Maze, building your own randomized place

In this tutorial we'll generate a maze with multiple distinct areas and navigate through it. You'll learn to

- fill a 2D rectangle with a maze-generating algorithm;
- visualize the algorithm by using a coroutine;
- place walls and doors;
- use object inheritance;
- use extension methods;
- move through the maze;
- combine first-person view and an overlay map;
- determine visible rooms.

You're assumed to know the basics of the Unity editor and scripting. If you've completed the Clock and Fractal tutorials you're good to go.

This tutorial requires at least Unity 4.5. It won't work with earlier versions.

Enter a random maze of your own creation.

Random Mazes

You've probably seen quite a few randomly generated mazes, either in digital form or in print.

There is a huge variety of maze types, but fundamentally they always boil down to the same thing. A maze is a collection or areas linked together such that you can start anywhere and from there be able to visit every other area. The shape and layout of these areas and how exactly they are connected defines the character of the maze.

It's time to generate our own maze! You can try out the final version of what we're going to make right here.

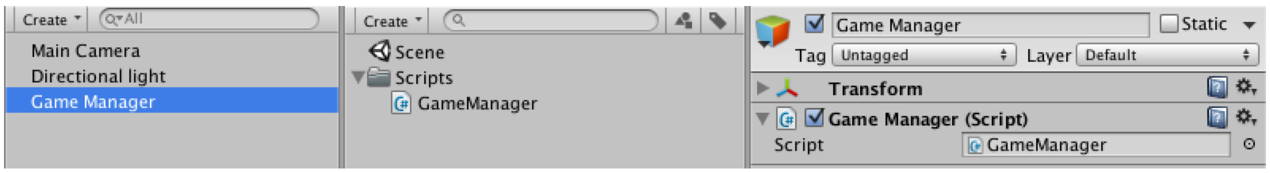
Try out the Maze

Press **space** to restart the maze generation. Once it's done, you can navigate the player avatar with the **arrow** keys or **WASD** and rotate with **QE**. You can **right-click** to go fullscreen.

Game Flow

If we were to make a game, we would first have to generate a maze and then spawn a player avatar that can navigate that maze. Then whenever a new game is started, we have to destroy the current maze, generate a new one, and place the avatar in it again. Let's create a game manager to take care of this.

Create a new project and place a default directional light somewhere out of the way for some basic lighting. Then add a new `GameManager` C# script. Let's arrange the assets by type, so put it in a new *Scripts* folder. Then create a new empty game object named *Game Manager* and add our new script component to it.



The Basics.

Our `GameManager` component simply begins the game when its `Start` method is called. We also let it restart the game whenever the player presses space. To support that, we need to check each update whether the **space** key has been pressed.

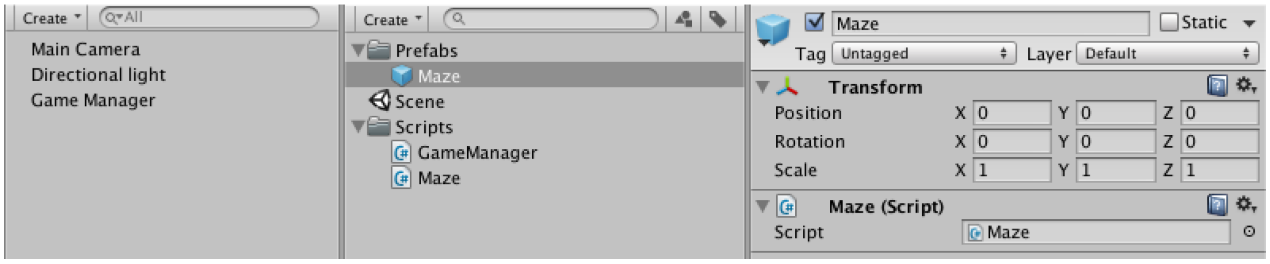
How does GetKeyDown work?

```
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {

	private void Start () {
		BeginGame();
	}

	private void Update () {
		if (Input.GetKeyDown(KeyCode.Space)) {
			RestartGame();
		}
	}

	private void BeginGame () {}

	private void RestartGame () {}
}
```

To begin a game we need to create a maze. So let's add a `Maze` script, then create a new empty game object named *Maze* and attach the script to it. Turn it into a prefab by dragging it into a new *Prefabs* folder that we also create to hold it. Once that's done, get rid of the instance in the hierarchy.

What's a prefab?

```
using UnityEngine;
using System.Collections;

public class Maze : MonoBehaviour {}
```


Create
Main Camera
Directional light
Game Manager

Create
Prefabs
Maze
Scene
Scripts
GameManager
Maze

Maze
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Maze (Script)
Script Maze

Maze prefab.

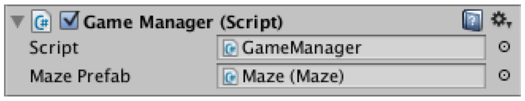
Now we can add a reference to this prefab to `GameManager` so it can create instances of it. Add a public variable for the prefab reference and a private one to hold the instance. Then we can instantiate a maze in `BeginGame` and destroy it in `RestartGame` before we begin a new game.

What does `Instantiate` do?

What does `as` do?

What does `Destroy` do?

```
	public Maze mazePrefab;

	private Maze mazeInstance;

	private void BeginGame () {
		mazeInstance = Instantiate(mazePrefab) as Maze;
	}

	private void RestartGame () {
		Destroy(mazeInstance.gameObject);
		BeginGame();
	}
```


Game Manager (Script)
Script GameManager
Maze Prefab Maze (Maze)

Game Manager can now create a maze.

Maze Fundamentals

Right now the game manager already does its job. When entering play mode, a maze instance is created, while pressing space destroys it and makes a new one. Now it's up to `Maze` to generate its contents.

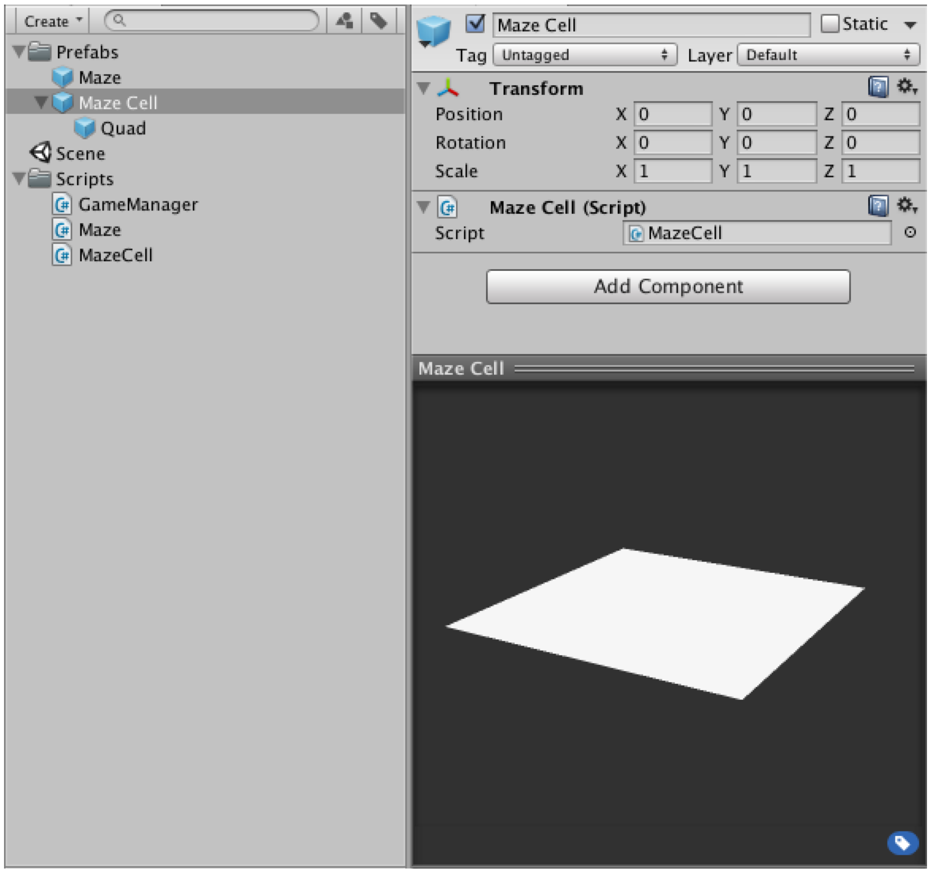
We are going to create a flat maze by filling a rectangular grid of configurable size. I'll make it 20 by 20. We'll store the cells in a 2D array and create a new `MazeCell` script to represent the cells. We also need a cell prefab to instantiate.

```
	public int sizeX, sizeZ;

	public MazeCell cellPrefab;

	private MazeCell[,] cells;
```

```
using UnityEngine;
```

```
public class MazeCell : MonoBehaviour {}
```

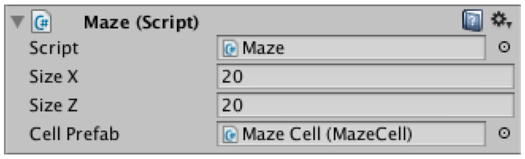
We need a 3D visualization for our cells. Create a new game object named *Maze Cell* and add the `MazeCell` component to it. Then create a default quad object, make it a child of the cell and set its rotation to (90,0,0). That gives us a very simple floor tile that fills the cell's area. Turn the whole thing into a prefab, get rid of the instance, and give `Maze` a reference to it.



Maze Cell   Static
Tag Untagged   Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Maze Cell (Script)
Script MazeCell



Maze (Script)
Script Maze
Size X 20
Size Z 20
Cell Prefab Maze Cell (MazeCell)

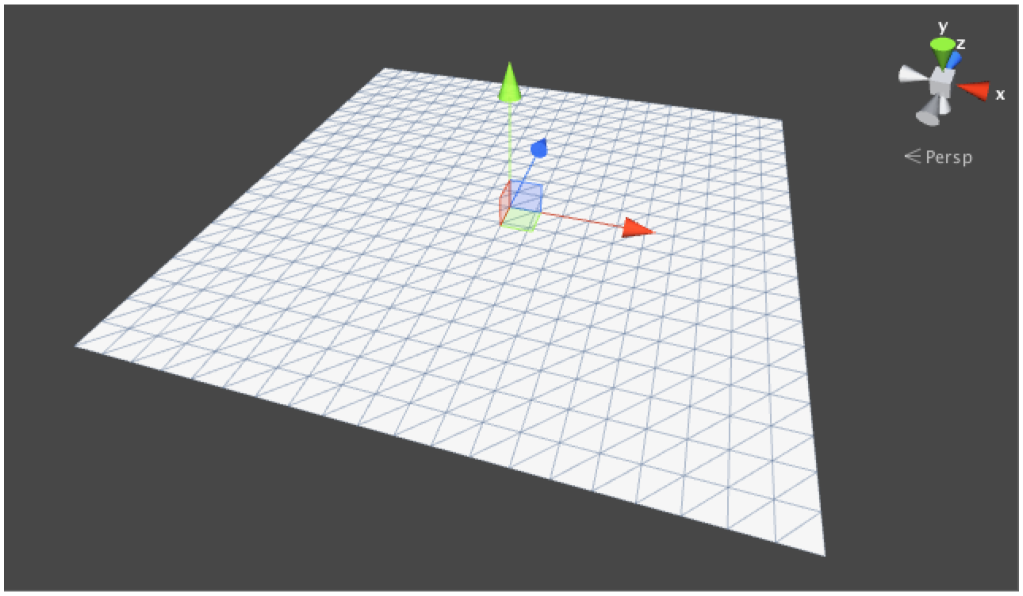
Maze cell prefab and a configured maze.

We should now add a `Generate` method to `Maze` that will take care of constructing the maze contents. We start with creating our 2D array and simply filling the entire grid with new cells by means of a double for-loop. We put the creation of individual cells in its own method. We instantiate a new cell, put it in the array and give it a descriptive name. We also make it a child object of our maze and position it so that the entire grid is centered.

```
	public void Generate () {
		cells = new MazeCell[sizeX, sizeZ];
		for (int x = 0; x < sizeX; x++) {
			for (int z = 0; z < sizeZ; z++) {
				CreateCell(x, z);
			}
		}
	}

	private void CreateCell (int x, int z) {
		MazeCell newCell = Instantiate(cellPrefab) as MazeCell;
		cells[x, z] = newCell;
		newCell.name = "Maze Cell " + x + ", " + z;
		newCell.transform.parent = transform;
		newCell.transform.localPosition = new Vector3(x - sizeX * 0.5f + 0.5f, 0f, z - sizeZ * 0.5f + 0.5f);
	}
```

Now let `GameManager` call `Generate` and the maze should appear when you enter play mode.

```
	private void BeginGame () {
		mazeInstance = Instantiate(mazePrefab) as Maze;
		mazeInstance.Generate();
	}
```



20 by 20 maze cells.

We get a full grid of cells, but we can't immediately see in what order the cells were generated. It would be useful – and even a bit of fun – to slow down the generation process so we could see how it works. We can do this by turning `Generate` into a coroutine and inserting some delay before each step. I'll set it to 0.01 seconds, which means generating 20 by 20 cells would take roughly four seconds, assuming your frame rate is high enough.

How does `WaitForSeconds` work?

