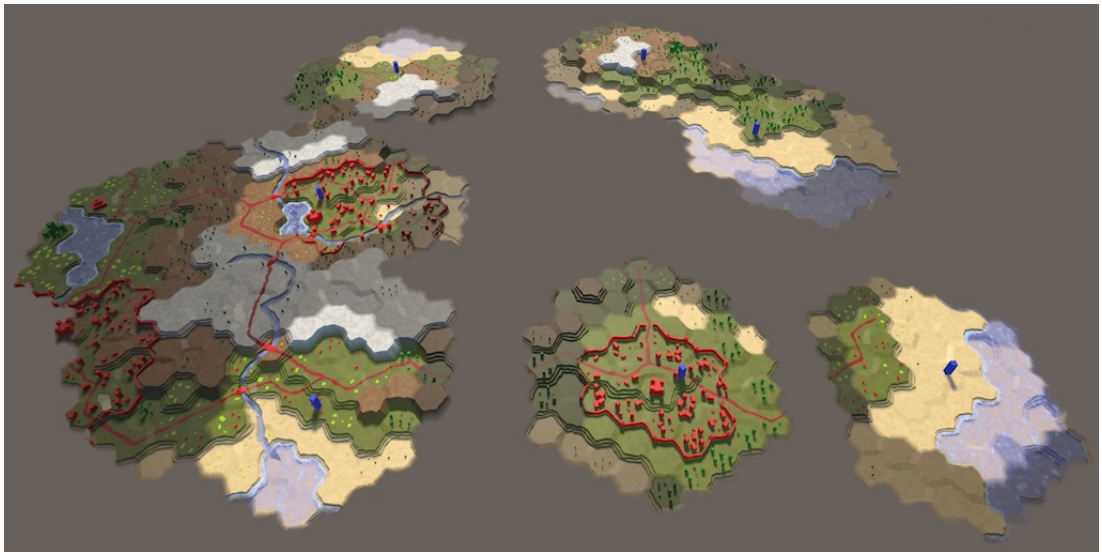
Hex Map 21 Exploration

See everything while editing.
Keep track of explored cells.
Hide what's yet unknown.
Have units avoid unexplored regions.

This is part 21 of a tutorial series about hexagon maps. The previous part added fog of war, which we'll now upgrade to support exploration.



We have some exploring to do.

1 Seeing Everything in Edit Mode

The idea of exploration is that cells that have not yet been seen are unknown, and thus invisible. Instead of darkening those cells, they shouldn't be shown at all. But it's hard to edit invisible cells. So before we add support for exploration, we're going to disable visibility while in edit mode.

1.1 Toggling Visibility

We can control whether shaders apply visibility via a keyword, like we do for the grid overlay. Let's use the *HEX_MAP_EDIT_MODE* keyword to indicated whether we are in edit mode. Because multiple shaders will need to be aware of this keyword, we'll define it globally, via the static `Shader.EnableKeyWord` and `Shader.DisableKeyword` methods. Invoke the appropriate one when the edit mode is changed, in `HexGameUI.SetEditMode`.

```
	public void SetEditMode (bool toggle) {
		enabled = !toggle;
		grid.ShowUI(!toggle);
		grid.ClearPath();
		if (toggle) {
			Shader.EnableKeyword("HEX_MAP_EDIT_MODE");
		}
		else {
			Shader.DisableKeyword("HEX_MAP_EDIT_MODE");
		}
	}
```

1.2 Edit Mode Shaders

When *HEX_MAP_EDIT_MODE* is defined, the shaders should ignore visibility. This boils down to always treating the visibility of a cell as 1. Let's add a function to the top of our *HexCellData* include file to filter cell data based on the keyword.

```
sampler2D _HexCellData;
float4 _HexCellData_TexelSize;

float4 FilterCellData (float4 data) {
	#if defined(HEX_MAP_EDIT_MODE)
		data.x = 1;
	#endif
	return data;
}
```

Pass the result of both `GetCellData` functions through this function before returning it.

```
float4 GetCellData (appdata_full v, int index) {
	…
	return FilterCellData(data);
}

float4 GetCellData (float2 cellDataCoordinates) {
	…
	return FilterCellData(tex2Dlod(_HexCellData, float4(uv, 0, 0)));
}
```

To make this work, all relevant shaders should get a multi-compile directive to create variants for when the *HEX_MAP_EDIT_MODE* keyword is defined. Add the following line to the *Estuary*, *Feature*, *River*, *Road*, *Terrain*, *Water*, and *Water Shore* shaders, between the target directive and the first include directive.

```
		#pragma multi_compile _ HEX_MAP_EDIT_MODE
```

Now the fog of war will disappear when we switch to map-editing mode.

