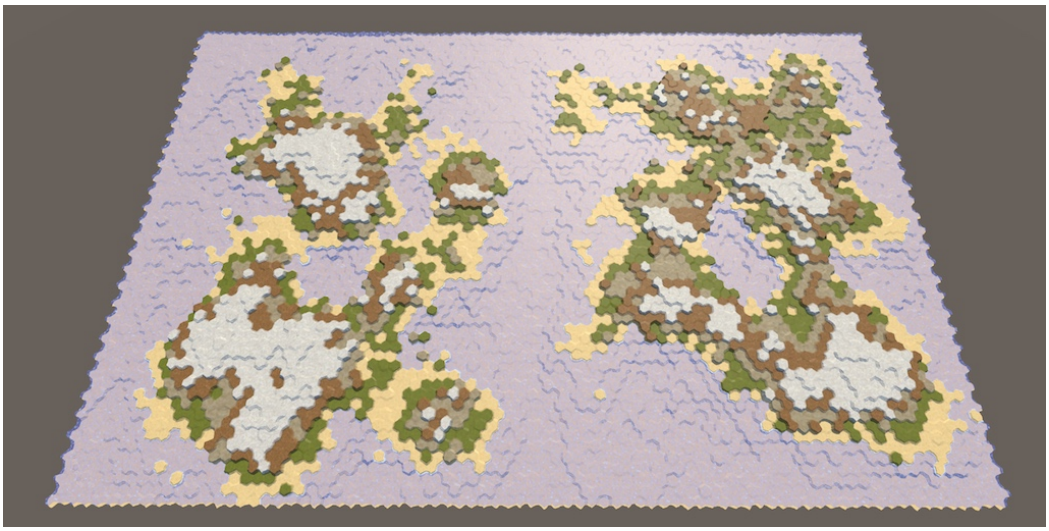
published 2023-10-04

Hex Map 2.3.0 Leaner Cells

Store cell data in bit flags.
Eliminate arrays per cell.
Retrieve neighbors via the grid.



When maps get multiple continents you start caring about memory usage.

This tutorial is made with Unity 2021.3.24f1 and follows Hex Map 2.2.0.

1 Roads

Each cell of the hex map has its own game object, which in turn stores its data in fields and arrays. This is fine for small maps but for larger maps the memory size of cells becomes problematic. In this tutorial we're going to drastically reduce the memory footprint of cells, with the ultimate goal of entirely doing away with cell game objects in a future release.

The first data that we will compress are the roads, which are currently stored in an array per cell. We will replace that array with a single bit flag field, similar to the one used for cell flags in the Maze project.

1.1 Hex Flags

Create a `HexFlags` flags enum type with six bits representing the directions in which a road could exist, along with a mask for all roads and an empty state. Give it extension methods to check whether any, all, or none of the bits for a given mask are set, and methods to get flags with or without a mask set.

```
[System.Flags]
public enum HexFlags
{
	Empty = 0,

	RoadNE = 0b000001,
	RoadE  = 0b000010,
	RoadSE = 0b000100,
	RoadSW = 0b001000,
	RoadW  = 0b010000,
	RoadNW = 0b100000,

	Roads = 0b111111
}

public static class HexFlagsExtensions
{
	public static bool HasAny (this HexFlags flags, HexFlags mask) => (flags & mask) != 0;

	public static bool HasAll (this HexFlags flags, HexFlags mask) =>
		(flags & mask) == mask;

	public static bool HasNone (this HexFlags flags, HexFlags mask) =>
		(flags & mask) == 0;

	public static HexFlags With (this HexFlags flags, HexFlags mask) => flags | mask;

	public static HexFlags Without (this HexFlags flags, HexFlags mask) => flags & ~mask;
}
```

For hex map specifically we also add `Has`, `With`, and `Without` methods that act on a specific direction relative to a starting bit. These methods are private to the extension class.

```
	static bool Has (this HexFlags flags, HexFlags start, HexDirection direction) =>
		((int)flags & ((int)start << (int)direction)) != 0;

	static HexFlags With (this HexFlags flags, HexFlags start, HexDirection direction) =>
		flags | (HexFlags)((int)start << (int)direction);

	static HexFlags Without (
		this HexFlags flags, HexFlags start, HexDirection direction
	) =>
		flags & ~(HexFlags)((int)start << (int)direction);
```