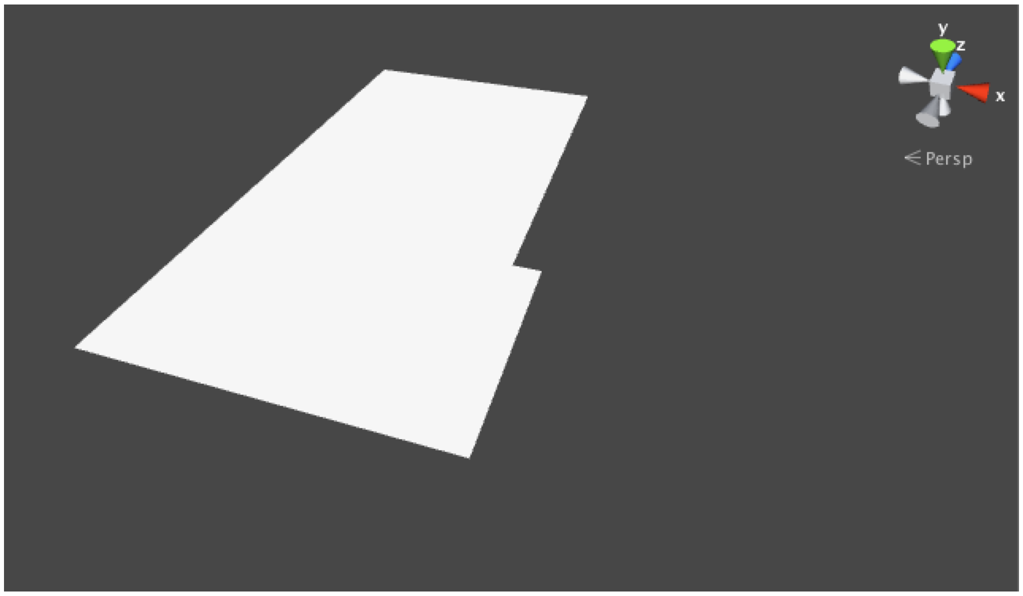
```
	public float generationStepDelay;

	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[sizeX, sizeZ];
		for (int x = 0; x < sizeX; x++) {
			for (int z = 0; z < sizeZ; z++) {
				yield return delay;
				CreateCell(x, z);
			}
		}
	}
```

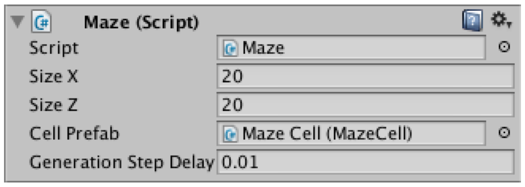
We now have to change `GameManager` so it starts the coroutine properly. Also, it is important to stop the coroutine when the game is restarted, because it might not have finished generating yet. As we only have to worry about one coroutine, we can take care of this by simply calling `StopAllCoroutines`. So yes, you can press space while a maze is still being generated and it will immediately start generating a new one.

Where does a coroutine live?

```
	private void BeginGame () {
		mazeInstance = Instantiate(mazePrefab) as Maze;
		StartCoroutine(mazeInstance.Generate());
	}
```

```
        private void RestartGame () {
                StopAllCoroutines();
                Destroy(mazeInstance.gameObject);
                BeginGame();
        }
```




Maze (Script)

Script	Maze
Size X	20
Size Z	20
Cell Prefab	Maze Cell (MazeCell)
Generation Step Delay	0.01

Maze generation with step delay.

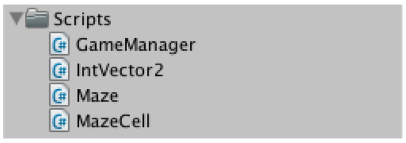
Cell Coordinates and Integer Vectors

To generate a real maze, we will be adding cells to our maze in a random way instead of using the double loop that we're using at this moment. So we will probably be using maze coordinates to figure out where we are at any given step. As we are operating in a 2D space, we need to use two integers. It would be convenient if we could manipulate the coordinates as a single value, like `Vector2` but with ints instead of floats. Unfortunately such a structure does not exist, but we can create one ourselves.

Let's add a new `IntVector2` script and make it a `struct` instead of a `class`. We give it a public `x` and `z` integer. That gives us two integers bundled together as a single value. We'll also add a special constructor method to it, which allows us to define values via `new IntVector2(1, 2)`.

Why isn't `IntVector2` immutable?

```
public struct IntVector2 {

        public int x, z;

        public IntVector2 (int x, int z) {
                this.x = x;
                this.z = z;
        }
}
```



We now have an integer vector.

We will most likely be adding these vectors together at some point. We could create a method for that. But it would be even more convenient if we could simply use the + operator. Fortunately, we can do this by creating an operator method, which is how Unity's vectors support operation as well. So yes, adding two vectors means that you're calling a method.

Let's add support for the + operator now. You can define the other operators as well, but addition is all we need here.

Aren't we changing a here?

```
	public static IntVector2 operator + (IntVector2 a, IntVector2 b) {
		a.x += b.x;
		a.z += b.z;
		return a;
	}
```

Now we can use our integer vector type to add coordinates to `MazeCell`.

Shouldn't cell coordinates be fixed?

```
	public IntVector2 coordinates;
```

And we can adjust `Maze` so it uses `IntVector2` when creating the cells and for its size as well, instead of using two separate integers.

```
	public IntVector2 size;

	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[size.x, size.z];
		for (int x = 0; x < size.x; x++) {
			for (int z = 0; z < size.z; z++) {
				yield return delay;
				CreateCell(new IntVector2(x, z));
			}
		}
	}

	private void CreateCell (IntVector2 coordinates) {
		MazeCell newCell = Instantiate(cellPrefab) as MazeCell;
		cells[coordinates.x, coordinates.z] = newCell;
		newCell.coordinates = coordinates;
		newCell.name = "Maze Cell " + coordinates.x + ", " + coordinates.z;
		newCell.transform.parent = transform;
		newCell.transform.localPosition =
			new Vector3(coordinates.x - size.x * 0.5f + 0.5f, 0f, coordinates.z - size.z * 0.5f + 0.5f);
	}
```

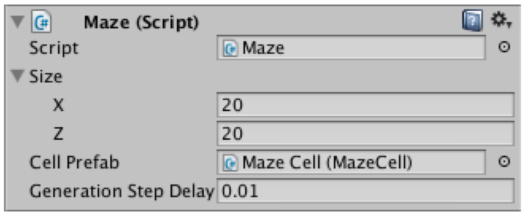
Unfortunately there's something wrong now. The maze's size no longer shows up in the inspector. This is because Unity does not save our custom struct. Fortunately, this is easy to solve by adding the `Serializable` attribute from the `System` namespace to `IntVector2`.

What's an attribute?

How does serialization work?

Can a struct be Serializable?

```
[System.Serializable]
public struct IntVector2
```



Maze size as an integer vector.

Random Cell Generation

Let's do away with our double loop that `Maze` uses to generate a regular pattern of cells. Instead we'll pick some random coordinates inside the maze and start generating a line of cells from there, until we run out of the maze.

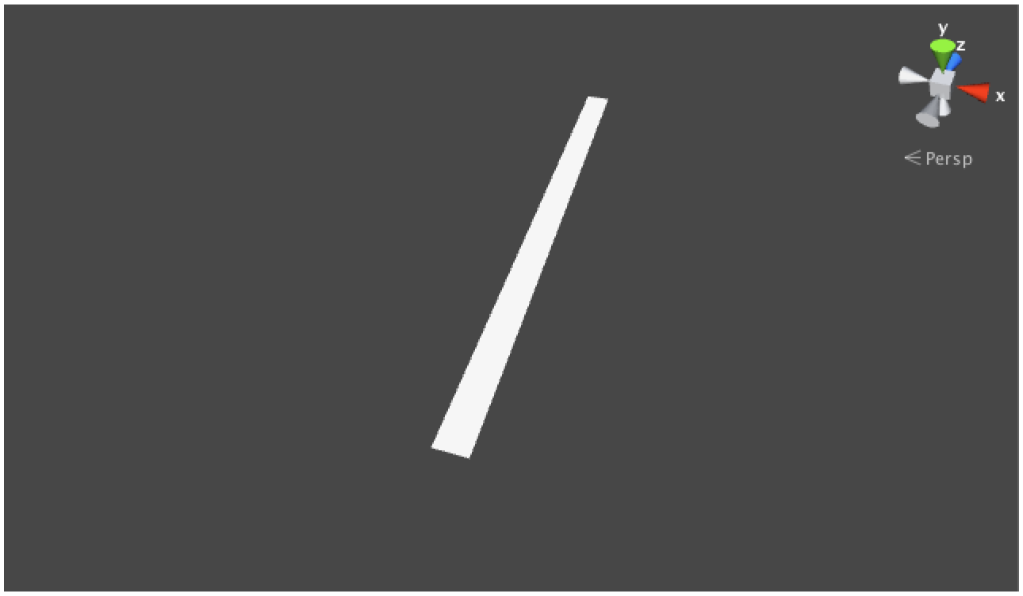
```
	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[size.x, size.z];
		IntVector2 coordinates = RandomCoordinates;
		while (ContainsCoordinates(coordinates)) {
			yield return delay;
			CreateCell(coordinates);
			coordinates.z += 1;
		}
	}
```

To make this work we have to also add a `RandomCoordinates` property to `Maze` that produces some coordinates inside it, plus a `ContainsCoordinates` method that checks whether some coordinates fall inside the maze. Let's make them public as they would be useful for anything that deals with mazes.

How do properties work?

What does && do?

```
	public IntVector2 RandomCoordinates {
		get {
			return new IntVector2(Random.Range(0, size.x), Random.Range(0, size.z));
		}
	}

	public bool ContainsCoordinates (IntVector2 coordinate) {
		return coordinate.x >= 0 && coordinate.x < size.x && coordinate.z >= 0 && coordinate.z < size.z;
	}
```

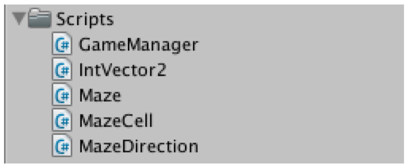


A random line of cells along the Z axis.

But we don't want to walk in a straight line, we want to move in a random direction each step. But what directions are there to choose from? Let's create a `MazeDirection` enum type to explicitly define that we have the north, east, south, and west directions. Place it in its own script file.

What's an **enum**?

```
using UnityEngine;

public enum MazeDirection {
	North,
	East,
	South,
	West
}
```



Getting a sense of direction.

Now it would be handy if we could ask for a random direction. Unfortunately an enum is not a class or a struct, so we cannot define methods or properties inside it. What we could do is add another static class and put a random property there. Let's use the plural version as its name and place it in the same file as `MazeDirection`. We also add a `Count` constant so we have an official way to know how many directions there are.

```
public static class MazeDirections {

	public const int Count = 4;

	public static MazeDirection RandomValue {
		get {
			return (MazeDirection)Random.Range(0, Count);
		}
	}
}
```

Now we can get a random direction, but how do we adjust the current coordinates based on that? It would be convenient if we could convert a direction into an integer vector somehow. Let's add a method to `MazeDirections` to take care of that. We'll use a private static array of vectors to make this conversion easy.

```
	private static IntVector2[] vectors = {
		new IntVector2(0, 1),
		new IntVector2(1, 0),
		new IntVector2(0, -1),
		new IntVector2(-1, 0)
	};

	public static IntVector2 ToIntVector2 (MazeDirection direction) {
		return vectors[(int)direction];
	}
```

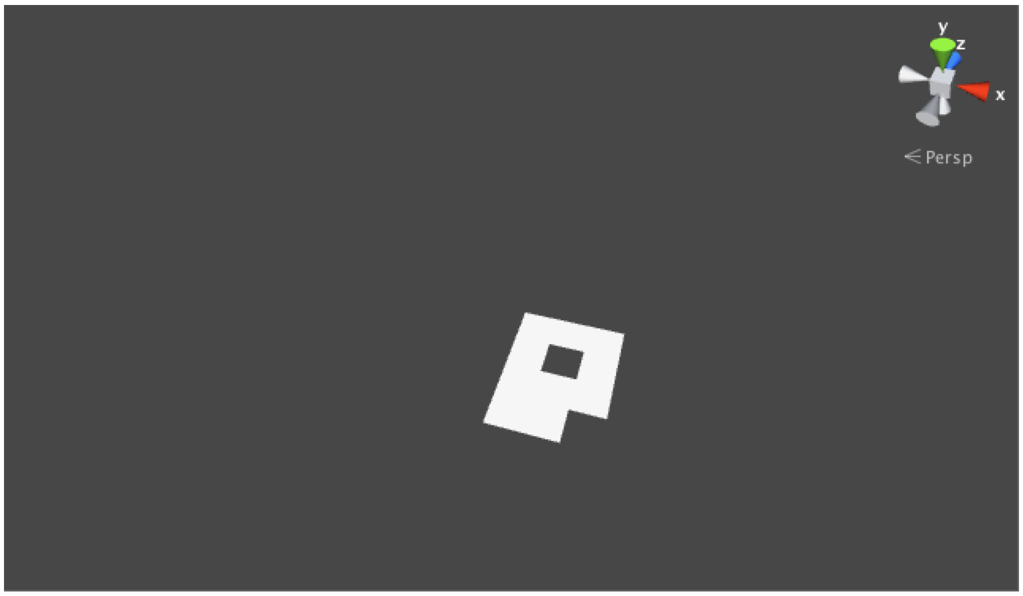
This allows us to convert an arbitrary direction into an integer vector via `MazeDirections.ToIntVector2(someDirection)`. While this works, it looks aweful. It would've been convenient if we could do it via `someDirection.ToIntVector2()` instead. The good news is that we can achieve this by using an extension method. With just a slight change to `ToIntVector2` it will behave as if it were an instance method of `MazeDirection`.

What's an extension method?

```
	public static IntVector2 ToIntVector2 (this MazeDirection direction) {
		return vectors[(int)direction];
	}
```

With these additions it is now easy to have `Maze` generate a new cell in a random direction each step. We do have to guard against visiting a cell more than once, so let's add a convenient method to retrieve the maze's cell at some coordinates.

```
	public MazeCell GetCell (IntVector2 coordinates) {
		return cells[coordinates.x, coordinates.z];
	}

	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[size.x, size.z];
		IntVector2 coordinates = RandomCoordinates;
		while (ContainsCoordinates(coordinates) && GetCell(coordinates) == null) {
			yield return delay;
			CreateCell(coordinates);
			coordinates += MazeDirections.RandomValue.ToIntVector2();
		}
	}
```



A short random walk.