2 Exploring Cells

By default, cells should be unexplored. They become explored as soon as a unit sees them. From then on they remain explored, regardless whether a unit can see them.

2.1 Keeping Track of Exploration

To support keeping track of the exploration state, add a public `IsExplored` property to `HexCell`.

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

Whether a cell is explored is determined by the cell itself. So only `HexCell` should be able to set this property. To enforce this, make the setter private.

```
	public bool IsExplored { get; private set; }
```

The first time a cell's visibility goes above zero, the cell is explored, and thus `IsExplored` should be set to `true`. Actually, we can suffice by simply always marking the cell as explored whenever visibility increases to 1. This should be done before invoking `RefreshVisibility`.

```
	public void IncreaseVisibility () {
		visibility += 1;
		if (visibility == 1) {
			IsExplored = true;
			ShaderData.RefreshVisibility(this);
		}
	}
```

2.2 Passing Exploration to Shaders

Like a cell's visibility, we can send its exploration state to the shaders via the shader data. It's another type of visibility, after all. `HexCellShaderData.RefreshVisibility` stores the visibility state in the data's R channel. Let's store the exploration state in the data's G channel.

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

2.3 Black Unexplored Terrain

Now we can use the shaders to visualize the exploration state of cells. To verify that it works as intended, we'll simply make unexplored terrain black. But first, to keep edit mode functional, adjust `FilterCellData` so it also filters the exploration data.

```
float4 FilterCellData (float4 data) {
	#if defined(HEX_MAP_EDIT_MODE)
		data.xy = 1;
	#endif
	return data;
}
```

The *Terrain* shader sends the visibility data of all three potential cells to the fragment program. In the case of exploration state, we'll combine them in the vertex program and send a single value to the fragment program. Add a fourth component to the `visibility` input data to make room for this.

```
		struct Input {
			float4 color : COLOR;
			float3 worldPos;
			float3 terrain;
			float4 visibility;
		};
```

In the vertex program, we now have to explicitly access `data.visibility.xyz` when adjusting the visibility factor.

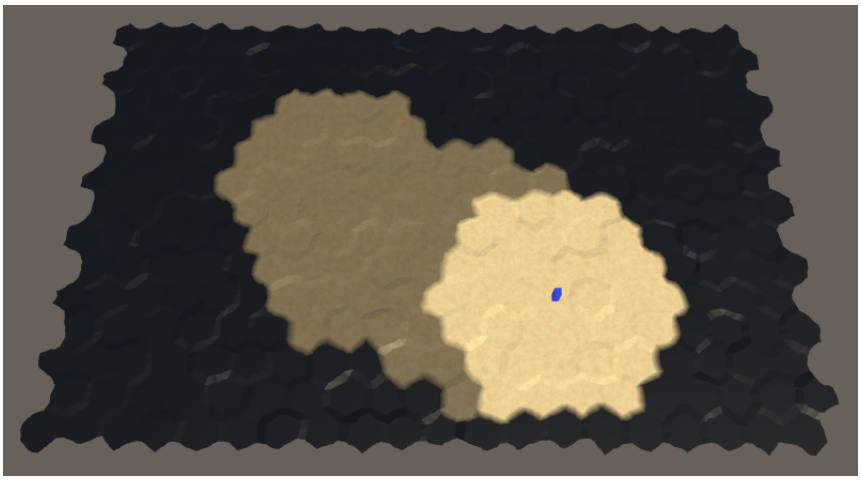
```
		void vert (inout appdata_full v, out Input data) {
			…
			data.visibility.xyz = lerp(0.25, 1, data.visibility.xyz);
		}
```

After that, combine the exploration states and put the result in `data.visibility.w`. This is done like combining the visibility in the other shaders, but using the Y component of the cell data.

```
			data.visibility.xyz = lerp(0.25, 1, data.visibility.xyz);
			data.visibility.w =
				cell0.y * v.color.x + cell1.y * v.color.y + cell2.y * v.color.z;
```

In the fragment program, the explorations state is now available via `IN.visibility.w`. Factor it into the albedo.

```
	void surf (Input IN, inout SurfaceOutputStandard o) {
		…

		float explored = IN.visibility.w;
		o.Albedo = c.rgb * grid * _Color * explored;
		o.Metallic = _Metallic;
		o.Smoothness = _Glossiness;
		o.Alpha = c.a;
	}
```



Unexplored terrain is black.

The terrain of unexplored cells in now black. Features, roads, and water aren't affected yet. This is enough to verify that exploration works.

