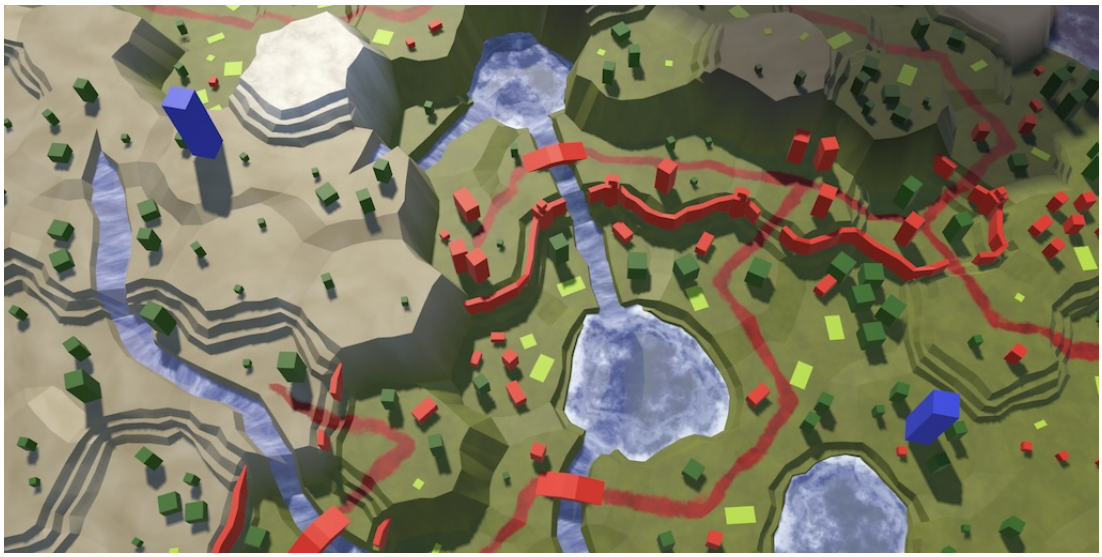
published 2024-04-09

Hex Map 3.3.0 Hex Cell Data



Generating and triangulating a map, without using cell objects.

This tutorial is made with Unity 2022.3.22f1 and follows Hex Map 3.2.0.

1 Cell Data

We're on a quest to get rid of the `HexCell` class. This will allows us to at some point convert some of our code into Burst jobs. This time we remove the dependency on the class from code that generates and visualizes maps.

1.1 Hex Cell Data

We're going to bundle most of the cell data in a new struct type, named `HexCellData`. Give it public fields for `HexFlags`, `HexValues`, and `HexCoordinates`. This data is used together when generating and visualizing maps. Thus the size of this struct is equal to four 32-bit values, as the coordinates are stored as two integers, so sixteen bytes in total.

To simplify conversion from `HexCell` to `HexCellData` we include some getter properties and methods from the class, all readonly, as listed below.

```
[System.Serializable]
public struct HexCellData
{
	public HexFlags flags;

	public HexValues values;

	public HexCoordinates coordinates;

	public readonly int Elevation => values.Elevation;

	public readonly int WaterLevel => values.WaterLevel;

	public readonly int TerrainTypeIndex => values.TerrainTypeIndex;

	public readonly int UrbanLevel => values.UrbanLevel;

	public readonly int FarmLevel => values.FarmLevel;

	public readonly int PlantLevel => values.PlantLevel;

	public readonly int SpecialIndex => values.SpecialIndex;

	public readonly bool Walled => flags.HasAny(HexFlags.Walled);

	public readonly bool HasRoads => flags.HasAny(HexFlags.Roads);

	public readonly bool IsExplored =>
		flags.HasAll(HexFlags.Explored | HexFlags.Explorable);

	public readonly bool IsSpecial => values.SpecialIndex > 0;

	public readonly bool IsUnderwater => values.WaterLevel > values.Elevation;

	public readonly bool HasIncomingRiver => flags.HasAny(HexFlags.RiverIn);

	public readonly bool HasOutgoingRiver => flags.HasAny(HexFlags.RiverOut);
```

```
	public readonly bool HasRiver => flags.HasAny(HexFlags.River);

	public readonly bool HasRiverBeginOrEnd =>
		HasIncomingRiver != HasOutgoingRiver;

	public readonly HexDirection IncomingRiver => flags.RiverInDirection();

	public readonly HexDirection OutgoingRiver => flags.RiverOutDirection();

	public readonly float StreamBedY =>
		(values.Elevation + HexMetrics.streamBedElevationOffset) *
		HexMetrics.elevationStep;

	public readonly float RiverSurfaceY =>
		(values.Elevation + HexMetrics.waterElevationOffset) *
		HexMetrics.elevationStep;

	public readonly float WaterSurfaceY =>
		(values.WaterLevel + HexMetrics.waterElevationOffset) *
		HexMetrics.elevationStep;

	public readonly int ViewElevation =>
		Elevation >= WaterLevel ? Elevation : WaterLevel;

	public readonly HexEdgeType GetEdgeType(HexCellData otherCell) =>
		HexMetrics.GetEdgeType(values.Elevation, otherCell.values.Elevation);

	public readonly bool HasIncomingRiverThroughEdge(HexDirection direction) =>
		flags.HasRiverIn(direction);

	public readonly bool HasRiverThroughEdge(HexDirection direction) =>
		flags.HasRiverIn(direction) || flags.HasRiverOut(direction);

	public readonly bool HasRoadThroughEdge(HexDirection direction) =>
		flags.HasRoad(direction);
}
```