Backtracking

As you will see when entering play mode and pressing space a few times, only a few cells get generated each time. This happens because it is likely that we bump into an already filled cell. One way to improve our approach is to keep track of a list of active cells. Each time we create a cell, we add it to this list. Then the next generation step we try to move one random step from the last cell in this list. If we cannot do this move, instead of immediately stopping, we remove the current cell from the active list. This way we will do a step backward and try again each time we fail, until the list is empty.

We're going to use a list of maze cells, so start by adding the `Systems.Collections.Generic` namespace to those used by `Maze`.

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

public class Maze : MonoBehaviour
```

Then we create a temporary list inside the `Generate` method. To keep the method simple, let's put the generation steps in their own methods and supply the active list to them as an argument.

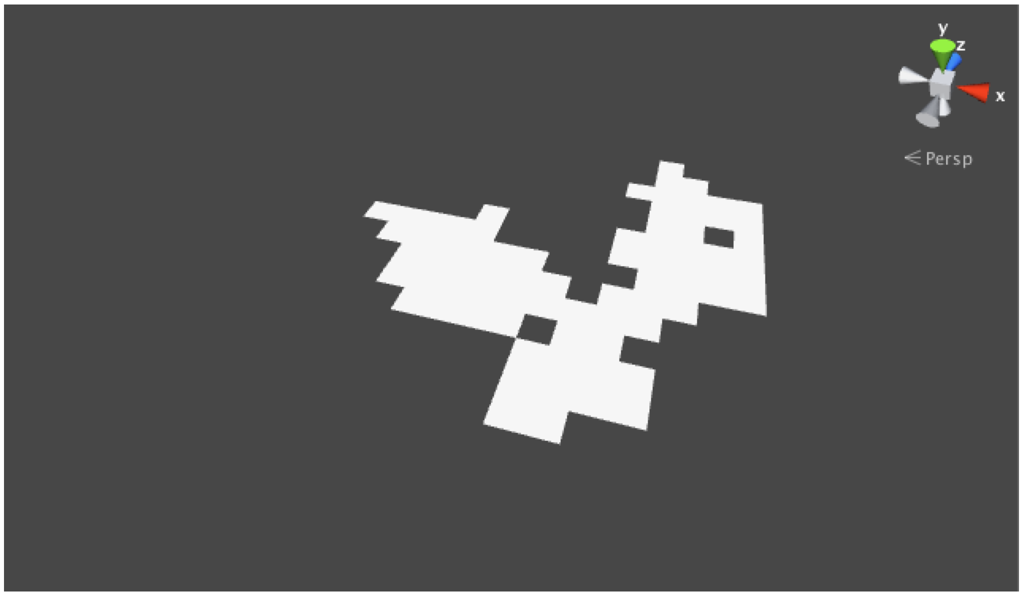
```
	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[size.x, size.z];
		List<MazeCell> activeCells = new List<MazeCell>();
		DoFirstGenerationStep(activeCells);
		while (activeCells.Count > 0) {
			yield return delay;
			DoNextGenerationStep(activeCells);
		}
	}
```

The `DoFirstGenerationStep` method is very short right now. The `DoNextGenerationStep` is a bit longer, because it has to retrieve the current cell, check whether the move is possible, and take care of removing cells from the list.

```
	private void DoFirstGenerationStep (List<MazeCell> activeCells) {
		activeCells.Add(CreateCell(RandomCoordinates));
	}

	private void DoNextGenerationStep (List<MazeCell> activeCells) {
		int currentIndex = activeCells.Count - 1;
		MazeCell currentCell = activeCells[currentIndex];
		MazeDirection direction = MazeDirections.RandomValue;
		IntVector2 coordinates = currentCell.coordinates + direction.ToIntVector2();
		if (ContainsCoordinates(coordinates) && GetCell(coordinates) == null) {
			activeCells.Add(CreateCell(coordinates));
		}
		else {
			activeCells.RemoveAt(currentIndex);
		}
	}
```

One additional change needed to make this work is to let `CreateCell` return the new cell that it creates.

```
	private MazeCell CreateCell (IntVector2 coordinates) {
		MazeCell newCell = Instantiate(cellPrefab) as MazeCell;
		cells[coordinates.x, coordinates.z] = newCell;
		newCell.coordinates = coordinates;
		newCell.name = "Maze Cell " + coordinates.x + ", " + coordinates.z;
		newCell.transform.parent = transform;
		newCell.transform.localPosition =
			new Vector3(coordinates.x - size.x * 0.5f + 0.5f, 0f, coordinates.z - size.z * 0.5f + 0.5f);
		return newCell;
	}
```



A larger walk with backtracking.

Connecting the Cells

While we now tend to generate longer paths of cells, it's still far from a complete maze. We should really be smart about how we move from cell to cell.

It's time to keep track of the connections between cells. Each cell has four edges, each of which connects to a neighboring cell, unless it would lead outside of the maze. We could either create a a single bidirectional edge between two cells, or give each their own unidirectional edge. We choose the latter approach, because it is more flexible.

Add a script for the new `MazeCellEdge` component type. Give it a reference to the cell it belongs to and one to the other cell that it connects with. Also give it a direction so we remember its orientation.

```
using UnityEngine;

public class MazeCellEdge : MonoBehaviour {

	public MazeCell cell, otherCell;

	public MazeDirection direction;
}
```

We want to make the edges children of their cells and place them in the same location. Also, once an edge is created its cell should know about it as well. Let's create an `Initialize` method to take care of this.

```
	public void Initialize (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		this.cell = cell;
		this.otherCell = otherCell;
		this.direction = direction;
		cell.SetEdge(direction, this);
		transform.parent = cell.transform;
		transform.localPosition = Vector3.zero;
	}
```

Now we need to add a `SetEdge` method to `MazeCell`. Let's add a `GetEdge` method as well, because that is bound to be useful later. Our cells will store their edges in an array, but no one else needs to know how that works, so we make it private.

```
	private MazeCellEdge[] edges = new MazeCellEdge[MazeDirections.Count];

	public MazeCellEdge GetEdge (MazeDirection direction) {
		return edges[(int)direction];
	}

	public void SetEdge (MazeDirection direction, MazeCellEdge edge) {
		edges[(int)direction] = edge;
	}
```

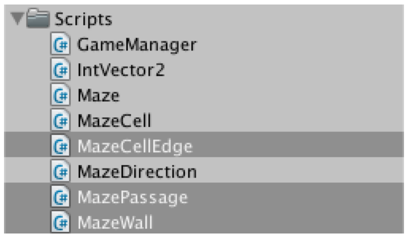
Whenever we move from one cell to a new one, we should tell both cells that the edges that connect them are now passages. Whenever we move out of the maze or bump into an already created cell, the edges should become walls instead of passages. So we really have two types of cell edges. Let's add a `MazePassage` and a `MazeWall` component that both extend `MazeCellEdge` and place them in their own files. Because we only want to use these types and never create an instance of the generic `MazeCellEdge`, we mark it as abstract.

What does it mean to be abstract?

```
public abstract class MazeCellEdge : MonoBehaviour
```

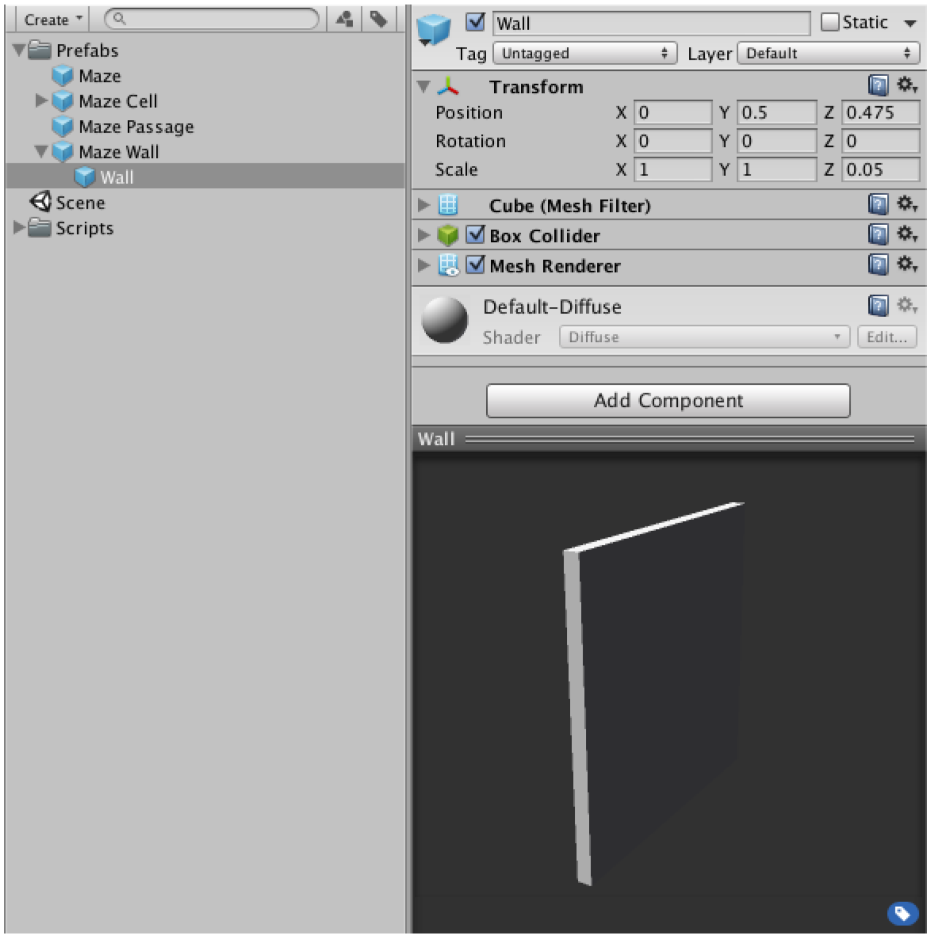
```
using UnityEngine;

public class MazePassage : MazeCellEdge {}
```

```
using UnityEngine;

public class MazeWall : MazeCellEdge {}
```



Three new edgy scripts.

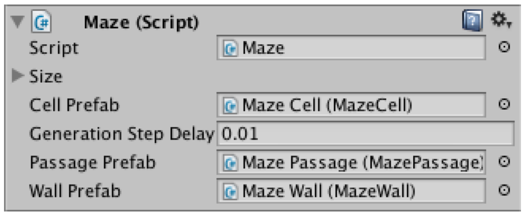
Now we can create a prefab for the passage. It's simply an empty game object with a `MazePassage` component added to it. A wall prefab can be created the same way, except that we also give it a default cube as a child. This cube is our wall's 3D representation. Let's make it 0.05 units thick and position it so that it will end up flush with the north edge of a cell.



Two new prefabs, showing the wall's cube.

With the prefabs ready, `Maze` can now get a reference to both, so it can generate instances of them.

```
public MazePassage passagePrefab;
public MazeWall wallPrefab;
```



Maze (Script)	
Script	Maze
Size	
Cell Prefab	Maze Cell (MazeCell)
Generation Step Delay	0.01
Passage Prefab	Maze Passage (MazePassage)
Wall Prefab	Maze Wall (MazeWall)

Maze with edge prefab references.

Now we can create passages and walls in `DoNextGenerationStep`. Let's assume we have convenient methods for that. When we would go out of the maze, we add a wall. If we're still inside the maze, we need to check if the current cell's neightbor doesn't exist yet. If so, we create it and place a passage in between them. But if the neighbor already exists, we separate them with a wall.

```
private void DoNextGenerationStep (List<MazeCell> activeCells) {
	int currentIndex = activeCells.Count - 1;
	MazeCell currentCell = activeCells[currentIndex];
	MazeDirection direction = MazeDirections.RandomValue;
	IntVector2 coordinates = currentCell.coordinates + direction.ToIntVector2();
	if (ContainsCoordinates(coordinates)) {
		MazeCell neighbor = GetCell(coordinates);
		if (neighbor == null) {
			neighbor = CreateCell(coordinates);
```

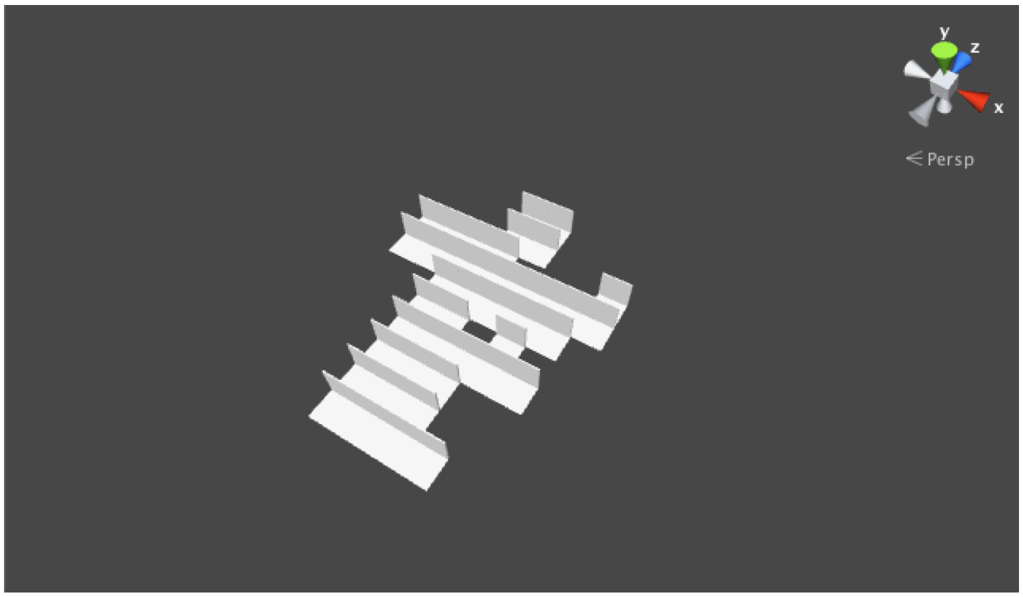
```
					CreatePassage(currentCell, neighbor, direction);
					activeCells.Add(neighbor);
				}
				else {
					CreateWall(currentCell, neighbor, direction);
					activeCells.RemoveAt(currentIndex);
				}
			}
			else {
				CreateWall(currentCell, null, direction);
				activeCells.RemoveAt(currentIndex);
			}
		}
```

The `CreatePassage` and `CreateWall` methods simply instantiate their respective prefabs and initialize them, once for both cells. The only real difference between them is that `CreateWall`'s second cell won't exist at the edge of the maze.

