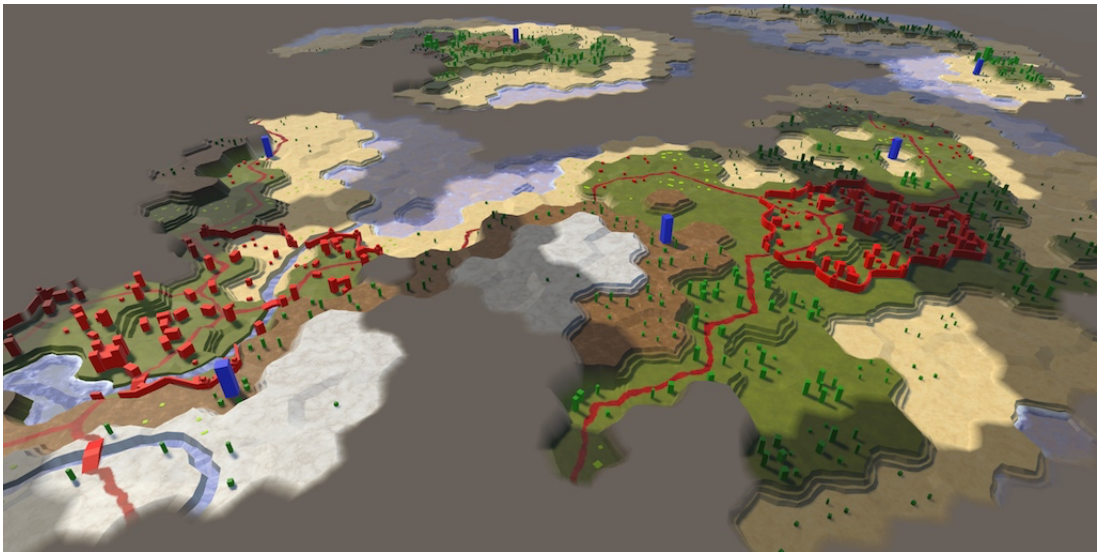
Catlike Coding

Unity C# Tutorials

Hex Map 22 Advanced Vision

Smoothly adjust visibility.
Use elevation to determine sight.
Hide the edge of the map.

This is part 22 of a tutorial series about hexagon maps. After adding support for exploration, we'll upgrade our vision calculations and transitions.



To see far, go high.

1 Visibility Transitions

A cell is either visible or invisible, because it is either in vision range of a unit, or not. Even though it looks like a unit takes a while to travel between cells, its vision jumps from cell to cell instantaneously. As a result, the visibility of cells around it change suddenly. The unit's movement appears smooth, while the visibility changes are sudden.

Ideally, visibility also changes smoothly. Cells would light up gradually as they come into view and darken slowly once they're no longer visible. Or maybe you prefer immediate transitions? Let's add a property to `HexCellShaderData` to toggle whether we want immediate transitions. We'll make smooth transitions the default.

```
	public bool ImmediateMode { get; set; }
```

1.1 Tracking Transitioning Cells

Even when showing smooth transitions, the actual visibility data is still binary. So it's purely a visual effect. This means that it's up to `HexCellShaderData` to keep track of visibility transitions. Give it a list to keep track of the transitioning cells. Make sure that it's empty after each initialization.

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

public class HexCellShaderData : MonoBehaviour {

	Texture2D cellTexture;
	Color32[] cellTextureData;

	List<HexCell> transitioningCells = new List<HexCell>();

	public bool ImmediateMode { get; set; }

	public void Initialize (int x, int z) {
		…

		transitioningCells.Clear();
		enabled = true;
	}

	…
}
```

Currently, we directly set the cell data in `RefreshVisibility`. This is still correct when immediate mode is active. But when it's not, we should instead add the cell to the list of transitioning cells.