We use them to publicly expose methods specific for road directions.

```
	public static bool HasRoad (this HexFlags flags, HexDirection direction) =>
		flags.Has(HexFlags.RoadNE, direction);

	public static HexFlags WithRoad (this HexFlags flags, HexDirection direction) =>
		flags.With(HexFlags.RoadNE, direction);

	public static HexFlags WithoutRoad (this HexFlags flags, HexDirection direction) =>
		flags.Without(HexFlags.RoadNE, direction);
```

1.2 Flags instead of Array

Add a flags field to `HexCell`.

```
	HexFlags flags;
```

We will now begin replacing the usage of the roads array with these flags. First is the `HasRoads` property, which can simply check if any roads flag is set.

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

Second is the `HasRoadsThroughEdge` method, which can use the `HasRoad` flags method to check it.

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

Next we will remove the `SetRoad` method and replace it with an explicit `RemoveRoad` method, because adding a road is done in only a single place.

```
	//void SetRoad (int index, bool state) { … }

	void RemoveRoad (HexDirection direction)
	{
		flags = flags.WithoutRoad(direction);
		HexCell neighbor = GetNeighbor(direction);
		neighbor.flags = neighbor.flags.WithoutRoad(direction.Opposite());
		neighbor.RefreshSelfOnly();
		RefreshSelfOnly();
	}
```

Adjust `AddRoad` and `RemoveRoads` to work with the flags.

```
	public void AddRoad (HexDirection direction)
	{
		if (
			!flags.HasRoad(direction) && !HasRiverThroughEdge(direction) &&
			!IsSpecial && !GetNeighbor(direction).IsSpecial &&
			GetElevationDifference(direction) <= 1
		)
		{
			//SetRoad((int)direction, true);
			flags = flags.WithRoad(direction);
			HexCell neighbor = GetNeighbor(direction);
			neighbor.flags = neighbor.flags.WithRoad(direction.Opposite());
			neighbor.RefreshSelfOnly();
			RefreshSelfOnly();
		}
	}

	public void RemoveRoads ()
	{
		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
		{
			if (flags.HasRoad(d))
			{
				RemoveRoad(d);
			}
		}
	}
```

Adjust the `Elevation` setter as well.

```
	public int Elevation
	{
		get => elevation;
		set
		{
			…

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				if (flags.HasRoad(d) && GetElevationDifference(d) > 1)
				{
					RemoveRoad(d);
				}
			}

			Refresh();
		}
	}
```

We also have to adjust saving and loading the roads data. As we already stored them as bit flags the code becomes simpler as a direct cast is possible.

```
	public void Save (BinaryWriter writer)
	{
		…

		//int roadFlags = 0;
		//for (int i = 0; i < roads.Length; i++) { … }
		writer.Write((byte)(flags & HexFlags.Roads));
		writer.Write(IsExplored);
	}

	public void Load (BinaryReader reader, int header)
	{
		…

		//int roadFlags = reader.ReadByte();
		//for (int i = 0; i < roads.Length; i++) { … }
		flags |= (HexFlags)reader.ReadByte();

		IsExplored = header >= 3 ? reader.ReadBoolean() : false;
		ShaderData.RefreshTerrain(this);
		ShaderData.RefreshVisibility(this);
	}
```

Finally we remove the roads array field. Cells have now become smaller and even a bit more performant as well, because the indirection via an array has been removed.

```
	//[SerializeField]
	//bool[] roads;
```

2 Rivers

What we did for roads we can also do for rivers. We store river data in two boolean and two direction fields instead of an array, but there is plenty of room for all river data in our single flags field.