2.4 Saving and Loading Exploration

Now that we support exploration, we should also make sure that the exploration state of cells is included when saving and loading maps. So we have to increase the map file version to 3. To make these changes more convenient, let's add a constant for this to `SaveLoadMenu`.

```
	const int mapFileVersion = 3;
```

Use this constant when writing the file version in `Save` and to check whether a file is supported in `Load`.

```
	void Save (string path) {
		using (
			BinaryWriter writer =
			new BinaryWriter(File.Open(path, FileMode.Create))
		) {
			writer.Write(mapFileVersion);
			hexGrid.Save(writer);
		}
	}

	void Load (string path) {
		if (!File.Exists(path)) {
			Debug.LogError("File does not exist " + path);
			return;
		}
		using (BinaryReader reader = new BinaryReader(File.OpenRead(path))) {
			int header = reader.ReadInt32();
			if (header <= mapFileVersion) {
				hexGrid.Load(reader, header);
				HexMapCamera.ValidatePosition();
			}
			else {
				Debug.LogWarning("Unknown map format " + header);
			}
		}
	}
```

In `HexCell.Save`, we'll write the exploration state as the last step.

```
	public void Save (BinaryWriter writer) {
		…
		writer.Write(IsExplored);
	}
```

And read it at the end of `Load`. After that, invoke `RefreshVisibility` in case the exploration state is now different than before.

```
	public void Load (BinaryReader reader) {
		…

		IsExplored = reader.ReadBoolean();
		ShaderData.RefreshVisibility(this);
	}
```

To remain backwards compatible with older save files, we should skip reading the explored state when the file version is less than 3. Let's default to unexplored when that's the case. To be able to do this, we have to add the header data as a parameter to `Load`.

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

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

From now on, `HexGrid.Load` has to pass the header data on to `HexCell.Load`.

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

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

Whether cells are explored is now included when saving and loading maps.

3 Hiding Unknown Cells

Currently, unexplored cells are visually indicated by giving them a solid black terrain. What we really want is for those cells to be invisible, because they are unknown. It is possible to make normally opaque geometry transparent, so it can no longer be seen. However, we're using Unity's surface shader framework which is not designed with this effect in mind. Instead of going for actual transparency, we'll adapt our shaders to match the background so they're also unnoticeable.

3.1 Making the Terrain Truly Black

Although unexplored terrain is solid black, we can still determine its features because it still has specular lighting. To get rid of the highlights we have to make it perfectly matte black. To do this without messing with other surface properties, it's easiest to simply fade the specular color to black. This is possible when using a surface shader with the specular workflow, but we're currently using the default metallic workflow. So let's begin by switching the *Terrain* shader to the specular workflow.

Replace the *_Metallic* property with a *_Specular* color property. Its default color value should be (0.2, 0.2, 0.2). This makes sure that it matches the appearance of the metallic version.

```
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Terrain Texture Array", 2DArray) = "white" {}
		_GridTex ("Grid Texture", 2D) = "white" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
//		_Metallic ("Metallic", Range(0,1)) = 0.0
		_Specular ("Specular", Color) = (0.2, 0.2, 0.2)
	}
```

Change the corresponding shader variables as well. The specular color of surface shaders is defined as a `fixed3`, so let's use that as well.

```
		half _Glossiness;
//		half _Metallic;
		fixed3 _Specular;
		fixed4 _Color;
```

Change the surface surf pragma from *Standard* to *StandardSpecular*. This causes Unity to generate shaders using the specular workflow.

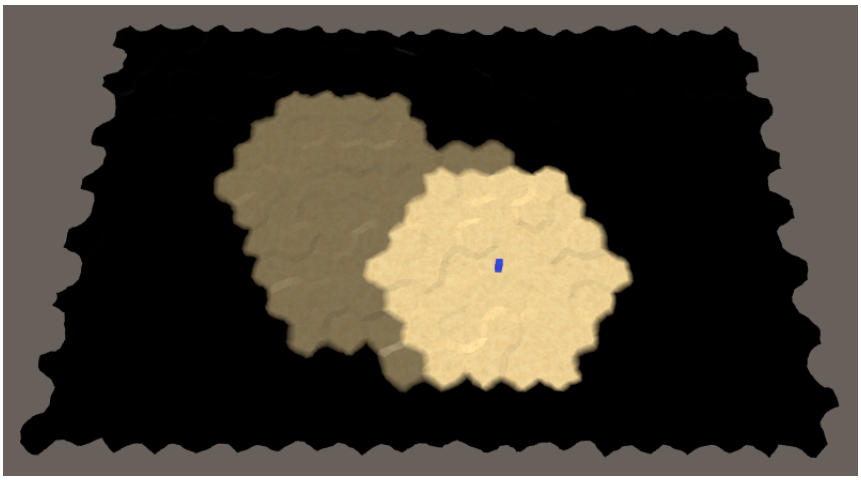
```
		#pragma surface surf StandardSpecular fullforwardshadows vertex:vert
```