```
	private void CreatePassage (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazePassage passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(cell, otherCell, direction);
		passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(otherCell, cell, direction.GetOpposite());
	}

	private void CreateWall (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazeWall wall = Instantiate(wallPrefab) as MazeWall;
		wall.Initialize(cell, otherCell, direction);
		if (otherCell != null) {
			wall = Instantiate(wallPrefab) as MazeWall;
			wall.Initialize(otherCell, cell, direction.GetOpposite());
		}
	}
```

The code above makes use of a convenient `GetOpposite` method that doesn't exist yet, so let's quickly add it to `MazeDirections`.

```
	private static MazeDirection[] opposites = {
		MazeDirection.South,
		MazeDirection.West,
		MazeDirection.North,
		MazeDirection.East
	};

	public static MazeDirection GetOpposite (this MazeDirection direction) {
		return opposites[(int)direction];
	}
```



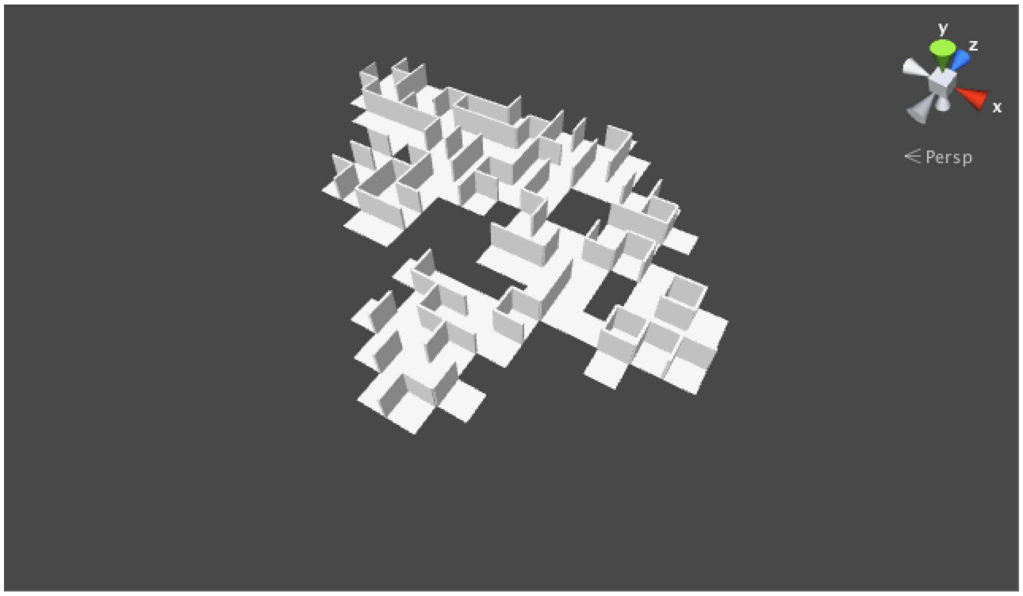
Walls without rotation.

We have now added some invisible passages and visible walls to the maze. So we can see that the walls are always on the north side of cells, which is incorrect. We fix this by rotating in the right direction in `MazeCellEdge`'s `Initialize` method.

```
	public void Initialize (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		this.cell = cell;
		this.otherCell = otherCell;
		this.direction = direction;
		cell.SetEdge(direction, this);
		transform.parent = cell.transform;
		transform.localPosition = Vector3.zero;
		transform.localRotation = direction.ToRotation();
	}
```

And yes, this means we're going to add yet another convenient method to `MazeDirections`.

```
	private static Quaternion[] rotations = {
		Quaternion.identity,
		Quaternion.Euler(0f, 90f, 0f),
		Quaternion.Euler(0f, 180f, 0f),
		Quaternion.Euler(0f, 270f, 0f)
	};

	public static Quaternion ToRotation (this MazeDirection direction) {
		return rotations[(int)direction];
	}
```



Walls with rotation.

Generating the Entire Maze

While the walls are now correctly rotated, we still don't fill the entire maze. Even worse, we're also generating completely walled-off sections, making them unreachable from anywhere else in the maze. This can happen because we choose a completely random direction each step, which could lead to us placing a wall where a passage had already been defined.

To completely fill the maze, we should only remove a cell from the active list when all its edges have been initialized. This is the first thing we should check for in `DoNextGenerationStep`, because a cell in the active list will have become fully initialized when all its neighbors have been visited. And to prevent placing incorrect walls, we should only pick a random direction that is not yet initialized for the current cell.

```
	private void DoNextGenerationStep (List<MazeCell> activeCells) {
		int currentIndex = activeCells.Count - 1;
		MazeCell currentCell = activeCells[currentIndex];
		if (currentCell.IsFullyInitialized) {
			activeCells.RemoveAt(currentIndex);
			return;
		}
		MazeDirection direction = currentCell.RandomUninitializedDirection;
		IntVector2 coordinates = currentCell.coordinates + direction.ToIntVector2();
		if (ContainsCoordinates(coordinates)) {
			MazeCell neighbor = GetCell(coordinates);
			if (neighbor == null) {
				neighbor = CreateCell(coordinates);
				CreatePassage(currentCell, neighbor, direction);
				activeCells.Add(neighbor);
			}
			else {
				CreateWall(currentCell, neighbor, direction);
			}
		}
		else {
			CreateWall(currentCell, null, direction);
		}
	}
```

We can easily check whether a cell is fully initialized by having `MazeCell` keep track of how often an edge has been set.

```
	private int initializedEdgeCount;

	public bool IsFullyInitialized {
		get {
			return initializedEdgeCount == MazeDirections.Count;
		}
	}

	public void SetEdge (MazeDirection direction, MazeCellEdge edge) {
		edges[(int)direction] = edge;
		initializedEdgeCount += 1;
	}
```

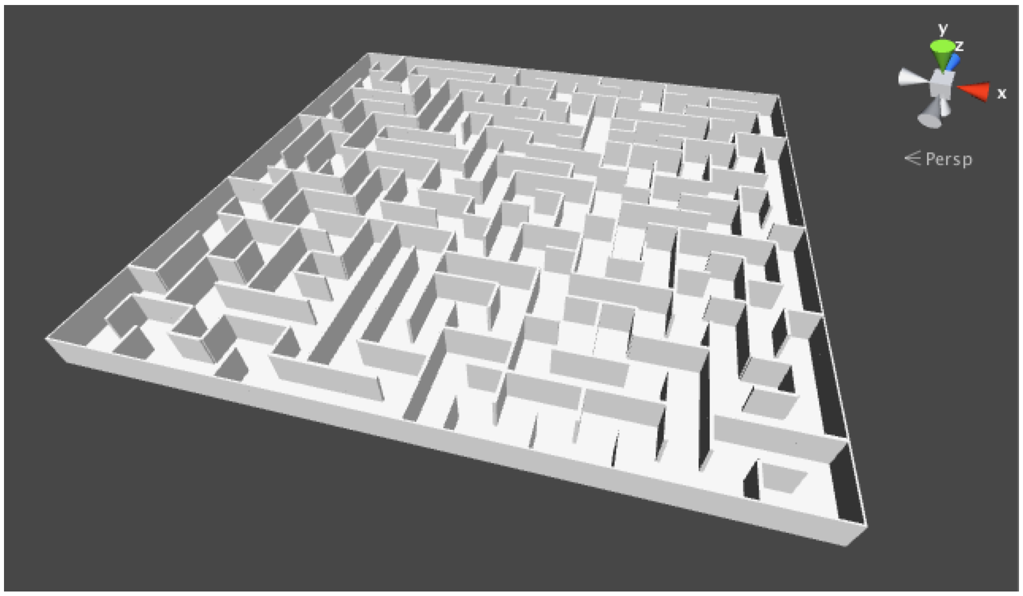
To get an unbiased random uninitialized direction is a little less straightforward. One way is to randomly decide how many uninitialized directions we should skip. Then we loop through our edges array and whenever we find a hole we check whether we are out of skips. If so, this is our direction. Otherwise, we decrease our amount of skips by one.

```
	public MazeDirection RandomUninitializedDirection {
		get {
			int skips = Random.Range(0, MazeDirections.Count - initializedEdgeCount);
			for (int i = 0; i < MazeDirections.Count; i++) {
				if (edges[i] == null) {
					if (skips == 0) {
						return (MazeDirection)i;
					}
					skips -= 1;
				}
			}
		}
	}
```

This will work as long as there are uninitialized edges remaining, otherwise we shouldn't call this method. If we did, we would run through the loop without returning and won't have any result. In fact, the compiler will complain that not all code paths return a value. We solve this by throwing an `InvalidOperationException` at the end of the method, which is the most appropriate exception for this case. This will result in a useful error message in Unity's console if we make a mistake somewhere and call this method when we shouldn't.

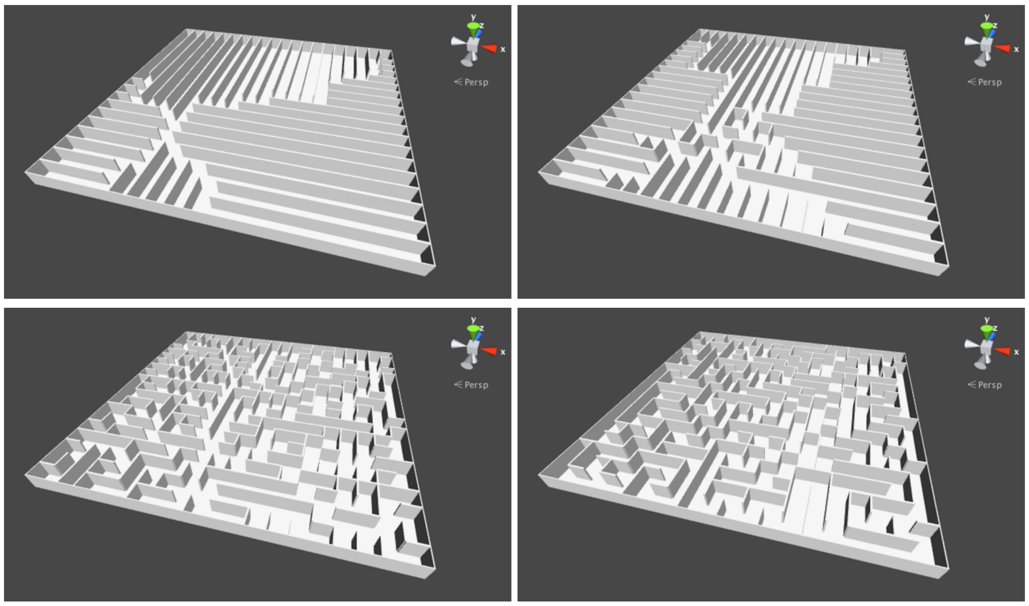
What does `throw` do?

