enjoyed the tutorial? Help me make more by becoming a patron!

Downloads

marching-squares-3-01.unitypackage
The project after Hermite Data.

marching-squares-3-finished.unitypackage
The finished project.

What's Hermite data?

Using Hermite data means that you not only store data points of some function, but also derivatives of that function at those points. This data can then be used to approximate the real function through Hermite interpolation. As the normals can be interpreted as the derivatives of a density field, it qualifies as Hermite data. It's named after Charles Hermite, a French mathematician.

Why the cosine?

While people are used to working with degrees, when doing math with them you typically end up converting to radians. And when performing trigonometry to find angles, you end up working with sines and cosines of angles.

When you have two lines crossing at an angle, you can always add a third line so you end up with a right triangle. If you place it so that the hypotenuse of that triangle has unit length, then the lengths of its other sides are equal to the sine and the cosine of the angle at which the lines cross. This is thanks to the Pythagorean theorem. That we end up using the cosine instead of the sine is because of the way the math is performed.

Why add a method for case 0?

For the sake of consistency. And to hint that in a later tutorial we might end up doing something in this case as well.

Isn't this bad for performance?

We've inserted two method invocations into the code path traversed when triangulating a cell. Does this make our code less efficient?

Taken at face value, it does. Whether it ends up being significant is questionable. However, good compilers should inline most of this, eliminating the call overhead while keeping the advantage while programming. This won't happen in debug builds though, and neither inside the Unity editor.

Why does dot lead to cosine?

It follows from the geometric definition of the dot product of two vectors, which states that it is equal to the length of both vectors and the cosine of the angle between them, all multiplied.

To visualize that this is so, remember that for 2D vectors **A** · **B** is equal to **A.x × B.x + A.y × B.y** and define **A** as **(1, 0)**. You have now simply eliminated the Y component of **B**, ending up with the length of its X component. If **B** was a vector with unit length, then the result must be equal to the cosine of the angle between **B** and the X axis. And in this case that's also the angle between **A** and **B**.

This also works for vectors with any orientation. You're effectively projecting one vector straight down to the other. In doing so you end up with a right triangle of which the bottom side's length is the result of the dot product. And if both

vectors were unit length, that's the cosine of their angle.

How to rewrite the dot product?

First you need to know that the dot product is distributive, which means that $\mathbf{A} \cdot (\mathbf{B} + \mathbf{C})$ is equal to $\mathbf{A} \cdot \mathbf{B} + \mathbf{A} \cdot \mathbf{C}$. This is because you can split a vector into any number of sub-vectors, the intermediate steps do not matter. So we can rewrite to $\mathbf{N}_1 \cdot (\mathbf{P}_2 - \mathbf{P}_1) + \mathbf{N}_1 \cdot \mathbf{D}_2 u_2 = 0$, which means that $\mathbf{N}_1 \cdot \mathbf{D}_2 u_2 = -\mathbf{N}_1 \cdot (\mathbf{P}_2 - \mathbf{P}_1)$.

Next, we can pull u_2 out of the dot product because we're doing nothing but multiplying, for which the order doesn't matter. So we get $u_2(\mathbf{N}_1 \cdot \mathbf{D}_2) = -\mathbf{N}_1 \cdot (\mathbf{P}_2 - \mathbf{P}_1)$ and thus $u_2 = -\mathbf{N}_1 \cdot (\mathbf{P}_2 - \mathbf{P}_1) / (\mathbf{N}_1 \cdot \mathbf{D}_2)$.

Why a static method?

If a method doesn't need any object state to do its job, I make it static. Just to point out that it's a bit of functionality that stands on its own.

Shouldn't we avoid using `ref`?

As mentioned in the Curves and Splines tutorial, it is generally best to avoid using reference parameters. In that tutorial we used a `System.Array` method, in this tutorial we're creating such a method ourselves.

You should be very careful with reference parameters because their behavior is atypical, they are rarely used, and as such are not well understood by many programmers. It is easy to make mistakes or create needlessly complex code. Especially if you're creating a public API, avoid using any.

Having said that, we are only using it as a helper method inside one class so its reach is very limited. Convenience beats caution in this case, plus it is a good opportunity to demonstrate its usage. An alternative solution to returning two results would be to create a special struct just for that purpose, but that's rather unwieldy.

Can we generalize this code?

We're basically doing the same thing four times here, with just a few differences. It makes sense to generalize this and create a single method for it. However, in this case it's not that straightforward to do. The resulting method needs to know which voxels to use, how to clamp, and how to triangulate. Parameterizating that is possible, but it's unwieldy and quickly leads to murky code. In this tutorial I prefer clarity over code reduction.

What's the determinant?

The determinant is a value associated with square matrices. It involves multiplying along diagonals in both directions. If you put two 2D vectors **A** and **B** under each other, you end up with a matrix.

A.x A.y

B.x B.y

The determinant of this matrix is equal to **A.x × B.y - A.y × B.x**. This is equal to the oriented area of the parallelogram defined by the two vectors. It's oriented because the sign of the value is determined by the relative direction of the vectors. It is this orientation that we can use to determine on which side of a line a point lies.