```
	public void RefreshVisibility (HexCell cell) {
		int index = cell.Index;
		if (ImmediateMode) {
			cellTextureData[index].r = cell.IsVisible ? (byte)255 : (byte)0;
			cellTextureData[index].g = cell.IsExplored ? (byte)255 : (byte)0;
		}
		else {
			transitioningCells.Add(cell);
		}
		enabled = true;
	}
```

Visibility now no longer appears to work, because we don't do anything with the cells in the list yet.

1.2 Looping through Transitioning Cells

Instead of immediately setting the relevant values to either 255 or 0, we going to gradually increase or decrease these values. How quickly we do that determines how smooth the transitions appear. We shouldn't do it too fast, but also not too slow. One second is a good compromise between nice visuals and playability. Let's define a constant for that, so it's easy to change.

```
	const float transitionSpeed = 255f;
```

In `LateUpdate`, we can now determine the delta to apply to the values, by multiplying the time delta with the speed. This has to be an integer, because we don't know how large it could get. A freak frame-rate dip could make the delta larger than 255.

Also, we must keep updating as long as there are cells in transition. So make sure we remain enabled while there's something in the list.

```
	void LateUpdate () {
		int delta = (int)(Time.deltaTime * transitionSpeed);

		cellTexture.SetPixels32(cellTextureData);
		cellTexture.Apply();
		enabled = transitioningCells.Count > 0;
	}
```

It is also theoretically possible to get very high frame rates. Together with a low transition speed, this could result in a delta of zero. To guarantee progress, force the delta to have a minimum of 1.

```
		int delta = (int)(Time.deltaTime * transitionSpeed);
		if (delta == 0) {
			delta = 1;
		}
```

After we have our delta, we can loop through all transitioning cells and update their data. Let's assume we have an `UpdateCellData` method for that, which has the relevant cell and the delta as parameters.

```
		int delta = (int)(Time.deltaTime * transitionSpeed);
		if (delta == 0) {
			delta = 1;
		}
		for (int i = 0; i < transitioningCells.Count; i++) {
			UpdateCellData(transitioningCells[i], delta);
		}
```

At some point, a cell's transition should be finished. Let's assume that the method returns whether it's still transitioning. When that's no longer the case, we can remove the cell from the list. Afterwards, we have to decrement the iterator to not skip any cells.

```
		for (int i = 0; i < transitioningCells.Count; i++) {
			if (!UpdateCellData(transitioningCells[i], delta)) {
				transitioningCells.RemoveAt(i--);
			}
		}
```

The order in which we process the transitioning cells doesn't matter. So we don't have to remove the cell at the current index, which forces `RemoveAt` to shift all cells after it. Instead, move the last cell to the current index and them remove the last one.

```
			if (!UpdateCellData(transitioningCells[i], delta)) {
				transitioningCells[i--] =
					transitioningCells[transitioningCells.Count - 1];
				transitioningCells.RemoveAt(transitioningCells.Count - 1);
			}
```

Now we have to create the `UpdateCellData` method. It's going to need to need the cell's index and data to do its work, so begin by fetching those. It also has to determine whether this cell still requires further updating. By default, we'll assume this is not the case. Once the work is done, the adjusted data has to be applied and the still-updating status returned.

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

		cellTextureData[index] = data;
		return stillUpdating;
	}
```

1.3 Updating Cell Data

At this point, we have a cell that is in transition, or maybe it is already finished. First, let's consider the cell's exploration state. If the cell is explored but its G value isn't 255 yet, then indeed it is still in transition, so keep track of this fact.

```
		bool stillUpdating = false;

		if (cell.IsExplored && data.g < 255) {
			stillUpdating = true;
		}

		cellTextureData[index] = data;
```

To progress the transition, add the delta to the cell's G value. Arithmatic operations don't work on bytes, they are always converted to integers first. So the sum is an integer, which has to be cast to a byte.

```
		if (cell.IsExplored && data.g < 255) {
			stillUpdating = true;
			int t = data.g + delta;
			data.g = (byte)t;
		}
```

But we must ensure that we do no exceed 255 before casting.

```
			int t = data.g + delta;
			data.g = t >= 255 ? (byte)255 : (byte)t;
```