```
	public MazeDirection RandomUninitializedDirection {
			get {
				int skips = Random.Range(0, MazeDirections.Count - initializedEdgeCount);
				for (int i = 0; i < MazeDirections.Count; i++) {
					if (edges[i] == null) {
						if (skips == 0) {
							return (MazeDirection)i;
						}
						skips -= 1;
					}
				}
				throw new System.InvalidOperationException("MazeCell has no uninitialized directions left.");
			}
		}
```



A complete maze.

Finally, we can generate a complete maze! We are now using one flavor of the Growing Tree algorithm. In case you're curious, you can change the nature of the maze you generate by using a different method to select the current index in `DoNextGenerationStep`. I have chosen to always select the last index, which causes the algorithm to dive into narrow paths that run all over the maze. Always selecting the first or the middle index will produce very different behavior. Another option is to just pick a random index. Or to choose between two approaches each step. You could even make this configurable if you like, see the Graphs tutorial for a way to do that.

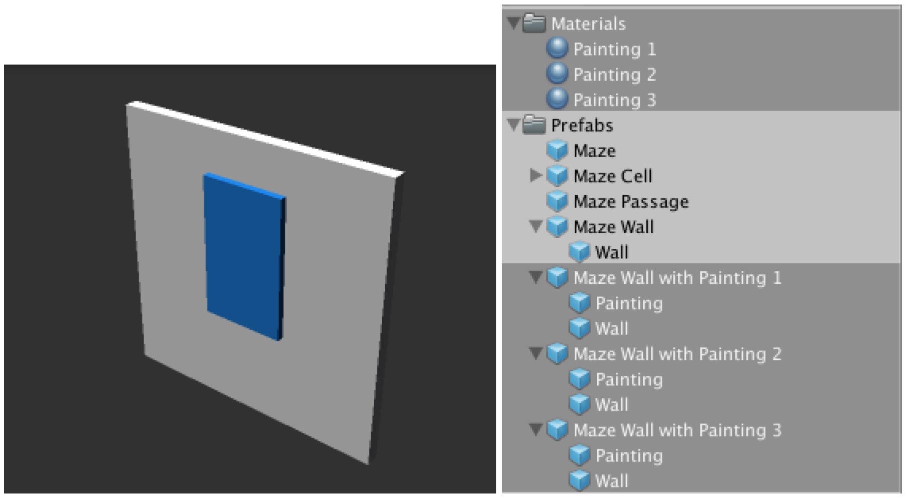


Comparing first, middle, random, and last index approaches.

Decorating the Maze

Our maze looks rather dull. Let's add some variety by introducing different wall sections. We can do this by hanging some very simple paintings on the walls.

First, create a few materials for the paintings, just to add some color variety. Put them together in a new *Materials* folder and give them any color you like. Next, drag the wall prefab into the scene and name it *Wall with Painting 1*. Add a new default cube named *Painting* to this instance. Give the cube a material, then scale and position it so it looks like it's hanging on the wall. Then turn the whole thing into a new prefab. Repeat this until you have enough paintings. I made three.

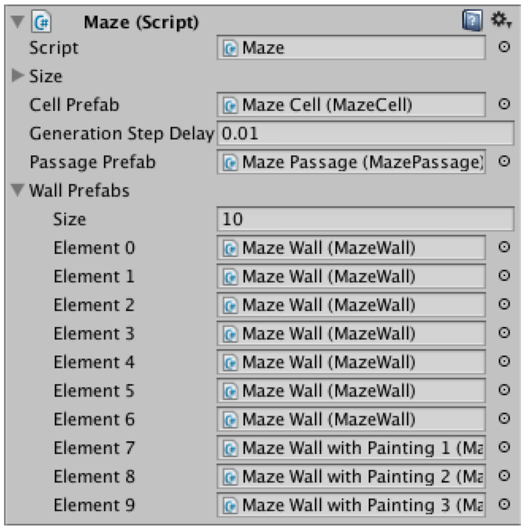


Walls with paintings.

Now change `Maze` so it has an array of wall prefabs instead of a single one. Then we can pick one at random from the array whenever we need to instantiate a new wall.

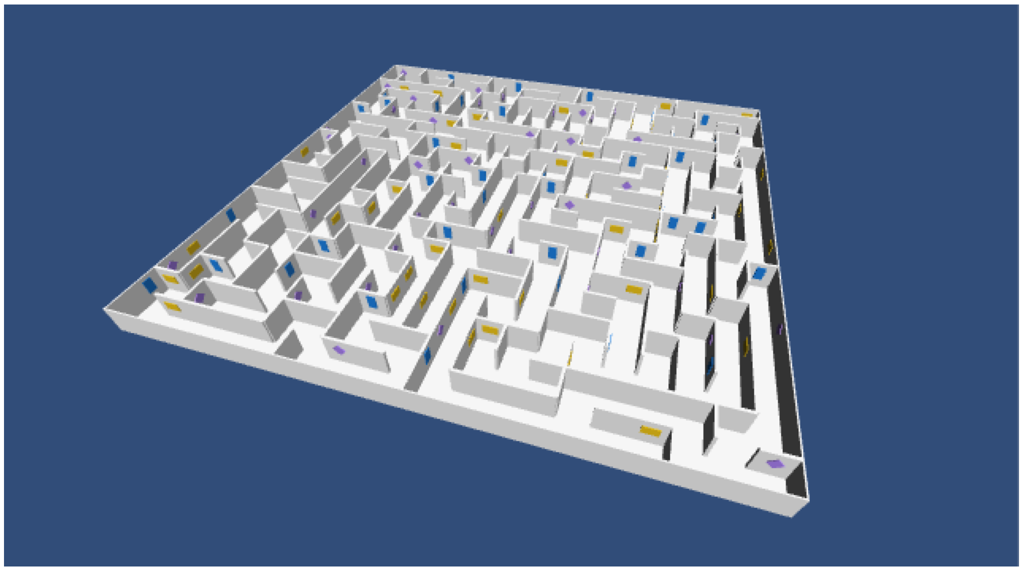
```
	public MazeWall[] wallPrefabs;

	private void CreateWall (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazeWall wall = Instantiate(wallPrefabs[Random.Range(0, wallPrefabs.Length)]) as MazeWall;
		wall.Initialize(cell, otherCell, direction);
		if (otherCell != null) {
			wall = Instantiate(wallPrefabs[Random.Range(0, wallPrefabs.Length)]) as MazeWall;
			wall.Initialize(otherCell, cell, direction.GetOpposite());
		}
	}
```

Then you can add all your wall prefabs to the array of the maze prefab. I added the empty wall multiple times, so it is more likely to be picked. Otherwise the maze will be brimming with paintings.



Maze (Script)

Script	Maze
Size	
Cell Prefab	Maze Cell (MazeCell)
Generation Step Delay	0.01
Passage Prefab	Maze Passage (MazePassage)
Wall Prefabs	
Size	10
Element 0	Maze Wall (MazeWall)
Element 1	Maze Wall (MazeWall)
Element 2	Maze Wall (MazeWall)
Element 3	Maze Wall (MazeWall)
Element 4	Maze Wall (MazeWall)
Element 5	Maze Wall (MazeWall)
Element 6	Maze Wall (MazeWall)
Element 7	Maze Wall with Painting 1 (Ma
Element 8	Maze Wall with Painting 2 (Ma
Element 9	Maze Wall with Painting 3 (Ma



Making an artsy maze.

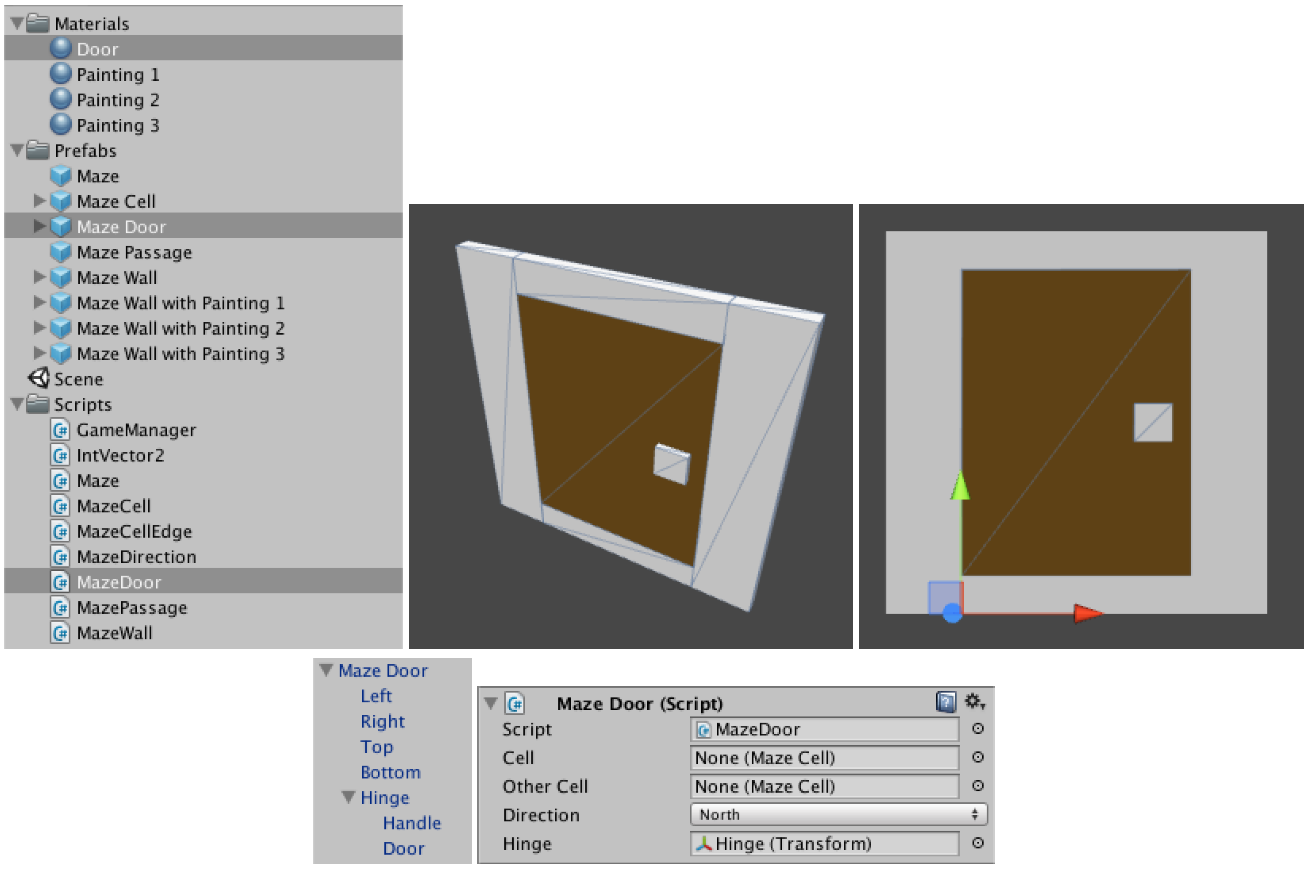
Placing Doors

Doors are another interesting element to add to our maze. Let's add a `MazeDoor` component that extends `MazePassage`. Because it will have a rotating part, add a public `Transform` variable to it named *hinge*.

```
using UnityEngine;

public class MazeDoor : MazePassage {

        public Transform hinge;
}
```

We will build a door frame from four cubes and put another cube in it with a new *Door* material, plus a door handle on the right side of it. To allow the door to rotate properly, add an empty game object named *Hinge* on the left side of the door with a Z-position of 0.5. Make the door and handle objects children of it. Add the `MazeDoor` component to the root object and connect its hinge, then turn it into a prefab.

Where are the prefab hinge's children?


Materials
Door
Painting 1
Painting 2
Painting 3
Prefabs
Maze
Maze Cell
Maze Door
Maze Passage
Maze Wall
Maze Wall with Painting 1
Maze Wall with Painting 2
Maze Wall with Painting 3
Scene
Scripts
GameManager
IntVector2
Maze
MazeCell
MazeCellEdge
MazeDirection
MazeDoor
MazePassage
MazeWall

Maze Door
Left
Right
Top
Bottom
Hinge
Handle
Door

Maze Door (Script)	
Script	MazeDoor
Cell	None (Maze Cell)
Other Cell	None (Maze Cell)
Direction	North
Hinge	Hinge (Transform)

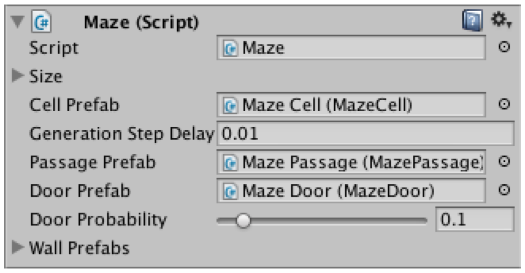
Creating a door.

`Maze` can now get a reference to the door prefab. We want to spawns doors instead of passages some of the time, but really not that often because otherwise the maze will get flooded with doors. So let's add a `doorProbabilty` configuration option and use that to decide whether we place a door or a passage. I set it to 0.1, which means that one out of ten passage will become a door.

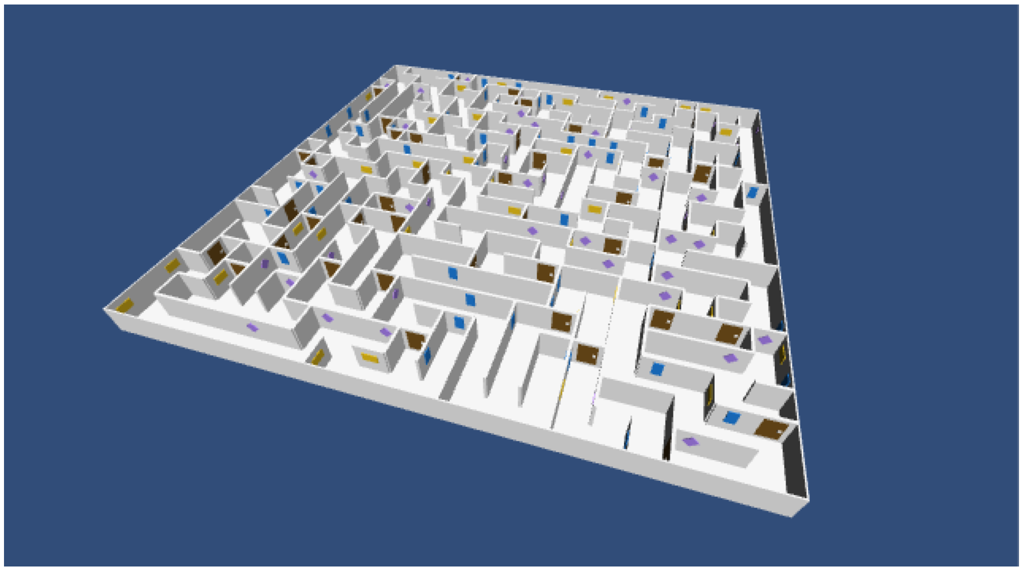
What does the question mark do?

What does **Range** do?

```
	public MazeDoor doorPrefab;

	[Range(0f, 1f)]
	public float doorProbability;

	private void CreatePassage (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazePassage prefab = Random.value < doorProbability ? doorPrefab : passagePrefab;
		MazePassage passage = Instantiate(prefab) as MazePassage;
		passage.Initialize(cell, otherCell, direction);
		passage = Instantiate(prefab) as MazePassage;
		passage.Initialize(otherCell, cell, direction.GetOpposite());
	}
```



Maze (Script)	
Script	Maze
Size	
Cell Prefab	Maze Cell (MazeCell)
Generation Step Delay	0.01
Passage Prefab	Maze Passage (MazePassage)
Door Prefab	Maze Door (MazeDoor)
Door Probability	0.1
Wall Prefabs	



Doors added to the maze.

We now get doors, but there is something wrong. All the doors are have their handle on the right side, even opposite sides of the same door! We need to make sure that the other side of a door swivels in the opposite direction.

We know that the sides of a door are created one after the other. If a door could somehow know that it was created second, it can then mirror itself. What we could do is add a convenient private property to `MazeDoor` that somehow gives us the opposite side of the door. Then we can mirror and reposition our hinge if the other side already exists. We add this functionality to `Initialize` by overriding that method with out own version. Inside it, we first call the original version and then do the additional work.

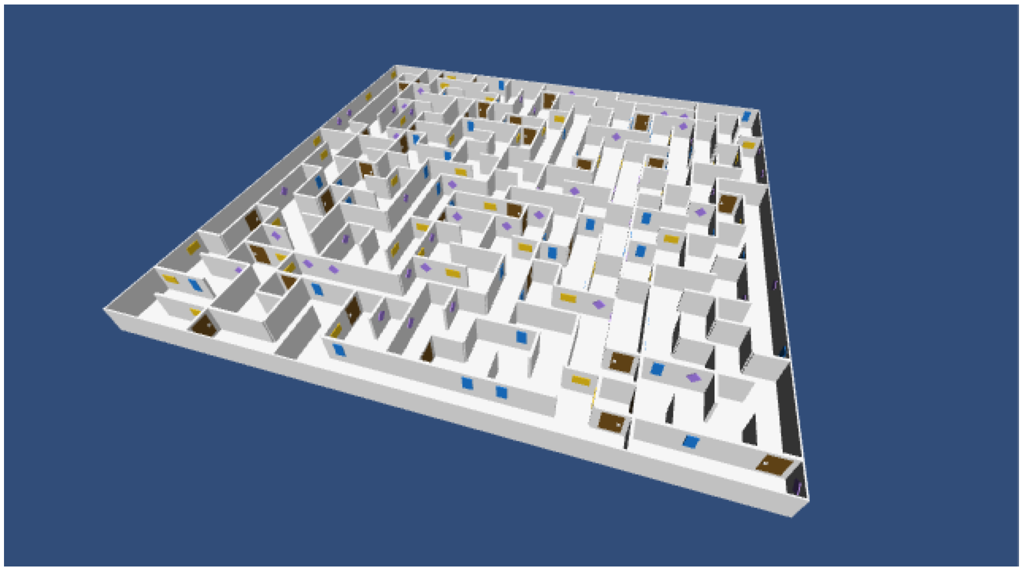
What's **base**?

