Catlike Coding

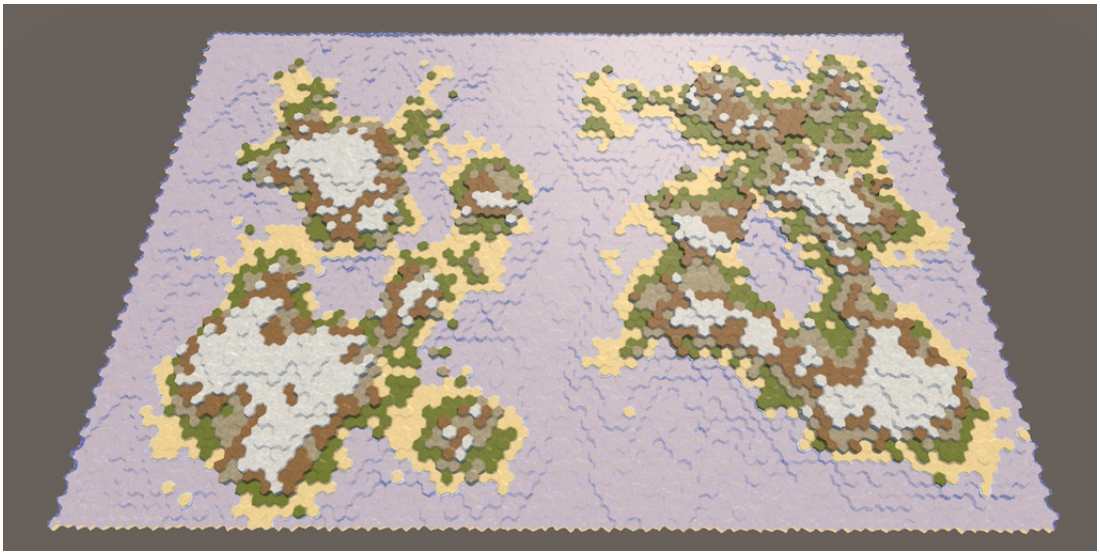
Unity C# Tutorials

Hex Map 24 Regions and Erosion

Add a border of water around the map.
Split the map into multiple regions.
Apply erosion to grind away cliffs.
Move land around to smoothen the terrain.

This is part 24 of a tutorial series about hexagon maps. We laid the foundation for procedural map generation in the previous part. This time we'll constrain where land may appear and have it affected by erosion.

This tutorial is made with Unity 2017.1.0.



Splitting the land and smoothing it out.

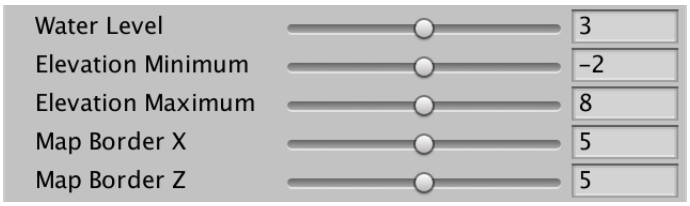
1 Map Border

Because we're pushing up chunks of land at random, it is possible that land ends up touching the edge of the map. This might not be desirable. A map bordered with water contains a natural barrier to keep players away from the edge. So it would be nice if we could prevent land from rising above the water level close to the edge.

1.1 Border Size

How close should the land be allowed to get to the edge of the map? There isn't a universal answer to this, so let's make it configurable. We'll do this by adding two sliders to our `HexMapGenerator` component, one for the borders along the X edges and one for the borders along the Z edges. That makes it possible to use a wider border in one dimension, or only have a border for a single dimension. Let's use a range from 0 to 10 cells, with 5 as the default for both.

```
	[Range(0, 10)]
	public int mapBorderX = 5;

	[Range(0, 10)]
	public int mapBorderZ = 5;
```



Map border sliders.

1.2 Constraining Chunk Centers

Without a border, all cells are valid. When a border is in effect, the minimum valid offset coordinates are increased, while the maximum valid coordinates are decreased. As we'll need to know the valid range when generating chunks, let's keep track of this range with four integer fields.

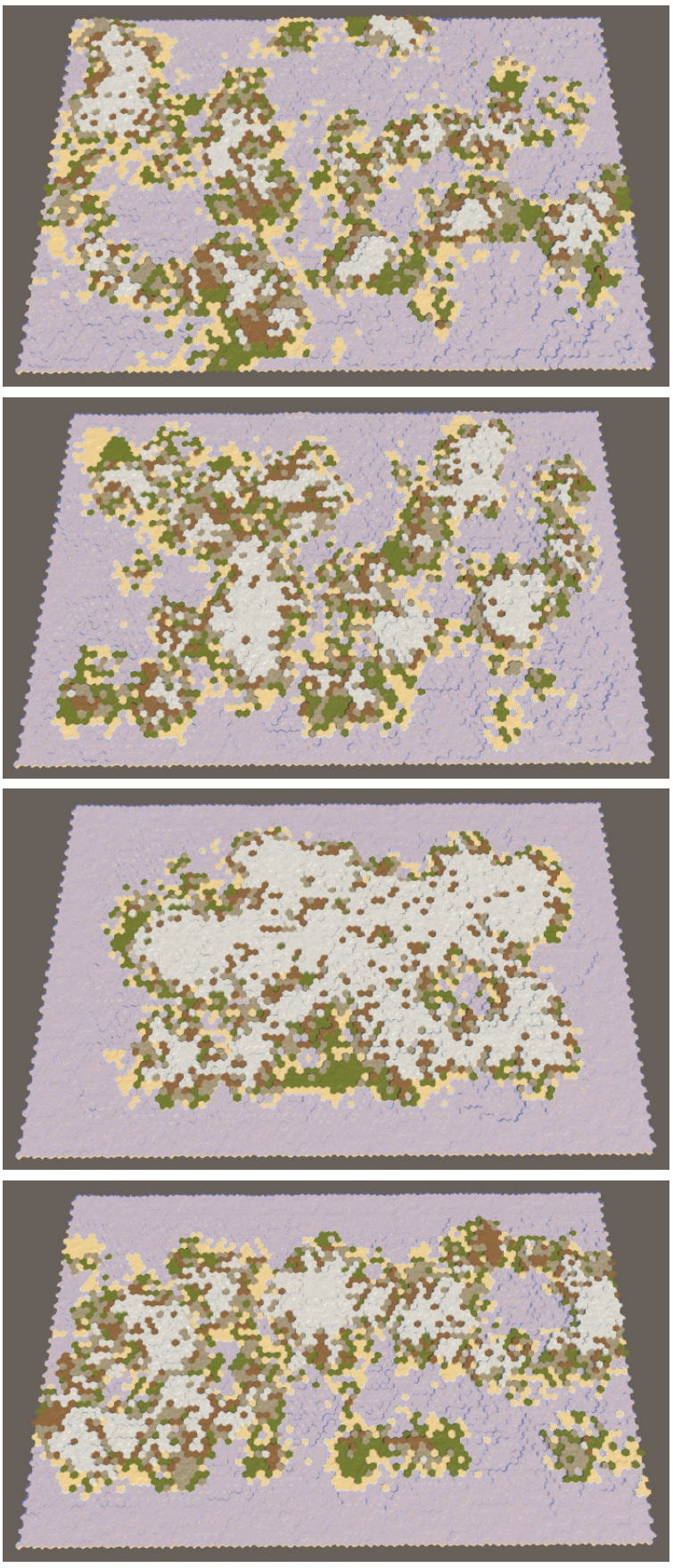
```
	int xMin, xMax, zMin, zMax;
```

Initialize the constraints before creating the land, in `GenerateMap`. We'll use these values as parameters for `Random.Range` invocations, so the maximums are actually exclusive. Without a border, they're equal to the dimension's cell count, so not minus 1.

```
	public void GenerateMap (int x, int z) {
		…
		for (int i = 0; i < cellCount; i++) {
			grid.GetCell(i).WaterLevel = waterLevel;
		}
		xMin = mapBorderX;
		xMax = x - mapBorderX;
		zMin = mapBorderZ;
		zMax = z - mapBorderZ;
		CreateLand();
		…
	}
```