2.1 River Flags

Use the next six bits of `HexFlags` to represent the incoming river direction and the six bits after that for the outgoing river direction. Also define masks for the incoming, the outgoing, and any river. Note that this data structure would allow for multiple incoming and outgoing rivers per cell, but our hex map does not support that.

```
	Roads = 0b111111,

	RiverInNE = 0b000001_000000,
	RiverInE  = 0b000010_000000,
	RiverInSE = 0b000100_000000,
	RiverInSW = 0b001000_000000,
	RiverInW  = 0b010000_000000,
	RiverInNW = 0b100000_000000,

	RiverIn = 0b111111_000000,

	RiverOutNE = 0b000001_000000_000000,
	RiverOutE  = 0b000010_000000_000000,
	RiverOutSE = 0b000100_000000_000000,
	RiverOutSW = 0b001000_000000_000000,
	RiverOutW  = 0b010000_000000_000000,
	RiverOutNW = 0b100000_000000_000000,

	RiverOut = 0b111111_000000_000000,

	River = 0b111111_111111_000000
```

Add extension methods for rivers like for roads.

```
	public static bool HasRiverIn (this HexFlags flags, HexDirection direction) =>
		flags.Has(HexFlags.RiverInNE, direction);

	public static HexFlags WithRiverIn (this HexFlags flags, HexDirection direction) =>
		flags.With(HexFlags.RiverInNE, direction);

	public static HexFlags WithoutRiverIn (this HexFlags flags, HexDirection direction) =>
		flags.Without(HexFlags.RiverInNE, direction);

	public static bool HasRiverOut (this HexFlags flags, HexDirection direction) =>
		flags.Has(HexFlags.RiverOutNE, direction);

	public static HexFlags WithRiverOut (this HexFlags flags, HexDirection direction) =>
		flags.With(HexFlags.RiverOutNE, direction);

	public static HexFlags WithoutRiverOut (
		this HexFlags flags, HexDirection direction
	) =>
		flags.Without(HexFlags.RiverOutNE, direction);
```

We also need to get the river directions, so add a private method to convert flags to a direction, shifted to the lowest six bits. This assumes that exactly one of the six bits is set.

```
	static HexDirection ToDirection (this HexFlags flags, int shift) =>
		(((int)flags >> shift) & 0b111111) switch
		{
			0b000001 => HexDirection.NE,
			0b000010 => HexDirection.E,
			0b000100 => HexDirection.SE,
			0b001000 => HexDirection.SW,
			0b010000 => HexDirection.W,
			_ => HexDirection.NW
		};
```

Using that method, add public methods for getting the incoming and outgoing river direction.

```
	public static HexDirection RiverInDirection (this HexFlags flags) =>
		flags.ToDirection(6);

	public static HexDirection RiverOutDirection (this HexFlags flags) =>
		flags.ToDirection(12);
```

2.2 Flags instead of Fields

This time we begin by removing the old river fields.

```
	//bool hasIncomingRiver, hasOutgoingRiver;
	//HexDirection incomingRiver, outgoingRiver;
```

Adjust the `HasIncomingRiver`, `HasOutGoingRiver`, and `HasRiver` properties.

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

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

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

Adjust the `HasRiverBeginOrEnd` property to use these properties instead of the removed fields.

```
	public bool HasRiverBeginOrEnd => HasIncomingRiver != HasOutgoingRiver;
```

Remove the `RiverBeginOrEndDirection` property, because with the fields removed getting river directions is no longer trivial.

```
	//public HexDirection RiverBeginOrEndDirection =>
		//hasIncomingRiver ? incomingRiver : outgoingRiver;
```

We keep and adjust the properties that explicitly get the incoming or outgoing river direction.

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

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

The `HasRiverThroughEdge` method also needs to be updated.

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

And we add a new `HasIncomingRiverThroughEdge` method, which checks the flags based on a given direction.