The `surf` function now requires its second parameter to be of the type `SurfaceOutputStandardSpecular`. Also, we should assign to `o.Specular` instead of to `o.Metallic`.

```
		void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
			…

			float explored = IN.visibility.w;
			o.Albedo = c.rgb * grid * _Color * explored;
//			o.Metallic = _Metallic;
			o.Specular = _Specular;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
```

Now we can fade out the highlights by factoring `explored` into the specular color.

```
			o.Specular = _Specular * explored;
```

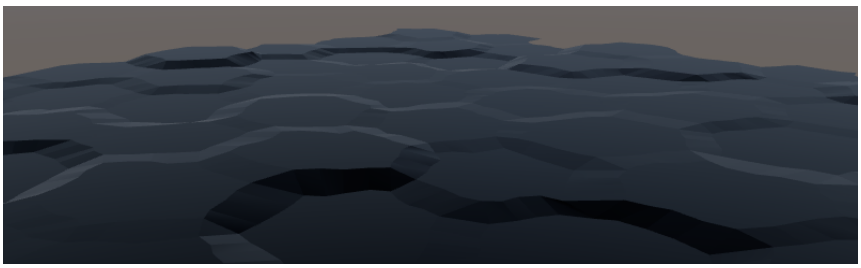


Unexplored without specular lighting.

When seen from above, unexplored terrain now appears matte black. However, when viewed at grazing angles surfaces become mirrors, which causes the terrain to reflect the environment, which is the skybox.

Why do surfaces become mirrors?

This is known as the Fresnel effect. See the Rendering series for more information.



Unexplored still reflects the environment.

To get rid of these reflections, treat unexplored terrain as fully occluded. This is done by using `explored` as the occlusion value, which acts as a mask for reflections.

```
            float explored = IN.visibility.w;
            o.Albedo = c.rgb * grid * _Color * explored;
            o.Specular = _Specular * explored;
            o.Smoothness = _Glossiness;
            o.Occlusion = explored;
            o.Alpha = c.a;
```

Unexplored without reflections.

3.2 Matching the Background

Now that unexplored terrain ignores all lighting, the next step is to make it match the background. As our camera is always looking from above, the background is always grey. To tell the *Terrain* shader which color to use, add a *_BackgroundColor* property to it, with black as default.

```
    Properties {
        …
        _BackgroundColor ("Background Color", Color) = (0,0,0)
    }

    …

        half _Glossiness;
        fixed3 _Specular;
        fixed4 _Color;
        half3 _BackgroundColor;
```

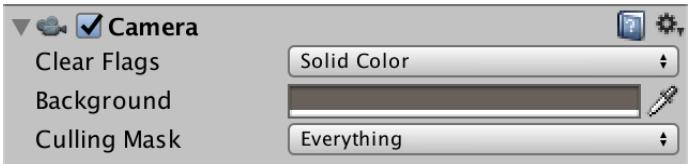
To use this color, we add it as emissive light. This is done by assigning the background color multiplied by one-minus-explored to `o.Emission`.

```
		o.Occlusion = explored;
		o.Emission = _BackgroundColor * (1 -  explored);
```

Because we're using the default skybox, the visible background color isn't actually uniform. Overall the best color is slightly reddish grey. You can use the *Hex Color* code 68615BFF when adjusting the terrain material.

Terrain material with gray background color.

This mostly works, although you could still perceive very faint silhouettes if you know where to look. To ensure that this is not possible for the player, you can adjust the camera to use 68615BFF as its solid background color, instead of the skybox.

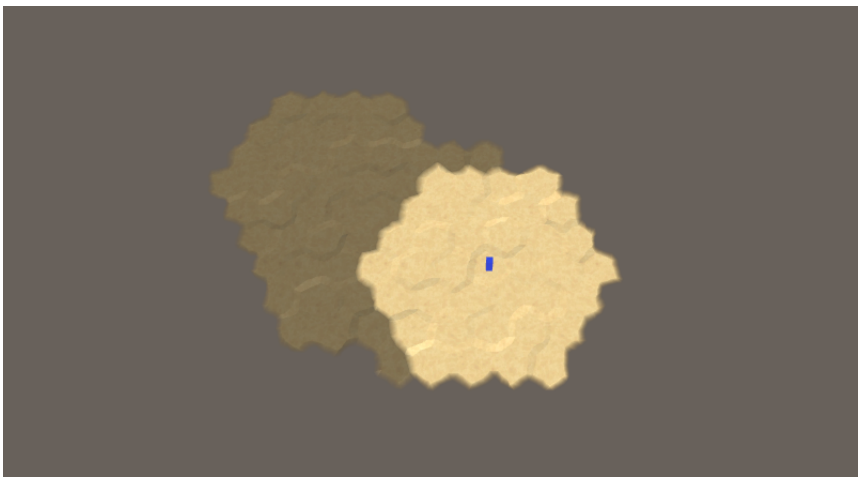


Camera with solid background color.