We're not going to strictly enforce that land won't appear beyond the border's edge, because that would just create hard cut-off edges. Instead, we'll only constrain the cells used to start generating chunks. So the rough centers of chunks are constrained, but parts of the chunks can extend into the border region. This is done by adjusting `GetRandomCell` so it picks a cell in the allow offset range.

```
	HexCell GetRandomCell () {
//		return grid.GetCell(Random.Range(0, cellCount));
		return grid.GetCell(Random.Range(xMin, xMax), Random.Range(zMin, zMax));
	}
```



Map borders 0×0, 5×5, 10×10, and 0×10.

With all map settings at their default values, a border of 5 will reliably prevent land from touching the map's edge. However, it is not guaranteed. Land can get close to the edge and sometimes touch it in multiple places.

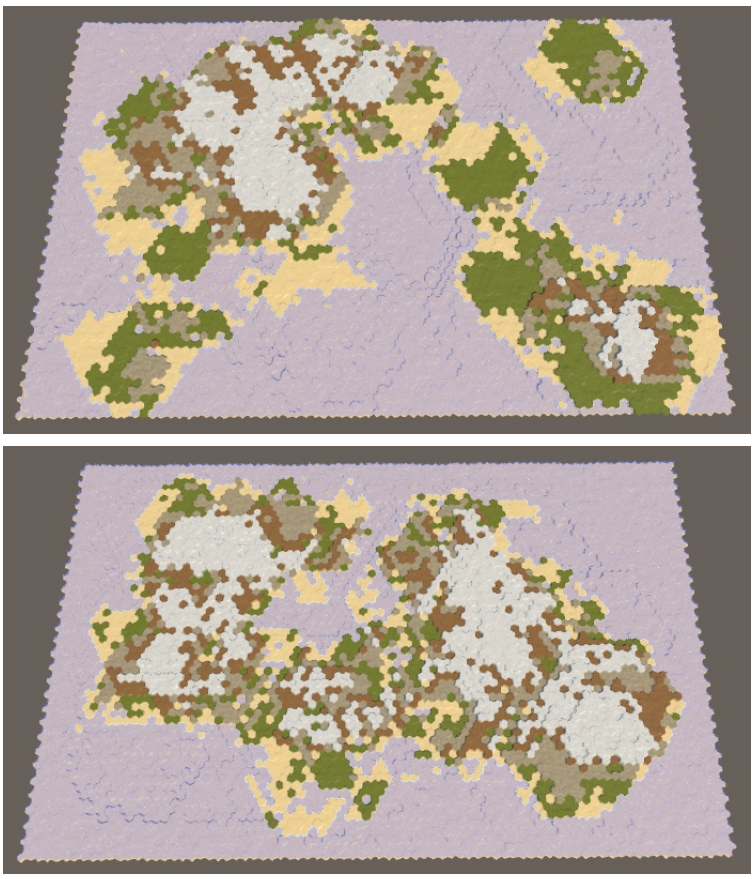
How likely it is for land to cross the entire border region depends on both the border size and the maximum chunk size. Without jitter, chunks are hexagons. A full hexagon with radius r contains $3r^2 + 3r + 1$ cells. If there are hexagons with a radius equal to the border size, then they'll be able to cross it. A full hexagon with a radius of 5 contains 91 cells. As the default maximum is 100 cells per chunk, this means that it is possible for land to bridge a gap of 5 cells, especially when there's jitter. To guarantee this doesn't happen, either decrease the maximum chunk size or increase the border size.

How do you derive how many cells a hexagonal area has?

At radius 0, we're dealing with a single cell. That's where the 1 comes from. At radius 1, there are six additional cells around the center, so $6 + 1$. You can think of these six cells are the tips of six triangles that touch the center. At radius 2, a second row is added to these triangles, so two more cells per triangle, for a total of $6(1 + 2) + 1$. At radius 3, a third row gets added, so three more cells per triangle, for a total of $6(1 + 2 + 3) + 1$, and so on. So in general the formula is

$$6\left(\sum_{i=1}^{r} i\right) + 1 = 6\left(\frac{r(r+1)}{2}\right) + 1 = 3r(r+1) + 1 = 3r^2 + 3r + 1.$$

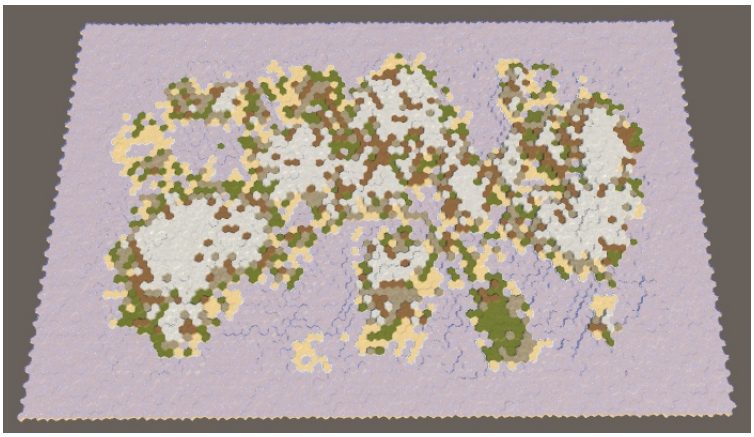
To see this clearly, you can fix the border size at 200. As a full hexagon with radius 8 contains 217 cells, land touching the map edge will be likely. At least, when using the default border size of 5. Increasing the border to 10 will make this much less likely.



Chunk size fixed at 200, map borders 5 and 10.

1.3 Pangea

Note that when you increase the map border while keeping the land percentage the same, you force the land to form in a smaller area. As a result, the default large map will likely produce a single large landmass—a supercontinent or Pangea—with maybe a few small islands. Increasing the border size will make this even more likely, until you're almost guaranteed to get a supercontinent. However, when the land percentage is too high, most of the available region gets filled and you end up with a landmass that looks quite rectangular. To prevent that from happening, you can lower the land percentage.



40% land with map border 10.

Where does the name Pangea come from?

It is the name of the last known supercontinent that existed on earth, long ago. The name is also written an Pangaea. It's derived from the Greek words pan and Gaia. It means something like the entire mother earth, or the whole land.