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

Next, adapt `RemoveIncomingRiver` and `RemoveOutgiongRiver` to work with the flags.

```
	public void RemoveIncomingRiver ()
	{
		if (!HasIncomingRiver)
		{
			return;
		}
		//hasIncomingRiver = false;
		//RefreshSelfOnly();

		HexCell neighbor = GetNeighbor(IncomingRiver);
		//neighbor.hasOutgoingRiver = false;
		flags = flags.Without(HexFlags.RiverIn);
		neighbor.flags = neighbor.flags.Without(HexFlags.RiverOut);
		neighbor.RefreshSelfOnly();
		RefreshSelfOnly();
	}

	public void RemoveOutgoingRiver ()
	{
		if (!HasOutgoingRiver)
		{
			return;
		}
		//hasOutgoingRiver = false;
		//RefreshSelfOnly();

		HexCell neighbor = GetNeighbor(OutgoingRiver);
		//neighbor.hasIncomingRiver = false;
		flags = flags.Without(HexFlags.RiverOut);
		neighbor.flags = neighbor.flags.Without(HexFlags.RiverIn);
		neighbor.RefreshSelfOnly();
		RefreshSelfOnly();
	}
```

Followed by `SetOutgoingRiver`.

```
	public void SetOutgoingRiver (HexDirection direction)
	{
		if (flags.HasRiverOut(direction))
		{
			return;
		}

		…

		RemoveOutgoingRiver();
		if (flags.HasRiverIn(direction))
		{
			RemoveIncomingRiver();
		}
		//hasOutgoingRiver = true;
		//outgoingRiver = direction;

		flags = flags.WithRiverOut(direction);
		specialIndex = 0;
		neighbor.RemoveIncomingRiver();
		//neighbor.hasIncomingRiver = true;
		//neighbor.incomingRiver = direction.Opposite();
		neighbor.flags = neighbor.flags.WithRiverIn(direction.Opposite());
		neighbor.specialIndex = 0;

		RemoveRoad(direction);
	}
```

`ValidateRiver` now also has to use the river properties.

```
	void ValidateRivers ()
	{
		if (HasOutgoingRiver && !IsValidRiverDestination(GetNeighbor(OutgoingRiver)))
		{
			RemoveOutgoingRiver();
		}
		if (
			HasIncomingRiver && !GetNeighbor(IncomingRiver).IsValidRiverDestination(this)
		)
		{
			RemoveIncomingRiver();
		}
	}
```

We keep the save format the same, so `Save` only has to switch to using properties.

```
		if (HasIncomingRiver)
		{
			writer.Write((byte)(IncomingRiver + 128));
		}
		else
		{
			writer.Write((byte)0);
		}

		if (HasOutgoingRiver)
		{
			writer.Write((byte)(OutgoingRiver + 128));
		}
		else
		{
			writer.Write((byte)0);
		}
```

While `Load` has to set the flags.

```
		byte riverData = reader.ReadByte();
		if (riverData >= 128)
		{
			//hasIncomingRiver = true;
			//incomingRiver = (HexDirection)(riverData - 128);
			flags = flags.WithRiverIn((HexDirection)(riverData - 128));
		}
		//else { … }

		riverData = reader.ReadByte();
		if (riverData >= 128)
		{
			//hasOutgoingRiver = true;
			//outgoingRiver = (HexDirection)(riverData - 128);
			flags = flags.WithRiverOut((HexDirection)(riverData - 128));
		}
		//else { … }
```

2.3 Chunks

We also have to tweak `HexGridChunk` to work with the new approach. Make `TriangulateWaterShore` use the new `HasIncomingRiverThroughEdge` method.

```
	void TriangulateWaterShore (
		HexDirection direction, HexCell cell, HexCell neighbor, Vector3 center
	) {
		…

		if (cell.HasRiverThroughEdge(direction)) {
			TriangulateEstuary(
				e1, e2, cell.HasIncomingRiverThroughEdge(direction), indices
			);
		}
		else { … }

		…
	}
```