Why not remove the skybox?

You could do that, but keep in mind that it is used for environmental lighting of the map. If you switch it to a solid color, the lighting of the map will change as well.

Now we're no longer able to distinguish between the background and unexplored cells. It is still possible for high unexplored terrain to occlude low explored terrain, when using low camera angles. Also, the unexplored parts still cast shadows on the explored parts. These minimal clues are fine.



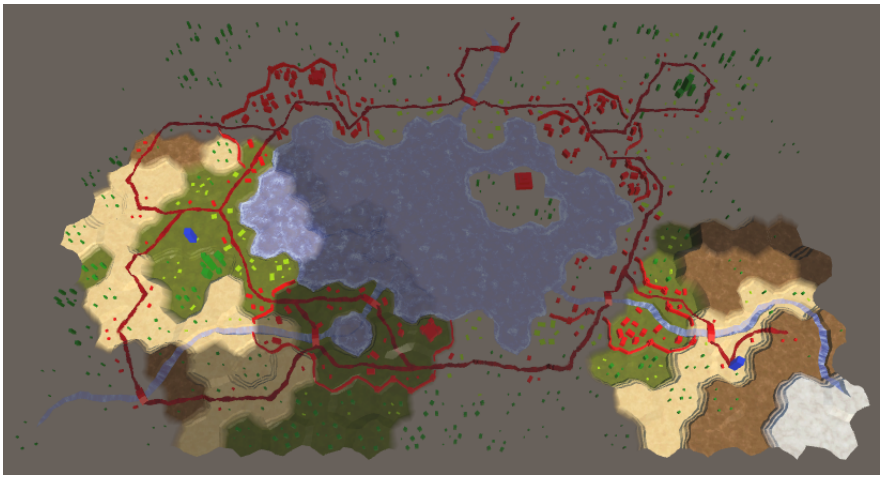
Unexplored cells no longer visible.

What if I'm not using a solid background color?

You could create your own shader which actually makes the terrain transparent, while still writing to the depth buffer, which might require some shader queue tricks. If you're using a screen-space texture, you can simply sample this texture instead of using a background color. If you're using a texture in world space, underneath the terrain, you'll have to use some math to figure out which texture UV coordinates to use, based on the view angle and fragment's world position.

3.3 Hiding Features

At this point only the terrain mesh is hidden. Everything else is still unaffected by the exploration state.



Only terrain is hidden so far.

Let's adjust the *Feature* shader next, which is an opaque shader like *Terrain*. Turn it into a specular shader and add a background color to it. First, the properties.

```
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
//		_Metallic ("Metallic", Range(0,1)) = 0.0
		_Specular ("Specular", Color) = (0.2, 0.2, 0.2)
		_BackgroundColor ("Background Color", Color) = (0,0,0)
		[NoScaleOffset] _GridCoordinates ("Grid Coordinates", 2D) = "white" {}
	}
```

Next, the surface pragma and variables, like before.

```
		#pragma surface surf StandardSpecular fullforwardshadows vertex:vert

		…

		half _Glossiness;
//		half _Metallic;
		fixed3 _Specular;
		fixed4 _Color;
		half3 _BackgroundColor;
```

Again, `visibility` needs another component. Because *Feature* combines the visibility per vertex, it only needed a single float. Now we need two.

```
		struct Input {
			float2 uv_MainTex;
			float2 visibility;
		};
```

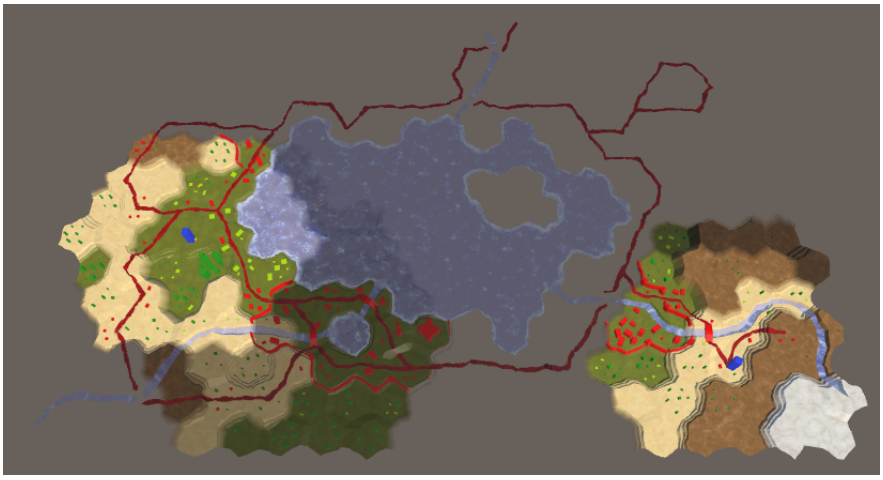
Adjust `vert` so it explicitly uses `data.visibility.x` for the visibility data, then assign the exploration data to `data.visibility.y`.

```
		void vert (inout appdata_full v, out Input data) {
			…

			float4 cellData = GetCellData(cellDataCoordinates);
			data.visibility.x = cellData.x;
			data.visibility.x = lerp(0.25, 1, data.visibility.x);
			data.visibility.y = cellData.y;
		}
```