1.4 Guarding Against Impossible Maps

We generate the desired amount of land by simply continuing to raise chunks until we've achieved the desired landmass. This works because eventually we could end up raising every single cell above the water level. However, when using a map border it can become impossible to reach every cell. When too high a land percentage is desired, this will lead to the generator trying—and failing—to raise more land forever, stuck in an infinite loop. This will freeze our app, which should never happen.

We cannot reliably detect impossible configurations ahead of time, but we can guard against infinite loops. Simply keep track of how many times we've looped in `CreateLand`. If we've iterated a ridiculously large amount of times, we're likely stuck and should stop.

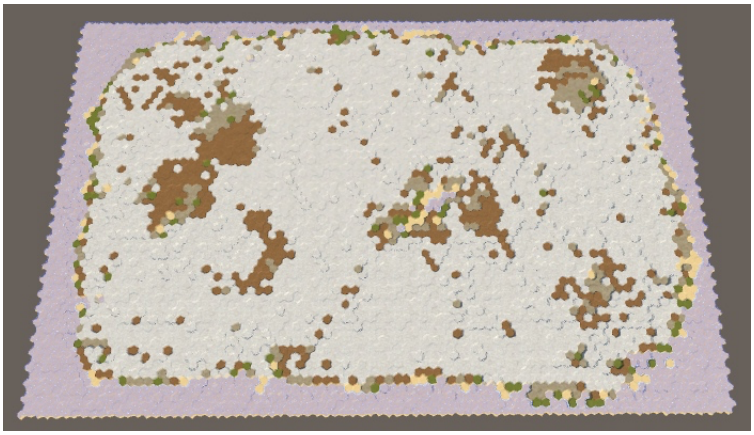
Up to a thousand iterations for a large maps seems acceptable, but 10,000 really is absurd. So let's use that a a cutoff point.

```
	void CreateLand () {
		int landBudget = Mathf.RoundToInt(cellCount * landPercentage * 0.01f);
//		while (landBudget > 0) {
		for (int guard = 0; landBudget > 0 && guard < 10000; guard++) {
			int chunkSize = Random.Range(chunkSizeMin, chunkSizeMax - 1);
			…
		}
	}
```

If we end up with a degenerate map, it won't take that much time to go through 10,000 iterations, because most cells will have achieved maximum elevation quickly, preventing new chunks from growing.

Even after aborting the loop, we still have a valid map. It just won't have the desired amount of land and won't look very interesting. Let's log a warning about this, reporting how much land budget we failed to use up.

```
	void CreateLand () {
		…
		if (landBudget > 0) {
			Debug.LogWarning("Failed to use up " + landBudget + " land budget.");
		}
	}
```



95% land with map border 10, failed to use up the budget.

Why does a failed map still have variety?

The coastline has variety because once elevations inside the spawn region become too high, new chunks are prevent from growing outward. The same concept prevents chunks from growing into small pockets of land that aren't at maximum elevation yet, and just happened to be missed. Also, variety keeps getting added by sinking chunks.

2 Partitioning the Map

Now that we have a map border, we've effectively split the map in two different regions. The border region, and the chunk spawn region. As the spawn region is what really matters, we can consider this a single-region scenario. The region just doesn't cover the entire map. But if that's possible, there's nothing stopping us from cutting the map into multiple disconnected spawn regions. That would make it possible to force multiple landmasses to form independently, representing different continents.

2.1 Map Region

Let's begin by representing a single map region with a struct. That makes it easier to work with multiple regions. Create a `MapRegion` struct for this, which simply contains the boundary fields. As we won't be using this struct outside of `HexMapGenerator`, we can defined it inside this class as a private inner struct. The four integer fields can then be replaced with a single `MapRegion` field.

```
	//	int xMin, xMax, zMin, zMax;
	struct MapRegion {
		public int xMin, xMax, zMin, zMax;
	}

	MapRegion region;
```

To keep things working, we now have to prefix our min-max fields with `region.` in `GenerateMap`.

```
		region.xMin = mapBorderX;
		region.xMax = x - mapBorderX;
		region.zMin = mapBorderZ;
		region.zMax = z - mapBorderZ;
```

And also in `GetRandomCell`.

```
	HexCell GetRandomCell () {
		return grid.GetCell(
			Random.Range(region.xMin, region.xMax),
			Random.Range(region.zMin, region.zMax)
		);
	}
```

2.2 Multiple Regions

To support multiple regions, replace the single `MapRegion` field with a list of regions.

```
	// MapRegion region;
	List<MapRegion> regions;
```

At this point it is a good idea to add a dedicated method for the creation of regions. It should create the required list, or clear it if there already is one. After that, define the single region like we did earlier and add it to the list.

```
	void CreateRegions () {
		if (regions == null) {
			regions = new List<MapRegion>();
		}
		else {
			regions.Clear();
		}

		MapRegion region;
		region.xMin = mapBorderX;
		region.xMax = grid.cellCountX - mapBorderX;
		region.zMin = mapBorderZ;
		region.zMax = grid.cellCountZ - mapBorderZ;
		regions.Add(region);
	}
```

Invoke this method in `GenerateMap` instead of directly creating a region.

```
//		region.xMin = mapBorderX;
//		region.xMax = x - mapBorderX;
//		region.zMin = mapBorderZ;
//		region.zMax = z - mapBorderZ;
		CreateRegions();
		CreateLand();
```

To make `GetRandomCell` work with an arbitrary region, give it a `MapRegion` parameter.

```
	HexCell GetRandomCell (MapRegion region) {
		return grid.GetCell(
			Random.Range(region.xMin, region.xMax),
			Random.Range(region.zMin, region.zMax)
		);
	}
```

The `RaiseTerraion` and `SinkTerrain` methods must now pass the correct region to `GetRandomCell`. To do this, they also need a region parameter each.

```
	int RaiseTerrain (int chunkSize, int budget, MapRegion region) {
		searchFrontierPhase += 1;
		HexCell firstCell = GetRandomCell(region);
		…
	}

	int SinkTerrain (int chunkSize, int budget, MapRegion region) {
		searchFrontierPhase += 1;
		HexCell firstCell = GetRandomCell(region);
		…
	}
```

The `CreateLand` method has to determine which region to raise or sink chunks for. To balance the land between regions, simply loop through the region list repeatedly.

```
	void CreateLand () {
		int landBudget = Mathf.RoundToInt(cellCount * landPercentage * 0.01f);
		for (int guard = 0; landBudget > 0 && guard < 10000; guard++) {
			for (int i = 0; i < regions.Count; i++) {
				MapRegion region = regions[i];
				int chunkSize = Random.Range(chunkSizeMin, chunkSizeMax - 1);
				if (Random.value < sinkProbability) {
					landBudget = SinkTerrain(chunkSize, landBudget, region);
				}
				else {
					landBudget = RaiseTerrain(chunkSize, landBudget, region);
				}
			}
		}
		if (landBudget > 0) {
			Debug.LogWarning("Failed to use up " + landBudget + " land budget.");
		}
	}
```