```
	private MazeDoor OtherSideOfDoor {
		get {
			return otherCell.GetEdge(direction.GetOpposite()) as MazeDoor;
		}
	}

	public override void Initialize (MazeCell primary, MazeCell other, MazeDirection direction) {
		base.Initialize(primary, other, direction);
		if (OtherSideOfDoor != null) {
			hinge.localScale = new Vector3(-1f, 1f, 1f);
			Vector3 p = hinge.localPosition;
			p.x = -p.x;
			hinge.localPosition = p;
		}
	}
```

However, we can't just override any method of the class that we're extending. The original class must have declared that this is possible by marking the method as virtual. So we add the virtual keyword to the `Initialize` method of `MazeCellEdge`.

Why is **virtual** needed?

```
	public virtual void Initialize (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		this.cell = cell;
		this.otherCell = otherCell;
		this.direction = direction;
		cell.SetEdge(direction, this);
		transform.parent = cell.transform;
		transform.localPosition = Vector3.zero;
		transform.localRotation = direction.ToRotation();
	}
```


Now with correct doors.

Adding Rooms

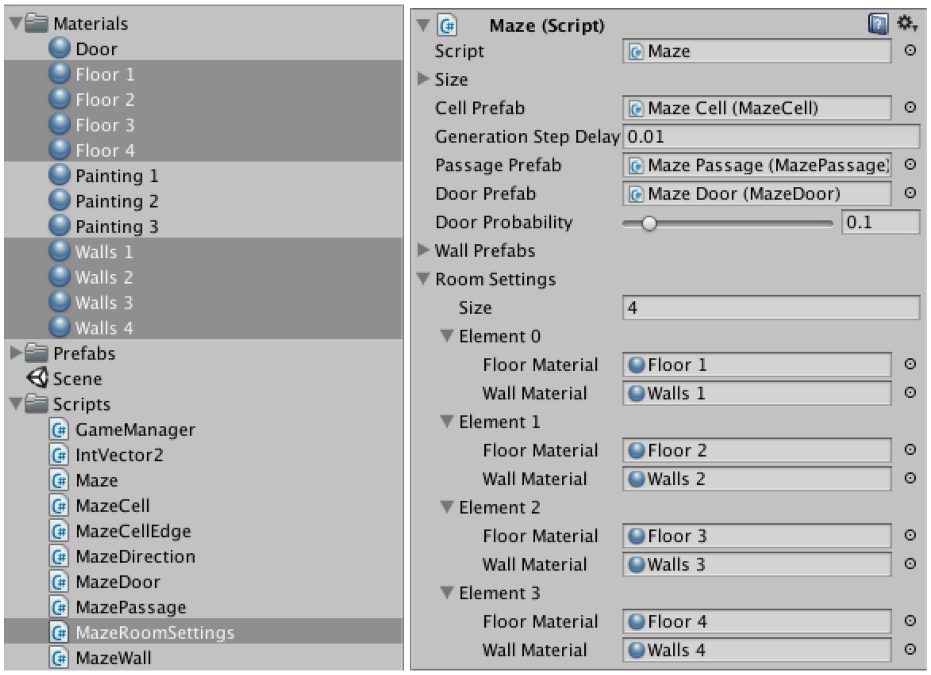
Our maze is uniformly white and that is rather boring. Let's spice things up by breaking the maze into rooms of different types. Then we can adjust the appearance of each room based on its type.

Create a new serialized `MazeRoomSettings` class with a public material references for floors and walls.

```
using UnityEngine;
using System;

[Serializable]
public class MazeRoomSettings {

	public Material floorMaterial, wallMaterial;
}
```

Now we can give `Maze` an array of these settings, defining the available room types. Then create a few materials with varying colors and populate the array. Four room types is a good amount for a 20 by 20 maze.

```
	public MazeRoomSettings[] roomSettings;
```



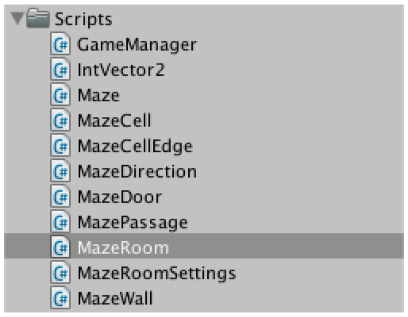
```
Maze (Script)
Script                  Maze
Size
Cell Prefab             Maze Cell (MazeCell)
Generation Step Delay   0.01
Passage Prefab          Maze Passage (MazePassage)
Door Prefab             Maze Door (MazeDoor)
Door Probability        0.1
Wall Prefabs
Room Settings
  Size                  4
  Element 0
    Floor Material      Floor 1
    Wall Material       Walls 1
  Element 1
    Floor Material      Floor 2
    Wall Material       Walls 2
  Element 2
    Floor Material      Floor 3
    Wall Material       Walls 3
  Element 3
    Floor Material      Floor 4
    Wall Material       Walls 4
```

Room settings.

Now we are also going to add a `MazeRoom` class so we can easily keep track of which cell belongs to which room. For now it simply is a wrapper for a list of cells, and it also has a reference to its settings and settings index. We have it extend `ScriptableObject` so Unity will keep the references intact if we were to cause a recompile while in play mode.

What's a `ScriptableObject`?

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

public class MazeRoom : ScriptableObject {

	public int settingsIndex;

	public MazeRoomSettings settings;

	private List<MazeCell> cells = new List<MazeCell>();

	public void Add (MazeCell cell) {
		cell.room = this;
		cells.Add(cell);
	}
}
```



Maze room script.

We also give `MazeCell` a reference to its room. While we're at it, let's also give it an `Initialize` method that takes care of assigning the right materials. As we only have the floor quad to worry about, we just grab the first child and be done with it.

```
	public MazeRoom room;

	public void Initialize (MazeRoom room) {
		room.Add(this);
		transform.GetChild(0).GetComponent<Renderer>().material = room.settings.floorMaterail;
	}
```

Now add a room list and a method to create a new room to `Maze`. We'll use it to create a new room for the first cell and each time we spawn a door.

If we were to just pick a random room type, it would be possible for two adjacent rooms to have the same type. While this is not really a problem, we get more variety by making sure that this won't happen. This can be done by checking whether we picked the same index as the room we came from. If so, we'll just add one to the index and wrap around. It's biased, but that's not a big deal here. Initially we'll pass a negative index so any room is fine.

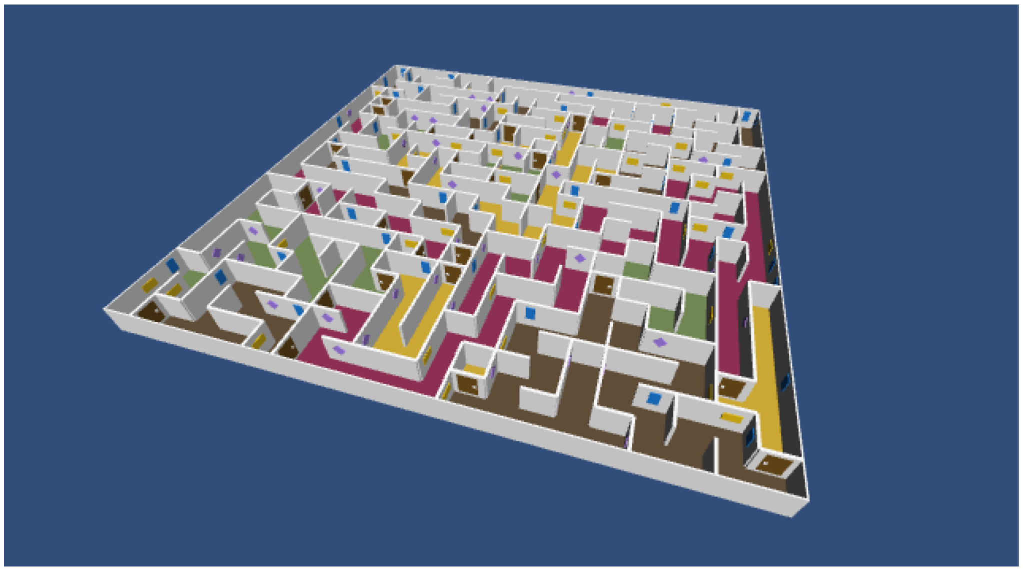
What does % do?

```
	private List<MazeRoom> rooms = new List<MazeRoom>();

	private MazeRoom CreateRoom (int indexToExclude) {
		MazeRoom newRoom = ScriptableObject.CreateInstance<MazeRoom>();
		newRoom.settingsIndex = Random.Range(0, roomSettings.Length);
		if (newRoom.settingsIndex == indexToExclude) {
			newRoom.settingsIndex = (newRoom.settingsIndex + 1) % roomSettings.Length;
		}
		newRoom.settings = roomSettings[newRoom.settingsIndex];
		rooms.Add(newRoom);
		return newRoom;
	}
```

It is now possible to create a new room in `DoFirstGenerationStep`. That will take care of the first cell. To put all the other cells in a room as well, we modify `CreatePassage` so it checks whether a door has been placed. If so, the other cell is the first of a new room. If not, it belongs to the same room as the previous cell.

What does **is** do?

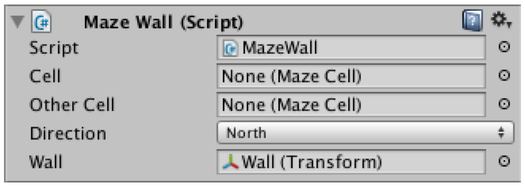
```
	private void DoFirstGenerationStep (List<MazeCell> activeCells) {
		MazeCell newCell = CreateCell(RandomCoordinates);
		newCell.Initialize(CreateRoom(-1));
		activeCells.Add(newCell);
	}

	private void CreatePassage (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazePassage prefab = Random.value < doorProbability ? doorPrefab : passagePrefab;
		MazePassage passage = Instantiate(prefab) as MazePassage;
		passage.Initialize(cell, otherCell, direction);
		passage = Instantiate(prefab) as MazePassage;
		if (passage is MazeDoor) {
			otherCell.Initialize(CreateRoom(cell.room.settingsIndex));
		}
		else {
			otherCell.Initialize(cell.room);
		}
		passage.Initialize(otherCell, cell, direction.GetOpposite());
	}
```



Now with colored floors.

To also color the walls, we need to adjust `MazeWall`. Give it a reference to its wall child and configure it for all the wall prefabs that you have created. This allows us to set the wall's material in an override of the `Initialize` method.

```
	public Transform wall;

	public override void Initialize (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		base.Initialize(cell, otherCell, direction);
		wall.GetComponent<Renderer>().material = cell.room.settings.wallMaterial;
	}
```



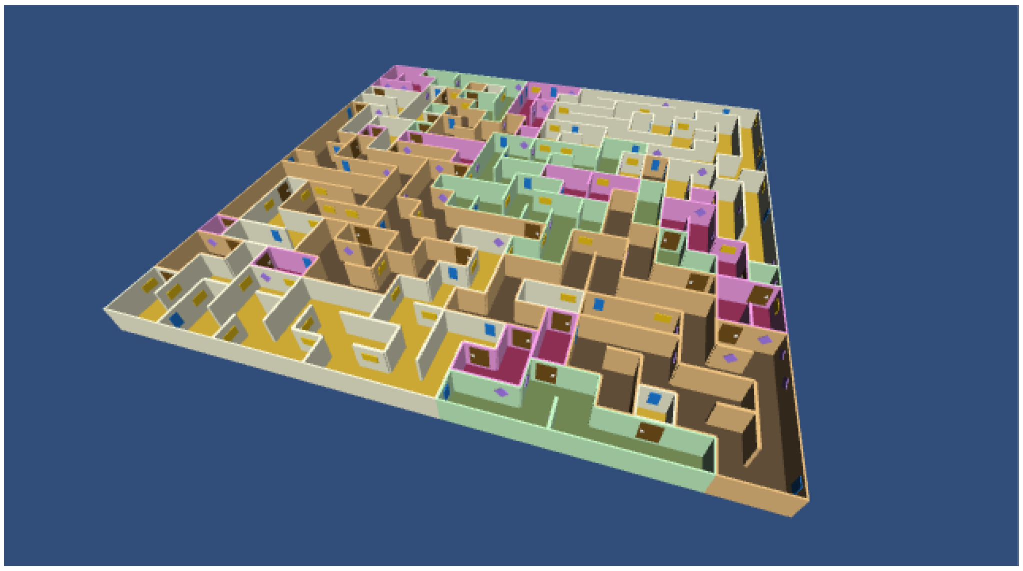
Maze Wall (Script)

Script	MazeWall
Cell	None (Maze Cell)
Other Cell	None (Maze Cell)
Direction	North
Wall	Wall (Transform)

Configuring a wall reference.