1.2 Moving Data to Arrays

The data that we defined in `HexCellData` will no longer be stored in `HexCell`. We instead move this data to a publicly accessible `CellData` array in `HexGrid`. While we're at it, let's also introduce a publicly accessible `CellPositions` array as well, to separately store the cell positions, which are less frequently accessed.

```
	public HexCellData[] CellData
	{ get; private set; }

	public Vector3[] CellPositions
	{ get; private set; }

	…

	void CreateCells()
	{
		cells = new HexCell[CellCountZ * CellCountX];
		CellData = new HexCellData[cells.Length];
		CellPositions = new Vector3[cells.Length];
		…
	}

	…

	void CreateCell(int x, int z, int i)
	{
		…

		var cell = cells[i] = new HexCell();
		cell.Grid = this;
		CellPositions[i] = position;
		CellData[i].coordinates = HexCoordinates.FromOffsetCoordinates(x, z);
		…
	}
```

Refactor the `flags` and `values` fields of `HexCell` into properties that act as proxies for the data stored in the grid's array.

```
	HexFlags Flags
	{
		get => Grid.CellData[Index].flags;
		set => Grid.CellData[Index].flags = value;
	}

	HexValues Values
	{
		get => Grid.CellData[Index].values;
		set => Grid.CellData[Index].values = value;
	}
```

Do the same for `Coordinates` and `Position`, making them readonly properties. The only place where the position is updated can directly set the position array's data.

```
	public HexCoordinates Coordinates => Grid.CellData[Index].coordinates;

	…

	public Vector3 Position => Grid.CellPositions[Index];

	…

	void RefreshPosition()
	{
		…
		Grid.CellPositions[Index] = position;

		…
	}
```

2 Switching from Class to Struct

We're not going to eliminate all usage of `HexCell` in this tutorial, we limit ourselves to the map generation and visualization code. This includes `HexCellShaderData`, `HexFeatureManager`, `HexGridChunk`, and `HexMapGenerator`, plus whatever supporting code needs to change.

2.1 Hex Cell Shader Data

We begin with `HexCellShaderData`. It has three methods that have a `HexCell` parameter. We'll change those parameters into cell indices and retrieve the required `HexCellData` in the methods. First is `RefreshTerrain`.

```
	public void RefreshTerrain(int cellIndex)
	{
		HexCellData cell = Grid.CellData[cellIndex];
		Color32 data = cellTextureData[cellIndex];
		data.b = cell.IsUnderwater ?
			(byte)(cell.WaterSurfaceY * (255f / 30f)) : (byte)0;
		data.a = (byte)cell.TerrainTypeIndex;
		cellTextureData[cellIndex] = data;
		enabled = true;
	}
```

Followed by `RefreshVisibility`, which only needs to access the cell data in one place so we don't needs to store it in a variable.

```
	public void RefreshVisibility(int cellIndex)
	{
		//int index = cell.Index;
		if (ImmediateMode)
		{
			cellTextureData[cellIndex].r = Grid.IsCellVisible(cellIndex) ?
				(byte)255 : (byte)0;
			cellTextureData[cellIndex].g = Grid.CellData[cellIndex].IsExplored ?
				(byte)255 : (byte)0;
		}
		else if (!visibilityTransitions[cellIndex])
		{
			visibilityTransitions[cellIndex] = true;
			transitioningCellIndices.Add(cellIndex);
		}
		enabled = true;
	}
```

Third is `ViewElevationChanged`.

```
	public void ViewElevationChanged(int cellIndex)
	{
		HexCellData cell = Grid.CellData[cellIndex];
		cellTextureData[cellIndex].b = cell.IsUnderwater ?
			(byte)(cell.WaterSurfaceY * (255f / 30f)) : (byte)0;
		needsVisibilityReset = true;
		enabled = true;
	}
```

We also have to change `UpdateCellData` so it retrieves **`HexCellData`**. After that change **`HexCellShaderData`** no longer relies on **`HexCell`**.

```
	bool UpdateCellData(int index, int delta)
	{
		//HexCell cell = Grid.GetCell(index);
		Color32 data = cellTextureData[index];
		bool stillUpdating = false;

		if (Grid.CellData[index].IsExplored && data.g < 255)
		{
			…
		}

		if (Grid.IsCellVisible(index))
		{
			…
		}
		…
	}
```

Now we have to fix the invocation of these methods in **`HexCell`**, by replacing the **`this`** argument with `Index` everywhere they're used. We also have to fix the methods that increase, decrease, and reset visibility in **`HexGrid`**.

```
	public void IncreaseVisibility(HexCell fromCell, int range)
	{
		List<HexCell> cells = GetVisibleCells(fromCell, range);
		for (int i = 0; i < cells.Count; i++)
		{
			int cellIndex = cells[i].Index;
			if (++cellVisibility[cellIndex] == 1)
			{
				cells[i].MarkAsExplored();
				cellShaderData.RefreshVisibility(cellIndex);
			}
		}
		ListPool<HexCell>.Add(cells);
	}

	public void DecreaseVisibility(HexCell fromCell, int range)
	{
		List<HexCell> cells = GetVisibleCells(fromCell, range);
		for (int i = 0; i < cells.Count; i++)
		{
			int cellIndex = cells[i].Index;
			if (--cellVisibility[cellIndex] == 0)
			{
				cellShaderData.RefreshVisibility(cellIndex);
			}
		}
		ListPool<HexCell>.Add(cells);
	}

	public void ResetVisibility()
	{
		for (int i = 0; i < cells.Length; i++)
		{
			if (cellVisibility[i] > 0)
			{
				cellVisibility[i] = 0;
				cellShaderData.RefreshVisibility(i);
			}
		}
		…
	}
```

2.2 Triangulating

We're going to make the code that triangulates the map rely on `HexCellData` instead of `HexCell`. First, we need to know a cell's column index in some cases when triangulating water. We don't store that index in the cell data and we don't have to, because we can derive it from the cell coordinates. Add a `ColumnIndex` property to `HexCoordinates` for that.

```
	public readonly int ColumnIndex => (x + z / 2) / HexMetrics.chunkSizeX;
```

Second, in `HexFeatureManager` we can simply replace all `HexCell` parameter types with `HexCellData`, without having to change any other code in that class.