Change `TriangulateRoadAdjacentToRiver` so it caches the river directions, as these aren't simple field lookups anymore.

```
	void TriangulateRoadAdjacentToRiver (
		HexDirection direction, HexCell cell, Vector3 center, EdgeVertices e
	) {
		…

		HexDirection riverIn = cell.IncomingRiver, riverOut = cell.OutgoingRiver;

		if (cell.HasRiverBeginOrEnd) {
			roadCenter += HexMetrics.GetSolidEdgeMiddle(
				(cell.HasIncomingRiver ? riverIn : riverOut).Opposite()
			) * (1f / 3f);
		}
		else if (riverIn == riverOut.Opposite()) {
			…
			roadCenter += corner * 0.5f;
			if (riverIn == direction.Next() && (
				cell.HasRoadThroughEdge(direction.Next2()) ||
				cell.HasRoadThroughEdge(direction.Opposite())
			)) {
				features.AddBridge(roadCenter, center - corner * 0.5f);
			}
			center += corner * 0.25f;
		}
		else if (riverIn == riverOut.Previous()) {
			roadCenter -= HexMetrics.GetSecondCorner(riverIn) * 0.2f;
		}
		else if (riverIn == riverOut.Next()) {
			roadCenter -= HexMetrics.GetFirstCorner(riverIn) * 0.2f;
		}
		…
	}
```

Make `TriangulateWithRiver` also use `HasIncomingRiverThroughEdge`.

```
	void TriangulateWithRiver (
		HexDirection direction, HexCell cell, Vector3 center, EdgeVertices e
	) {
		…

		if (!cell.IsUnderwater) {
			bool reversed = cell.HasIncomingRiverThroughEdge(direction);
			…
		}
	}
```

And `TriangulateConnection` as well.

```
	void TriangulateConnection (
		HexDirection direction, HexCell cell, EdgeVertices e1
	) {
		…

		if (hasRiver) {
			…

			if (!cell.IsUnderwater) {
				if (!neighbor.IsUnderwater) {
					TriangulateRiverQuad(
						e1.v2, e1.v4, e2.v2, e2.v4,
						cell.RiverSurfaceY, neighbor.RiverSurfaceY, 0.8f,
						cell.HasIncomingRiverThroughEdge(direction),
						indices
					);
				}
				…
			}
			…
		}

		…
	}
```

3 Neighbors

Neighbor data is also stored an array, which we'll remove next. While we could store whether neighbors exist in flags we won't, as they usually do exist.

3.1 Going via the Grid

Without direct references to its neighbors `HexCell` needs a reference to the hex grid. Give it a property for this.

```
	public HexGrid Grid
	{ get; set; }
```

We make a slight adjustment to `HexGrid`, consolidating the bounds check for `GetCell` so it has only two exit paths instead of three.

```
	public HexCell GetCell (HexCoordinates coordinates)
	{
		int z = coordinates.Z;
		//if (z < 0 || z >= CellCountZ) {
			//return null;
		//}
		int x = coordinates.X + z / 2;
		if (z < 0 || z >= CellCountZ || x < 0 || x >= CellCountX)
		{
			return null;
		}
		return cells[x + z * CellCountX];
	}
```

And we add a `TryGetCell` method alternative to `GetCell` which is slightly more performant and more convenient to use.

```
	public bool TryGetCell (HexCoordinates coordinates, out HexCell cell)
	{
		int z = coordinates.Z;
		int x = coordinates.X + z / 2;
		if (z < 0 || z >= CellCountZ || x < 0 || x >= CellCountX)
		{
			cell = null;
			return false;
		}
		cell = cells[x + z * CellCountX];
		return true;
	}
```

Also add a `Step` method to `HexCoordinates` that makes it easy to get the hex coordinates of a direct neighbor.

```
	public HexCoordinates Step (HexDirection direction) => direction switch
	{
		HexDirection.NE => new HexCoordinates(x, z + 1),
		HexDirection.E => new HexCoordinates(x + 1, z),
		HexDirection.SE => new HexCoordinates(x + 1, z - 1),
		HexDirection.SW => new HexCoordinates(x, z - 1),
		HexDirection.W => new HexCoordinates(x - 1, z),
		_ => new HexCoordinates(x - 1, z + 1)
	};
```

3.2 No More Array

Remove the neighbors array from `HexCell`.

```
	//[SerializeField]
	//HexCell[] neighbors;
```