However, we should take care to distribute the sinking of chunks evenly as well. This can be done by determining whether we sink or not for all regions at once.

```
		for (int guard = 0; landBudget > 0 && guard < 10000; guard++) {
			bool sink = Random.value < sinkProbability;
			for (int i = 0; i < regions.Count; i++) {
				MapRegion region = regions[i];
				int chunkSize = Random.Range(chunkSizeMin, chunkSizeMax - 1);
//				if (Random.value < sinkProbability) {
				if (sink) {
					landBudget = SinkTerrain(chunkSize, landBudget, region);
				}
				else {
					landBudget = RaiseTerrain(chunkSize, landBudget, region);
				}
			}
		}
```

Finally, to ensure we exactly use up our land budget, we have to stop the process as soon as the budget reaches zero. This can happen at any point in the region loop. So move the check for zero budget to the inner loop. Actually, we can limit this check to only after land has been raised, as sinking a chunk will never use up budget. When we're done, we can directly exit the `CreateLand` method.

```
//		for (int guard = 0; landBudget > 0 && guard < 10000; guard++) {
		for (int guard = 0; guard < 10000; guard++) {
			bool sink = Random.value < sinkProbability;
			for (int i = 0; i < regions.Count; i++) {
				MapRegion region = regions[i];
				int chunkSize = Random.Range(chunkSizeMin, chunkSizeMax - 1);
					if (sink) {
					landBudget = SinkTerrain(chunkSize, landBudget, region);
				}
				else {
					landBudget = RaiseTerrain(chunkSize, landBudget, region);
					if (landBudget == 0) {
						return;
					}
				}
			}
		}
```

2.3 Two Regions

Although we support multiple regions now, we still only define a single one. Let's change that by adjusting `CreateRegions`, so it vertically splits the map in two. To do so, halve the `xMax` value of the region that we add. Then use that same value for `xMin` and use the original value again for `xMax`, using that as a second region.

```
		MapRegion region;
		region.xMin = mapBorderX;
		region.xMax = grid.cellCountX / 2;
		region.zMin = mapBorderZ;
		region.zMax = grid.cellCountZ - mapBorderZ;
		regions.Add(region);
		region.xMin = grid.cellCountX / 2;
		region.xMax = grid.cellCountX - mapBorderX;
		regions.Add(region);
```

Generating maps at this point doesn't make any difference. Even though we've defined two regions, they cover the same area as the old single region. To pull them apart, we have to leave empty space in between them. We'll do this by adding a slider for a region border, using the same range and default as for the map borders.

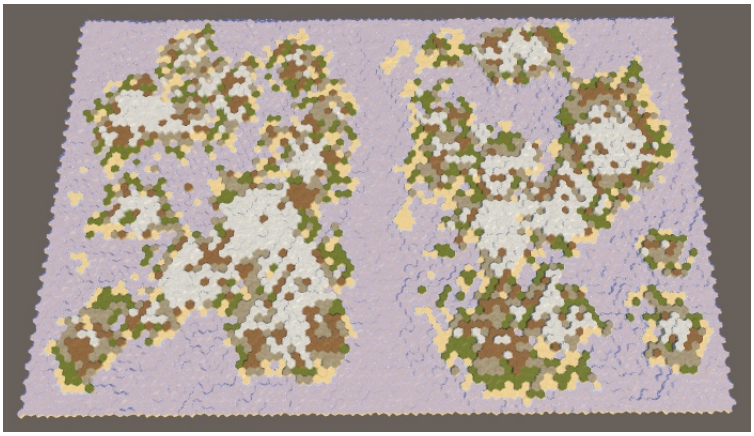
```
	[Range(0, 10)]
	public int regionBorder = 5;
```

Region border slider.

As land could form on either side of the space between regions, it will be much more likely that a land bridge forms then on the edges of the map. To counter this, we'll use the region border to define a spawn-free zone between the dividing line and the area in which chunks are allowed to start. This means that the distance between adjacent regions is double the region border size.

To apply the region border, subtract it from `xMax` of the first region and add it to `xMin` of the second region.

```
		MapRegion region;
		region.xMin = mapBorderX;
		region.xMax = grid.cellCountX / 2 - regionBorder;
		region.zMin = mapBorderZ;
		region.zMax = grid.cellCountZ - mapBorderZ;
		regions.Add(region);
		region.xMin = grid.cellCountX / 2 + regionBorder;
		region.xMax = grid.cellCountX - mapBorderX;
		regions.Add(region);
```

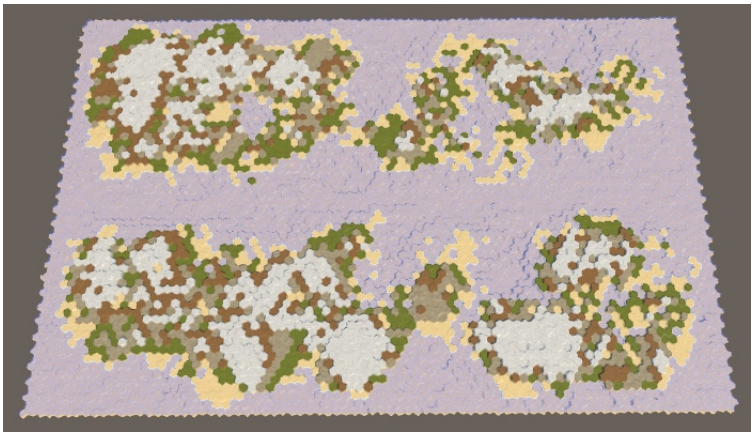


Map vertically split in two regions.

Using default settings, this will produce maps with two clearly separate regions, though just as with a single region and a large map border, we're not guaranteed to get exactly two landmasses. Most of the time, it will be two large continents, maybe with a few islands each. Occasionally, a region will end up containing two or more large islands instead. And sometimes the two continents will be connected by a land bridge.

Of course it is also possible to split the map horizontally, swapping the approach for the X and Z dimensions. Let's randomly pick one of the two possible orientations.

```
		MapRegion region;
		if (Random.value < 0.5f) {
			region.xMin = mapBorderX;
			region.xMax = grid.cellCountX / 2 - regionBorder;
			region.zMin = mapBorderZ;
			region.zMax = grid.cellCountZ - mapBorderZ;
			regions.Add(region);
			region.xMin = grid.cellCountX / 2 + regionBorder;
			region.xMax = grid.cellCountX - mapBorderX;
			regions.Add(region);
		}
		else {
			region.xMin = mapBorderX;
			region.xMax = grid.cellCountX - mapBorderX;
			region.zMin = mapBorderZ;
			region.zMax = grid.cellCountZ / 2 - regionBorder;
			regions.Add(region);
			region.zMin = grid.cellCountZ / 2 + regionBorder;
			region.zMax = grid.cellCountZ - mapBorderZ;
			regions.Add(region);
		}
```



Map horizontally split in two regions.

Because we're using a wide map, a horizontal split will produce wider and thinner regions. This makes it more likely that regions will end up with multiple disconnected landmasses.

2.4 Up to Four Regions

Let's make the amount of regions configurable, supporting between 1 and 4 four of them.

```
	[Range(1, 4)]
	public int regionCount = 1;
```

Slider for the amount of regions.