Third, we're going to directly work with cell coordinates instead of cells, so we refactor `HexGrid.TryGetCell` to get a cell index instead. If no index is found it should be set to −1.

```
	public bool TryGetCellIndex(HexCoordinates coordinates, out int cellIndex)
	{
		int z = coordinates.Z;
		int x = coordinates.X + z / 2;
		if (z < 0 || z >= CellCountZ || x < 0 || x >= CellCountX)
		{
			cellIndex = -1;
			return false;
		}
		cellIndex = x + z * CellCountX;
		return true;
	}
```

Now we move on to `HexGridChunk`, which does the triangulation. We'll have to adjust a lot of methods and we'll work through them from top to bottom.

The starting point is `Triangulate` with a `HexCell` parameter, which we replace with a cell index parameter. This requires us to retrieve the cell data, which replaces the old cell reference, and the cell position. We then pass the index and position as extra arguments to the `Triangulate` variant with a direction.

```
	public void Triangulate()
	{
		…
		for (int i = 0; i < cellIndices.Length; i++)
		{
			Triangulate(cellIndices[i]);
		}
		…
	}

	void Triangulate(int cellIndex)
	{
		HexCellData cell = Grid.CellData[cellIndex];
		Vector3 cellPosition = Grid.CellPositions[cellIndex];
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
		{
			Triangulate(d, cell, cellIndex, cellPosition);
		}
		if (!cell.IsUnderwater)
		{
			if (!cell.HasRiver && !cell.HasRoads)
			{
				features.AddFeature(cell, cellPosition);
			}
			if (cell.IsSpecial)
			{
				features.AddSpecialFeature(cell, cellPosition);
			}
		}
	}
```

Adjust the parameter list of the direction-based `Triangulate` method. We're going to also pass the cell index as an extra argument to all other triangulation methods that we invoke here. In case of `TriangulateConnection` we also pass along the Y coordinate of the center position.

```
	void Triangulate(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center)
	{
		var e = new EdgeVertices(
			center + HexMetrics.GetFirstSolidCorner(direction),
			center + HexMetrics.GetSecondSolidCorner(direction));

		if (cell.HasRiver)
		{
			if (cell.HasRiverThroughEdge(direction))
			{
				e.v3.y = cell.StreamBedY;
				if (cell.HasRiverBeginOrEnd)
				{
					TriangulateWithRiverBeginOrEnd(cell, cellIndex, center, e);
				}
				else
				{
					TriangulateWithRiver(direction, cell, cellIndex, center, e);
				}
			}
			else
			{
				TriangulateAdjacentToRiver(
					direction, cell, cellIndex, center, e);
			}
		}
		else
		{
			TriangulateWithoutRiver(direction, cell, cellIndex, center, e);
			if (!cell.IsUnderwater && !cell.HasRoadThroughEdge(direction))
			{
				features.AddFeature(
					cell, (center + e.v1 + e.v5) * (1f / 3f));
			}
		}

		if (direction <= HexDirection.SE)
		{
			TriangulateConnection(direction, cell, cellIndex, center.y, e);
		}

		if (cell.IsUnderwater)
		{
			TriangulateWater(direction, cell, cellIndex, center);
		}
	}
```

Going deeper, we move to `TriangulateWater` first. Adjust its parameter list as needed, then have it fetch the neighbor coordinates and use those to check whether there is a neighbor that isn't underwater. If so, when triangulating the water shore, also pass along the cell index and the neighbor's column index. Otherwise, when triangulating open water, provide the cell coordinates, cell index, and neighbor index as extra arguments.

```
	void TriangulateWater(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center)
	{
		center.y = cell.WaterSurfaceY;
		HexCoordinates neighborCoordinates = cell.coordinates.Step(direction);
		if (Grid.TryGetCellIndex(neighborCoordinates, out int neighborIndex) &&
			!Grid.CellData[neighborIndex].IsUnderwater)
		{
			TriangulateWaterShore(
				direction, cell, cellIndex, neighborIndex,
				neighborCoordinates.ColumnIndex, center);
		}
		else
		{
			TriangulateOpenWater(
				cell.coordinates, direction, cellIndex, neighborIndex, center);
		}
	}
```

We adjust `TriangulateOpenWater` next. Again change its parameter list and use the new data. We now check for the neighbor's existence by comparing its index with −1. Also, accessing next neighbor becomes a bit more verbose, having to explicitly get the coordinates for stepping in the direction for the next neighbor.

```
	void TriangulateOpenWater(
		HexCoordinates coordinates,
		HexDirection direction,
		int cellIndex,
		int neighborIndex,
		Vector3 center)
	{
		…
		indices.x = indices.y = indices.z = cellIndex;
		water.AddTriangleCellData(indices, weights1);

		if (direction <= HexDirection.SE && neighborIndex != -1)
		{
			…
			indices.y = neighborIndex;
			water.AddQuadCellData(indices, weights1, weights2);

			if (direction <= HexDirection.E)
			{
				if (!Grid.TryGetCellIndex(
					coordinates.Step(direction.Next()),
					out int nextNeighborIndex) ||
					!Grid.CellData[nextNeighborIndex].IsUnderwater)
				{
					return;
				}
				water.AddTriangle(
					c2, e2, c2 + HexMetrics.GetWaterBridge(direction.Next()));
				indices.z = nextNeighborIndex;
				water.AddTriangleCellData(
					indices, weights1, weights2, weights3);
			}
		}
	}
```