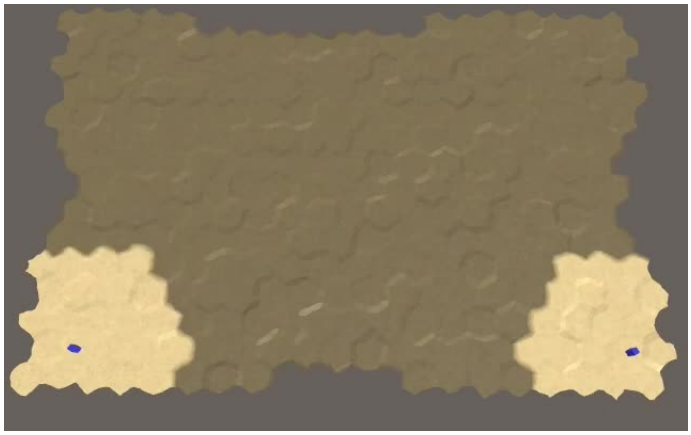
Next, we have to do the same thing for the visibility, which uses the R value.

```
		if (cell.IsExplored && data.g < 255) {
			…
		}

		if (cell.IsVisible && data.r < 255) {
			stillUpdating = true;
			int t = data.r + delta;
			data.r = t >= 255 ? (byte)255 : (byte)t;
		}
```

As cells can also become invisible again, we must also check whether we have to decrease the R value. This is the case when the cell isn't visible while R is larger than zero.

```
		if (cell.IsVisible) {
			if (data.r < 255) {
				stillUpdating = true;
				int t = data.r + delta;
				data.r = t >= 255 ? (byte)255 : (byte)t;
			}
		}
		else if (data.r > 0) {
			stillUpdating = true;
			int t = data.r - delta;
			data.r = t < 0 ? (byte)0 : (byte)t;
		}
```

Now `UpdateCellData` is complete and the visibility transitions are functional.



Visibility transitions.

1.4 Preventing Duplicate Transition Entries

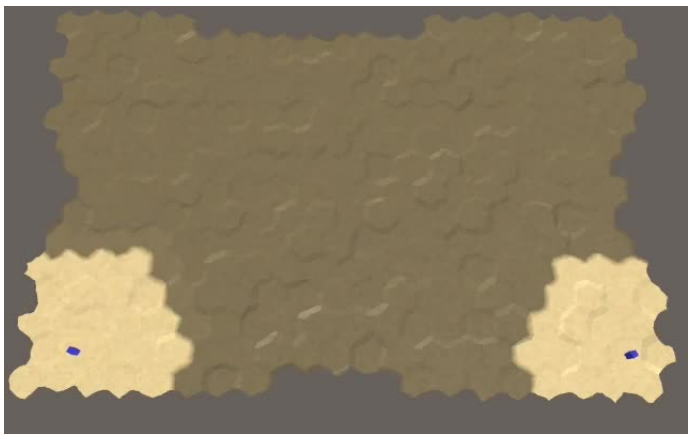
Although the transitions work, we can end up with duplicate entries in the list. This happens when a cell's visibility state changes while it is still in transition. For example, when a cell is only visible for a short while during a unit's journey.

The result of duplicate entries is that the cell's transition gets updated multiple times per frame, which leads to faster transitions and more work than necessary. We could prevent this by checking whether the cell is already in the list before adding it. However, searching a list each time `RefreshVisibility` is invoked is expensive, especially when many cells are already in transition. Instead, let's use one of the yet-unused data channels to store whether a cell in transition, like the B value. Set this value to 255 when the cell is added to the list. Then only add cells whose B value isn't 255.

```
	public void RefreshVisibility (HexCell cell) {
		int index = cell.Index;
		if (ImmediateMode) {
			cellTextureData[index].r = cell.IsVisible ? (byte)255 : (byte)0;
			cellTextureData[index].g = cell.IsExplored ? (byte)255 : (byte)0;
		}
		else if (cellTextureData[index].b != 255) {
			cellTextureData[index].b = 255;
			transitioningCells.Add(cell);
		}
		enabled = true;
	}
```

To make this work, we have to set the B value back to zero once a cell's transition is finished.

```
	bool UpdateCellData (HexCell cell, int delta) {
		…

		if (!stillUpdating) {
			data.b = 0;
		}
		cellTextureData[index] = data;
		return stillUpdating;
	}
```



Transitions without duplicates.