We can use a `switch` statement to select the correct region code to execute. Begin by reintroducing the code for a single region, using it as the default, while keeping the code for two regions for case 2.

```
		MapRegion region;
		switch (regionCount) {
		default:
			region.xMin = mapBorderX;
			region.xMax = grid.cellCountX - mapBorderX;
			region.zMin = mapBorderZ;
			region.zMax = grid.cellCountZ - mapBorderZ;
			regions.Add(region);
			break;
		case 2:
			if (Random.value < 0.5f) {
				region.xMin = mapBorderX;
				region.xMax = grid.cellCountX / 2 - regionBorder;
				region.zMin = mapBorderZ;
				region.zMax = grid.cellCountZ - mapBorderZ;
				regions.Add(region);
				region.xMin = grid.cellCountX / 2 + regionBorder;
				region.xMax = grid.cellCountX - mapBorderX;
				regions.Add(region);
			}
			else {
				region.xMin = mapBorderX;
				region.xMax = grid.cellCountX - mapBorderX;
				region.zMin = mapBorderZ;
				region.zMax = grid.cellCountZ / 2 - regionBorder;
				regions.Add(region);
				region.zMin = grid.cellCountZ / 2 + regionBorder;
				region.zMax = grid.cellCountZ - mapBorderZ;
				regions.Add(region);
			}
			break;
		}
```

What's a `switch` statement?

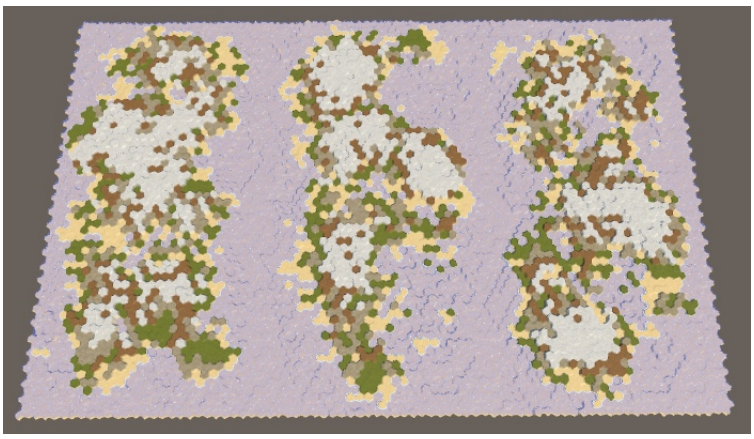
It is an alternative to writing a sequence if-else-if-else statements. The switch is applied to a variable and labels are used to indicate which code to execute. There is also a `default` label, which functions like a final `else` block. Each case has to be terminated by either a `break` statement or a `return` statement.

To keep a `switch` block legible, it is generally a good idea to keep the cases short, ideally a single statement or method invocation. I didn't bother doing that for the example region code, but if you're going to make more interesting regions I suggest you use separate methods. For example:

```
	switch (regionCount) {
		default: CreateOneRegion(); break;
		case 2: CreateTwoRegions(); break;
		case 3: CreateThreeRegions(); break;
		case 4: CreateFourRegions(); break;
	}
```

Three regions work similar as two, except we're using thirds instead of halves. In this case, a horizontal split would produce regions that are too narrow, so we'll only support a vertical split. Note also that we end up with twice as much region border space, so there's less space to spawn chunks than for two regions.

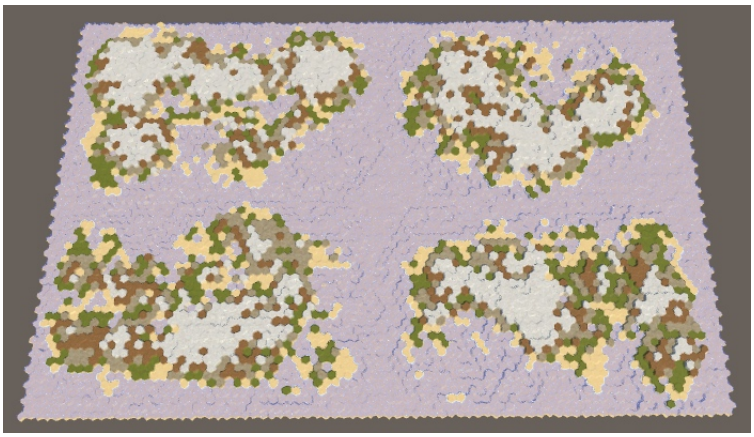
```
	switch (regionCount) {
	default:
		…
		break;
	case 2:
		…
		break;
	case 3:
		region.xMin = mapBorderX;
		region.xMax = grid.cellCountX / 3 - regionBorder;
		region.zMin = mapBorderZ;
		region.zMax = grid.cellCountZ - mapBorderZ;
		regions.Add(region);
		region.xMin = grid.cellCountX / 3 + regionBorder;
		region.xMax = grid.cellCountX * 2 / 3 - regionBorder;
		regions.Add(region);
		region.xMin = grid.cellCountX * 2 / 3 + regionBorder;
		region.xMax = grid.cellCountX - mapBorderX;
		regions.Add(region);
		break;
	}
```



Three regions.

Four regions can be done by combining a horizontal and vertical split, creating one region in each corner of the map.

```
		switch (regionCount) {
		…
		case 4:
			region.xMin = mapBorderX;
			region.xMax = grid.cellCountX / 2 - regionBorder;
			region.zMin = mapBorderZ;
			region.zMax = grid.cellCountZ / 2 - regionBorder;
			regions.Add(region);
			region.xMin = grid.cellCountX / 2 + regionBorder;
			region.xMax = grid.cellCountX - mapBorderX;
			regions.Add(region);
			region.zMin = grid.cellCountZ / 2 + regionBorder;
			region.zMax = grid.cellCountZ - mapBorderZ;
			regions.Add(region);
			region.xMin = mapBorderX;
			region.xMax = grid.cellCountX / 2 - regionBorder;
			regions.Add(region);
			break;
		}
	}
```



Four regions.

The approach that we used here is the most straightforward way to partition the map. It generates regions that are roughly equal in landmass and their variety can be controlled via the other map generation settings. However, it will always be at least fairly obvious that the map has been split along straight lines. The more control you want, the less organic the result will look. So it's fine if you need multiple fairly equal regions for gameplay reasons. But if you want the most varied and unconstrained land, you have to make do with a single region.

Having said that, there are other ways to partition the map. You're not limited to straight lines. You're also not limited to using regions of the same size, nor do you need to cover the entire map with regions. You can leave holes too. You could also have regions overlap, or change the land distribution between regions. It's even possible to define different generator settings per region—though that's more complex—for example ensuring that a map gets both a large continent and an archipelago.