`TriangulateWaterShore` requires more changes, but they are of a similar nature. We access the same data in multiple places here, and as that data has to be extracted or calculated now let's store it in temporary variables.

```
	void TriangulateWaterShore(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		int neighborIndex,
		int neighborColumnIndex,
		Vector3 center)
	{
		…
		indices.x = indices.z = cellIndex;
		indices.y = neighborIndex;
		…

		Vector3 center2 = Grid.CellPositions[neighborIndex];
		int cellColumnIndex = cell.coordinates.ColumnIndex;
		if (neighborColumnIndex < cellColumnIndex - 1)
		{
			center2.x += HexMetrics.wrapSize * HexMetrics.innerDiameter;
		}
		else if (neighborColumnIndex > cellColumnIndex + 1)
		{
			center2.x -= HexMetrics.wrapSize * HexMetrics.innerDiameter;
		}
		…

		HexCoordinates nextNeighborCoordinates = cell.coordinates.Step(
			direction.Next());
		if (Grid.TryGetCellIndex(
			nextNeighborCoordinates, out int nextNeighborIndex))
		{
			Vector3 center3 = Grid.CellPositions[nextNeighborIndex];
			bool nextNeighborIsUnderwater =
				Grid.CellData[nextNeighborIndex].IsUnderwater;
			int nextNeighborColumnIndex = nextNeighborCoordinates.ColumnIndex;
			if (nextNeighborColumnIndex < cellColumnIndex - 1)
			{
				center3.x += HexMetrics.wrapSize * HexMetrics.innerDiameter;
			}
			else if (nextNeighborColumnIndex > cellColumnIndex + 1)
			{
				center3.x -= HexMetrics.wrapSize * HexMetrics.innerDiameter;
			}
			Vector3 v3 = center3 + (nextNeighborIsUnderwater ?
				HexMetrics.GetFirstWaterCorner(direction.Previous()) :
				HexMetrics.GetFirstSolidCorner(direction.Previous()));
			v3.y = center.y;
			waterShore.AddTriangle(e1.v5, e2.v5, v3);
			waterShore.AddTriangleUV(
				new Vector2(0f, 0f),
				new Vector2(0f, 1f),
				new Vector2(0f, nextNeighborIsUnderwater ? 0f : 1f));
			indices.z = nextNeighborIndex;
			waterShore.AddTriangleCellData(
				indices, weights1, weights2, weights3);
		}
	}
```

`TriangulateWithoutRiver` requires minimal changes, replacing **`HexCell`** with **`HexCellData`** and adding a new cell index parameter. The `GetRoadInterpolators` method only needs its cell parameter type changed to **`HexCellData`**.

```
	void TriangulateWithoutRiver(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center,
		EdgeVertices e)
	{
		TriangulateEdgeFan(center, e, cellIndex);

		if (cell.HasRoads)
		{
			Vector2 interpolators = GetRoadInterpolators(direction, cell);
			TriangulateRoad(
				center,
				Vector3.Lerp(center, e.v1, interpolators.x),
				Vector3.Lerp(center, e.v5, interpolators.y),
				e, cell.HasRoadThroughEdge(direction), cellIndex);
		}
	}

	Vector2 GetRoadInterpolators(HexDirection direction, HexCellData cell) { … }
```

`TriangulateAdjacentToRiver` also only requires index-related changes, besides changing the cell parameter type to `HexCellData`.

```
	void TriangulateAdjacentToRiver(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center,
		EdgeVertices e)
	{
		if (cell.HasRoads)
		{
			TriangulateRoadAdjacentToRiver(
				direction, cell, cellIndex, center, e);
		}

		…

		TriangulateEdgeStrip(
			m, weights1, cellIndex,
			e, weights1, cellIndex);
		TriangulateEdgeFan(center, m, cellIndex);

		…
	}
```

The same goes for `TriangulateRoadAdjacentToRiver`.

```
	void TriangulateRoadAdjacentToRiver(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center,
		EdgeVertices e)
	{
		…
		TriangulateRoad(roadCenter, mL, mR, e, hasRoadThroughEdge, cellIndex);
		if (previousHasRiver)
		{
			TriangulateRoadEdge(roadCenter, center, mL, cellIndex);
		}
		if (nextHasRiver)
		{
			TriangulateRoadEdge(roadCenter, mR, center, cellIndex);
		}
	}
```

And for `TriangulateWithRiverBeginOrEnd`.

```
	void TriangulateWithRiverBeginOrEnd(
		HexCellData cell, int cellIndex, Vector3 center, EdgeVertices e)
	{
		…

		TriangulateEdgeStrip(
			m, weights1, cellIndex,
			e, weights1, cellIndex);
		TriangulateEdgeFan(center, m, cellIndex);

		if (!cell.IsUnderwater)
		{
			bool reversed = cell.HasIncomingRiver;
			Vector3 indices;
			indices.x = indices.y = indices.z = cellIndex;
			…
		}
	}
```

And `TriangulateWithRiver`.

```
	void TriangulateWithRiver(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		Vector3 center,
		EdgeVertices e)
	{
		…

		TriangulateEdgeStrip(
			m, weights1, cellIndex,
			e, weights1, cellIndex);

		…

		Vector3 indices;
		indices.x = indices.y = indices.z = cellIndex;
		…
	}
```