Adjust `surf` so it uses the new data, like *Terrain*.

```
		void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
			float explored = IN.visibility.y;
			o.Albedo = c.rgb * (IN.visibility.x * explored);
//			o.Metallic = _Metallic;
			o.Specular = _Specular * explored;
			o.Smoothness = _Glossiness;
			o.Occlusion = explored;
			o.Emission = _BackgroundColor * (1 -  explored);
			o.Alpha = c.a;
		}
```

Adjust the features so they use materials with the proper settings.



Hidden features.

3.4 Hiding Water

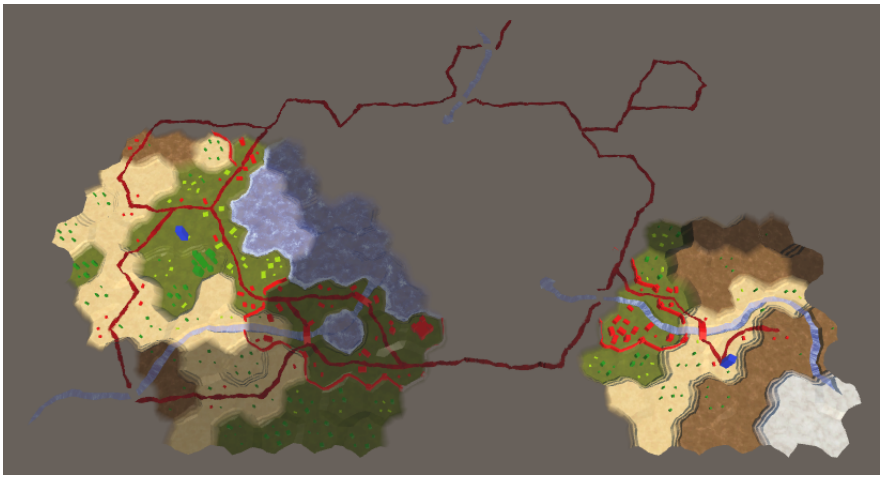
Next up are the *Water* and *Water Shore* shaders. Begin by converting them to specular shaders, to stay consistent. However, they do not need a background color, because they are transparent shaders.

After the conversion, add another component to `visibility` and adjust `vert` accordingly. Both shaders combine the data of three cells.

```
		struct Input {
			…
			float2 visibility;
		};

		…

		void vert (inout appdata_full v, out Input data) {
			…

			data.visibility.x =
				cell0.x * v.color.x + cell1.x * v.color.y + cell2.x * v.color.z;
			data.visibility.x = lerp(0.25, 1, data.visibility.x);
			data.visibility.y =
				cell0.y * v.color.x + cell1.y * v.color.y + cell2.y * v.color.z;
		}
```

Water and *Water Shore* do different things in `surf`, but they set their surface properties in the same way. Because they're transparent, factor `explore` into alpha instead of setting emission.

```
		void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
			…

			float explored = IN.visibility.y;
			o.Albedo = c.rgb * IN.visibility.x;
			o.Specular = _Specular * explored;
			o.Smoothness = _Glossiness;
			o.Occlusion = explored;
			o.Alpha = c.a * explored;
		}
```



Hidden water.

3.5 Hiding Estuaries, Rivers, and Roads

The remaining shaders are *Estuary*, *River*, and *Road*. All three are transparent shaders and combine the data of two cells. Switch all of them to the specular workflow. Then add the explored data to `visibility`.