3 Erosion

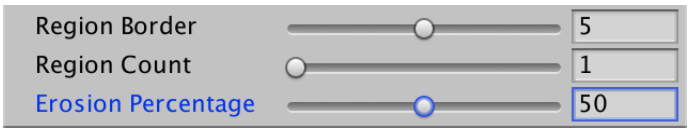
All the maps that we have generated so far appear rather rough and jagged. Real terrain can look like this, but over time it becomes more smooth and polished, its sharp features fading away due to erosion. To improve our maps, we should apply this erosion process as well. We'll do this after creating the rough land, in a separate method.

```
	public void GenerateMap (int x, int z) {
		…
		CreateRegions();
		CreateLand();
		ErodeLand();
		SetTerrainType();
		…
	}

	…

	void ErodeLand () {}
```

3.1 Erosion Percentage

The more time has passed, the more erosion has taken place. So how much erosion we want isn't fixed, it has to be configurable. At minimum, there is zero erosion, which is the case for the maps that we have so far generated. At maximum, there is total erosion, meaning that further application of eroding forces will no longer change the terrain. So the erosion setting should be a percentage from 0 to 100, and we'll use 50 as the default.

```
	[Range(0, 100)]
	public int erosionPercentage = 50;
```



Erosion slider.

3.2 Finding Erodible Cells

Erosion makes the terrain more smooth. In our case, the only real sharp terrain features that we have are cliffs. So these are the targets of our erosion process. If a cliff exists, erosion should shrink it, until it has ultimately been reduced to a slope. We won't flatten slopes further, because that would produce uninteresting terrain. To do this, we have to figure out which cells sit at the top of cliffs, and lower their elevation. These are our erodible cells.

Let's create a method to determine whether a cell is erodible. It does this by looking through the cell's neighbors until it finds a sufficiently large elevation difference. As cliffs require at least an elevation difference of two, a cell is erodible if one or more of its neighbors is at least two steps below it. If there is no such neighbor, then the cell isn't erodible.

```
	bool IsErodible (HexCell cell) {
		int erodibleElevation = cell.Elevation - 2;
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
			HexCell neighbor = cell.GetNeighbor(d);
			if (neighbor && neighbor.Elevation <= erodibleElevation) {
				return true;
			}
		}
		return false;
	}
```

We can use this method in `ErodeLand` to loop through all cells and keep track of all erodible ones in a temporary list.

```
	void ErodeLand () {
		List<HexCell> erodibleCells = ListPool<HexCell>.Get();
		for (int i = 0; i < cellCount; i++) {
			HexCell cell = grid.GetCell(i);
			if (IsErodible(cell)) {
				erodibleCells.Add(cell);
			}
		}

		ListPool<HexCell>.Add(erodibleCells);
	}
```

Once we know the total amount of erodible cells, we can use the erosion percentage to determine how many erodible cells should remain. For example, if the percentage is 50, then we should erode cells until we end up with half the original amount. If the percentage is 100 instead, we won't stop until all erodible cells are gone.

```
	void ErodeLand () {
		List<HexCell> erodibleCells = ListPool<HexCell>.Get();
		for (int i = 0; i < cellCount; i++) {
			…
		}

		int targetErodibleCount =
			(int)(erodibleCells.Count * (100 - erosionPercentage) * 0.01f);

		ListPool<HexCell>.Add(erodibleCells);
	}
```

Shouldn't we only count erodible land cells?

Erosion also happens underwater. There are different types of erosion, but we don't have to worry about those details and can make do with a single generic approach.

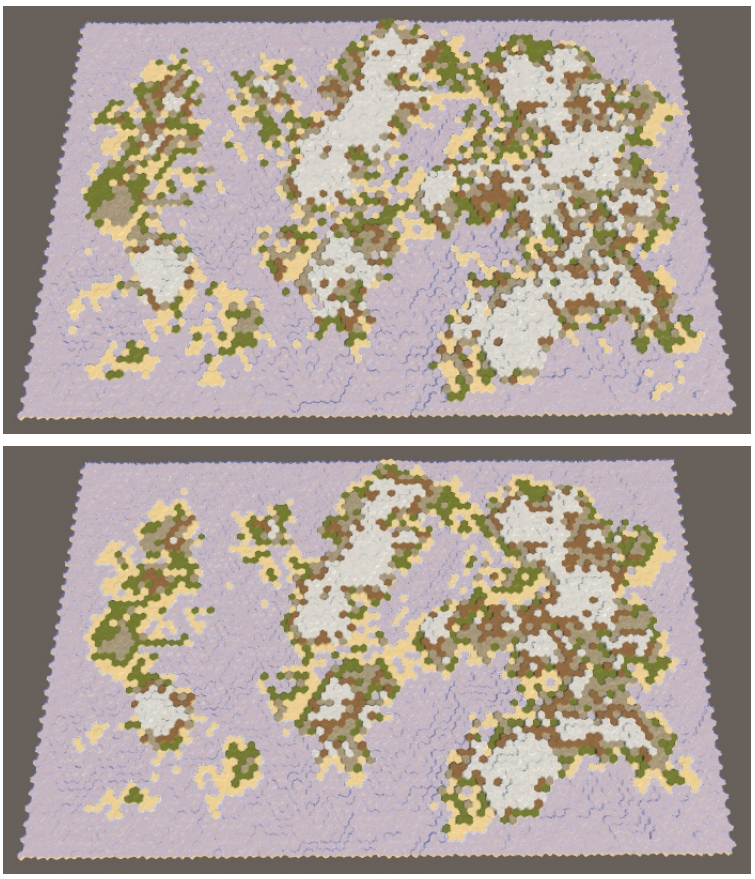
3.3 Lowering Cells

Let's start by being naive and assume that simply decrementing an erodible cell's elevation will make it no longer erodible. If that were true, we simply have to keep picking random cells from the list, decrement their elevation, then remove them from the list. We repeat this until we reach the desired amount of erodible cells.

```
		int targetErodibleCount =
			(int)(erodibleCells.Count * (100 - erosionPercentage) * 0.01f);

		while (erodibleCells.Count > targetErodibleCount) {
			int index = Random.Range(0, erodibleCells.Count);
			HexCell cell = erodibleCells[index];

			cell.Elevation -= 1;

			erodibleCells.Remove(cell);
		}

		ListPool<HexCell>.Add(erodibleCells);
```

To prevent the searching required by `erodibleCells.Remove`, just override the current cell with the last one in the list, then remove the last element. We don't care about their order anyway.

```
//			erodibleCells.Remove(cell);
			erodibleCells[index] = erodibleCells[erodibleCells.Count - 1];
			erodibleCells.RemoveAt(erodibleCells.Count - 1);
```

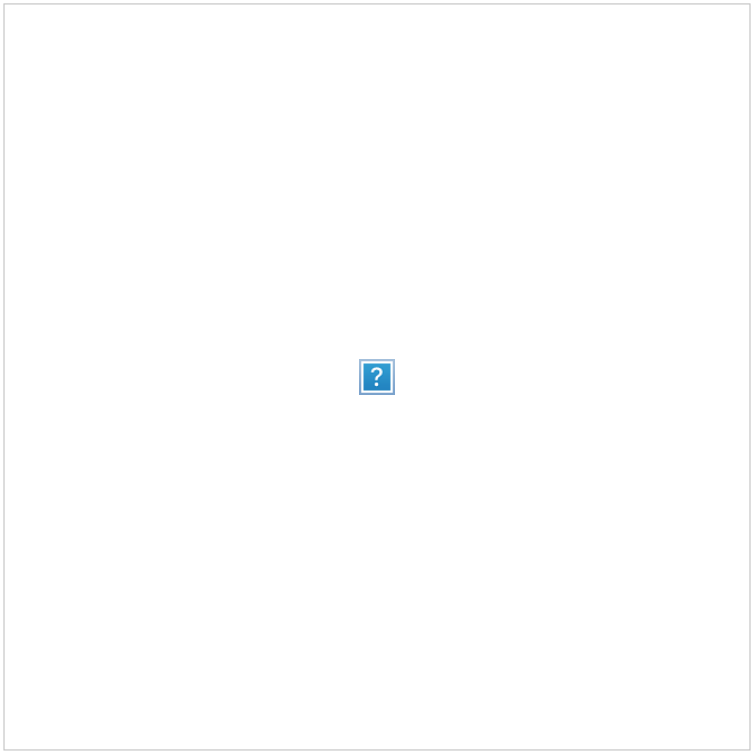


Naive lowering 0% and 100% erodible cells, map seed 1957632474.