`TriangulateConnection` is more complicated. It gains a parameter for the center Y coordinate. For each corner we have to pass both the appropriate cell index and data when invoking `TriangulateCorner`, and just the indices when invoking `TriangulateEdgeTerraces`.

```
	void TriangulateConnection(
		HexDirection direction,
		HexCellData cell,
		int cellIndex,
		float centerY,
		EdgeVertices e1)
	{
		if (!Grid.TryGetCellIndex(
			cell.coordinates.Step(direction), out int neighborIndex))
		{
			return;
		}
		HexCellData neighbor = Grid.CellData[neighborIndex];
		Vector3 bridge = HexMetrics.GetBridge(direction);
		bridge.y = Grid.CellPositions[neighborIndex].y - centerY;
		…

		if (hasRiver)
		{
			e2.v3.y = neighbor.StreamBedY;
			Vector3 indices;
			indices.x = indices.z = cellIndex;
			indices.y = neighborIndex;

			…
		}

		if (cell.GetEdgeType(neighbor) == HexEdgeType.Slope)
		{
			TriangulateEdgeTerraces(e1, cellIndex, e2, neighborIndex, hasRoad);
		}
		else
		{
			TriangulateEdgeStrip(
```

```
                e1, weights1, cellIndex,
                e2, weights2, neighborIndex, hasRoad);
        }

        features.AddWall(e1, cell, e2, neighbor, hasRiver, hasRoad);

        if (direction <= HexDirection.E &&
            Grid.TryGetCellIndex(
                cell.coordinates.Step(direction.Next()),
                out int nextNeighborIndex))
        {
            HexCellData nextNeighbor = Grid.CellData[nextNeighborIndex];
            Vector3 v5 = e1.v5 + HexMetrics.GetBridge(direction.Next());
            v5.y = Grid.CellPositions[nextNeighborIndex].y;

            if (cell.Elevation <= neighbor.Elevation)
            {
                if (cell.Elevation <= nextNeighbor.Elevation)
                {
                    TriangulateCorner(
                        e1.v5, cellIndex, cell,
                        e2.v5, neighborIndex, neighbor,
                        v5, nextNeighborIndex, nextNeighbor);
                }
                else
                {
                    TriangulateCorner(
                        v5, nextNeighborIndex, nextNeighbor,
                        e1.v5, cellIndex, cell,
                        e2.v5, neighborIndex, neighbor);
                }
            }
            else if (neighbor.Elevation <= nextNeighbor.Elevation)
            {
                TriangulateCorner(
                    e2.v5, neighborIndex, neighbor,
                    v5, nextNeighborIndex, nextNeighbor,
                    e1.v5, cellIndex, cell);
            }
            else {
                TriangulateCorner(
                    v5, nextNeighborIndex, nextNeighbor,
                    e1.v5, cellIndex, cell,
                    e2.v5, neighborIndex, neighbor);
            }
        }
    }
```

Adjust the parameter list of `TriangulateCorner` to match. Pass the indices to `TriangulateCornerTerraces` instead of cells, and do that in addition to the cell data for the other triangulate method invocations.

```
    void TriangulateCorner(
        Vector3 bottom, int bottomCellIndex, HexCellData bottomCell,
        Vector3 left, int leftCellIndex, HexCellData leftCell,
        Vector3 right, int rightCellIndex, HexCellData rightCell)
    {
        …

        if (leftEdgeType == HexEdgeType.Slope)
        {
```

```
            if (rightEdgeType == HexEdgeType.Slope)
            {
                TriangulateCornerTerraces(
                    bottom, bottomCellIndex,
                    left, leftCellIndex,
                    right, rightCellIndex);
            }
            else if (rightEdgeType == HexEdgeType.Flat)
            {
                TriangulateCornerTerraces(
                    left, leftCellIndex,
                    right, rightCellIndex,
                    bottom, bottomCellIndex);
            }
            else
            {
                TriangulateCornerTerracesCliff(
                    bottom, bottomCellIndex, bottomCell,
                    left, leftCellIndex, leftCell,
                    right, rightCellIndex, rightCell);
            }
        }
        else if (rightEdgeType == HexEdgeType.Slope)
        {
            if (leftEdgeType == HexEdgeType.Flat)
            {
                TriangulateCornerTerraces(
                    right, rightCellIndex,
                    bottom, bottomCellIndex,
                    left, leftCellIndex);
            }
            else
            {
                TriangulateCornerCliffTerraces(
                    bottom, bottomCellIndex, bottomCell,
                    left, leftCellIndex, leftCell,
                    right, rightCellIndex, rightCell);
            }
        }
        else if (leftCell.GetEdgeType(rightCell) == HexEdgeType.Slope)
        {
            if (leftCell.Elevation < rightCell.Elevation)
            {
                TriangulateCornerCliffTerraces(
                    right, rightCellIndex, rightCell,
                    bottom, bottomCellIndex, bottomCell,
                    left, leftCellIndex, leftCell);
            }
            else
            {
                TriangulateCornerTerracesCliff(
                    left, leftCellIndex, leftCell,
                    right, rightCellIndex, rightCell,
                    bottom, bottomCellIndex, bottomCell);
            }
        }
        else
        {
            terrain.AddTriangle(bottom, left, right);
            Vector3 indices;
            indices.x = bottomCellIndex;
            indices.y = leftCellIndex;
            indices.z = rightCellIndex;
            terrain.AddTriangleCellData(indices, weights1, weights2, weights3);
        }
```

```
		features.AddWall(
			bottom, bottomCell, left, leftCell, right, rightCell);
	}
```