Adjust its `GetNeighbor` method to go via the grid and include a `TryGetNeighbor` variant as well. Going via the grid is comparable to going via a neighbors array. It requires a little more work to find the neighbor coordinates, but everything now goes through a central point instead of via separate arrays that are scattered around, so memory access is improved.

```
	public HexCell GetNeighbor (HexDirection direction) =>
		Grid.GetCell(Coordinates.Step(direction));

	public bool TryGetNeighbor (HexDirection direction, out HexCell cell) =>
		Grid.TryGetCell(Coordinates.Step(direction), out cell);
```

Remove the `SetNeighbor` method as it is no longer applicable.

```
	//public void SetNeighbor (HexDirection direction, HexCell cell) { … }
```

`GetEdgeType` now has to use `GetNeighbor`. We still assume that the neighbor exists, because only then is this method invoked.

```
	public HexEdgeType GetEdgeType (HexDirection direction) => HexMetrics.GetEdgeType(
		elevation, GetNeighbor(direction).elevation
	);
```

Also adjust `Refresh` so it uses `TryGetNeighbor`.

```
	void Refresh ()
	{
		if (Chunk)
		{
			Chunk.Refresh();
			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				//HexCell neighbor = neighbors[i];
				if (TryGetNeighbor(d, out HexCell neighbor) && neighbor.Chunk != Chunk)
				{
					neighbor.Chunk.Refresh();
				}
			}
			…
		}
	}
```

3.3 The Grid

`HexGrid.CreateCell` now has to set the cell's grid and no longer needs to hook up neighbors.

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

		HexCell cell = cells[i] = Instantiate<HexCell>(cellPrefab);
		cell.Grid = this;
		…

		//if (x > 0) {
			//cell.SetNeighbor(HexDirection.W, cells[i - 1]);
			//…
		//}
		//if (z > 0) {
			//…
		//}

		Text label = Instantiate<Text>(cellLabelPrefab);
		…
	}
```

Also adjust `Search` so it uses `TryGetNeighbor`.

```
	bool Search (HexCell fromCell, HexCell toCell, HexUnit unit)
	{
		…
		while (searchFrontier.Count > 0)
		{
			…

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				//HexCell neighbor = current.GetNeighbor(d);
				if (
					//neighbor == null ||
					!current.TryGetNeighbor(d, out HexCell neighbor) ||
					neighbor.SearchPhase > searchFrontierPhase
				)
				{
					continue;
				}
				…
			}
		}
		return false;
	}
```

The same goes for `GetVisibleCells`.

```
	List<HexCell> GetVisibleCells (HexCell fromCell, int range)
	{
		…
		while (searchFrontier.Count > 0)
		{
			…

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++)
			{
				//HexCell neighbor = current.GetNeighbor(d);
				if (
					//neighbor == null ||
					!current.TryGetNeighbor(d, out HexCell neighbor) ||
					neighbor.SearchPhase > searchFrontierPhase ||
					!neighbor.Explorable
				)
				{
					continue;
				}

				…
			}
		}
		return visibleCells;
	}
```

3.4 Chunks

Adjust `HexGridChunk` so it also relies on `TryGetNeighbor`, beginning with `TriangulateWater`.

```
	void TriangulateWater (
		HexDirection direction, HexCell cell, Vector3 center
	) {
		center.y = cell.WaterSurfaceY;

		//HexCell neighbor = cell.GetNeighbor(direction);
		if (
			cell.TryGetNeighbor(direction, out HexCell neighbor) && !neighbor.IsUnderwater
		) {
			TriangulateWaterShore(direction, cell, neighbor, center);
		}
		…
	}
```

Do the same for `TriangulateOpenWater`, `TriangulateWaterShore`, and `TriangulateConnection`.

3.5 Map Generator

HexMapGenerator can also use `TryGetNeighbor`. First in `RaiseTerrain`.

