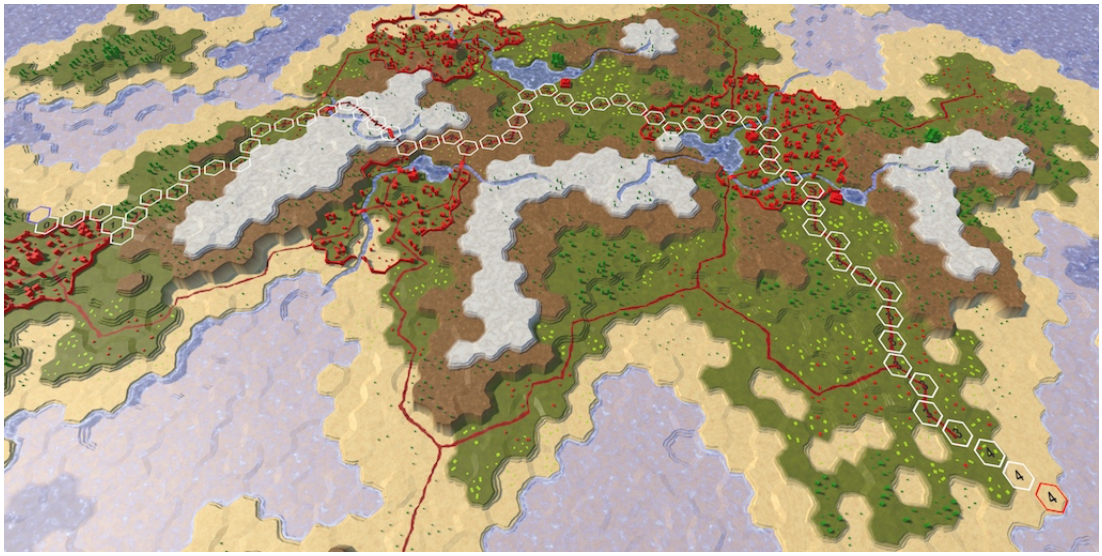
Hex Map 17 Limited Movement

Finds paths for turn-based movement.
Immediately show paths.
Search more efficiently.
Only visualize the path.

This is part 17 of a tutorial series about hexagon maps. This time, we'll split movement into turns and search as quickly as possible.



A journey of a few turns.

1 Turn-based Movement

Strategy games that use hexagon grids are nearly always turn-based. Units that navigate the map have a limited speed, which constrains how far they can move in a single turn.

1.1 Speed

To support limited movement, let's add a `speed` integer parameter to `HexGrid.FindPath` and `HexGrid.Search`. It defines the movement budget for one turn.

```
	public void FindPath (HexCell fromCell, HexCell toCell, int speed) {
		StopAllCoroutines();
		StartCoroutine(Search(fromCell, toCell, speed));
	}

	IEnumerator Search (HexCell fromCell, HexCell toCell, int speed) {
		…
	}
```

In a game, different unit types will often have different speeds. Cavalry is fast, infantry is slow, and so on. We don't have units yet, so we'll use a fixed speed for now. Let's use 24. This is a reasonably large value which isn't divisible by 5, which is our default movement cost. Add the constant speed as an argument for `FindPath` in `HexMapEditor.HandleInput`.

```
			if (editMode) {
				EditCells(currentCell);
			}
			else if (
				Input.GetKey(KeyCode.LeftShift) && searchToCell != currentCell
			) {
				if (searchFromCell) {
					searchFromCell.DisableHighlight();
				}
				searchFromCell = currentCell;
				searchFromCell.EnableHighlight(Color.blue);
				if (searchToCell) {
					hexGrid.FindPath(searchFromCell, searchToCell, 24);
				}
			}
			else if (searchFromCell && searchFromCell != currentCell) {
				searchToCell = currentCell;
				hexGrid.FindPath(searchFromCell, searchToCell, 24);
			}
```

1.2 Turns

Besides keeping track of the total movement cost of a path, we now also have to know how many turns it requires to travel along it. But we do not have to store this information per cell. We can derive it by dividing the traveled distance by the speed. As these are integers, we use an integer division. So total distances of at most 24 correspond to turn 0. This means that the entire path can be traveled in the current turn. If the destination were at distance 30, then its turn would be 1. The unit will have to use all of its movement of its current turn and part of its next turn before it would reach the destination.

Let's figure out the turn of the current cell and that of its neighbors, inside `HexGrid.Search`. The current cell's turn can be computed once, just before looping through the neighbors. The neighbor's turn can be determined once we have found its distance.

```
		int currentTurn = current.Distance / speed;

		for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
			…
			int distance = current.Distance;
			if (current.HasRoadThroughEdge(d)) {
				distance += 1;
			}
			else if (current.Walled != neighbor.Walled) {
				continue;
			}
			else {
				distance += edgeType == HexEdgeType.Flat ? 5 : 10;
				distance += neighbor.UrbanLevel + neighbor.FarmLevel +
					neighbor.PlantLevel;
			}

			int turn = distance / speed;

			…
		}
```

1.3 Lost Movement

If the neighbor's turn is larger than the current turn, then we have passed a turn boundary. If the movement required to enter the neighbor was 1, then all is fine. But when it's more costly to enter the neighbor, then it gets more complicated.

Suppose that we're moving across a featureless map, so all cells require 5 movement to enter. Our speed is 24. After four steps, we have used 20 of our movement budget, with 4 remaining. The next step again requires 5 movement, which is one more than we have. What should we do at this point?

There are two ways to deal with this situation. The first is to allow the fifth cell to be entered during the current turn, even if we don't have enough movement. The second is to disallow the movement during the current turn, which means that the leftover movement points cannot be used and are lost.

Which option is best depends on the game. In general, the first approach works well for games with units that can only move a few steps per turn, like the Civilization games. This ensures that units can always move at least a single cell per turn. If the units can move many cells per turn, like in Age of Wonders or Battle for Wesnoth, then the second approach works better.

As we use speed 24, let's go for the second option. To make this work, we have to isolate the cost to the enter the neighboring cell, before adding it to the current distance.

```
//                  int distance = current.Distance;
                    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;
                    }

                    int distance = current.Distance + moveCost;
                    int turn = distance / speed;
```

If we end up crossing a turn boundary, then we first use up all the movement of the current turn. We can do this by simply multiplying the turn by the speed. After that, we add the movement cost.

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

The result of this will be that we end the first turn in the fourth cell, with 4 movement points left unused. Those wasted points are factored into the cost of the fifth cell, so its distance will become 29 instead of 25. As as result, distances will end up longer than before. For example, the tenth cell used to have a distance of 50. But now two turn boundaries are crossed before we get there, wasting 8 movement points, so its distance is now 58.

Taking longer than expected.

Because the unused movement points are added to the cell distances, they are taken into consideration when determining the shortest path. The most efficient path wastes as few points as possible. So different speeds can result in different paths.

1.4 Showing Turns instead of Distances

When playing a game, we don't really care about the distance values used to find the shortest path. We are interested in how many turns are required to reach the destination. So let's shows the turns instead of the distances.

First, get rid of `UpdateDistanceLabel` and its invocation in **`HexCell`**.

```
	public int Distance {
		get {
			return distance;
		}
		set {
			distance = value;
//			UpdateDistanceLabel();
		}
	}

	…

//	void UpdateDistanceLabel () {
//		UnityEngine.UI.Text label = uiRect.GetComponent<Text>();
//		label.text = distance == int.MaxValue ? "" : distance.ToString();
//	}
```

In its place, add a public `SetLabel` method to **`HexCell`** which accept an arbitrary string.

```
	public void SetLabel (string text) {
		UnityEngine.UI.Text label = uiRect.GetComponent<Text>();
		label.text = text;
	}
```

Use this new method in **`HexGrid`**.`Search` when clearing the cells. To hide the labels, just set them to **`null`**.

```
		for (int i = 0; i < cells.Length; i++) {
			cells[i].Distance = int.MaxValue;
			cells[i].SetLabel(null);
			cells[i].DisableHighlight();
		}
```

Then set the neighbor's label to its turn. After that, we'll be able so see how many extra turns it would take to traverse the entire path.

```
				if (neighbor.Distance == int.MaxValue) {
					neighbor.Distance = distance;
					neighbor.SetLabel(turn.ToString());
					neighbor.PathFrom = current;
					neighbor.SearchHeuristic =
						neighbor.coordinates.DistanceTo(toCell.coordinates);
					searchFrontier.Enqueue(neighbor);
				}
				else if (distance < neighbor.Distance) {
					int oldPriority = neighbor.SearchPriority;
					neighbor.Distance = distance;
					neighbor.SetLabel(turn.ToString());
					neighbor.PathFrom = current;
					searchFrontier.Change(neighbor, oldPriority);
				}
```

Turns required to move along the path.

2 Instant Paths

When playing a game, we also don't care how the pathfinding algorithm finds its way. We want to immediately see the path that we request. By now we can be confident that our algorithm works, so let's get rid of the search visualization.

2.1 No More Coroutine

We used a coroutine to slowly step through our algorithm. We no longer have to do this, so get rid of the invocations of `StartCoroutine` and `StopAllCoroutines` in **`HexGrid`**. Instead, we simply invoke `Search` like a regular method.

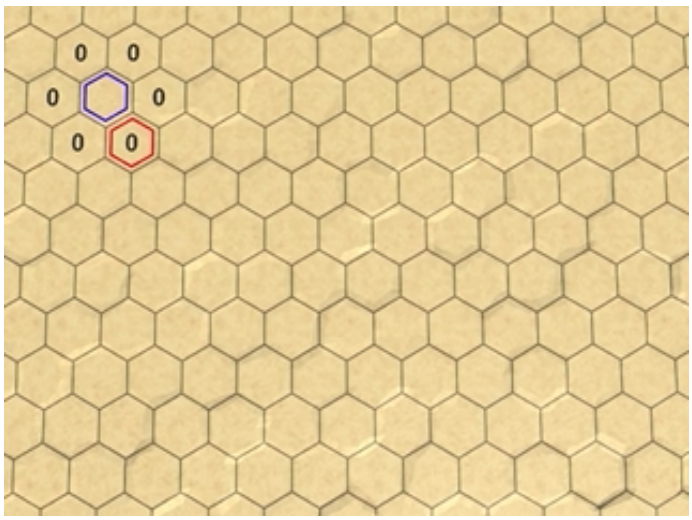
```
	public void Load (BinaryReader reader, int header) {
//		StopAllCoroutines();
		…
	}

	public void FindPath (HexCell fromCell, HexCell toCell, int speed) {
//		StopAllCoroutines();
//		StartCoroutine(Search(fromCell, toCell, speed));
		Search(fromCell, toCell, speed);
	}
```

As we're no longer using `Search` as a coroutine, it no longer has to yield, so get rid of that statement. That means we can also remove the **`WaitForSeconds`** declaration and change the method's return type to **`void`**.

```
	void Search (HexCell fromCell, HexCell toCell, int speed) {
		…

//		WaitForSeconds delay = new WaitForSeconds(1 / 60f);
		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
		while (searchFrontier.Count > 0) {
//			yield return delay;
			HexCell current = searchFrontier.Dequeue();
			…
		}
	}
```



Immediate results.

2.2 Timing Searches

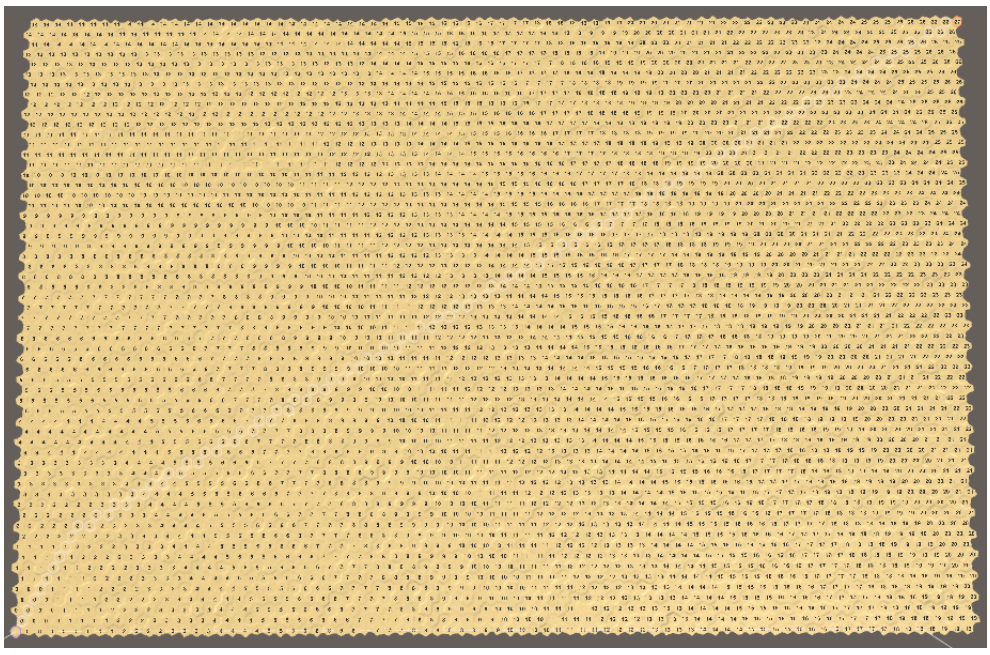
We now get immediate paths, but how quickly do we get them, really? Short paths appear seemingly instantaneous, but long paths on large maps might feel sluggish.

Let's measure how long it takes to find and display a path. We could use the profiler to time the search, but that's a bit overkill and generates additional overhead. Let's use a `Stopwatch` instead, which is found in the `System.Diagnostics` namespace. As we're only using it for a short while, I won't bother adding a `using` statement for it to the top of the script.

Right before performing the search, create a new stopwatch and start it. Once the search is finished, stop the stopwatch and log how many milliseconds have elapsed.

```
	public void FindPath (HexCell fromCell, HexCell toCell, int speed) {
		System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
		sw.Start();
		Search(fromCell, toCell, speed);
		sw.Stop();
		Debug.Log(sw.ElapsedMilliseconds);
	}
```

Let's use the worst case for our algorithm, which is a search from the bottom left to the top right of a large map. A featureless map is worst, because then the algorithm will have to process all of the 4,800 cells of the map.



Worst-case search.

How long it takes will vary, because the Unity editor is not the only process running on your machine. So try it a few times and to get an indication of the average duration. In my case, it takes about 45 milliseconds. That isn't very fast, corresponding to 22.22 paths per second, let's say 22 pps. This means that the game's frame rate will also drop to at most 22 fps on the frame that this path is being computed. And that ignores all the other work that has to be done, like actually rendering the frame. So we get a pretty significant frame rate drop, below 20 fps.

When performing performance tests like these, keep in mind that the performance in the Unity editor will not be as good as the performance of a stand-alone app. If I perform the same test with a build, it only takes 15 milliseconds on average. That's 66 pps, which is a lot better. However, that's still a large chunk of the frame budget, which will drop the frame rate below 60 fps.

Where can I see the debug log for a build?

Unity apps write to a log file, which is stored somewhere on your system. Its location depends on the platform. See Unity's Log Files documentation for for how to find the log files on your system.

2.3 Only Search When Needed

A quick optimization that we can do is to make sure that we only search when needed. Currently, we initiate a new search every frame that we hold the mouse button down. As a result the frame rate will be lowered consistently while dragging. We can prevent this by only initiating a new search in `HexMapEditor.HandleInput` when we're actually dealing with a new end point. If not, the currently visible path is still valid.

```
			if (editMode) {
				EditCells(currentCell);
			}
			else if (
				Input.GetKey(KeyCode.LeftShift) && searchToCell != currentCell
			) {
				if (searchFromCell != currentCell) {
					if (searchFromCell) {
						searchFromCell.DisableHighlight();
					}
					searchFromCell = currentCell;
					searchFromCell.EnableHighlight(Color.blue);
					if (searchToCell) {
						hexGrid.FindPath(searchFromCell, searchToCell, 24);
					}
				}
			}
			else if (searchFromCell && searchFromCell != currentCell) {
				if (searchToCell != currentCell) {
					searchToCell = currentCell;
					hexGrid.FindPath(searchFromCell, searchToCell, 24);
				}
			}
```

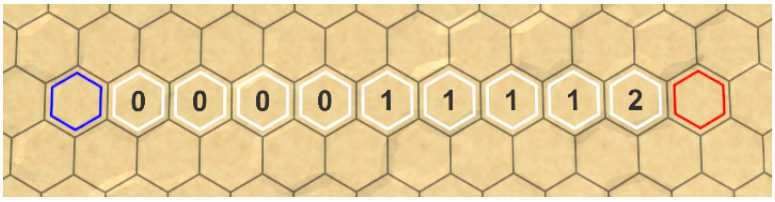
2.4 Only Show Labels on the Path

Displaying turn labels is fairly costly, especially because we're not using an optimized approach. Doing it for all cells surely slows us down. So let's omit setting the labels in `HexGrid.Search`.

```
                if (neighbor.Distance == int.MaxValue) {
                    neighbor.Distance = distance;
//                  neighbor.SetLabel(turn.ToString());
                    neighbor.PathFrom = current;
                    neighbor.SearchHeuristic =
                        neighbor.coordinates.DistanceTo(toCell.coordinates);
                    searchFrontier.Enqueue(neighbor);
                }
                else if (distance < neighbor.Distance) {
                    int oldPriority = neighbor.SearchPriority;
                    neighbor.Distance = distance;
//                  neighbor.SetLabel(turn.ToString());
                    neighbor.PathFrom = current;
                    searchFrontier.Change(neighbor, oldPriority);
                }
```

We really only need to see this information for the path that is found. So compute the turn and set the label of the cells on the path only, when the destination is reached.

```
            if (current == toCell) {
                current = current.PathFrom;
                while (current != fromCell) {
                    int turn = current.Distance / speed;
                    current.SetLabel(turn.ToString());
                    current.EnableHighlight(Color.white);
                    current = current.PathFrom;
                }
                break;
            }
```



Only show labels of cells on the path.

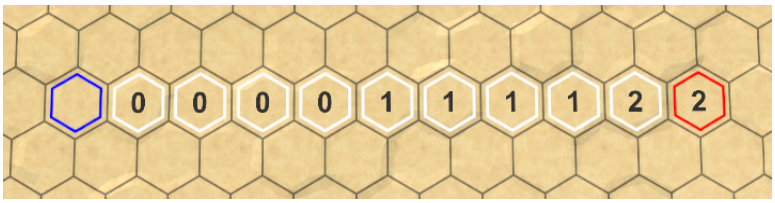
Now we only see the turn labels on the cell in between the start and destination cells. But the destination cell is most important, so we have to set its label too. We can do this by starting the path loop at the destination cell, instead of the cell before it. This will set the destination's highlight to white instead of red, so move the highlighting of the destination cell to directly below the loop.

```
		fromCell.EnableHighlight(Color.blue);
//		toCell.EnableHighlight(Color.red);

		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
		while (searchFrontier.Count > 0) {
			HexCell current = searchFrontier.Dequeue();

			if (current == toCell) {
//				current = current.PathFrom;
				while (current != fromCell) {
					int turn = current.Distance / speed;
					current.SetLabel(turn.ToString());
					current.EnableHighlight(Color.white);
					current = current.PathFrom;
				}
				toCell.EnableHighlight(Color.red);
				break;
			}

			…
		}
```



Turn info is most important for the destination.

After these changes, my worst case is now improved to 23 milliseconds in the editor and 6 milliseconds in a stand-alone build. That's 43 pps and 166 pps respectively, which is a significant improvement.

3 Smartest Search

In the previous tutorial, we made our search routine smarter by implementing the A* algorithm. However, we actually do not yet search in the most optimal way. Each iteration, we calculate the distances from the current cell to all of its neighbors. This is correct for cells that are not yet or currently part of the search frontier. But cells that have already been taken out of the frontier no longer need to be considered. That's because we've already found the shortest path to those cells. Skipping those cells is exactly what a proper A* implementation does, so we should do that as well.

3.1 Cell Search Phase

How do we know whether a cell has already exited the frontier? Currently, we cannot determine this. So we have to keep track of which phase of the search a cell is in. It's either not yet in the frontier, currently part of the frontier, or behind the frontier. We can track this by adding a simple integer property to `HexCell`.

```
	public int SearchPhase { get; set; }
```

For example, 0 means the cell has not yet been reached, 1 indicated that the cell is currently in the frontier, while 2 means it has been taken out of the frontier.

3.2 Entering the Frontier

In `HexGrid.Search`, we could reset all cells to 0 and always use 1 for the frontier. Or we could increment the frontier number for each new search. That way, we do not have to bother with resetting the cells, as long as we increment the frontier by two each time.

```
	int searchFrontierPhase;

	…

	void Search (HexCell fromCell, HexCell toCell, int speed) {
		searchFrontierPhase += 2;
		…
	}
```

Now we have to set the search phase of cells when we add them to the frontier. This begins with the starting cell, when it is added to the frontier.

```
		fromCell.SearchPhase = searchFrontierPhase;
		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
```

And also whenever we add a neighbor to the frontier.

```
				if (neighbor.Distance == int.MaxValue) {
					neighbor.SearchPhase = searchFrontierPhase;
					neighbor.Distance = distance;
					neighbor.PathFrom = current;
					neighbor.SearchHeuristic =
						neighbor.coordinates.DistanceTo(toCell.coordinates);
					searchFrontier.Enqueue(neighbor);
				}
```

3.3 Checking the Frontier

Up to this point, we used a distance equal to `int.MaxValue` to check whether a cell has not yet been added to the frontier. We can now compare the cell's search phase with the current frontier instead.

```
//				if (neighbor.Distance == int.MaxValue) {
				if (neighbor.SearchPhase < searchFrontierPhase) {
					neighbor.SearchPhase = searchFrontierPhase;
					neighbor.Distance = distance;
					neighbor.PathFrom = current;
					neighbor.SearchHeuristic =
						neighbor.coordinates.DistanceTo(toCell.coordinates);
					searchFrontier.Enqueue(neighbor);
				}
```

This means that we no longer have to reset the cell distances before searching. This means that we can get away with doing less work, which is good.

```
		for (int i = 0; i < cells.Length; i++) {
//			cells[i].Distance = int.MaxValue;
			cells[i].SetLabel(null);
			cells[i].DisableHighlight();
		}
```

3.4 Exiting the Frontier

When a cell is taken out of the frontier, we indicate this by incrementing its search phase. That puts it behind the current frontier and in front of the next.

```
		while (searchFrontier.Count > 0) {
			HexCell current = searchFrontier.Dequeue();
			current.SearchPhase += 1;

			…
		}
```

Finally, we can skip cells that have been taken out of the frontier, avoiding a pointless distance computation and comparison.

```
			for (HexDirection d = HexDirection.NE; d <= HexDirection.NW; d++) {
				HexCell neighbor = current.GetNeighbor(d);
				if (
					neighbor == null ||
					neighbor.SearchPhase > searchFrontierPhase
				) {
					continue;
				}
				…
			}
```

At this point, our algorithm still produces the same results, but more efficiently. For me, the worst-case search now takes 20 milliseconds in the editor and 5 milliseconds in a build.

You could also log how many times cells were processed by the algorithm, by incrementing a counter when a cell's distance is computer. Earlier, the our algorithm computed 28,239 distances for the worst-case search. Now, with our complete A* algorithm, we compute only 14,120 distances. That's a reduction of 50%. How big of an impact this has on performance depends on how costly it is to calculate the movement costs. In our case, it's not that much work, so the improvement is not that great in a build, though still pretty significant in the editor.

4 Cleaning the Path

When a new search is initiated, we first have to clean up the visualization of the previous path. Right now, we do this by disabling the highlight and removing the label of every cell in the grid. This is a heavy-handed approach. Ideally, we only reset the cells that are part of the previous path.

4.1 Only Searching

Let's start by completely removing the visualization code from `Search`. All that it should do is search for a path, regardless of what we do with that information.

```
	void Search (HexCell fromCell, HexCell toCell, int speed) {
		searchFrontierPhase += 2;
		if (searchFrontier == null) {
			searchFrontier = new HexCellPriorityQueue();
		}
		else {
			searchFrontier.Clear();
		}

//		for (int i = 0; i < cells.Length; i++) {
//			cells[i].SetLabel(null);
//			cells[i].DisableHighlight();
//		}
//		fromCell.EnableHighlight(Color.blue);

		fromCell.SearchPhase = searchFrontierPhase;
		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
		while (searchFrontier.Count > 0) {
			HexCell current = searchFrontier.Dequeue();
			current.SearchPhase += 1;

			if (current == toCell) {
//				while (current != fromCell) {
//					int turn = current.Distance / speed;
//					current.SetLabel(turn.ToString());
//					current.EnableHighlight(Color.white);
//					current = current.PathFrom;
//				}
//				toCell.EnableHighlight(Color.red);
//				break;
			}

			…
		}
	}
```

To communicate whether `Search` found a path, have it return a boolean.

```
	bool Search (HexCell fromCell, HexCell toCell, int speed) {
		searchFrontierPhase += 2;
		if (searchFrontier == null) {
			searchFrontier = new HexCellPriorityQueue();
		}
		else {
			searchFrontier.Clear();
		}

		fromCell.SearchPhase = searchFrontierPhase;
		fromCell.Distance = 0;
		searchFrontier.Enqueue(fromCell);
		while (searchFrontier.Count > 0) {
			HexCell current = searchFrontier.Dequeue();
			current.SearchPhase += 1;

			if (current == toCell) {
				return true;
			}

			…
		}
		return false;
	}
```