1.5 Immediately Loading Visibility

Visibility changes are now always gradual, even when loading a map. This doesn't make much sense, because the map represent a state where cells were already visible, so a transition is not appropriate. Also, starting transitions for many visible cells on a large map could slow down the game after a load. So let's switch to immediate mode in `HexGrid.Load` before the cells and units are loaded.

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

		cellShaderData.ImmediateMode = true;

		for (int i = 0; i < cells.Length; i++) {
			cells[i].Load(reader, header);
		}
		…
	}
```

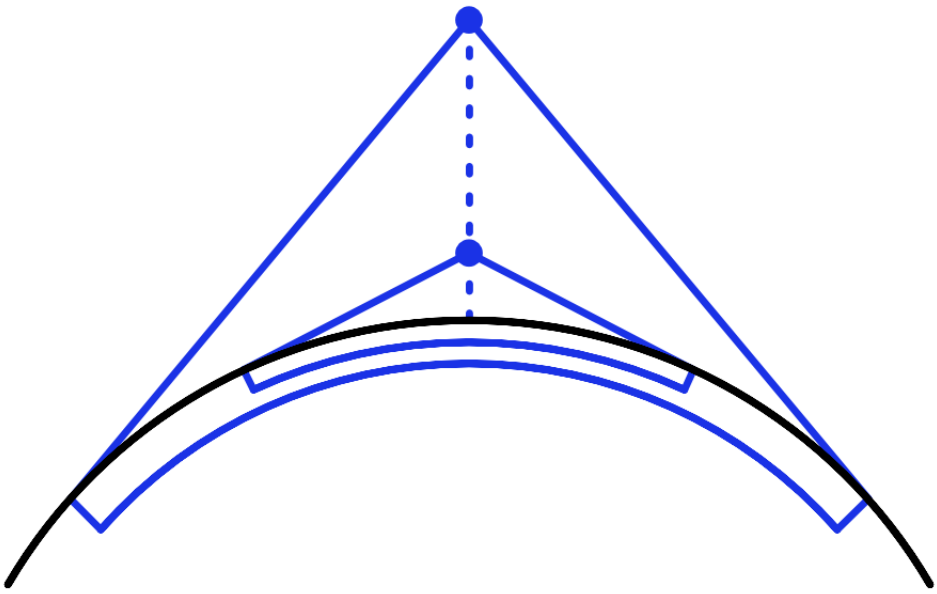
This overrides the original immediate mode setting, whatever it was. Maybe it's always disabled, or maybe it's been made into a configuration option. So remember the original mode and switch back to that after the work is done.

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

		bool originalImmediateMode = cellShaderData.ImmediateMode;
		cellShaderData.ImmediateMode = true;

		…

		cellShaderData.ImmediateMode = originalImmediateMode;
	}
```

2 Vision Based on Elevation

So far, we've simply used a fixed view range of three for all units, but vision is more complex than that. In general, there are two reasons that we cannot see something. The first is that there's something else in front of it, blocking our view. The second is that something is visually too small for us to perceive, either because it's really tiny or very far away. We're only going to concern ourselves with blocked vision.

We cannot see what's on the other side of the earth, because the planet blocks our view. We can only see up to the horizon. Because a planet is roughly a sphere, the higher our point of view, the more of its surface we can see. So the horizon depends on elevation.



Horizon depends on view elevation.

The limited vision range of our units simulates the horizon effect caused by the earth's curvature. How far they should be able to see depends on the planet's size and the map's scale. At least, that's the rationale. The main reason that we limit vision is for gameplay, the restriction of information known as fog of war. But knowing the physics behind it, we can conclude that a high vantage point should be strategically valuable, as it extends the horizon and makes it possible to see over lower obstacles. However, that is currently not the case.