`TriangulateEdgeTerraces` now works directly with cell indices.

```
	void TriangulateEdgeTerraces(
		EdgeVertices begin, int beginCellIndex,
		EdgeVertices end, int endCellIndex,
		bool hasRoad)
	{
		…
		float i1 = beginCellIndex;
		float i2 = endCellIndex;

		…
	}
```

So does `TriangulateCornerTerraces`.

```
	void TriangulateCornerTerraces(
		Vector3 begin, int beginCellIndex,
		Vector3 left, int leftCellIndex,
		Vector3 right, int rightCellIndex)
	{
		…
		indices.x = beginCellIndex;
		indices.y = leftCellIndex;
		indices.z = rightCellIndex;

		…
	}
```

The same goes for `TriangulateCornerTerracesCliff`, though it needs the cell data as well.

```
	void TriangulateCornerTerracesCliff(
		Vector3 begin, int beginCellIndex, HexCellData beginCell,
		Vector3 left, int leftCellIndex, HexCellData leftCell,
		Vector3 right, int rightCellIndex, HexCellData rightCell)
	{
		…
		indices.x = beginCellIndex;
		indices.y = leftCellIndex;
		indices.z = rightCellIndex;

		…
	}
```

Which is also the case for `TriangulateCornerCliffTerraces`.

```
	void TriangulateCornerCliffTerraces(
		Vector3 begin, int beginCellIndex, HexCellData beginCell,
		Vector3 left, int leftCellIndex, HexCellData leftCell,
		Vector3 right, int rightCellIndex, HexCellData rightCell)
	{
		…
		indices.x = beginCellIndex;
		indices.y = leftCellIndex;
		indices.z = rightCellIndex;

		…
	}
```

That covers all triangulation code. To completely remove all dependencies on `HexCell` from `HexGridChunk` let's also adjust `AddCell`. Replace its `HexCell` parameter with both a cell index and a `RectTransform` reference for the cell's UI. Then no longer set the cell's chunk here.

```
	public void AddCell(int index, int cellIndex, RectTransform cellUI)
	{
		cellIndices[index] = cellIndex;
		//cell.Chunk = this;
		cellUI.SetParent(gridCanvas.transform, false);
	}
```

Adjust `HexGrid.AddCellToChunk` so the cell's chunk is set here and the correct arguments are passed to `AddCell`.

```
	void AddCellToChunk(int x, int z, HexCell cell)
	{
		…
		cell.Chunk = chunk;
		chunk.AddCell(
			localX + localZ * HexMetrics.chunkSizeX, cell.Index, cell.UIRect);
	}
```

2.3 Hex Map Generator

The last class that we adapt is `HexMapGenerator`. We begin by refactoring `HexGrid.GetCell(int xOffset, int zOffset)` so it returns a cell index. This method is only used by the generator.

```
	public int GetCellIndex(int xOffset, int zOffset) =>
		xOffset + zOffset * CellCountX;
```

Then we can refactor `HexMapGenerator.GetCell` the same way.

```
	int GetRandomCellIndex (MapRegion region) => grid.GetCellIndex(
		Random.Range(region.xMin, region.xMax),
		Random.Range(region.zMin, region.zMax));
```

Also, because from now on we're directly setting cell data and not going through `HexCell` the cell and its visualization will no longer automatically refresh itself. To work around that we introduce a public `HexCell.RefreshAll` method that performs a full refresh.

```
	public void RefreshAll()
	{
		RefreshPosition();
		Grid.ShaderData.RefreshTerrain(Index);
		Grid.ShaderData.RefreshVisibility(Index);
	}
```

Change `HexMapGenerator.GenerateMap` so it modifies the cell data directly when setting the water level, and invokes `RefreshAll` on all cells after the map has been generated. This is the only place where it still relies on `HexCell`.

```
	public void GenerateMap(int x, int z, bool wrapping)
	{
		…
		for (int i = 0; i < cellCount; i++)
		{
			grid.CellData[i].values = grid.CellData[i].values.WithWaterLevel(
				waterLevel);
		}
		…
		for (int i = 0; i < cellCount; i++)
		{
			grid.SearchData[i].searchPhase = 0;
			grid.GetCell(i).RefreshAll();
		}

		Random.state = originalRandomState;
	}
```

Moving on to the generating code, we begin with `RaiseTerrain`. Change it so it works with cell indices, retrieves cell data, and directly adjusts the elevation in the cell data array.

```
	int RaiseTerrain(int chunkSize, int budget, MapRegion region)
	{
		searchFrontierPhase += 1;
		int firstCellIndex = GetRandomCellIndex(region);
		grid.SearchData[firstCellIndex] = new HexCellSearchData
		{
			searchPhase = searchFrontierPhase
		};
		searchFrontier.Enqueue(firstCellIndex);
		HexCoordinates center = grid.CellData[firstCellIndex].coordinates;

		…
		while (size < chunkSize && searchFrontier.TryDequeue(out int index))
		{
			HexCellData current = grid.CellData[index];
			…
			grid.CellData[index].values =
				current.values.WithElevation(newElevation);
			…

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				if (grid.TryGetCellIndex(
					current.coordinates.Step(d), out int neighborIndex) &&
					grid.SearchData[neighborIndex].searchPhase <
						searchFrontierPhase)
				{
					grid.SearchData[neighborIndex] = new HexCellSearchData
					{
						searchPhase = searchFrontierPhase,
						distance = grid.CellData[neighborIndex].coordinates.
							DistanceTo(center),
						heuristic = Random.value < jitterProbability ? 1 : 0
					};
					searchFrontier.Enqueue(neighborIndex);
				}
			}
		}
		…
	}
```

`SinkTerrain` is nearly identical and should be changed in the same way.