We have to do the same for `MazeDoor`, except now we set the material of all its direct children except for the hinge.

```
	public override void Initialize (MazeCell primary, MazeCell other, MazeDirection direction) {
		base.Initialize(primary, other, direction);
		if (OtherSideOfDoor != null) {
			hinge.localScale = new Vector3(-1f, 1f, 1f);
			Vector3 p = hinge.localPosition;
			p.x = -p.x;
			hinge.localPosition = p;
		}
		for (int i = 0; i < transform.childCount; i++) {
			Transform child = transform.GetChild(i);
			if (child != hinge) {
				child.GetComponent<Renderer>().material = cell.room.settings.wallMaterial;
			}
		}
	}
```



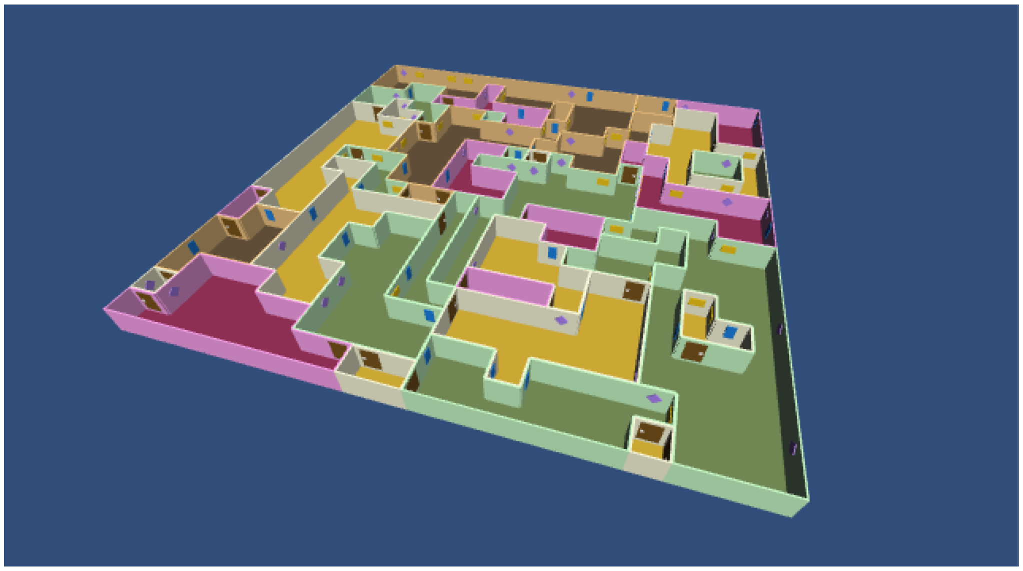
Fully colored rooms.

Expanding Rooms

We've been talking about rooms all this time, but they're really more like winding corridors. If we can prevent walls from being placed between two cell that belong to the same room, then the rooms will become more open areas.

Add a new `CreatePassageInSameRoom` method that simply creates a passage between two cells, with no chance of a door. Then update `DoNextGenerationStep` so it calls this method when two cells share a room, instead of placing a wall.

```
	private void CreatePassageInSameRoom (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazePassage passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(cell, otherCell, direction);
		passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(otherCell, cell, direction.GetOpposite());
	}

	private void DoNextGenerationStep (List activeCells) {
		int currentIndex = activeCells.Count - 1;
		MazeCell currentCell = activeCells[currentIndex];
		if (currentCell.IsFullyInitialized) {
			activeCells.RemoveAt(currentIndex);
			return;
		}
		MazeDirection direction = currentCell.RandomUninitializedDirection;
		IntVector2 coordinates = currentCell.coordinates + direction.ToIntVector2();
		if (ContainsCoordinates(coordinates)) {
			MazeCell neighbor = GetCell(coordinates);
			if (neighbor == null) {
				neighbor = CreateCell(coordinates);
				CreatePassage(currentCell, neighbor, direction);
				activeCells.Add(neighbor);
			}
			else if (currentCell.room == neighbor.room) {
				CreatePassageInSameRoom(currentCell, neighbor, direction);
			}
			else {
				CreateWall(currentCell, neighbor, direction);
			}
		}
		else {
			CreateWall(currentCell, null, direction);
		}
	}
```


Some breathing room.

We can go a step further and even join together adjacent rooms if they share the same settings. Besides creating larger rooms that way, fusing rooms from different parts of the maze creates loops. This means that there will be multiple ways to navigate it and you could end up walking in circles.

All that's really needed for this change is to relax our room comparison.

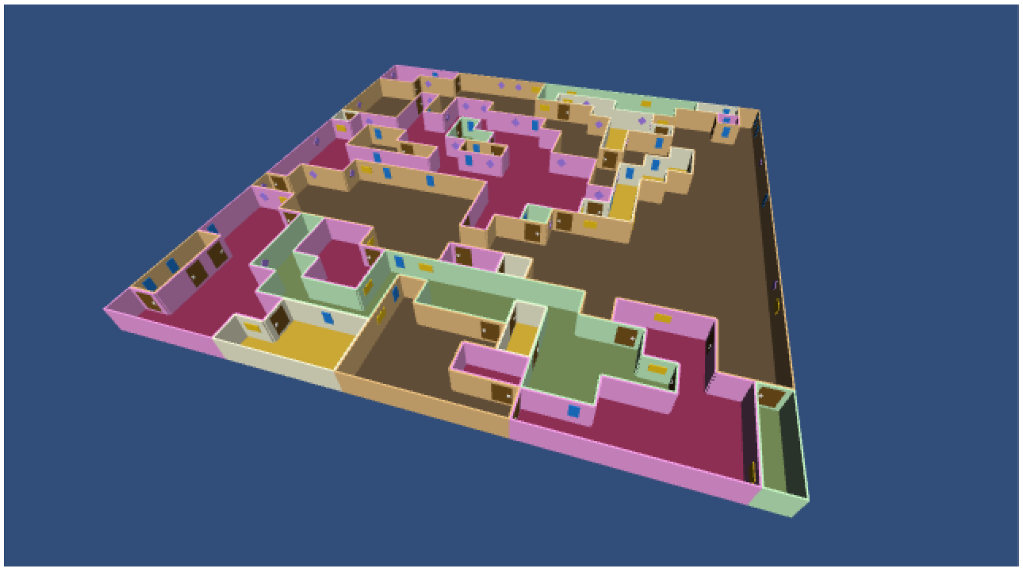
```
	private void DoNextGenerationStep (List<MazeCell> activeCells) {
		int currentIndex = activeCells.Count - 1;
		MazeCell currentCell = activeCells[currentIndex];
		if (currentCell.IsFullyInitialized) {
			activeCells.RemoveAt(currentIndex);
			return;
		}
		MazeDirection direction = currentCell.RandomUninitializedDirection;
		IntVector2 coordinates = currentCell.coordinates + direction.ToIntVector2();
		if (ContainsCoordinates(coordinates)) {
			MazeCell neighbor = GetCell(coordinates);
			if (neighbor == null) {
				neighbor = CreateCell(coordinates);
				CreatePassage(currentCell, neighbor, direction);
				activeCells.Add(neighbor);
			}
			else if (currentCell.room.settingsIndex == neighbor.room.settingsIndex) {
				CreatePassageInSameRoom(currentCell, neighbor, direction);
			}
			else {
				CreateWall(currentCell, neighbor, direction);
			}
		}
		else {
			CreateWall(currentCell, null, direction);
		}
	}
```

While this looks correct, we will now have different rooms with open passages connecting them. We should also get rid of one of the room instances as it is assimilated by the other. So let's add an `Assimilate` method to `MazeRoom`.

```
	public void Assimilate (MazeRoom room) {
		for (int i = 0; i < room.cells.Count; i++) {
			Add(room.cells[i]);
		}
	}
```

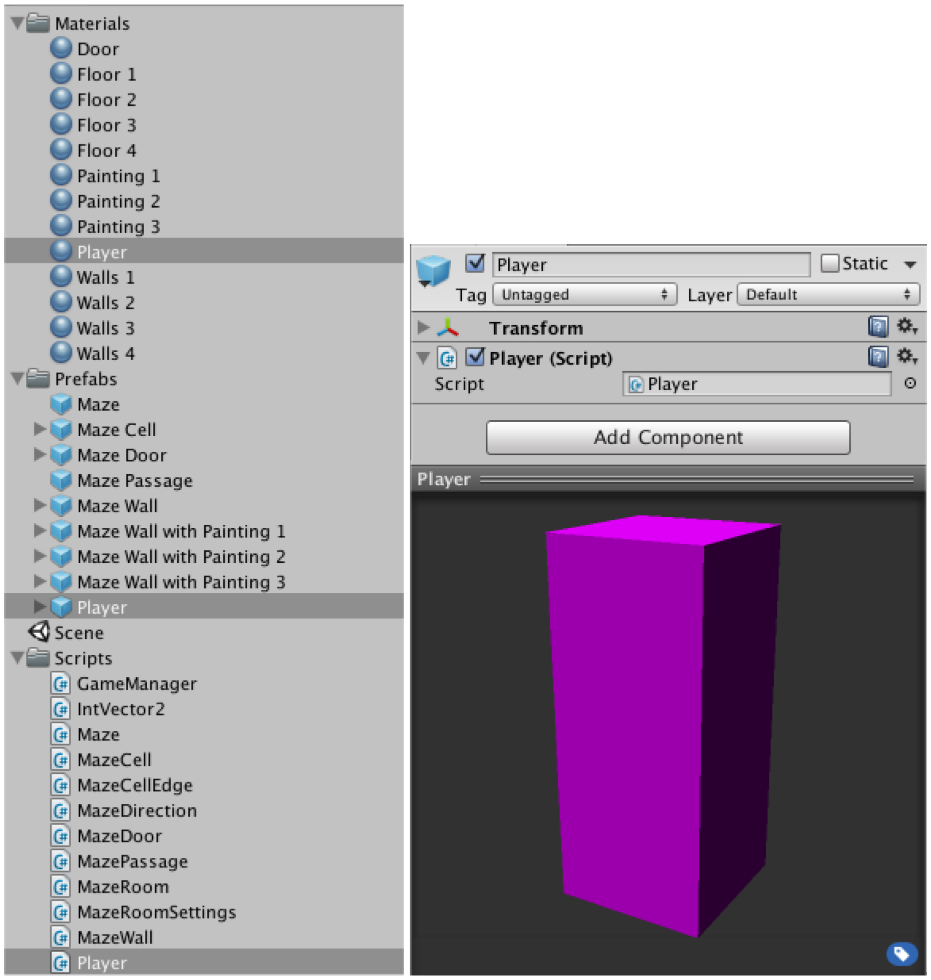
Then we have `Maze` check whether it's connecting different rooms, in which case it assimilates and removes the other room.

```
	private void CreatePassageInSameRoom (MazeCell cell, MazeCell otherCell, MazeDirection direction) {
		MazePassage passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(cell, otherCell, direction);
		passage = Instantiate(passagePrefab) as MazePassage;
		passage.Initialize(otherCell, cell, direction.GetOpposite());
		if (cell.room != otherCell.room) {
			MazeRoom roomToAssimilate = otherCell.room;
			cell.room.Assimilate(roomToAssimilate);
			rooms.Remove(roomToAssimilate);
			Destroy(roomToAssimilate);
		}
	}
```


Now with large rooms and loops.

Walking Around

It's high time we walked around in our own maze. Create a simple player model, attach a new `Player` component that we create as well, and turn it into a prefab.



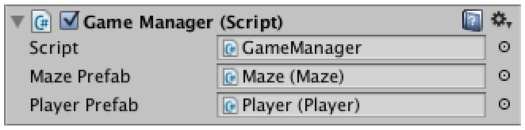
A square player.

Give `Player` a public method so we can tell it what cell it's in. Also give it an `Update` method that moves the player when an arrow key is pressed. Movement should only happen if the edge we would cross is a passage, otherwise we're blocked.

```
using UnityEngine;

public class Player : MonoBehaviour {

	private MazeCell currentCell;

	public void SetLocation (MazeCell cell) {
		currentCell = cell;
		transform.localPosition = cell.transform.localPosition;
	}

	private void Move (MazeDirection direction) {
		MazeCellEdge edge = currentCell.GetEdge(direction);
		if (edge is MazePassage) {
			SetLocation(edge.otherCell);
		}
	}

	private void Update () {
		if (Input.GetKeyDown(KeyCode.UpArrow)) {
			Move(MazeDirection.North);
		}
		else if (Input.GetKeyDown(KeyCode.RightArrow)) {
			Move(MazeDirection.East);
		}
		else if (Input.GetKeyDown(KeyCode.DownArrow)) {
			Move(MazeDirection.South);
		}
		else if (Input.GetKeyDown(KeyCode.LeftArrow)) {
			Move(MazeDirection.West);
		}
	}
}
```

Now give `GameManager` both a reference to the player prefab and a local reference to a current player.

```
	public Player playerPrefab;

	private Player playerInstance;
```

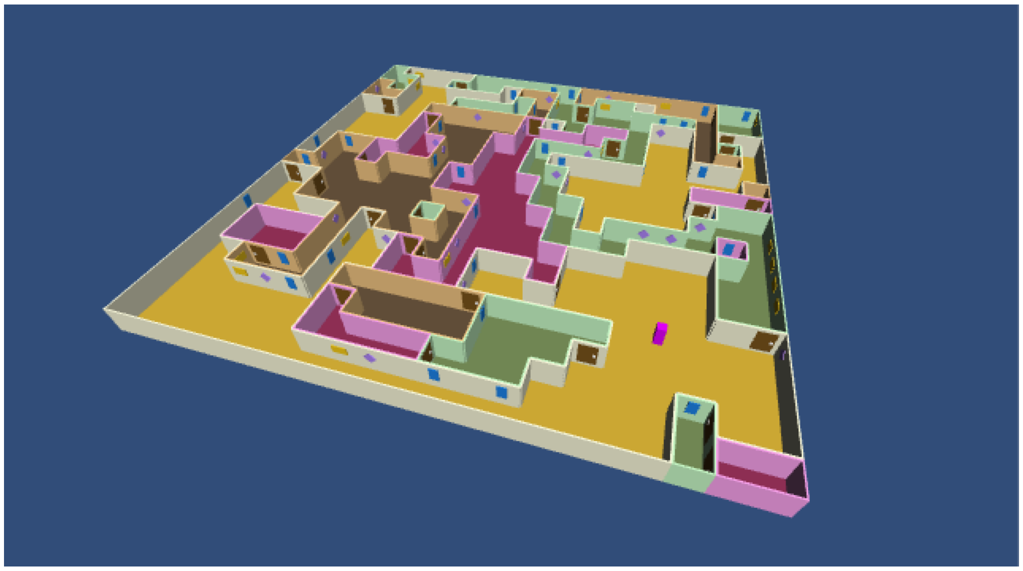


Script GameManager
Maze Prefab Maze (Maze)
Player Prefab Player (Player)

Game manager wants a player now.