2.1 Elevation for Viewing

To take elevation into consideration for vision, we need to know a cell's elevation for the purpose of sight. This is either its regular elevation or its water level, depending on whether it's dry or submerged. Let's add a convenient property for that to `HexCell`.

```
	public int ViewElevation {
		get {
			return elevation >= waterLevel ? elevation : waterLevel;
		}
	}
```

But if vision is influenced by elevation, then when a cell's view elevation changes the visibility situation might change as well. As the cell could have been or is now blocking the vision of multiple units, it isn't trivial to determine what has to be changed. It's not possible for the changed cell to solve this problem itself, so let's have it notify `HexCellShaderData` that the situation has changed. Let's assume `HexCellShaderData` has a convenient `ViewElevationChanged` method for that. Invoke it when `HexCell.Elevation` is set, if needed.

```
	public int Elevation {
		get {
			return elevation;
		}
		set {
			if (elevation == value) {
				return;
			}
			int originalViewElevation = ViewElevation;
			elevation = value;
			if (ViewElevation != originalViewElevation) {
				ShaderData.ViewElevationChanged();
			}
			…
		}
	}
```

The same goes for `WaterLevel`.

```
	public int WaterLevel {
		get {
			return waterLevel;
		}
		set {
			if (waterLevel == value) {
				return;
			}
			int originalViewElevation = ViewElevation;
			waterLevel = value;
			if (ViewElevation != originalViewElevation) {
				ShaderData.ViewElevationChanged();
			}
			ValidateRivers();
			Refresh();
		}
	}
```

2.2 Resetting Visibility

Now we have to create the `HexCellShaderData.ViewElevationChanged` method. Figuring out how the overall visibility situation could have changed is a hard problem, especially when multiple cells are altered together. So we're not going to be smart about this. Instead, we'll schedule a reset of all cell visibility. Add a boolean field to keep track of whether this is required. Inside the method, simply set it to true and enable the component. No matter how many cells get changed at once, it will result in a single reset.

```
	bool needsVisibilityReset;

	…

	public void ViewElevationChanged () {
		needsVisibilityReset = true;
		enabled = true;
	}
```

Resetting the visibility of all cells requires access to them, which `HexCellShaderData` doesn't have. So let's delegate that responsibility to `HexGrid`. This requires that we add a property for a reference to the grid to `HexCellShaderData`. Then we can use it in `LateUpdate` to request a reset.

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

	…

	void LateUpdate () {
		if (needsVisibilityReset) {
			needsVisibilityReset = false;
			Grid.ResetVisibility();
		}

		…
	}
```

Moving on to `HexGrid`, setup the grid reference in `HexGrid.Awake` after creating the shader data.

```
	void Awake () {
		HexMetrics.noiseSource = noiseSource;
		HexMetrics.InitializeHashGrid(seed);
		HexUnit.unitPrefab = unitPrefab;
		cellShaderData = gameObject.AddComponent<HexCellShaderData>();
		cellShaderData.Grid = this;
		CreateMap(cellCountX, cellCountZ);
	}
```

`HexGrid` must also get a `ResetVisibility` method to reset all the cells. Just have it loop through the cells and delegate the resetting to them.

```
	public void ResetVisibility () {
		for (int i = 0; i < cells.Length; i++) {
			cells[i].ResetVisibility();
		}
	}
```

Now we have to add a `ResetVisibilty` method to `HexCell`. Simply have it set the cell's visibility to zero and trigger a visibility refresh. We only need to do this when the cell's visibility was actually larger than zero.

```
	public void ResetVisibility () {
		if (visibility > 0) {
			visibility = 0;
			ShaderData.RefreshVisibility(this);
		}
	}
```

After all visibility data has been reset, `HexGrid.ResetVisibility` has to apply the vision of all units again, for which its needs to know each unit's vision range. Let's suppose that's available via a `VisionRange` property.