Next up is `ErodeLand`, which requires quite a few changes but they're similar to those for `RaiseTerrain`. Besides that we switch from working with a list of cells to a list of cell indices and the erosion target also becomes an index. We now pass a cell index and cell elevation to `GetErosionTarget` and `IsErodible`. Also, the code for incrementing and decrementing elevation has to become more verbose.

```
	void ErodeLand()
	{
		List<int> erodibleIndices = ListPool<int>.Get();
		for (int i = 0; i < cellCount; i++)
		{
			//HexCell cell = grid.GetCell(i);
			if (IsErodible(i, grid.CellData[i].Elevation))
			{
				erodibleIndices.Add(i);
```

```
            }
        }

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

        while (erodibleIndices.Count > targetErodibleCount)
        {
            int index = Random.Range(0, erodibleIndices.Count);
            int cellIndex = erodibleIndices[index];
            HexCellData cell = grid.CellData[cellIndex];
            int targetCellIndex = GetErosionTarget(cellIndex, cell.Elevation);

            //cell.Elevation -= 1;
            grid.CellData[cellIndex].values = cell.values =
                cell.values.WithElevation(cell.Elevation - 1);

            //targetCell.Elevation += 1;
            HexCellData targetCell = grid.CellData[targetCellIndex];
            grid.CellData[targetCellIndex].values = targetCell.values =
                targetCell.values.WithElevation(targetCell.Elevation + 1);

            if (!IsErodible(cellIndex, cell.Elevation))
            {
                int lastIndex = erodibleIndices.Count - 1;
                erodibleIndices[index] = erodibleIndices[lastIndex];
                erodibleIndices.RemoveAt(lastIndex);
            }

            for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
            {
                if (grid.TryGetCellIndex(
                    cell.coordinates.Step(d), out int neighborIndex) &&
                    grid.CellData[neighborIndex].Elevation ==
                        cell.Elevation + 2 &&
                    !erodibleIndices.Contains(neighborIndex))
                {
                    erodibleIndices.Add(neighborIndex);
                }
            }

            if (IsErodible(targetCellIndex, targetCell.Elevation) &&
                !erodibleIndices.Contains(targetCellIndex))
            {
                erodibleIndices.Add(targetCellIndex);
            }

            for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
            {
                if (grid.TryGetCellIndex(
                    targetCell.coordinates.Step(d), out int neighborIndex) &&
                    neighborIndex != cellIndex &&
                    grid.CellData[neighborIndex].Elevation ==
                        targetCell.Elevation + 1 &&
                    !IsErodible(
                        neighborIndex, grid.CellData[neighborIndex].Elevation))
                {
                    erodibleIndices.Remove(neighborIndex);
                }
            }
        }

        ListPool<int>.Add(erodibleIndices);
    }
```

Change the parameter list of `IsErodible` to match, then use the cell index to retrieve the cell coordinates used for looping through its neighbors.

```
	bool IsErodible(int cellIndex, int cellElevation)
	{
		int erodibleElevation = cellElevation - 2;
		HexCoordinates coordinates = grid.CellData[cellIndex].coordinates;
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
		{
			if (grid.TryGetCellIndex(
				coordinates.Step(d), out int neighborIndex) &&
				grid.CellData[neighborIndex].Elevation <= erodibleElevation)
			{
				return true;
			}
		}
		return false;
	}
```

Change `GetErosionTarget` likewise, now returning a cell index.

```
	int GetErosionTarget (int cellIndex, int cellElevation)
	{
		List<int> candidates = ListPool<int>.Get();
		int erodibleElevation = cellElevation - 2;
		HexCoordinates coordinates = grid.CellData[cellIndex].coordinates;
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
		{
			if (grid.TryGetCellIndex(
				coordinates.Step(d), out int neighborIndex) &&
				grid.CellData[neighborIndex].Elevation <= erodibleElevation
			)
			{
				candidates.Add(neighborIndex);
			}
		}
		int target = candidates[Random.Range(0, candidates.Count)];
		ListPool<int>.Add(candidates);
		return target;
	}
```

Next up is `EvolveClimate`, which should also switch to using cell data and indices.

```
	void EvolveClimate(int cellIndex)
	{
		HexCellData cell = grid.CellData[cellIndex];
		…
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
		{
			if (!grid.TryGetCellIndex(
				cell.coordinates.Step(d), out int neighborIndex))
			{
				continue;
			}
			ClimateData neighborClimate = nextClimate[neighborIndex];
			…

			int elevationDelta = grid.CellData[neighborIndex].ViewElevation -
				cell.ViewElevation;
			…

			nextClimate[neighborIndex] = neighborClimate;
		}

		…
	}
```

Change `CreateRivers` to work with cell indices as well.

```
	void CreateRivers()
	{
		List<int> riverOrigins = ListPool<int>.Get();
		for (int i = 0; i < cellCount; i++)
		{
			HexCellData cell = grid.CellData[i];
			…
			if (weight > 0.75f)
			{
				riverOrigins.Add(i);
				riverOrigins.Add(i);
			}
			if (weight > 0.5f)
			{
				riverOrigins.Add(i);
			}
			if (weight > 0.25f)
			{
				riverOrigins.Add(i);
			}
		}

		int riverBudget = Mathf.RoundToInt(landCells * riverPercentage * 0.01f);
		while (riverBudget > 0 && riverOrigins.Count > 0)
		{
			int index = Random.Range(0, riverOrigins.Count);
			int lastIndex = riverOrigins.Count - 1;
			int originIndex = riverOrigins[index];
			HexCellData origin = grid.CellData[originIndex];
			riverOrigins[index] = riverOrigins[lastIndex];
			riverOrigins.RemoveAt(lastIndex);

			if (!origin.HasRiver)
			{
				bool isValidOrigin = true;
				for (HexDirection d = HexDirection.NE;
					d <= HexDirection.NW; d++)
				{
					if (grid.TryGetCellIndex(
						origin.coordinates.Step(d), out int neighborIndex) &&
						(grid.CellData[neighborIndex].HasRiver ||
							grid.CellData[neighborIndex].IsUnderwater))
					{
						isValidOrigin = false;
						break;
					}
				}
				if (isValidOrigin)
				{
					riverBudget -= CreateRiver(originIndex);
				}
			}
		}

		…

		ListPool<int>.Add(riverOrigins);
	}
```