3.4 Erodible Bookkeeping

Our naive approach does apply some erosion, but not nearly enough. That's because a cell might still be erodible after its elevation has been decremented once. So only remove the cell if it's no longer erodible.

```
			if (!IsErodible(cell)) {
				erodibleCells[index] = erodibleCells[erodibleCells.Count - 1];
				erodibleCells.RemoveAt(erodibleCells.Count - 1);
			}
```

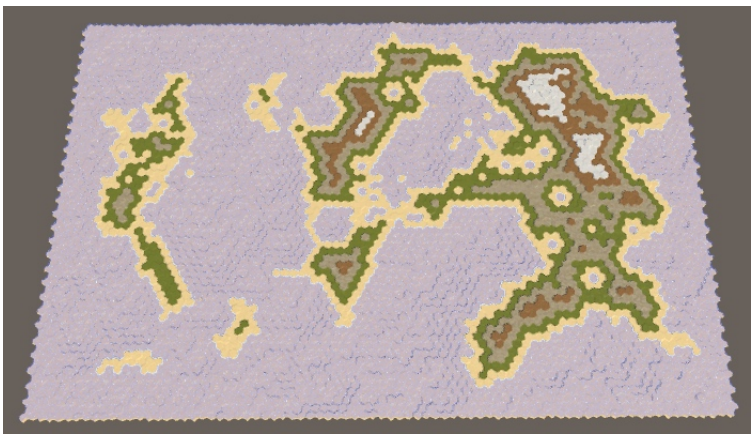


100% erosion, while keeping erodible cells in the list.

This produces much stronger erosion, but it still doesn't eliminate all cliffs when used at 100%. That's because when a cell's elevation is lowered, one of its neighbors might become erodible. So it's possible that we end up with more erodible cells than we started with.

After lowering the cell, we have to check all its neighbors. If they're now erodible but not yet in the list, we have to add them to it.

```
			if (!IsErodible(cell)) {
				erodibleCells[index] = erodibleCells[erodibleCells.Count - 1];
				erodibleCells.RemoveAt(erodibleCells.Count - 1);
			}

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				HexCell neighbor = cell.GetNeighbor(d);
				if (
					neighbor && IsErodible(neighbor) &&
					!erodibleCells.Contains(neighbor)
				) {
					erodibleCells.Add(neighbor);
				}
			}
```



All erodible cells lowered.

3.5 Conserving Landmass

Our erosion process can now continue until all cliffs have been eliminated. The effect on the land is dramatic. A lot of the landmass is gone and we end up with a significantly lower land percentage than desired. This happens because we remove land from the map.

Actual erosion does not destroy matter. It takes it away from one place and deposits it somewhere else. We can do the same thing. Whenever we lower one cell, we should raise one of its neighbors. The single level of elevation effectively migrates to a lower cell. This conserves the total elevation of the map, it just smoothes it out.

To make this happen, we have to decide where to move the eroded material to. This is our erosion target. Let's create a method to determine the target, given a cell that we're about to erode. As that cell must have a cliff, it makes sense to pick the cell at the bottom of that cliff as the target. But the erodible cell could have multiple cliffs. So let's check all the neighbors and put all candidates in a temporary list, then pick one of them at random.

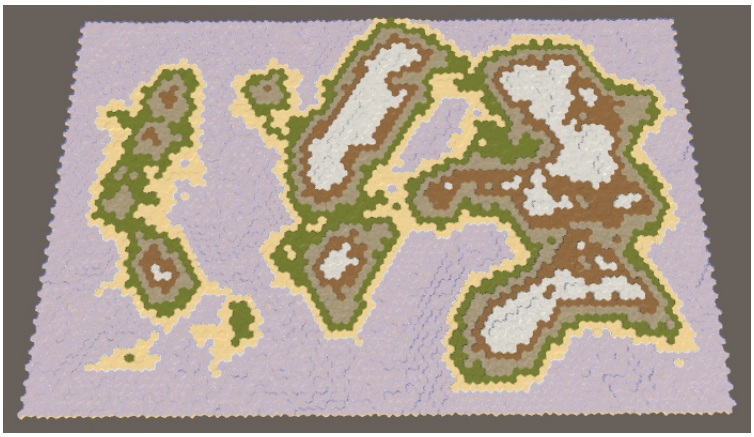
```
	HexCell GetErosionTarget (HexCell cell) {
		List<HexCell> candidates = ListPool<HexCell>.Get();
		int erodibleElevation = cell.Elevation - 2;
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
			HexCell neighbor = cell.GetNeighbor(d);
			if (neighbor && neighbor.Elevation <= erodibleElevation) {
				candidates.Add(neighbor);
			}
		}
		HexCell target = candidates[Random.Range(0, candidates.Count)];
		ListPool<HexCell>.Add(candidates);
		return target;
	}
```

In `ErodeLand`, determine the target cell directly after picking the erodible cell. Then decrement and increment the cell elevations directly after each other. This might make the target cell itself erodible, but that's covered when we check the neighbors of the cell we just eroded.

```
			HexCell cell = erodibleCells[index];
			HexCell targetCell = GetErosionTarget(cell);

			cell.Elevation -= 1;
			targetCell.Elevation += 1;

			if (!IsErodible(cell)) {
				erodibleCells[index] = erodibleCells[erodibleCells.Count - 1];
				erodibleCells.RemoveAt(erodibleCells.Count - 1);
			}
```

Because we've raised the target cell, some of that cell's neighbors might now no longer be erodible. We have to go though them and check whether they're erodible. If they're not but they are in the list, then we have to remove them from it.

```
			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				HexCell neighbor = cell.GetNeighbor(d);
				…
			}

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				HexCell neighbor = targetCell.GetNeighbor(d);
				if (
					neighbor && !IsErodible(neighbor) &&
					erodibleCells.Contains(neighbor)
				) {
					erodibleCells.Remove(neighbor);
				}
			}
```



100% erosion with conserved landmass.

Erosion will now smooth the terrain much better, lowering some areas while raising others. As a result, the landmass can both increase and shrink. This can adjust the land percentage by a few percent in either direction, but large deviations are rare. So the more erosion you apply, the less control you have over the final land percentage.