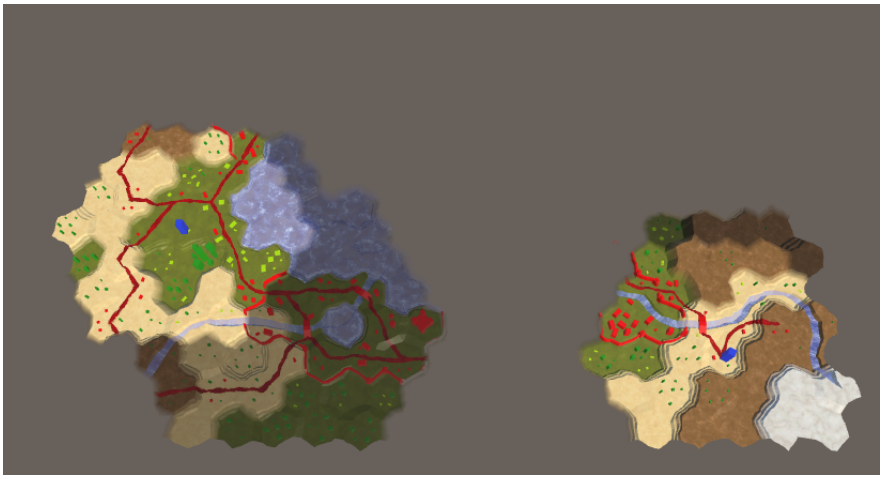
```
		struct Input {
			…
			float2 visibility;
		};

		…

		void vert (inout appdata_full v, out Input data) {
			…

			data.visibility.x = cell0.x * v.color.x + cell1.x * v.color.y;
			data.visibility.x = lerp(0.25, 1, data.visibility.x);
			data.visibility.y = cell0.y * v.color.x + cell1.y * v.color.y;
		}
```

Adjust the `surf` function of the *Estuary* and *River* shaders so it uses the new data. They both require the same changes.

```
		void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
			…

			float explored = IN.visibility.y;
			fixed4 c = saturate(_Color + water);
			o.Albedo = c.rgb * IN.visibility.x;
			o.Specular = _Specular * explored;
			o.Smoothness = _Glossiness;
			o.Occlusion = explored;
			o.Alpha = c.a * explored;
		}
```

The *Road* shader is a little different, because it uses an additional blend factor.

```
		void surf (Input IN, inout SurfaceOutputStandardSpecular o) {
			float4 noise = tex2D(_MainTex, IN.worldPos.xz * 0.025);
			fixed4 c = _Color * ((noise.y * 0.75 + 0.25) * IN.visibility.x);
			float blend = IN.uv_MainTex.x;
			blend *= noise.x + 0.5;
			blend = smoothstep(0.4, 0.7, blend);

			float explored = IN.visibility.y;
			o.Albedo = c.rgb;
			o.Specular = _Specular * explored;
			o.Smoothness = _Glossiness;
			o.Occlusion = explored;
			o.Alpha = blend * explored;
		}
```



Everything hidden.

4 Avoiding Unexplored Cells

Although everything that's yet unknown is visually hidden, pathfinding doesn't take the exploration state into consideration yet. As a result, units can be ordered to move into or through unexplored cells, magically knowing which path to take. We should force units to avoid unexplored cells.

Moving through unexplored cells.

4.1 Units Determine Move Cost

Before dealing with the unexplored cells, let's migrate the code to determine the cost of movement from `HexGrid` to `HexUnit`. That makes it easier to support units with varying movement rules in the future.

Add a public `GetMoveCost` method to `HexUnit` to determine the move cost. It needs to know which cells the movement is between as well as the direction. Copy the relevant move-cost code from `HexGrid.Search` into this method and adjust the variable names.

```
	public int GetMoveCost (
		HexCell fromCell, HexCell toCell, HexDirection direction)
	{
		HexEdgeType edgeType = fromCell.GetEdgeType(toCell);
		if (edgeType == HexEdgeType.Cliff) {
			continue;
		}
		int moveCost;
		if (fromCell.HasRoadThroughEdge(direction)) {
			moveCost = 1;
		}
		else if (fromCell.Walled != toCell.Walled) {
			continue;
		}
		else {
			moveCost = edgeType == HexEdgeType.Flat ? 5 : 10;
			moveCost +=
				toCell.UrbanLevel + toCell.FarmLevel + toCell.PlantLevel;
		}
	}
```

The method should return the move cost. While the old code uses `continue` to skip invalid moves, we cannot use this approach here. Instead, we'll return a negative move cost to indicate that movement is not possible.

```
	public int GetMoveCost (
		HexCell fromCell, HexCell toCell, HexDirection direction)
	{
		HexEdgeType edgeType = fromCell.GetEdgeType(toCell);
		if (edgeType == HexEdgeType.Cliff) {
			return -1;
		}
		int moveCost;
		if (fromCell.HasRoadThroughEdge(direction)) {
			moveCost = 1;
		}
		else if (fromCell.Walled != toCell.Walled) {
			return -1;
		}
		else {
			moveCost = edgeType == HexEdgeType.Flat ? 5 : 10;
			moveCost +=
				toCell.UrbanLevel + toCell.FarmLevel + toCell.PlantLevel;
		}
		return moveCost;
	}
```