And adjust `CreateRiver` to match. Instead of invoking `SetOutgoingRiver` on a cell we now have to explicitly set the flags of both cells correctly. The verification work done by `SetOutgoingRiver` is not needed here, because we only create a river connection if there are no obstructions.

```
	int CreateRiver(int originIndex)
	{
		int length = 1;
		int cellIndex = originIndex;
		HexCellData cell = grid.CellData[cellIndex];
		HexDirection direction = HexDirection.NE;
		while (!cell.IsUnderwater)
		{
			int minNeighborElevation = int.MaxValue;
			flowDirections.Clear();
			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				if (!grid.TryGetCellIndex(
					cell.coordinates.Step(d), out int neighborIndex))
				{
					continue;
				}
				HexCellData neighbor = grid.CellData[neighborIndex];

				…

				if (neighborIndex == originIndex || neighbor.HasIncomingRiver)
				{
					continue;
				}

				…

				if (neighbor.HasOutgoingRiver)
				{
					//cell.SetOutgoingRiver(d);
					grid.CellData[cellIndex].flags = cell.flags.WithRiverOut(d);
					grid.CellData[neighborIndex].flags =
						neighbor.flags.WithRiverIn(d.Opposite());
					return length;
				}

				…
			}

			if (flowDirections.Count == 0)
			{
				…

				if (minNeighborElevation >= cell.Elevation)
				{
					//cell.WaterLevel = minNeighborElevation;
					cell.values = cell.values.WithWaterLevel(
						minNeighborElevation);
					if (minNeighborElevation == cell.Elevation)
					{
						//cell.Elevation = minNeighborElevation - 1;
						cell.values = cell.values.WithElevation(
							minNeighborElevation - 1);
					}
					grid.CellData[cellIndex].values = cell.values;
```

```
                }
                break;
            }

            direction = flowDirections[Random.Range(0, flowDirections.Count)];
            //cell.SetOutgoingRiver(direction);
            cell.flags = cell.flags.WithRiverOut(direction);
            grid.TryGetCellIndex(
                cell.coordinates.Step(direction), out int outIndex);
            grid.CellData[outIndex].flags =
                grid.CellData[outIndex].flags.WithRiverIn(direction.Opposite());

            length += 1;

            if (minNeighborElevation >= cell.Elevation &&
                Random.value < extraLakeProbability)
            {
                //cell.WaterLevel = cell.Elevation;
                //cell.Elevation -= 1;
                cell.values = cell.values.WithWaterLevel(cell.Elevation);
                cell.values = cell.values.WithElevation(cell.Elevation - 1);
            }
            grid.CellData[cellIndex] = cell;
            cellIndex = outIndex;
            cell = grid.CellData[cellIndex];
        }
        return length;
    }
```

The last generation step to convert is `SetTerrainType`. Make it work with cell data and also pass the cell index to `DetermineTemperature`.

```
	void SetTerrainType()
	{
		…

		for (int i = 0; i < cellCount; i++)
		{
			HexCellData cell = grid.CellData[i];
			float temperature = DetermineTemperature(i, cell);
			float moisture = climate[i].moisture;
			if (!cell.IsUnderwater)
			{
				…

				//cell.TerrainTypeIndex = cellBiome.terrain;
				//cell.PlantLevel = cellBiome.plant;
				grid.CellData[i].values = cell.values.
					WithTerrainTypeIndex(cellBiome.terrain).
					WithPlantLevel(cellBiome.plant);
			}
			else
			{
				int terrain;
				if (cell.Elevation == waterLevel - 1)
				{
					int cliffs = 0, slopes = 0;
					for (HexDirection d = HexDirection.NE;
						d <= HexDirection.NW; d++)
					{
						if (!grid.TryGetCellIndex(
							cell.coordinates.Step(d), out int neighborIndex))
						{
							continue;
						}
						int delta = grid.CellData[neighborIndex].Elevation -
							cell.WaterLevel;
						…
					}

					…
				}
				…
				//cell.TerrainTypeIndex = terrain;
				grid.CellData[i].values =
					cell.values.WithTerrainTypeIndex(terrain);
			}
		}
	}
```

`DetermineTemperature` needs the cell index to get its position.

```
	float DetermineTemperature(int cellIndex, HexCellData cell)
	{
		…

		float jitter = HexMetrics.SampleNoise(
			grid.CellPositions[cellIndex] * 0.1f)[temperatureJitterChannel];

		…
	}
```

2.4 Hex Cell Cleanup

We have now removed the reliance on `HexCell` as much as possible from the code that generates and visualizes maps. The only place that the cell objects are still accessed is when refreshing them after generating a map.

At this point there are multiple members of `HexCell` that are no longer accessed, so let's remove them. They are `HasRiverBeginOrEnd`, `HasRoads`, `StreamBedY`, `RiverSurfaceY`, `WaterSurfaceY`, and `HasIncomingRiverThroughEdge`.

The next tutorial is Hex Map 3.4.0.