```
	public void ResetVisibility () {
		for (int i = 0; i < cells.Length; i++) {
			cells[i].ResetVisibility();
		}
		for (int i = 0; i < units.Count; i++) {
			HexUnit unit = units[i];
			IncreaseVisibility(unit.Location, unit.VisionRange);
		}
	}
```

To make this work, refactor rename the `HexUnit.visionRange` to `HexUnit.VisionRange` and turn it into a property. For now it always gets us the constant 3, but this will change in the future.

```
	public int VisionRange {
		get {
			return 3;
		}
	}
```

This will ensure that the visibility data gets reset and stays correct after the view elevation of cells are changed. But it's also possible that we change the rules for vision and trigger a recompile while in play mode. To ensure that the vision adjusts itself automatically, let's also trigger a reset in `HexGrid.OnEnable`, when we detect a recompile.

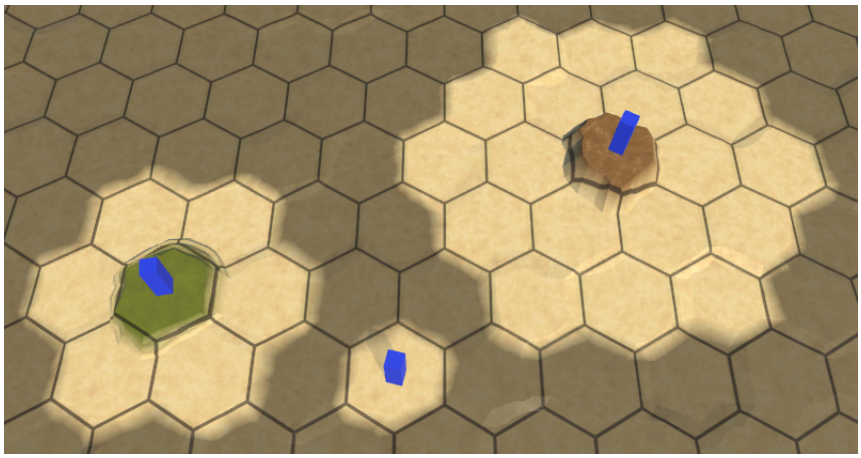
```
	void OnEnable () {
		if (!HexMetrics.noiseSource) {
			…
			ResetVisibility();
		}
	}
```

Now we can change our vision code and can see the results while staying in play mode.

2.3 Extending the Horizon

How vision works is determined by `HexGrid.GetVisibleCells`. To have elevation impact the vision range, we can simply use the view elevation of `fromCell`, temporarily overriding the provided range. That makes it easy to very that it works.

```
	List<HexCell> GetVisibleCells (HexCell fromCell, int range) {
		…

		range = fromCell.ViewElevation;
		fromCell.SearchPhase = searchFrontierPhase;
		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
		…
	}
```



Using elevation as range.

2.4 Blocking Vision

Using the view elevation as the range works correctly when all other cells are at elevation zero. But if all cells had the same elevation as the view point, then we should end up with an effective range of zero. Also, cells with high elevation should block vision to lower cells behind them. Neither is currently the case.

Unblocked vision.

While the most correct way to determine vision would be to perform some kind of raycast test, that would become expensive fast and could still produce weird results. We need an approach that is fast and provides results that are good enough, but need not be perfect. Also, it is important that vision rules are simple, intuitive, and easy to predict by players.

The solution we're going for is to add the neighbor's view elevation to the covered distance, when determining when we can see a cell. This effectively shortens the vision range when looking at those cells. And when they're skipped, this can also prevent us from reaching cells behind them.

```
int distance = current.Distance + 1;
if (distance + neighbor.ViewElevation > range) {
	continue;
}
```

Elevated cells block vision.

Shouldn't we be able to see high cells in the distance?

Like a mountain range, you can see their slopes adjacent to the cells that you can still see. But you cannot see the mountains from above, so you cannot see the cells themselves.

2.5 Not Looking Around Corners

It now looks like high cells block vision to lower cells, except sometimes vision gets through when it seemingly shouldn't. This happens because our search algorithm still found a path to those cells, going around the blocking cells. As a result, it appears as if our vision can bend around obstacles. To prevent this, we have to ensure that only the shortest paths are considered when determining a cell's visibility. This can be done by skipping paths that would become longer than that.