```
	int RaiseTerrain (int chunkSize, int budget, MapRegion region) {
		…
		while (size < chunkSize && searchFrontier.Count > 0) {
			…

			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				//HexCell neighbor = current.GetNeighbor(d);
				if (
					current.TryGetNeighbor(d, out HexCell neighbor) &&
					neighbor.SearchPhase < searchFrontierPhase
				) {
					…
				}
			}
		}
		…
	}
```

And also in `SinkTerrain`, `ErodeLand`, `IsErodible`, `GetErosionTarget`, `EvolveClimate`, `CreateRivers`, `CreateRiver`, and `SetTerrainType`.

3.6 Map Editor

HexMapEditor also has one place where we can use `TryGetNeighbor`, in `EditCell`.

```
	void EditCell (HexCell cell)
	{
		if (cell)
		{
			…
			if (
				isDrag &&
				cell.TryGetNeighbor(dragDirection.Opposite(), out HexCell otherCell)
			) {
				//HexCell otherCell = cell.GetNeighbor(dragDirection.Opposite());
				//if (otherCell) {
				if (riverMode == OptionalToggle.Yes)
				{
					otherCell.SetOutgoingRiver(dragDirection);
				}
				if (roadMode == OptionalToggle.Yes)
				{
					otherCell.AddRoad(dragDirection);
				}
				//}
			}
		}
	}
```

4 Walls and Exploration

The final data that we turn into flags in this tutorial are the wall and exploration states. These are boolean fields, taking up the same amount of space as our single flags field, while they could fit in a single bit each.

4.1 Walls

Add a bit for walls to `HexFlags`.

```
	River = 0b111111_111111_000000,

	Walled = 0b1_000000_000000_000000
```

Then remove the `walled` field from `HexCell`.

```
	//bool walled;
```

Adjust the `Walled` property so it uses the bit flag.

```
	public bool Walled
	{
		get => flags.HasAny(HexFlags.Walled);
		set
		{
			HexFlags newFlags =
				value ? flags.With(HexFlags.Walled) : flags.Without(HexFlags.Walled);
			if (flags != newFlags)
			{
				//walled = value;
				flags = newFlags;
				Refresh();
			}
		}
	}
```

Change `Save` to it uses that property.

```
		writer.Write(Walled);
```

And `Load` so it sets the flag if needed.

```
		//walled = reader.ReadBoolean();
		if (reader.ReadBoolean())
		{
			flags = flags.With(HexFlags.Walled);
		}
```

4.2 Exploration

Add bits for the explored and explorable states to `HexFlags`.

```
	Walled = 0b1_000000_000000_000000,

	Explored   = 0b010_000000_000000_000000,
	Explorable = 0b100_000000_000000_000000
```

Then remove the `explored` field from `HexCell`.

```
	//bool explored;
```

Change the `IsExplored` property to work with the flags.

```
	public bool IsExplored
	{
		get => flags.HasAll(HexFlags.Explored | HexFlags.Explorable);
		private set => flags = value ?
			flags.With(HexFlags.Explored) : flags.Without(HexFlags.Explored);
	}
```

`Explorable` was a stand-alone property, but it must now also refer to the flags.

```
	public bool Explorable
	{
		get => flags.HasAny(HexFlags.Explorable);
		set => flags = value ?
			flags.With(HexFlags.Explorable) : flags.Without(HexFlags.Explorable);
	}
```

Whether a cell is explorable is set when a map is created. So when `Load` gets invoked it is already appropriately set. To make this explicit clear all flags at the start of `Load`, except the explorable one.

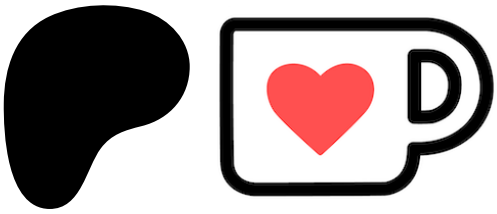
```
	public void Load (BinaryReader reader, int header)
	{
		flags &= HexFlags.Explorable;
		…
	}
```

Our cells have become a lot leaner at this point. We'll continue working on this in the future.

The next tutorial is Hex Map 3.0.0.

Enjoying the tutorials? Are they useful?

Please support me on Patreon or Ko-fi!



Or make a direct donation!

made by Jasper Flick