3.6 Faster Erosion

While we don't need to worry too much about the efficiency of our erosion algorithm, some quick gains can be made. First, note that we explicitly check whether the cell we eroded is still erodible. If not, we efficiently remove it from the list. So we can skip checking this cell when going through the target cell's neighbors.

```
			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				HexCell neighbor = targetCell.GetNeighbor(d);
				if (
					neighbor && neighbor != cell && !IsErodible(neighbor) &&
					erodibleCells.Contains(neighbor)
				) {
					erodibleCells.Remove(neighbor);
				}
			}
```

Second, we only have to bother checking the target cell's neighbors if there used to be a cliff between them, but not anymore. This is only the case if the neighbor is now one step higher than the target cell. If so, then the neighbor was guaranteed to be in the list, so we don't have to verify this, which means that we can skip a needless search.

```
				HexCell neighbor = targetCell.GetNeighbor(d);
				if (
					neighbor && neighbor != cell &&
					neighbor.Elevation == targetCell.Elevation + 1 &&
					!IsErodible(neighbor)
//					&& erodibleCells.Contains(neighbor)
				) {
					erodibleCells.Remove(neighbor);
				}
```

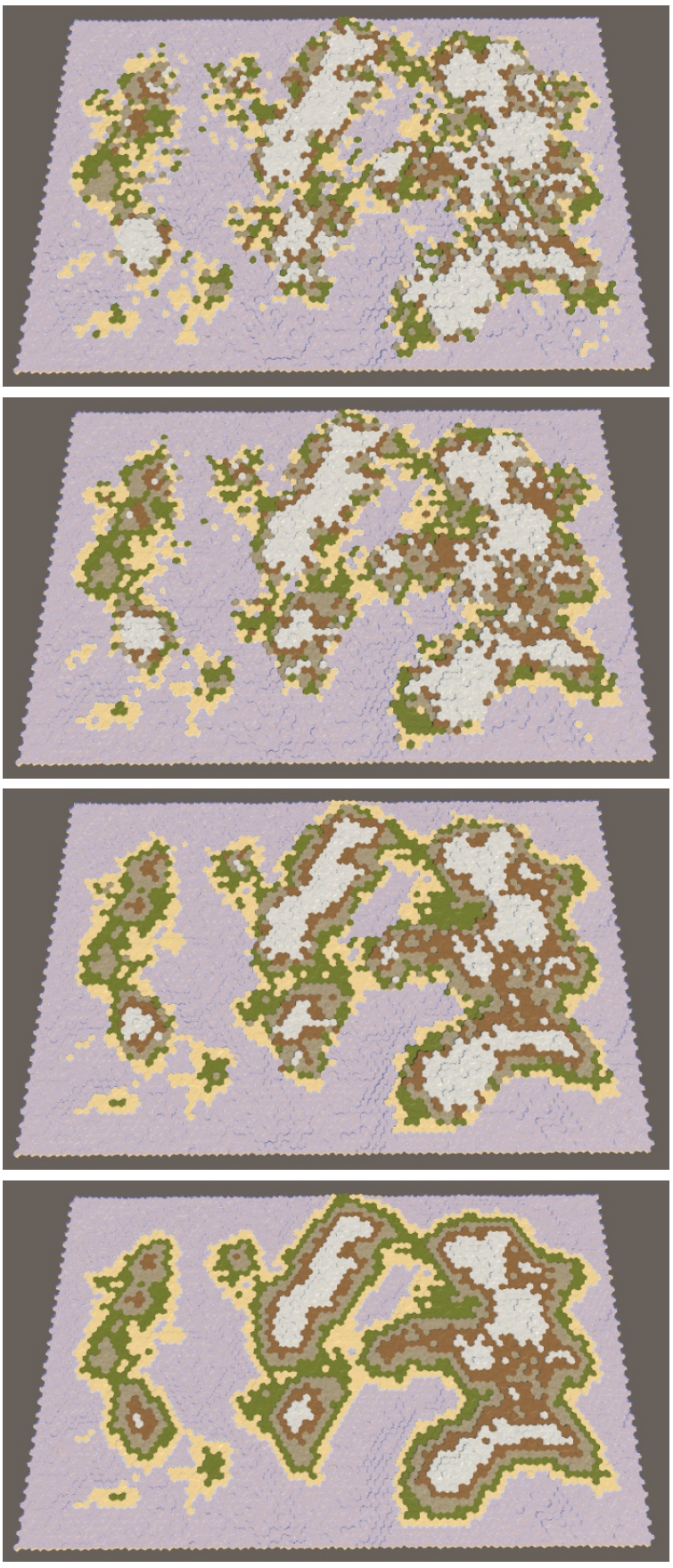
Third, we can use a similar trick when checking the neighbors of the erodible cell. If there's now a cliff between them, then the neighbor is erodible. We don't need to invoke `IsErodible` to find this out.

```
				HexCell neighbor = cell.GetNeighbor(d);
				if (
					neighbor && neighbor.Elevation == cell.Elevation + 2 &&
//					IsErodible(neighbor) &&
					!erodibleCells.Contains(neighbor)
				) {
					erodibleCells.Add(neighbor);
				}
```

However, we do still have to check whether the target cell is erodible, but the above loop now no longer takes care of that. So do this explicitly for the target cell.

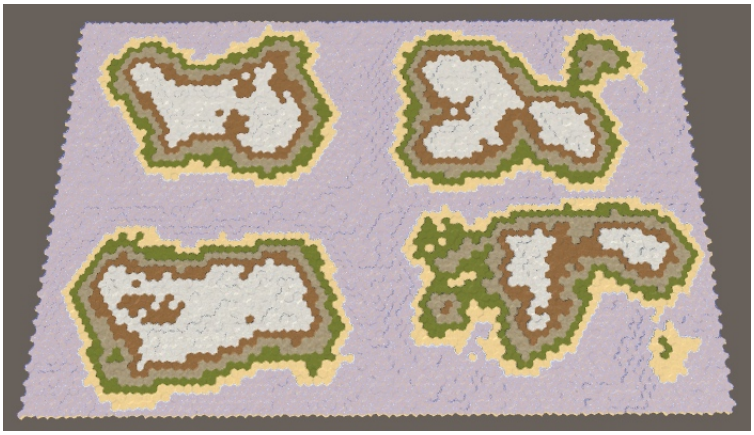
```
		if (!IsErodible(cell)) {
			erodibleCells[index] = erodibleCells[erodibleCells.Count - 1];
			erodibleCells.RemoveAt(erodibleCells.Count - 1);
		}

		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
			…
		}

		if (IsErodible(targetCell) && !erodibleCells.Contains(targetCell)) {
			erodibleCells.Add(targetCell);
		}

		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
			…
		}
```

We can now apply erosion fairly quickly, up to the percentage that we want, relative to the initial amount of cliffs that were generated. Note that because we slightly changed the point at which the target cell gets added to the erodible list, the final result will have changed a bit compared to before our optimizations.



25%, 50%, 75%, and 100% erosion.

Also note that although the coastlines change shape, the topology doesn't get fundamentally altered. Landmasses tend to remain either connected or separated. Only tiny islands could sink entirely. The details are smoothed out, but the overall shapes remain the same. A narrow connection might disappear, but it could also grow a bit. A narrow gaps might fill up, or it might widen a little. So erosion won't dramatically glue distant regions together.



Four fully-eroded regions, still separate.

The next tutorial is Water Cycle.

made by Jasper Flick