```
		HexCoordinates fromCoordinates = fromCell.coordinates;
		while (searchFrontier.Count > 0) {
			…

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

				int distance = current.Distance + 1;
				if (distance + neighbor.ViewElevation > range ||
					distance > fromCoordinates.DistanceTo(neighbor.coordinates)
				) {
					continue;
				}

				…
			}
		}
```

Only using shortest paths.

This fixes most of the obviously wrong cases. It works well for nearby cells, because there are only a few shortest paths to those. Cells further away have more possible paths, so vision bending can still occur a bit at long distances. This won't be a problem if vision ranges remain short and nearby elevation differences aren't too great.

Finally, add the view elevation to the provided range, instead of replacing it. The unit's inherent range represents its own height, flight altitude, or scouting potential.

```
		range += fromCell.ViewElevation;
```

Vision with full range, using a lower view point.

So our final vision rules are for sight to travel along shortest paths up to the vision range, modified by the view elevation difference of cells, relative to the view point. When a cell is out of range, it blocks all paths through it. The result is that high vantage points with unblocked vision are strategically valuable.

What about features blocking vision?

I decided to not have features impact vision, but you could for example have dense forests or walls add effective elevation to a cell. This does make the vision rules harder for players to estimate.

3 Inexplorable Cells

One final issue about vision concerns the edge of the map. The terrain suddenly ends, without transition, where cells on the edge lack neighbors.

Obvious map edge.

Ideally, the visuals of unexplored regions and the map edge are identical. We could achieve this by adding special cases for triangulating edges when there are no neighbors, but that requires extra logic, and we have to deal with missing cells. So that's not trivial. An alternative is to force the border cells of the map to remain unexplored, even if they would be in sight range of a unit. That approach is much simpler, so let's go with that. It also makes it possible to mark other cells inexplorable as well, making irregular map edges easier to achieve. Besides that, hidden edge cells make is possible to make rivers and roads appear to enter and leave the map, because their end points are out of sight. You could also have units enter or leave the map this way.

3.1 Marking Cells as Explorable

To indicate that a cell is explorable, add an `Explorable` property to `HexCell`.

```
	public bool Explorable { get; set; }
```

Now a cell can only be visible if it is also explorable, so adjust the `IsVisible` property to take this into account.

```
	public bool IsVisible {
		get {
			return visibility > 0 && Explorable;
		}
	}
```

The same is true for `IsExplored`. However, we used a default property for that. We have to convert it into an explicit property to be able to adjust its getter logic.

```
	public bool IsExplored {
		get {
			return explored && Explorable;
		}
		private set {
			explored = value;
		}
	}

	…

	bool explored;
```

3.2 Hiding the Map Edge

Hiding the edge of our rectangular map can be done in the `HexGrid.CreateCell` method. Cells that's aren't at the edge are explorable, while all others are inexplorable.

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

		HexCell cell = cells[i] = Instantiate<HexCell>(cellPrefab);
		cell.transform.localPosition = position;
		cell.coordinates = HexCoordinates.FromOffsetCoordinates(x, z);
		cell.Index = i;
		cell.ShaderData = cellShaderData;

		cell.Explorable =
			x > 0 && z > 0 && x < cellCountX - 1 && z < cellCountZ - 1;

		…
	}
```

Now our maps fade out at their edge, the great beyond being inexplorable. As a result, the explorable size of our maps are reduced by two in each dimension.

inexplorable map edge.

What about making the explorable state editable?

That is possible, giving you maximum flexibility. Then you also have to add it to the save data.

3.3 Inexplorable Cells Block Vision

Finally, if a cell cannot be explored then it should also block vision. Adjust `HexGrid.GetVisibleCells` to take this into account.

```
			if (
				neighbor == null ||
				neighbor.SearchPhase > searchFrontierPhase ||
				!neighbor.Explorable
			) {
				continue;
			}
```

The next tutorial is Generating Land.

made by Jasper Flick