Now we need to know the selected unit when finding a path, instead of only a speed. Adjust `HexGameUI.DoPathFinding` accordingly.

```
	void DoPathfinding () {
		if (UpdateCurrentCell()) {
			if (currentCell && selectedUnit.IsValidDestination(currentCell)) {
				grid.FindPath(selectedUnit.Location, currentCell, selectedUnit);
			}
			else {
				grid.ClearPath();
			}
		}
	}
```

As we still need to access the unit's speed, add a `Speed` property to `HexUnit`. It just returns the constant value 24 for now.

```
	public int Speed {
		get {
			return 24;
		}
	}
```

In `HexGrid`, adjust `FindPath` and `Search` so they work with the new approach.

```
	public void FindPath (HexCell fromCell, HexCell toCell, HexUnit unit) {
		ClearPath();
		currentPathFrom = fromCell;
		currentPathTo = toCell;
		currentPathExists = Search(fromCell, toCell, unit);
		ShowPath(unit.Speed);
	}

	bool Search (HexCell fromCell, HexCell toCell, HexUnit unit) {
		int speed = unit.Speed;
		…
	}
```

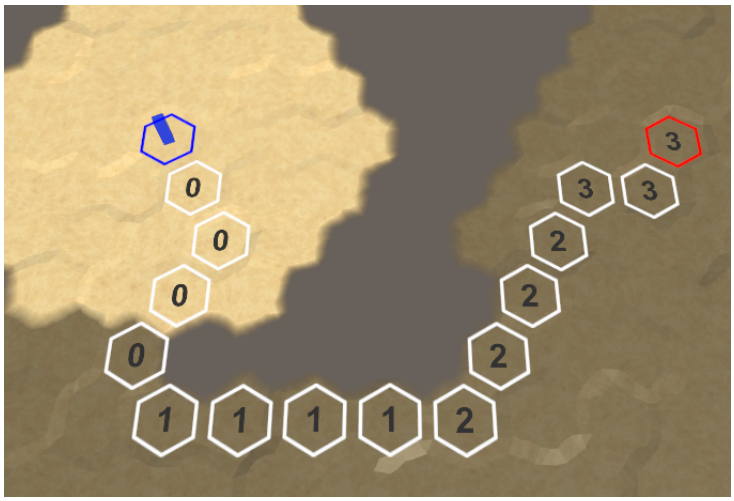
Finally, remove the old code in `Search` that determines whether a neighbor can be moved into and what the move cost is. Instead, invoke `HexUnit.IsValidDestination` and `HexUnit.GetMoveCost`. Skip the cell if the move cost ends up negative.

```
            for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
                HexCell neighbor = current.GetNeighbor(d);
                if (
                    neighbor == null ||
                    neighbor.SearchPhase > searchFrontierPhase
                ) {
                    continue;
                }
//                  if (neighbor.IsUnderwater || neighbor.Unit) {
//                      continue;
//                  }
//                  HexEdgeType edgeType = current.GetEdgeType(neighbor);
//                  if (edgeType == HexEdgeType.Cliff) {
//                      continue;
//                  }
//                  int moveCost;
//                  if (current.HasRoadThroughEdge(d)) {
//                      moveCost = 1;
//                  }
//                  else if (current.Walled != neighbor.Walled) {
//                      continue;
//                  }
//                  else {
//                      moveCost = edgeType == HexEdgeType.Flat ? 5 : 10;
//                      moveCost += neighbor.UrbanLevel + neighbor.FarmLevel +
//                          neighbor.PlantLevel;
//                  }
                if (!unit.IsValidDestination(neighbor)) {
                    continue;
                }
                int moveCost = unit.GetMoveCost(current, neighbor, d);
                if (moveCost < 0) {
                    continue;
                }

                int distance = current.Distance + moveCost;
                int turn = (distance - 1) / speed;
                if (turn > currentTurn) {
                    distance = turn * speed + moveCost;
                }

                …
            }
```

4.2 Moving Around Unexplored Areas

To avoid unexplored cells, we just have to make `HexUnit.IsValidDestination` check whether the cell is explored.

```
	public bool IsValidDestination (HexCell cell) {
		return cell.IsExplored && !cell.IsUnderwater && !cell.Unit;
	}
```



Units can no longer enter unexplored cells.

As unexplored cells are no longer valid destinations, units will avoid them when moving to a destination. Unexplored regions thus act as barriers, which can make a path longer or even impossible. You'll have to move units close to unknown terrain to explore the area first.

What if a shorter path becomes available during movement?

Our approach determines the path to take once and doesn't deviate from it while traveling. You could change this to looking for a new path after each step, but that could make unit movement unpredictable and erratic. It's best to stick to the path that was chosen instead of trying to be smart.

Having said that, when strictly enforcing single-turn movement it becomes necessary to find a new path every turn for long-range travel. In that case units will switch to a shorter path on their next turn, if one has become available.

The next tutorial is Advanced Vision.

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick