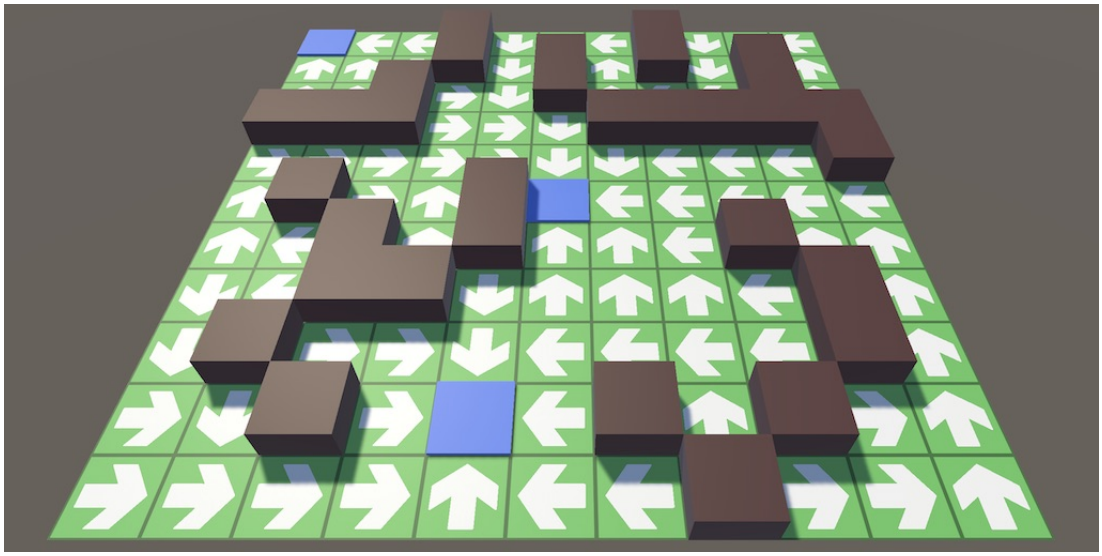
The Board Building a Maze

Create a tiled board.
Find paths with breadth-first search.
Support empty, destination, and wall tiles.
Edit tile content in play mode.
Optionally show grid and path visualizations.

This is the first installment of a tutorial series about creating a simple tower defense game. It covers the creation of the game board, pathfinding, and placement of destinations and walls.

This tutorial is made with Unity 2018.3.0f2.



A board ready for a tile-based tower-defense game.

1 A Tower Defense Game

Tower defense is a game genre where the goal is to destroy hordes of enemies before they reach their destination. This is done by building towers that attack those enemies. There are many variations of the genre. We'll create a game with a tiled board. Enemies will move across the board toward their destination, while the player places obstacles to hinder them.

This series assumes that you've gone through the Object Management series first.

1.1 The Board

The game board is the most important part of the game, so we'll create it first in this tutorial. It's going to be a game object with a custom `GameBoard` component that can be initialized with a 2D size, for which we can use a `Vector2Int` value. The board should work with any size, but we'll decide which to use somewhere else, so we'll provide a public `Initialize` method for that.

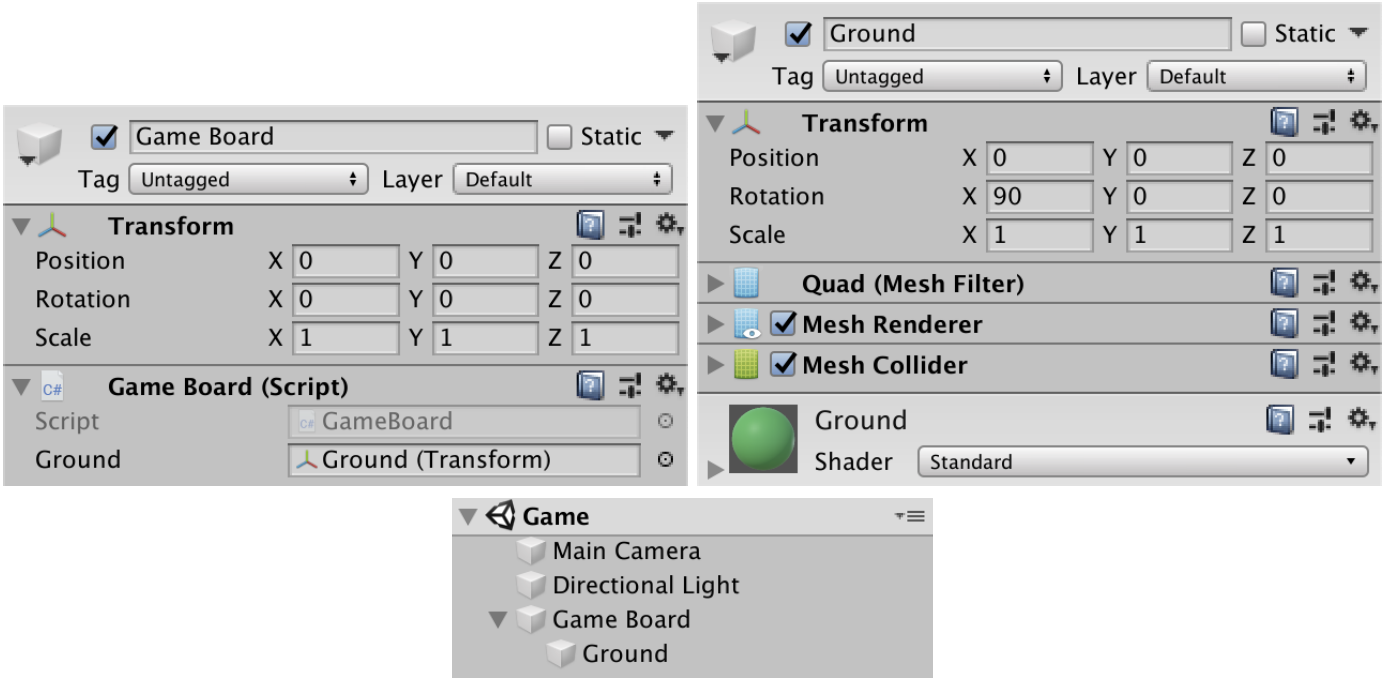
Besides that, we'll visualize the board with a single quad that represents the ground. We won't make the board object a quad itself, instead we'll give it a quad child object. When initializing, we make the ground's XY scale equal to the board's size. So each tile is one square unit.

```
using UnityEngine;

public class GameBoard : MonoBehaviour {

	[SerializeField]
	Transform ground = default;

	Vector2Int size;

	public void Initialize (Vector2Int size) {
		this.size = size;
		ground.localScale = new Vector3(size.x, size.y, 1f);
	}
}
```

Why explicitly set `ground` to its default value?

The idea is that everything that we configure via the Unity editor is exposed via serialized private fields. These fields should only be changed via an inspector. Unfortunately, the Unity editor will always show a compiler warning complaining that the value is never assigned to. We can suppress this warning by explicitly assigning a default value to the field. We could assign `null`, but I make it explicit that we just use the default value, which doesn't represent a valid ground reference, so use `default` instead.

Create the board object in a new scene and give it the required quad child, with a material that makes it look like ground. As we're creating a simple prototype-like game, a uniform green material will do. Also rotate the quad 90° around its X axis so it lies flat in the XZ plane.



Game board.

Why not put the game in the XY plane?

Although the game plays in 2D, we're going to render it in 3D, with 3D enemies and a camera that could be moved around at some point. The XZ plane is more convenient for that and matches the default skybox orientation used for environmental lighting.

1.2 The Game

Next, create a `Game` component that is in charge of the entire game. At this point that means initializing the board. We'll simply make the size configurable via its inspector and have it initialize the board when it awakens. Let's use 11×11 as the default size.

```
using UnityEngine;

public class Game : MonoBehaviour {

	[SerializeField]
	Vector2Int boardSize = new Vector2Int(11, 11);

	[SerializeField]
	GameBoard board = default;

	void Awake () {
		board.Initialize(boardSize);
	}
}
```

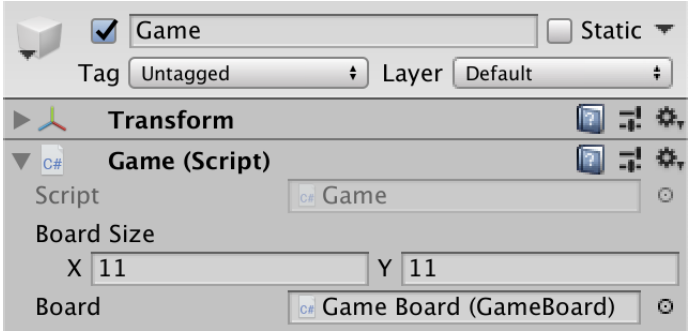
Board sizes can only be positive, and it doesn't make much sense to have a board with only a single tile. So let's enforce 2×2 as the minimum. We can do that by adding an `OnValidate` method that enforces the minimum.

```
	void OnValidate () {
		if (boardSize.x < 2) {
			boardSize.x = 2;
		}
		if (boardSize.y < 2) {
			boardSize.y = 2;
		}
	}
```

When does `Onvalidate` get invoked?

If it exists, the Unity editor invokes it on components after they might have changed. This includes when they're added to a game object, after a scene is loaded, after a recompilation, after an edit via the inspector, after an undo/redo, and after a component reset.

`OnValidate` is the only place in our code where we should ever assign values to component configuration fields.



The game object.

Now we get a properly-sized board after entering play mode. While playing, position the camera so the entire board is clearly visible, copy its transformation component, exit play mode, and paste the component values. For an 11×11 board at the origin, you get a nice top-down view by placing the camera at (0,10,0) and rotating it 90° around the X axis. We'll keep the camera at this fixed orientation, but might make it move in the future.

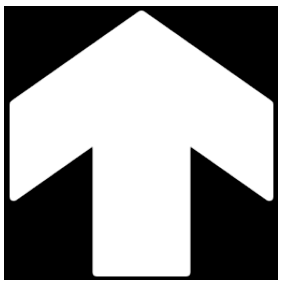
Camera above board.

How do you copy and paste component values?

Via the dropdown menu accessible via the gear button at the top right corner of the component.

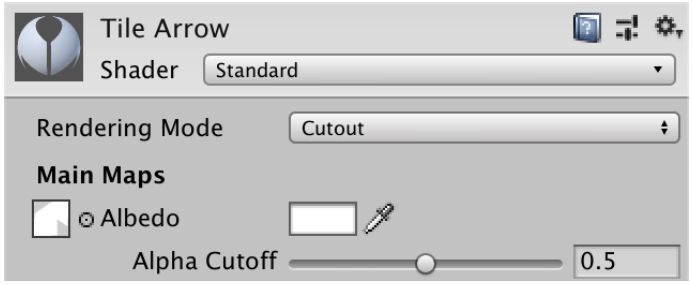
1.3 Tile Prefab

The board is made up of square tiles. Enemies will be able to walk from tile to tile, across edges but not diagonally. Movement will always be toward the nearest destination. Let's visualize the movement direction per tile, with an arrow. Here is an arrow texture for that.



Arrow, on black background.

Put the arrow texture in your project and enable its *Alpha As Transparency* option. Then create a material for the arrow, which can be a default material set to cutout mode, with the arrow as its main texture.



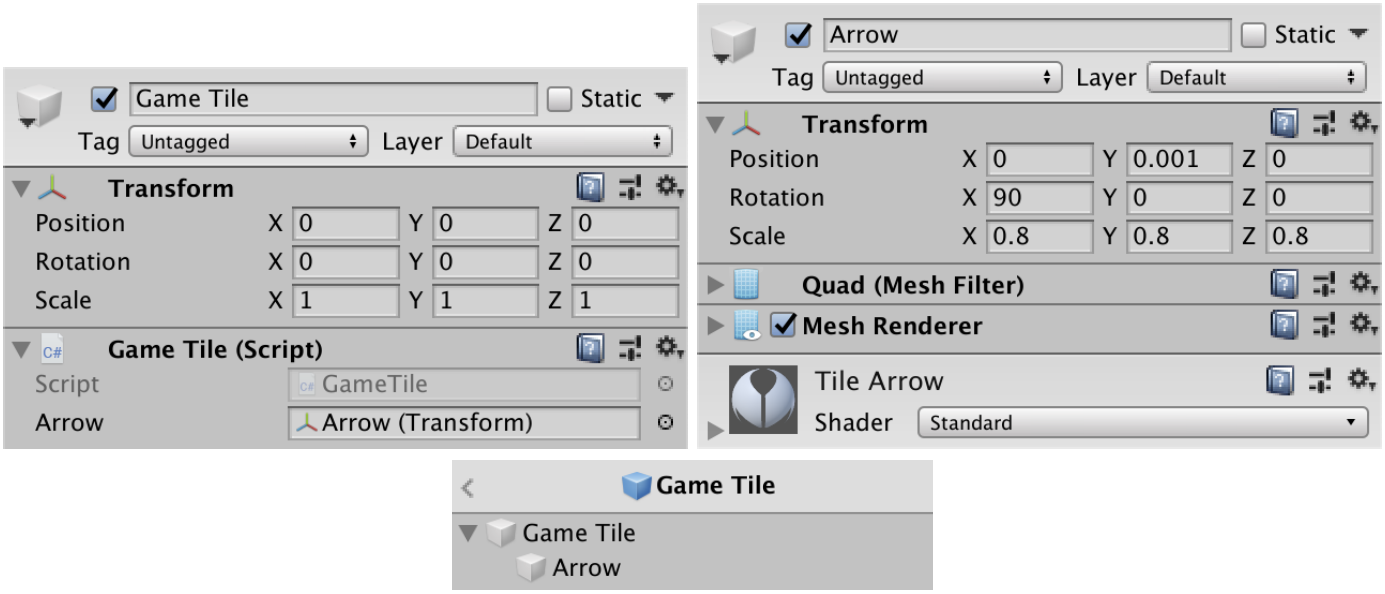
Arrow material.

> **Why use cutout rendering mode?**
>
> That makes it possible for the arrow to be shadowed when using Unity's default render pipeline.

We'll use a game object to represent each tile in the game. Each will have its own quad with the arrow material, just like the board has its ground quad. We'll also give tiles a custom `GameTile` component, with a reference to their arrow.

```
using UnityEngine;

public class GameTile : MonoBehaviour {

	[SerializeField]
	Transform arrow = default;
}
```

Construct a tile object and turn it into a prefab. The tiles will be aligned with the ground, so move the arrow up a bit to prevent depth issues during rendering. Also, scale the arrow down a bit so there's some space between adjacent arrows. An Y offset of 0.001 and a uniform scale of 0.8 will do.



Tile prefab.

Where is the tile prefab hierarchy?

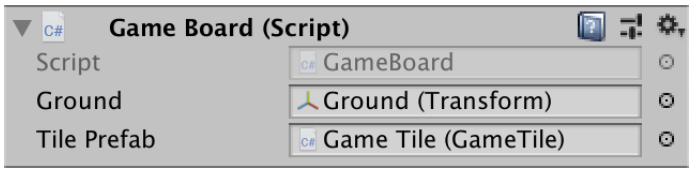
You can open prefab-editing mode by double-clicking the prefab asset or by selecting the prefab and clicking the *Open Prefab* button in its inspector. You can exit the prefab-editing mode via the arrow button at the top left of its hierarchy header.

Note that the tiles themselves don't really need to be game objects. They're purely for keeping track of the board state. We could've used the same approach as for behavior in the Object Management series. But game objects work just fine for the early stages of simple or prototype games. We might change this in the future.

1.4 Laying Tiles

To create tiles `GameBoard` needs to have a reference to the tile prefab.

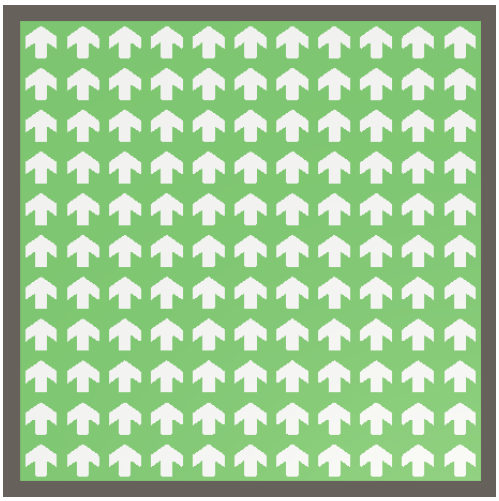
```
	[SerializeField]
	GameTile tilePrefab = default;
```



Linked to tile prefab.

Then it can instantiate them in `Initialize` with a double loop over the two dimensions of the grid. Although the size is expressed in X and Y, we place the tiles in the XZ plane, just like the board itself. As the board is centered on the origin, we have to subtract the relevant size minus one, divided by two, from the tile position's components. Note that this must be a floating-point division, otherwise it won't work for even sizes.

```
	public void Initialize (Vector2Int size) {
		this.size = size;
		ground.localScale = new Vector3(size.x, size.y, 1f);

		Vector2 offset = new Vector2(
			(size.x - 1) * 0.5f, (size.y - 1) * 0.5f
		);
		for (int y = 0; y < size.y; y++) {
			for (int x = 0; x < size.x; x++) {
				GameTile tile = Instantiate(tilePrefab);
				tile.transform.SetParent(transform, false);
				tile.transform.localPosition = new Vector3(
					x - offset.x, 0f, y - offset.y
				);
			}
		}
	}
```



Instantiated tiles.

We'll need to access the tiles later, so keep track of them with an array. We don't need to use a list because we won't change the board size once initialized.

```
	GameTile[] tiles;

	public void Initialize (Vector2Int size) {
		…
		tiles = new GameTile[size.x * size.y];
		for (int i = 0, y = 0; y < size.y; y++) {
			for (int x = 0; x < size.x; x++, i++) {
				GameTile tile = tiles[i] = Instantiate(tilePrefab);
				…
			}
		}
	}
```

How does that assignment work?

It is a chained assignment. In this case, it means that we assign a reference to the instantiated tile to both the array element and the local variable. It does the exact same thing as the below code.

```
GameTile t = Instantiate(tilePrefab);
tiles[i] = t;
GameTile tile = t;
```

2 Pathfinding

At this point each tile has an arrow, but they're all pointing in the positive Z direction, which we'll interpret as north. The next step is to figure out the correct direction per tile. We do that by finding the paths that the enemies will follow to get to their destination.

2.1 Tile Neighbors

Paths go from tile to tile, in either north, east, south, or west direction. To make searching easy, have `GameTile` keep track of references to its four neighbors.

```
	GameTile north, east, south, west;
```

The neighbor relationship is symmetrical. If a tile is the eastern neighbor of a second tile, then the second tile is the western neighbor of the first tile. Add a public static method to `GameTile` to establish that relationship between two tiles.

```
	public static void MakeEastWestNeighbors (GameTile east, GameTile west) {
		west.east = east;
		east.west = west;
	}
```

> **Why make it a static method?**
>
> We could make it an instance method with a single parameter as well, in which case we could invoke it as `eastTile.MakeEastWestNeighbors(westTile)` or something like that. But in cases where it isn't clear which of the tiles the method should be invoked on a static method is a good approach. Examples are the `Distance` and `Dot` methods of `Vector3`.

Once that relationship has been established it should never be changed. If that does happen we made a programming mistake. We can verify this by checking whether both references are `null` before assigning them and log an error if that's not the case. We can use the `Debug.Assert` method for that.

```
	public static void MakeEastWestNeighbors (GameTile east, GameTile west) {
		Debug.Assert(
			west.east == null && east.west == null, "Redefined neighbors!"
		);
		west.east = east;
		east.west = west;
	}
```

What does `Debug.Assert` do?

If the first argument is `false`, it logs an assertion error, using the second argument message if provided. This invocation is only included in development builds, not in release builds. So it's a good way to add checks during development that won't affect the final release.

Add a similar method to create a north-south neighbor relationship.

```
	public static void MakeNorthSouthNeighbors (GameTile north, GameTile south) {
		Debug.Assert(
			south.north == null && north.south == null, "Redefined neighbors!"
		);
		south.north = north;
		north.south = south;
	}
```

We can establish these relationships when the tiles are created in `GameBoard.Initialize`. If the X coordinate is larger than zero then we can make an east-west relationship between the current tile and the previous one. If the Y coordinate is larger than zero then we can make a north-south relationship between the current tile and the one from a row earlier.

```
		for (int i = 0, y = 0; y < size.y; y++) {
			for (int x = 0; x < size.x; x++, i++) {
				…

				if (x > 0) {
					GameTile.MakeEastWestNeighbors(tile, tiles[i - 1]);
				}
				if (y > 0) {
					GameTile.MakeNorthSouthNeighbors(tile, tiles[i - size.x]);
				}
			}
		}
```

Note that the tiles on the edge of the board have fewer than four neighbors. Either one or two or their neighbor references remain `null`.

2.2 Distance and Direction

We're not going to have all enemies search for a path all the time. We only have to do it once per tile. Then enemies can then query the tile they're in for where to go next. We store that information in `GameTile` by adding a reference to the next tile on the path. Besides that, we'll also store the distance to the destination, expressed as the amount of tiles that still have to be entered before reaching the destination. That's not useful information for enemies, but we'll use it when finding the shortest paths.

```
	GameTile north, east, south, west, nextOnPath;

	int distance;
```

Each time we decide to find the paths, we must initialize the path data. Before a path is found, there isn't a next tile yet and the distance can be considered infinite. We can represent that with the largest possible integer value, `int.MaxValue`. Add a public `ClearPath` method to reset `GameTile` to this state.

```
	public void ClearPath () {
		distance = int.MaxValue;
		nextOnPath = null;
	}
```

It is only possible to find any paths if there is a destination. That means that a tile has to become the destination. Such a tile has a distance of zero and there is not next tile as the path ends here. Add a public method to turn the tile into a destination.

```
	public void BecomeDestination () {
		distance = 0;
		nextOnPath = null;
	}
```

Eventually, all tiles should have a path, so their distance is no longer equal to `int.MaxValue`. Add a convenient getter property to check whether a tile currently has a path.

```
	public bool HasPath => distance != int.MaxValue;
```

How does that property work?

It's a shorthand to define a getter property that contains only a single expression. It does the exact the same thing as the below code.

```
	public bool HasPath {
		get {
			return distance != int.MaxValue;
		}
	}
```

The => arrow operator can also be used for the getter and setter parts of a property separately, for method bodies, constructors, and in some other places.

2.3 Growing the Path

If we have a tile with a path, we can let it grow the path toward one of its neighbors. Initially the only tile with a path is the destination, so we start at distance zero and increase it from there, going in the opposite direction that the enemies will move. So all direct neighbors of the destination will have distance 1, and all other neighbors of those tiles will have distance 2, and so on.

Give `GameTile` a private method to grow the path to one of its neighbors, defined via a parameter. The neighbor's distance becomes one longer than its own, and the neighbor's path points toward this tile. The method should only be invoked on tiles that already have a path, so assert that.

```
	void GrowPathTo (GameTile neighbor) {
		Debug.Assert(HasPath, "No path!");
		neighbor.distance = distance + 1;
		neighbor.nextOnPath = this;
	}
```

The idea is that we invoke this method once for each of the four neighbors of the tile. As some of those references will be `null`, check for that and abort if so. Also, if a neighbor already has a path then we have nothing to do and can abort as well.

```
	void GrowPathTo (GameTile neighbor) {
		Debug.Assert(HasPath, "No path!");
		if (neighbor == null || neighbor.HasPath) {
			return;
		}
		neighbor.distance = distance + 1;
		neighbor.nextOnPath = this;
	}
```

How `GameTile` keeps track of its neighbors is unknown to other code. That's why `GrowPathTo` is private. Instead, we'll add public methods that instruct a tile to grow its path in a specific direction, indirectly invoking `GrowPathTo`. But the code that takes care of searching the entire board must keep track of which tiles have been visited. So have it return the neighbor, or `null` if we aborted.

```
	GameTile GrowPathTo (GameTile neighbor) {
		if (!HasPath || neighbor == null || neighbor.HasPath) {
			return null;
		}
		neighbor.distance = distance + 1;
		neighbor.nextOnPath = this;
		return neighbor;
	}
```

Now add the methods to grow the path in specific directions.

```
	public GameTile GrowPathNorth () => GrowPathTo(north);

	public GameTile GrowPathEast () => GrowPathTo(east);

	public GameTile GrowPathSouth () => GrowPathTo(south);

	public GameTile GrowPathWest () => GrowPathTo(west);
```

2.4 Breadth-First Search

It is the responsibility of `GameBoard` to make sure all its tiles contain valid path data. We'll implement this by performing a breadth-first search. We start with a destination tile, then grow the path to its neighbors, then to the neighbors of those tiles, and so on. Each step the distance increases by one, and paths are never grown toward tiles that already have a path. This guarantees that all tiles end up pointing along the shortest path to the destination.

> **What about A* pathfinding?**
>
> A* is an evolution of breadth-first search. It is useful when you're searching for a single shortest path. But we need all shortest paths, so A* provides no benefit. See the Hex Map series for examples of both breadth-first search and A* applied to a hex grid, with animations.

To perform the search, we have to keep track of the tiles that we've added to the path, but haven't grown the path out from yet. That collection of tiles is often known as the search frontier. It is important that tiles are processed in the same order that they're added to the frontier, so let's use a `Queue`. We'll end up searching more than once later, so define it as a field in `GameBoard`.

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

public class GameBoard : MonoBehaviour {

	…

	Queue<GameTile> searchFrontier = new Queue<GameTile>();

	…
}
```

To always keep the board state valid, we should find the paths at the end of `Initialize`, but put the code in a separate `FindPaths` method. The first step is to clear the path of all tiles, then make one tile the destination and add it to the frontier. Let's just pick the fist tile. As `tiles` is an array we can use a `foreach` loop without worrying about memory pollution. If we switch to a list later, we should also replace the `foreach` loops with `for` loops.

```
	public void Initialize (Vector2Int size) {
		…

		FindPaths();
	}

	void FindPaths () {
		foreach (GameTile tile in tiles) {
			tile.ClearPath();
		}
		tiles[0].BecomeDestination();
		searchFrontier.Enqueue(tiles[0]);
	}
```

The next step is to take the single tile out of the frontier and grow the path to its neighbors, adding them all to the frontier. First go north, then east, then south, and finally west.

```
	public void FindPaths () {
		foreach (GameTile tile in tiles) {
			tile.ClearPath();
		}
		tiles[0].BecomeDestination();
		searchFrontier.Enqueue(tiles[0]);

		GameTile tile = searchFrontier.Dequeue();
		searchFrontier.Enqueue(tile.GrowPathNorth());
		searchFrontier.Enqueue(tile.GrowPathEast());
		searchFrontier.Enqueue(tile.GrowPathSouth());
		searchFrontier.Enqueue(tile.GrowPathWest());
	}
```

Repeat this step as long as there are tiles in the frontier.

```
		while (searchFrontier.Count > 0) {
			GameTile tile = searchFrontier.Dequeue();
			searchFrontier.Enqueue(tile.GrowPathNorth());
			searchFrontier.Enqueue(tile.GrowPathEast());
			searchFrontier.Enqueue(tile.GrowPathSouth());
			searchFrontier.Enqueue(tile.GrowPathWest());
		}
```

Growing the path doesn't always yield a new tile. We could check whether we got `null` before enqueueing, but we can also delay the `null` check until after we dequeue.

```
			GameTile tile = searchFrontier.Dequeue();
			if (tile != null) {
				searchFrontier.Enqueue(tile.GrowPathNorth());
				searchFrontier.Enqueue(tile.GrowPathEast());
				searchFrontier.Enqueue(tile.GrowPathSouth());
				searchFrontier.Enqueue(tile.GrowPathWest());
			}
```

2.5 Showing the Paths

We now end up with a board containing valid paths, but we don't see this yet. We have to adjust the arrows so they point along the path through their tile. We can do that by rotating them. As these rotations are always the same, add static `Quaternion` fields to `GameTile`, one per direction.

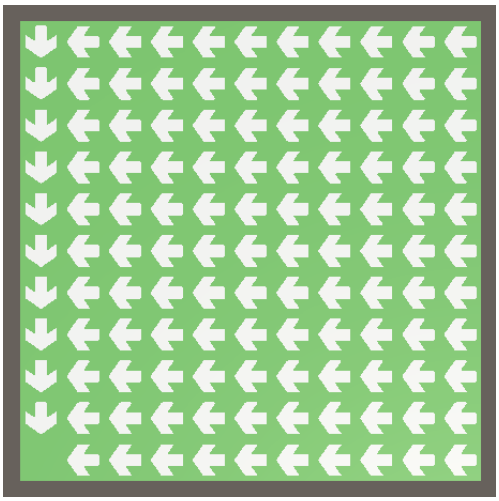
```
	static Quaternion
		northRotation = Quaternion.Euler(90f, 0f, 0f),
		eastRotation = Quaternion.Euler(90f, 90f, 0f),
		southRotation = Quaternion.Euler(90f, 180f, 0f),
		westRotation = Quaternion.Euler(90f, 270f, 0f);
```

Add a public `ShowPath` method as well. If the distance is zero then the tile is a destination and it has nowhere to point to, so deactivate its arrow. Otherwise, active the arrow and set its rotation. The correct rotation can be found by comparing `nextOnPath` with its neighbors.

```
	public void ShowPath () {
		if (distance == 0) {
			arrow.gameObject.SetActive(false);
			return;
		}
		arrow.gameObject.SetActive(true);
		arrow.localRotation =
			nextOnPath == north ? northRotation :
			nextOnPath == east ? eastRotation :
			nextOnPath == south ? southRotation :
			westRotation;
	}
```

Invoke this method on all tiles at the end of `GameBoard.FindPaths`.

```
	public void FindPaths () {
		…

		foreach (GameTile tile in tiles) {
			tile.ShowPath();
		}
	}
```



Found paths.

Why not rotate the arrow directly in `GrowPathTo`?

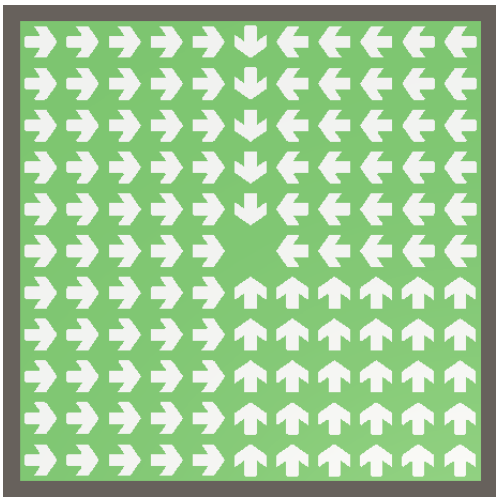
To keep search logic and visualization separate. Later on, we'll make the visualization optional. If arrows aren't shown, then we don't need to rotate them every time we invoke `FindPaths`.

2.6 Alternating Search Priority

It turns out that when the southwest corner tile is the destination all paths go straight west until they reach the edge of the board and then go south. This is valid, as indeed there are no shorter paths to the destination. That's because there is no diagonal movement. However, there are many other shortest paths possible, which might look better.

To get a better idea of why these paths were found, move the destination to the center of the map. That's simply the tile in the middle of the array, when using an odd board size.

```
		tiles[tiles.Length / 2].BecomeDestination();
		searchFrontier.Enqueue(tiles[tiles.Length / 2]);
```

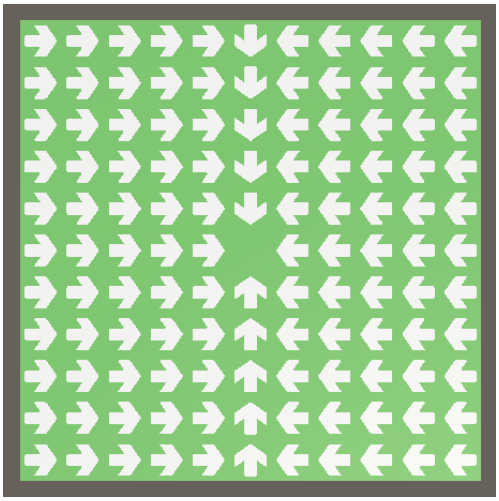


Center destination.

The result makes sense if you keep in mind how the search works. As we add neighbors in north-east-south-west order, north has the highest priority. As we search backwards, that means south is the last traveled direction. That's why there are only a few arrows pointing south, and so many pointing east.

We can change the result by adjusting the direction priorities. Let's swap east and south. That should result in north-south and east-west symmetry.

```
			searchFrontier.Enqueue(tile.GrowPathNorth());
			searchFrontier.Enqueue(tile.GrowPathSouth());
			searchFrontier.Enqueue(tile.GrowPathEast());
			searchFrontier.Enqueue(tile.GrowPathWest())
```



North-south-east-west search order.

That looks better, but it would be best if the paths alternate direction to approximate diagonal movement where it appears natural. We can do that by reversing the search priorities of adjacent tiles, in a checkerboard pattern.

Instead of figuring out which kind of tile we have while searching, we'll add a public property to `GameTile` to indicate whether it is an alternative tile.

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

Set this property in `GameBoard.Initialize`. First, mark tiles as alternative if their X coordinate is an even number.

```
		for (int i = 0, y = 0; y < size.y; y++) {
			for (int x = 0; x < size.x; x++, i++) {
				…

				tile.IsAlternative = (x & 1) == 0;
			}
		}
```

What does `(x & 1) == 0` do?

A single ampersand is the binary AND operator. It performs the logical AND operation on each individual pair of bits of its operands. So both bits of a pair need to be 1 for the resulting bit to be 1. For example, 10101010 and 00001111 produce 00001010.

Internally, numbers are binary. They only use 0s and 1s. In binary, the sequence 1, 2, 3, 4 is written as 1, 10, 11, 100. As you can see, the least significant digit of even numbers is zero.

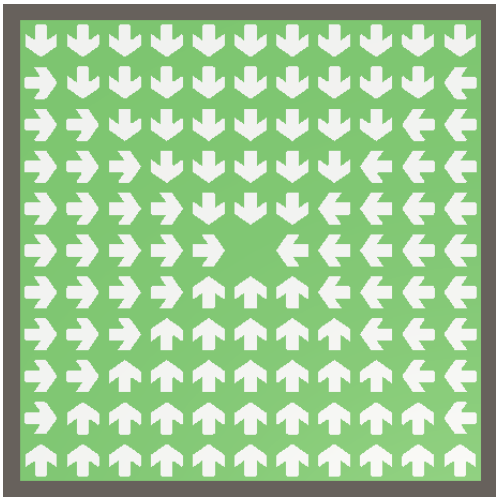
We use the binary AND as a mask, ignoring everything except the least significant bit. If the result is zero, then we have an even number.

Second, negate the result if their Y coordinate is even. That produces a checkerboard pattern.

```
            tile.IsAlternative = (x & 1) == 0;
            if ((y & 1) == 0) {
                tile.IsAlternative = !tile.IsAlternative;
            }
```

In `FindPaths`, keep the same search order for alternative tiles, but reverse it for all other tiles. That will make paths home in on diagonals and zigzag along them.

```
        if (tile != null) {
            if (tile.IsAlternative) {
                searchFrontier.Enqueue(tile.GrowPathNorth());
                searchFrontier.Enqueue(tile.GrowPathSouth());
                searchFrontier.Enqueue(tile.GrowPathEast());
                searchFrontier.Enqueue(tile.GrowPathWest());
            }
            else {
                searchFrontier.Enqueue(tile.GrowPathWest());
                searchFrontier.Enqueue(tile.GrowPathEast());
                searchFrontier.Enqueue(tile.GrowPathSouth());
                searchFrontier.Enqueue(tile.GrowPathNorth());
            }
        }
```



Alternating search order.

3 Changing Tiles

At this point all the tiles are empty. One tile is used as the destination, but besides not having a visible arrow it look the same as all other tiles. We're going to make it possible to change the tiles, by putting things in them.

3.1 Tile Content

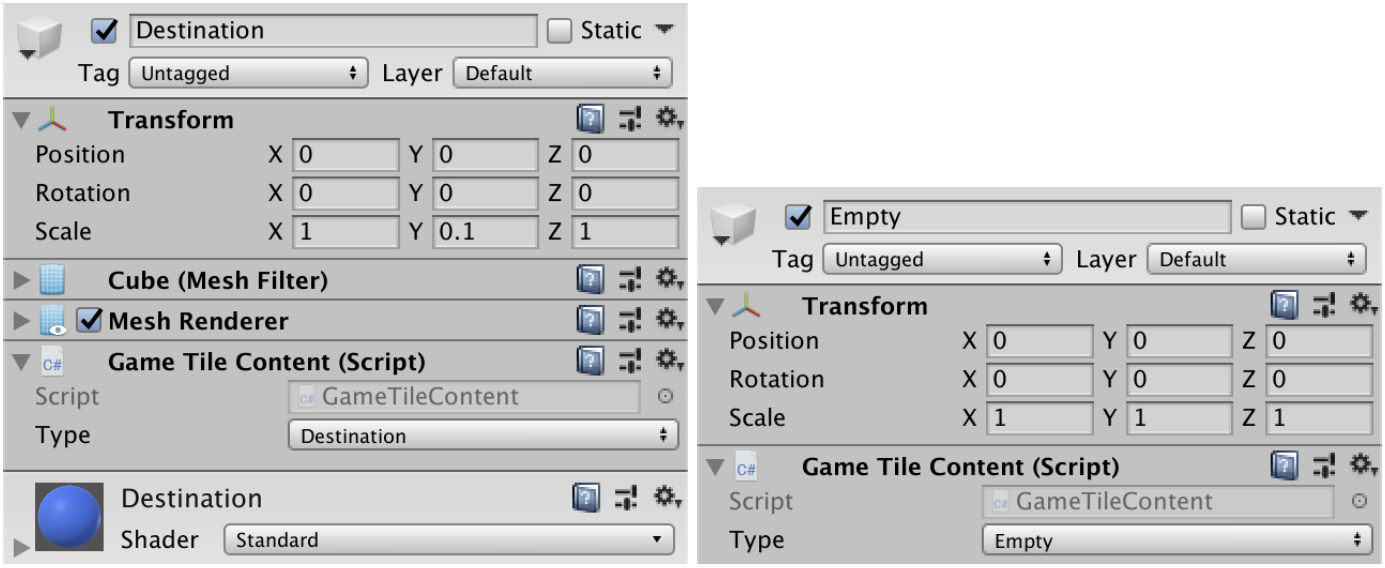
The tile objects themselves are just a means to keep track of tile information. We don't change these objects directly. Instead, we'll introduce separate content and place that on the board. Right now we only distinguish between tiles that are empty and the destination tile. Create a `GameTileContentType` enum type to identify these cases.

```
public enum GameTileContentType {
	Empty, Destination
}
```

Next, create a `GameTileContent` component type that allows its content type to be set via its inspector and is accessible via a public getter property.

```
using UnityEngine;

public class GameTileContent : MonoBehaviour {

	[SerializeField]
	GameTileContentType type = default;

	public GameTileContentType Type => type;
}
```

Then create prefabs for the two content types, each with a `GameTileContent` component with their type set correctly. Let's use a blue flattened cube to visualize destination tiles. Because it's almost flat it doesn't need to have a collider. Use an otherwise empty game object for the empty content prefab.



Destination and empty prefabs.

We'll give empty tiles a content object because then all tiles will always have content, which means that we won't have to check for `null` content references.

3.2 Content Factory

We'll make content editable, so we'll also create a factory for it, using the same approach that we used in the Object Management series. This means that `GameTileContent` should keep track of its origin factory, which should only be set once, and send itself back to the factory in a `Recycle` method.

```
	GameTileContentFactory originFactory;

	…

	public GameTileContentFactory OriginFactory {
		get => originFactory;
		set {
			Debug.Assert(originFactory == null, "Redefined origin factory!");
			originFactory = value;
		}
	}

	public void Recycle () {
		originFactory.Reclaim(this);
	}
```

That assumes the existence of a `GameTileContentFactory`, so create a scriptable object type for it with the required `Recycle` method. We won't actually bother with a fully functional factory that recycles content at this point, so just have it destroy the content. Object reuse can be added to the factory later without changing any other code.

```
using UnityEngine;
using UnityEngine.SceneManagement;

[CreateAssetMenu]
public class GameTileContentFactory : ScriptableObject {

	public void Reclaim (GameTileContent content) {
		Debug.Assert(content.OriginFactory == this, "Wrong factory reclaimed!");
		Destroy(content.gameObject);
	}
}
```

Give the factory a private `Get` method with a prefab parameter. Again, we'll skip object reuse. It instantiates the prefab, sets its origin factory, moves it to the factory scene, and returns it.

```
	GameTileContent Get (GameTileContent prefab) {
		GameTileContent instance = Instantiate(prefab);
		instance.OriginFactory = this;
		MoveToFactoryScene(instance.gameObject);
		return instance;
	}
```

The instance is moved to the content scene of the factory, which can be created on demand. If we're in the editor, first check whether the scene does exist before creating it, in case we lost track of it during a hot reload.

```
	Scene contentScene;

	…

	void MoveToFactoryScene (GameObject o) {
		if (!contentScene.isLoaded) {
			if (Application.isEditor) {
				contentScene = SceneManager.GetSceneByName(name);
				if (!contentScene.isLoaded) {
					contentScene = SceneManager.CreateScene(name);
				}
			}
			else {
				contentScene = SceneManager.CreateScene(name);
			}
		}
		SceneManager.MoveGameObjectToScene(o, contentScene);
	}
```

We only have two content types, so simply add two prefab configuration fields for them.

```
	[SerializeField]
	GameTileContent destinationPrefab = default;

	[SerializeField]
	GameTileContent emptyPrefab = default;
```

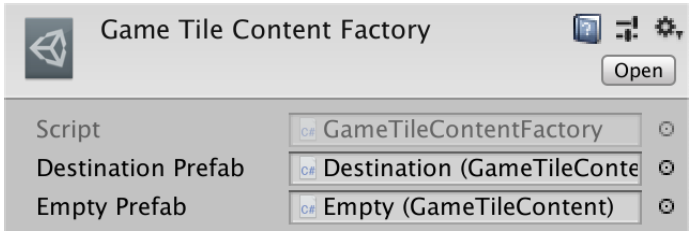
The last step to make the factory functional is to provide a public `Get` method with a `GameTileContentType` parameter, which gets an instance of the corresponding prefab.

```
	public GameTileContent Get (GameTileContentType type) {
		switch (type) {
			case GameTileContentType.Destination: return Get(destinationPrefab);
			case GameTileContentType.Empty: return Get(emptyPrefab);
		}
		Debug.Assert(false, "Unsupported type: " + type);
		return null;
	}
```

Do we need to give each tile its own empty content instance?

Because the empty content doesn't visualize anything, we could get away with instantiating a single empty content object and reusing it for all tiles. But that's an optimization that we don't need to bother with at this point. Also, it's possible to still add some visualization to empty tiles, like stones, grass, flowers, or something else appropriate. It's even possible to mix different visualizations, having the factory return a random one each time. That's currently not the case, but we could make that change later by only adjusting the factory.

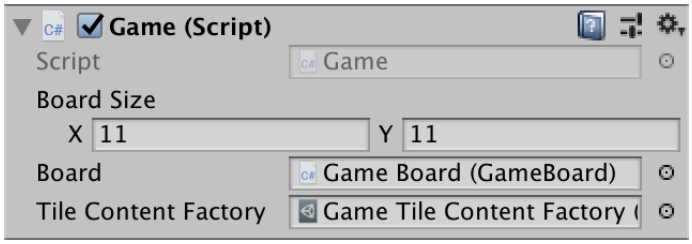
Create the factory asset and configure its prefab references.



Content factory.

Then give `Game` a reference to the factory.

```
	[SerializeField]
	GameTileContentFactory tileContentFactory = default;
```



Game with factory.

3.3 Touching a Tile

To edit the board it must be possible to select a tile. We'll make this possible in play mode, by casting a ray in the scene when the game window is clicked. If the ray hits a tile, it is touched by the player, which indicates that it must be changed. `Game` will handle player input, but we'll make it the responsibility of `GameBoard` to figure out which tile was touched, given a ray.

Not all rays will hit a tile, so it's possible that we end up with nothing. So add a `GetTile` method to `GameBoard` that initially always returns `null`, indicating that no tile was found.

```
	public GameTile GetTile (Ray ray) {
		return null;
	}
```

To determine whether a tile was hit we have to invoke `Physics.Raycast` with the ray as an argument. It returns whether something was hit. If so, we might be able to return a tile, though we don't yet know which, so return `null` right now.

```
	public GameTile GetTile (Ray ray) {
		if (Physics.Raycast(ray) {
			return null;
		}
		return null;
	}
```

To find out whether we hit a tile we need more information about the hit. `Physics.Raycast` can provide this information via a second `RaycastHit` struct parameter. This is an output parameter, which is indicated by writing `out` in front of it. This means that the method invocation will set the value of the variable that we pass to it.

```
		RaycastHit hit;
		if (Physics.Raycast(ray, out hit) {
			return null;
		}
```

It's possible to inline the declaration of variables used for output parameters, so let's do that.

```
		//RaycastHit hit;
		if (Physics.Raycast(ray, out RaycastHit hit) {
			return null;
		}
```

We don't care about what collider got hit, we'll simply use the XZ position of the hit point to determine the tile. We get the tile coordinates by adding half the relevant board size to the hit point's coordinates, then casting the results integers. The final tile index is then its X coordinate, plus its Y coordinate times the board width.

```
		if (Physics.Raycast(ray, out RaycastHit hit)) {
			int x = (int)(hit.point.x + size.x * 0.5f);
			int y = (int)(hit.point.z + size.y * 0.5f);
			return tiles[x + y * size.x];
		}
```

But this is only valid when the tile coordinates lie within the board's bounds, so make sure of that. If not, don't return a tile.

```
			int x = (int)(hit.point.x + size.x * 0.5f);
			int y = (int)(hit.point.z + size.y * 0.5f);
			if (x >= 0 && x < size.x && y >= 0 && y < size.y) {
				return tiles[x + y * size.x];
			}
```

3.4 Changing Content

To make it possible to change the content of a tile, add a public `Content` property to `GameTile`. Its getter simply returns the content, while its setter also recycles its previous content, if any, and positions the new content.

```
	GameTileContent content;

	public GameTileContent Content {
		get => content;
		set {
			if (content != null) {
				content.Recycle();
			}
			content = value;
			content.transform.localPosition = transform.localPosition;
		}
	}
```

This is the only place where we have to check whether the content is `null`, because tiles initially have no content. To be sure, assets that the setter isn't invoked with `null`.

```
		set {
			Debug.Assert(value != null, "Null assigned to content!");
			…
		}
```

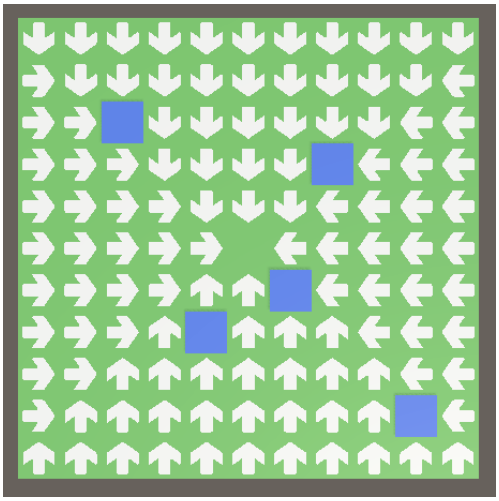
The last thing that we need is player input. Converting a click to a ray can be done by invoking `ScreenPointToRay` with `Input.mousePosition` as its argument on the main camera, which we can access via `Camera.main`. Add a convenient property for that to `Game`.

```
		Ray TouchRay => Camera.main.ScreenPointToRay(Input.mousePosition);
```

Then add an `Update` method that checks whether the primary mouse button was pressed per update, by invoking `Input.GetMouseButtonDown` with zero as an argument. If so, handle the player's touch, which means getting a tile from the board, and if we got one setting its content to a destination, by getting one from the factory.

```
	void Update () {
		if (Input.GetMouseButtonDown(0)) {
			HandleTouch();
		}
	}

	void HandleTouch () {
		GameTile tile = board.GetTile(TouchRay);
		if (tile != null) {
			tile.Content =
				tileContentFactory.Get(GameTileContentType.Destination);
		}
	}
```

It is now possible to turn any tile into a destination, with a primary cursor click.



Multiple destinations.

3.5 Keeping a Valid Board

Although we can turn tiles into destinations, this doesn't yet affect the paths. Also, we haven't set the empty content of tiles yet. Keeping the board consistent and valid is the responsibility of `GameBoard`, so let's move the responsibility of setting tile content to it as well. To make this possible, give it a reference to the content factory, provided via its `Intialize` method, and use it to give all tiles an empty content instance.

```
	GameTileContentFactory contentFactory;

	public void Initialize (
		Vector2Int size, GameTileContentFactory contentFactory
	) {
		this.size = size;
		this.contentFactory = contentFactory;
		ground.localScale = new Vector3(size.x, size.y, 1f);

		tiles = new GameTile[size.x * size.y];
		for (int i = 0, y = 0; y < size.y; y++) {
			for (int x = 0; x < size.x; x++, i++) {
				…

				tile.Content = contentFactory.Get(GameTileContentType.Empty);
			}
		}

		FindPaths();
	}
```

`Game` must now pass its factory to the board.

```
	void Awake () {
		board.Initialize(boardSize, tileContentFactory);
	}
```

Why not add a factory configuration field to `GameBoard`?

The board needs a factory, but doesn't need to know where it came from. We can end up with multiple factories at some point, used to change the appearance of the board content.

As we can now have multiple destinations, adjust `GameBoard.FindPaths` so it invokes `BecomeDestination` on each and adds them all to the frontier. That's all we need to do to support multiple destinations. All other tiles are simply cleared, as usual. Then remove the hard-coded central destination.

```
	void FindPaths () {
		foreach (GameTile tile in tiles) {
			if (tile.Content.Type == GameTileContentType.Destination) {
				tile.BecomeDestination();
				searchFrontier.Enqueue(tile);
			}
			else {
				tile.ClearPath();
			}
		}

		//tiles[tiles.Length / 2].BecomeDestination();
		//searchFrontier.Enqueue(tiles[tiles.Length / 2]);

		…
	}
```

But if we can turn tiles into destinations, it should also be possible to do the reverse, turning destinations into empty tiles. But then we could end up with a board with no destination at all. In that case `FindPaths` cannot do its job. That's the case when the frontier is empty after initializing the paths for all cells. Indicate this invalid board state by returning `false` and abort, otherwise return `true` at the end.

```
	bool FindPaths () {
		foreach (GameTile tile in tiles) {
			…
		}
		if (searchFrontier.Count == 0) {
			return false;
		}

		…
		return true;
	}
```

The simplest way to support clearing destinations is to make it a toggle operation. Clicking an empty tiles makes it a destination, while clicking a destination removes it. But content changes are now the responsibility of `GameBoard`, so give it a public `ToggleDestination` method with a tile parameter. If the tile is a destination, then make it empty and invoke `FindPaths`. Otherwise, make it a destination and also invoke `FindPaths`.

```
	public void ToggleDestination (GameTile tile) {
		if (tile.Content.Type == GameTileContentType.Destination) {
			tile.Content = contentFactory.Get(GameTileContentType.Empty);
			FindPaths();
		}
		else {
			tile.Content = contentFactory.Get(GameTileContentType.Destination);
			FindPaths();
		}
	}
```

Adding a destination can never result in a valid board state, but removing a destination can. So check whether `FindPaths` failed after making a tile empty. If so, undo the change by again turning the tile into a destination and invoke `FindPaths` once more to return to the previous valid state.

```
		if (tile.Content.Type == GameTileContentType.Destination) {
			tile.Content = contentFactory.Get(GameTileContentType.Empty);
			if (!FindPaths()) {
				tile.Content =
					contentFactory.Get(GameTileContentType.Destination);
				FindPaths();
			}
		}
```

Can we make that validity check more efficient?

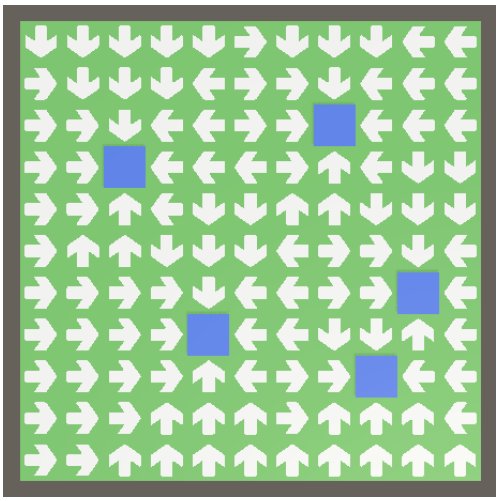
You could keep track of how many destinations there are, with an additional field. However, it only matters when the player tries to clear the last destination, which will seldom happen. Also, there will be other ways in which the board state can become invalid. We'll simply always rely on `FindPaths` to determine validity, which is very fast anyway.

We can now invoke `ToggleDestination` with the center tile at the end of `Initialize` instead of explicitly invoking `FindPaths`. This is the only time that we start with an invalid board state, but we're guaranteed to end up with a correct state.

```
	public void Initialize (
		Vector2Int size, GameTileContentFactory contentFactory
	) {
		…

		//FindPaths();
		ToggleDestination(tiles[tiles.Length / 2]);
	}
```

Finally, have `Game` invoke `ToggleDestination` instead of setting the tile content itself.

```
	void HandleTouch () {
		GameTile tile = board.GetTile(TouchRay);
		if (tile != null) {
			//tile.Content =
				//tileContentFactory.Get(GameTileContentType.Destination);
			board.ToggleDestination(tile);
		}
	}
```



Multiple destinations with valid paths.

Shouldn't we make it impossible for `Game` to set tile content directly?

Ideally, yes. We could make tiles private to the board. But we won't bother with that at this point, because `Game` and other code might need to access tiles for other purposes later. Once that's clear, we can come up with a better solution.

4 Walls

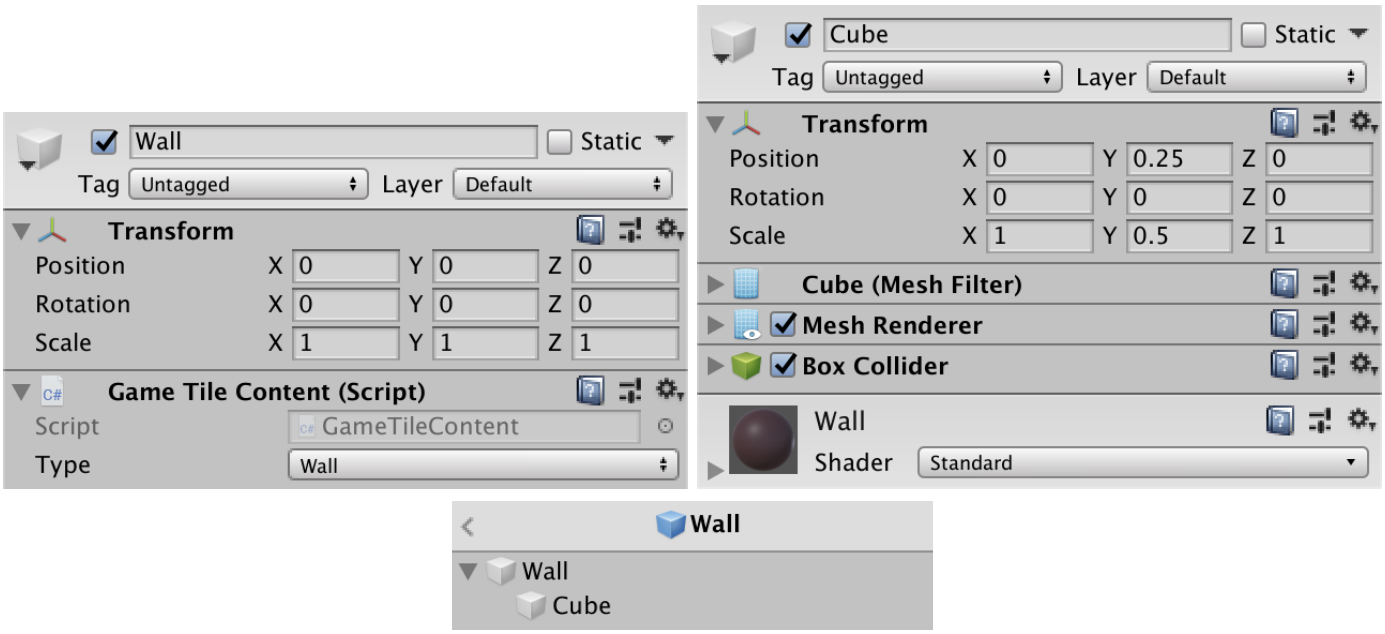
The point of a tower defense game is to make sure that enemies don't reach a destination. That's done in two ways. First, by killing them, and second by slowing them down so you have more time to kill them. On a tiled board, the primary way to give yourself more time is by increasing the distance that enemies have to travel. That's done by placing obstacles on the board, typically towers that also kill enemies, but in this tutorial we'll limit ourselves to walls.

4.1 Content

Walls are another type of content, so add an entry for them to `GameTileContentType`.

```
public enum GameTileContentType {
	Empty, Destination, Wall
}
```

Then create a wall prefab. This time, create a tile content game object and give it a cube child that's positioned so it sits on top of the board and fill the entire tile. Make it half a unit high and keep its collider, because walls can visually obstruct part of the tiles behind them. So when the player touches a wall it will affect the corresponding tile.



Wall prefab.

Add the wall prefab to the factory, both in code and the inspector.

```
	[SerializeField]
	GameTileContent wallPrefab = default;

	…

	public GameTileContent Get (GameTileContentType type) {
		switch (type) {
			case GameTileContentType.Destination: return Get(destinationPrefab);
			case GameTileContentType.Empty: return Get(emptyPrefab);
			case GameTileContentType.Wall: return Get(wallPrefab);
		}
		Debug.Assert(false, "Unsupported type: " + type);
		return null;
	}
```

Factory with wall prefab.

4.2 Toggling Walls

Add a toggle method for a wall to `GameBoard`, just like for a destination, initially without checking for an invalid board state.

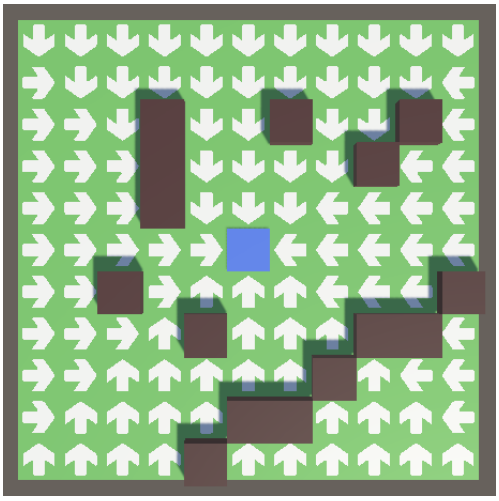
```
	public void ToggleWall (GameTile tile) {
		if (tile.Content.Type == GameTileContentType.Wall) {
			tile.Content = contentFactory.Get(GameTileContentType.Empty);
			FindPaths();
		}
		else {
			tile.Content = contentFactory.Get(GameTileContentType.Wall);
			FindPaths();
		}
	}
```

We're only going to support toggling between empty and wall tiles, not allowing walls to directly replace destinations. So only make a wall when the tile is empty. Also, the idea is that walls will block pathfinding. But every tile must have a path to a destination, otherwise enemies can get stuck. Once again, we'll have `FindPaths` check for this, and undo the change if we created an invalid board state.

```
		else if (tile.Content.Type == GameTileContentType.Empty) {
			tile.Content = contentFactory.Get(GameTileContentType.Wall);
			if (!FindPaths()) {
				tile.Content = contentFactory.Get(GameTileContentType.Empty);
				FindPaths();
			}
		}
```

Toggling walls will be much more common than toggling destinations, so have `Game` toggle a wall with the primary touch. Destinations can be toggled with the alternative touch—usually a right mouse click—which we can detect by passing 1 to `Input.GetMouseButtonDown`.

```
	void Update () {
		if (Input.GetMouseButtonDown(0)) {
			HandleTouch();
		}
		else if (Input.GetMouseButtonDown(1)) {
			HandleAlternativeTouch();
		}
	}

	void HandleAlternativeTouch () {
		GameTile tile = board.GetTile(TouchRay);
		if (tile != null) {
			board.ToggleDestination(tile);
		}
	}

	void HandleTouch () {
		GameTile tile = board.GetTile(TouchRay);
		if (tile != null) {
			board.ToggleWall(tile);
		}
	}
```



Now with walls.

Why do I get large gaps between shadows of diagonally adjacent walls?

That happens because the wall cubes barely touch along diagonals and the shadows are biased to avoid shadows artifacts. The gaps can be reduced by decreasing the shadow bias of the light, decreasing the camera's far clipping plane, and increasing the shadow map resolution. For example, I've reduces the far plane to 20 and set the light's normal bias to zero. Also, MSAA produces artifacts if combined with the default directional shadows, so I turned it off.

Let's also make it impossible for destinations to directly replace walls.

```
	public void ToggleDestination (GameTile tile) {
		if (tile.Content.Type == GameTileContentType.Destination) {
			…
		}
		else if (tile.Content.Type == GameTileContentType.Empty) {
			tile.Content = contentFactory.Get(GameTileContentType.Destination);
			FindPaths();
		}
	}
```

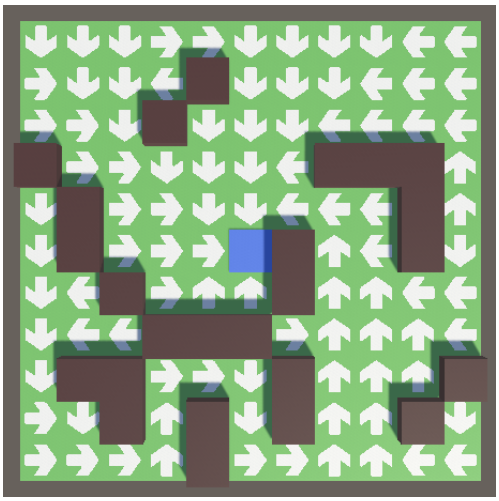
4.3 Blocking Pathfinding

To have walls block pathfinding, all we have to do is not add tiles with walls to the search frontier. We can do that by having `GameTile.GrowPathTo` not return tiles with a wall. But the path should still grown into the wall, to ensure that all tiles on the board have a path. That's needed because it might be possible that a tile with enemies in it suddenly becomes a wall.

```
	GameTile GrowPathTo (GameTile neighbor) {
		if (!HasPath || neighbor == null || neighbor.HasPath) {
			return null;
		}
		neighbor.distance = distance + 1;
		neighbor.nextOnPath = this;
		return
			neighbor.Content.Type != GameTileContentType.Wall ? neighbor : null;
	}
```

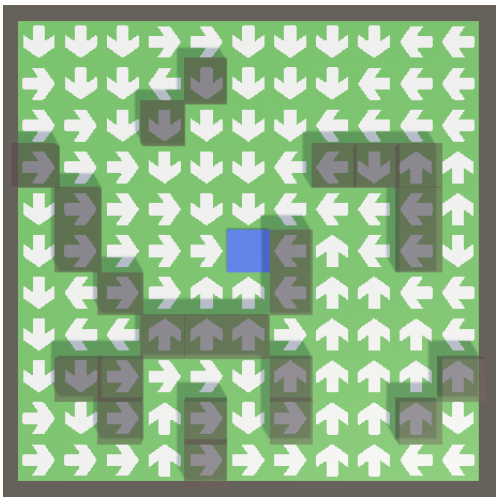
To ensure that all tiles indeed have a path, `GameBoard.FindPaths` must check this after completing the search. If this is not the case, the board state is invalid and `false` must be returned. It is not needed to update the path visualization for invalid states, because the board will be reverted to the previous state.

```
	bool FindPaths () {
		…

		foreach (GameTile tile in tiles) {
			if (!tile.HasPath) {
				return false;
			}
		}

		foreach (GameTile tile in tiles) {
			tile.ShowPath();
		}
		return true;
	}
```



Walls affecting paths.

To verify that walls indeed have valid paths as well, you can make the cubes semitransparent.



Transparent walls.

Note that the requirement that all paths are valid makes it impossible to wall off a section of the board without a destination in it. You can split the map, but only if each part has at least one destination. Also, each wall must be adjacent to an empty tile or a destination, otherwise it cannot have a path itself. For example, you cannot make a solid 3×3 block of walls.

4.4 Hiding the Paths

The path visualization allows us to see how pathfinding works and verify that it is indeed correct, but it's not intended to be shown to the player, at least not always. So let's make it possible to hide the arrows. We can do that by adding a public `HidePath` method to `GameTile` that simply deactivates its arrow.

```
	public void HidePath () {
		arrow.gameObject.SetActive(false);
	}
```

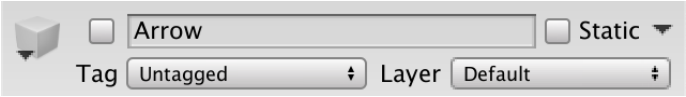
Whether the paths are shown is part of the board state. Give `GameBoard` a boolean field—set to `false` by default—to keep track of its state along with a public property to get and set it. The setter should either show or hide the paths of all tiles.

```
	bool showPaths;

	public bool ShowPaths {
		get => showPaths;
		set {
			showPaths = value;
			if (showPaths) {
				foreach (GameTile tile in tiles) {
					tile.ShowPath();
				}
			}
			else {
				foreach (GameTile tile in tiles) {
					tile.HidePath();
				}
			}
		}
	}
```

Now `FindPaths` only has to show the updated paths if the visualization is enabled.

```
	bool FindPaths () {
		…

		if (showPaths) {
			foreach (GameTile tile in tiles) {
				tile.ShowPath();
			}
		}
		return true;
	}
```

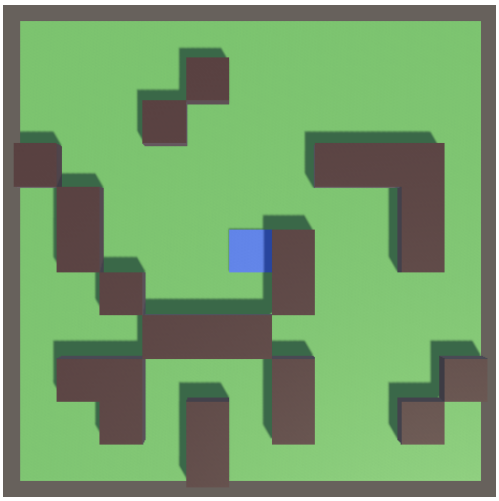
The path visualization is disabled by default. Disable the arrow of the tile prefab as well.



Prefab arrow inactive by default.

Make `Game` toggle the visualization with a key press. The P key is an obvious choice, but that interferes with the default keyboard shortcut to enter and exit play mode in the Unity editor. The result would be that the visualization toggles when the shortcut is used to exit play mode, which doesn't look nice. So let's use the V key instead, for visualization or verbose.

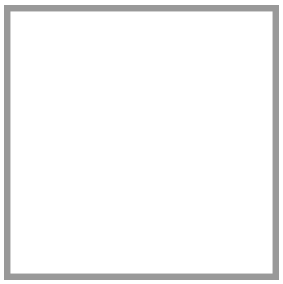
```
	void Update () {
		…

		if (Input.GetKeyDown(KeyCode.V)) {
			board.ShowPaths = !board.ShowPaths;
		}
	}
```



Without arrows.

4.5 Showing the Grid

When the arrows are hidden it become difficult to see the location of each tile. Let's add grid lines to make this easier. Here is a grid texture with a square border that can be used to outline a single tile.



Grid texture.

We won't add this texture to each tile individually, instead we'll apply it to the ground. But we'll make the grid optional, just like the path visualization. So add a `Texture2D` configuration field to `GameBoard` and set it to the grid texture.

```
	[SerializeField]
	Texture2D gridTexture = default;
```

Board with grid texture.

Add another boolean field and property to control the grid visualization state. In this case the setter has to get hold of the ground's material, which it can do by invoking `GetComponent<MeshRenderer>` on the ground and accessing the `material` property of the result. If the grid should be shown, assign the grid texture to the material's `mainTexture` property. Otherwise, assign `null` to it. Note that changing the material's texture will create a duplicate material instance, so it becomes independent of the material asset.

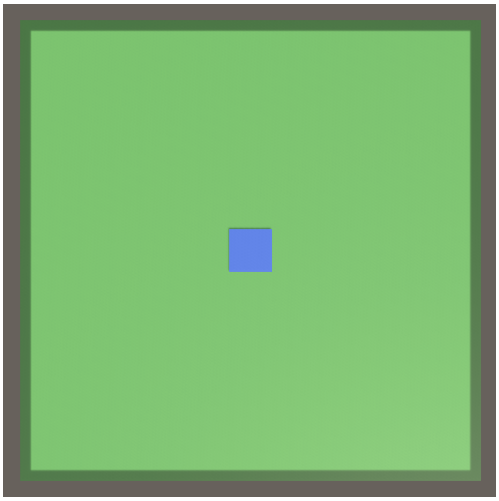
```
	bool showGrid, showPaths;

	public bool ShowGrid {
		get => showGrid;
		set {
			showGrid = value;
			Material m = ground.GetComponent<MeshRenderer>().material;
			if (showGrid) {
				m.mainTexture = gridTexture;
			}
			else {
				m.mainTexture = null;
			}
		}
	}
```

Make `Game` toggle the grid visualization with the G key.

```
	void Update () {
		…
		if (Input.GetKeyDown(KeyCode.G)) {
			board.ShowGrid = !board.ShowGrid;
		}
	}
```

Also, let's enable the grid visualization by default, in `Awake`.

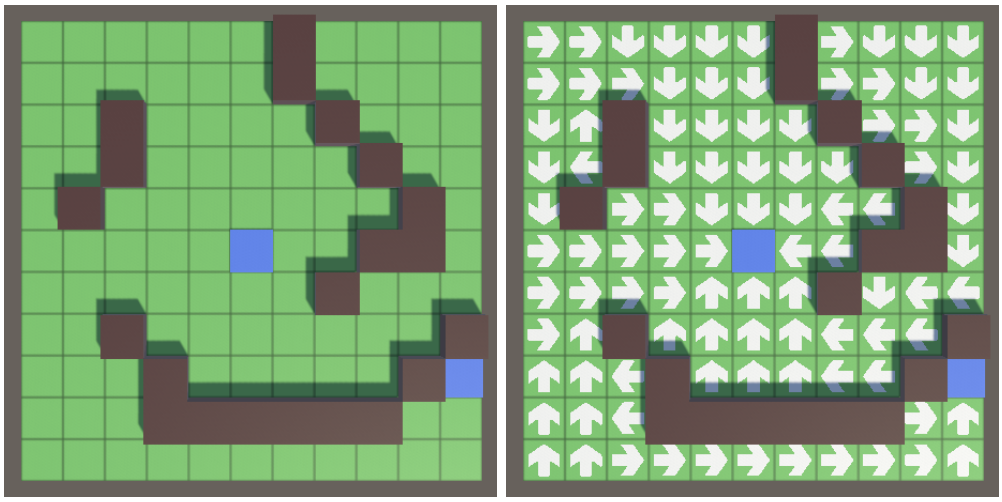
```
	void Awake () {
		board.Initialize(boardSize, tileContentFactory);
		board.ShowGrid = true;
	}
```



Unscaled grid.

At this point we end up with a border around the entire board. That matches the texture, but is not what we want. We have to scale the material's main texture so it matches the grid size. We can do that by invoking the material's `SetTextureScale` method with the texture property name—which is *_MainTex*— and a 2D size. We can directly use the board size, which gets implicitly converted to a `Vector2` value.

```
		if (showGrid) {
			m.mainTexture = gridTexture;
			m.SetTextureScale("_MainTex", size);
		}
```



Scaled grid, without and with path visualization.

At this point we have a functional board for our grid-based tower defense game. The next tutorial is Enemies.

repository

made by Jasper Flick