4.2 Remembering the Path

When a path is found, we have to remember it. That way, we can clean it up next time. So keep track of the end points and whether a path exists between them.

```
	HexCell currentPathFrom, currentPathTo;
	bool currentPathExists;

	…

	public void FindPath (HexCell fromCell, HexCell toCell, int speed) {
		System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
		sw.Start();
		currentPathFrom = fromCell;
		currentPathTo = toCell;
		currentPathExists = Search(fromCell, toCell, speed);
		sw.Stop();
		Debug.Log(sw.ElapsedMilliseconds);
	}
```

4.3 Showing the Path Again

We can use the search data that we remembered to once again visualize the path. Create a new `ShowPath` method for that. It will loop from the end to the beginning of the path, highlighting the cells and setting their labels to their turn. We need to know the speed to do that, so make that a parameter. If we don't have a path, the method will just highlight the end points.

```
	void ShowPath (int speed) {
		if (currentPathExists) {
			HexCell current = currentPathTo;
			while (current != currentPathFrom) {
				int turn = current.Distance / speed;
				current.SetLabel(turn.ToString());
				current.EnableHighlight(Color.white);
				current = current.PathFrom;
			}
		}
		currentPathFrom.EnableHighlight(Color.blue);
		currentPathTo.EnableHighlight(Color.red);
	}
```

Invoke this method in `FindPath`, after searching.

```
		currentPathExists = Search(fromCell, toCell, speed);
		ShowPath(speed);
```

4.4 Cleaning Up

We're seeing paths again, but they no longer get erased. To clean them up, create a `ClearPath` method. It's basically a copy of `ShowPath`, except that it disables the highlights and labels instead of showing them. After doing that, it should also clear the path data that we remembered, as it is no longer valid.

```
	void ClearPath () {
		if (currentPathExists) {
			HexCell current = currentPathTo;
			while (current != currentPathFrom) {
				current.SetLabel(null);
				current.DisableHighlight();
				current = current.PathFrom;
			}
			current.DisableHighlight();
			currentPathExists = false;
		}
		currentPathFrom = currentPathTo = null;
	}
```

Besides that, we must also make sure to clear the end points in case of an invalid path.

```
		if (currentPathExists) {
			…
		}
		else if (currentPathFrom) {
			currentPathFrom.DisableHighlight();
			currentPathTo.DisableHighlight();
		}
		currentPathFrom = currentPathTo = null;
```

With this method, we can clear the old path visualization by only visiting the necessary cells. The size of the map no longer matters. Invoke it in `FindPath`, before initiating a new search.

```
		sw.Start();
		ClearPath();
		currentPathFrom = fromCell;
		currentPathTo = toCell;
		currentPathExists = Search(fromCell, toCell, speed);
		if (currentPathExists) {
			ShowPath(speed);
		}
		sw.Stop();
```

Also, make sure to clear the path when creating a new map.

```
	public bool CreateMap (int x, int z) {
		…

		ClearPath();
		if (chunks != null) {
			for (int i = 0; i < chunks.Length; i++) {
				Destroy(chunks[i].gameObject);
			}
		}

		…
	}
```

And also before loading another map.

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

The path visualizations once again gets cleaned up, just like before we made this change. Now that we're using a more efficient approach, my worst-case search is down to 14 milliseconds in the editor. That's quite an improvement for simply cleaning up smarter. In a build, I'm now down to 3 milliseconds. That's 333 pps, which makes our pathfinding definitely usable in real time.

Now that we have fast pathfinding, we can remove the timing debug code.

```
	public void FindPath (HexCell fromCell, HexCell toCell, int speed) {
//		System.Diagnostics.Stopwatch sw = new System.Diagnostics.Stopwatch();
//		sw.Start();
		ClearPath();
		currentPathFrom = fromCell;
		currentPathTo = toCell;
		currentPathExists = Search(fromCell, toCell, speed);
		ShowPath(speed);
//		sw.Stop();
//		Debug.Log(sw.ElapsedMilliseconds);
	}
```

The next tutorial is Units.

made by Jasper Flick