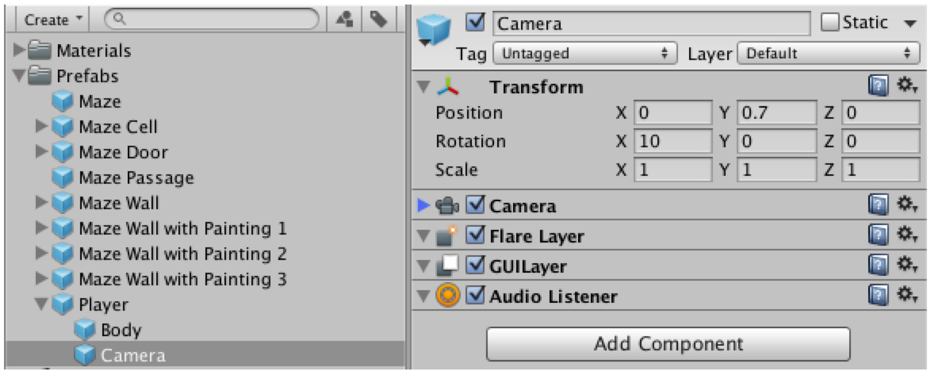
We should instantiate a new player after the maze has finished generating. However, we currently start a coroutine and move on. In order to wait, we turn `BeginGame` into a coroutine as well. Then we can yield the other coroutine, so it finishes before we continue ourselves and create the player and give it a random location. Also, make sure to destroy the current player in `RestartGame`, if it has already been created.

```
	private void Start () {
		StartCoroutine(BeginGame());
	}

	private IEnumerator BeginGame () {
		mazeInstance = Instantiate(mazePrefab) as Maze;
		yield return StartCoroutine(mazeInstance.Generate());
		playerInstance = Instantiate(playerPrefab) as Player;
		playerInstance.SetLocation(mazeInstance.GetCell(mazeInstance.RandomCoordinates));
	}

	private void RestartGame () {
		StopAllCoroutines();
		Destroy(mazeInstance.gameObject);
		if (playerInstance != null) {
			Destroy(playerInstance.gameObject);
		}
		StartCoroutine(BeginGame());
	}
```


A player exploring our maze.

What It Looks Like

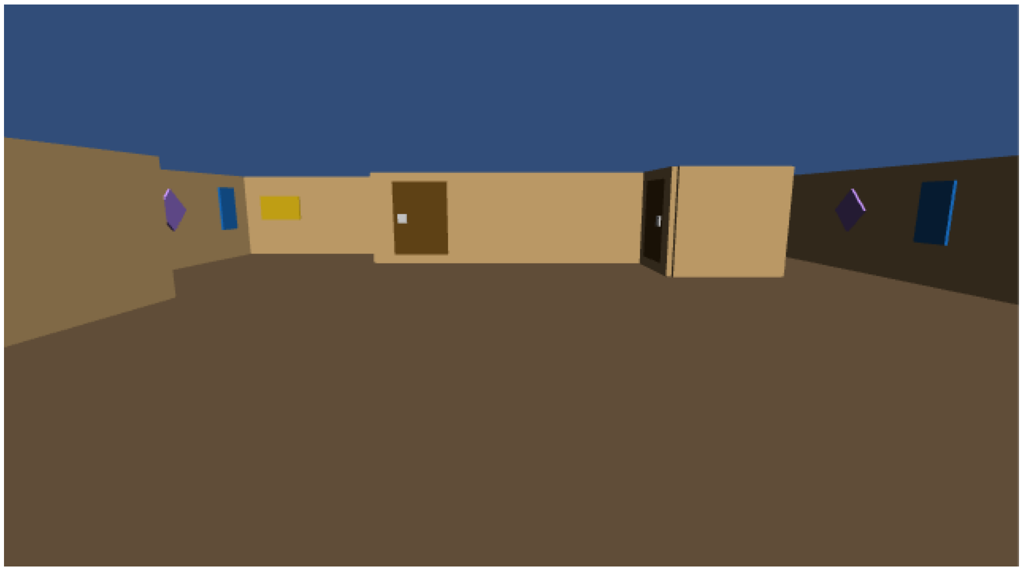
What would our maze look like, when viewing it through the eyes of the player? Let's find out by adding a camera to the player prefab! You can do so by dragging an instance of the player prefab into the scene, creating a default camera, making it a child of the player, and then clicking the prefab *Apply* button of the player instance. I position the camera at a height of 0.7 and rotate it ten degrees around its X axis so it doesn't stare straight ahead but looks a bit to the floor.



Camera

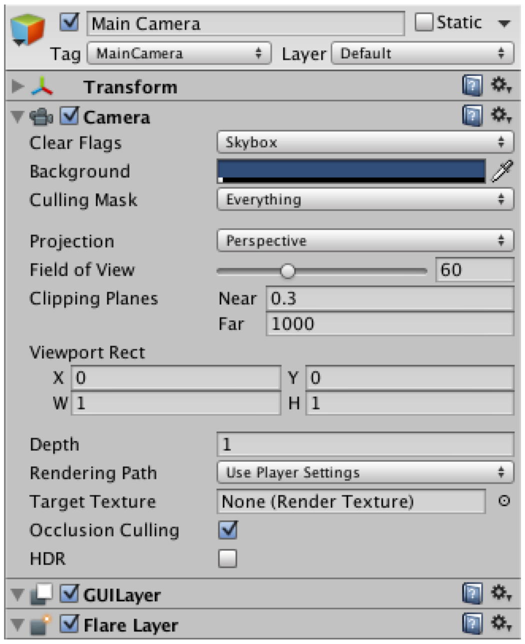
Tag: Untagged — Layer: Default

Position	X 0	Y 0.7	Z 0
Rotation	X 10	Y 0	Z 0
Scale	X 1	Y 1	Z 1


Player camera in action.

Keep in mind that the main camera is also still being rendered. The player camera is just rendered on top of it, because it has the default depth of zero while the default main camera has a depth of -1. Unity will also complain that there are two audio listeners in the scene at the same time, so we have to do something about that.

What about we keep both cameras, but turn the main camera view into a map overlay? First, remove the audio listener from the main camera and increase its depth value to one. That will make it render after and on top of the player's camera.



Main Camera ☐ Static
Tag MainCamera Layer Default

Transform

Camera

Clear Flags	Skybox
Background	
Culling Mask	Everything
Projection	Perspective
Field of View	60
Clipping Planes	Near 0.3
	Far 1000
Viewport Rect	
X 0	Y 0
W 1	H 1
Depth	1
Rendering Path	Use Player Settings
Target Texture	None (Render Texture)
Occlusion Culling	✓
HDR	☐

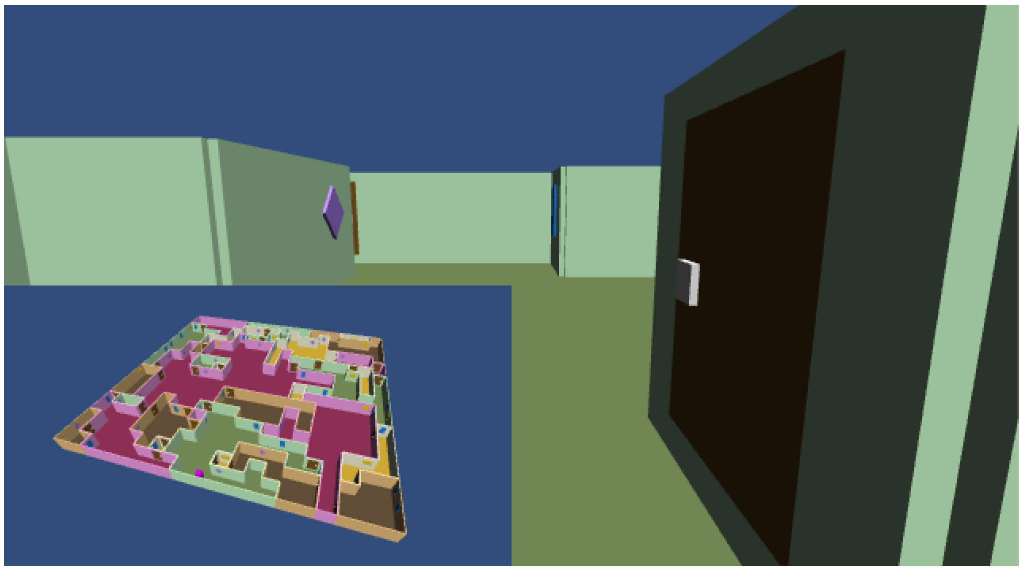
GUILayer

Flare Layer

Tweaked main camera.

Now we again only see the main camera as it's rendering after the player camera. To turn it into a smaller overlay, we reduce its view rectangle after a maze has been generated. We also set it to cover the entire view before we start generating. That ensures that we get a full-size view of the maze while it is being generated.

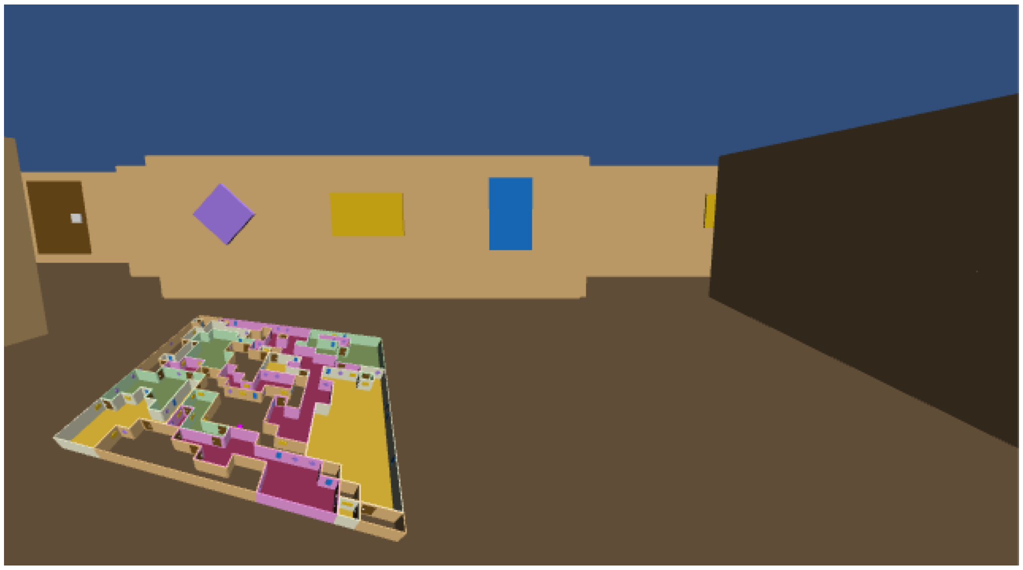
```
	private IEnumerator BeginGame () {
		Camera.main.rect = new Rect(0f, 0f, 1f, 1f);
		mazeInstance = Instantiate(mazePrefab) as Maze;
		yield return StartCoroutine(mazeInstance.Generate());
		playerInstance = Instantiate(playerPrefab) as Player;
		playerInstance.SetLocation(mazeInstance.GetCell(mazeInstance.RandomCoordinates));
		Camera.main.rect = new Rect(0f, 0f, 0.5f, 0.5f);
	}
```



An overlay map.

This works, but it would be nicer if the map was rendered on top of the player's view without its own background. Fortunately, we can easily achieve this by changing the camera's clear flags to *Depth*. When generating the maze, it should use its default flags value, which is *SkyBox*.

```
	private IEnumerator BeginGame () {
		Camera.main.clearFlags = CameraClearFlags.Skybox;
		Camera.main.rect = new Rect(0f, 0f, 1f, 1f);
		mazeInstance = Instantiate(mazePrefab) as Maze;
		yield return StartCoroutine(mazeInstance.Generate());
		playerInstance = Instantiate(playerPrefab) as Player;
		playerInstance.SetLocation(mazeInstance.GetCell(mazeInstance.RandomCoordinates));
		Camera.main.clearFlags = CameraClearFlags.Depth;
		Camera.main.rect = new Rect(0f, 0f, 0.5f, 0.5f);
	}
```



Now without nasty background.

The player can move and have a map, but we are stuck looking to the north all the time. To change this, `Player` need to keep track of where it's looking and respond to user input to change its rotation. Let's use **Q** to rotate counterclockwise and **E** to rotate clockwise. While we're at it, we can also support the common **WASD** key bindings.

```
	private MazeDirection currentDirection;

	private void Rotate (MazeDirection direction) {
		transform.localRotation = direction.ToRotation();
		currentDirection = direction;
	}

	private void Update () {
		if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow)) {
			Move(MazeDirection.North);
		}
		else if (Input.GetKeyDown(KeyCode.D) || Input.GetKeyDown(KeyCode.RightArrow)) {
			Move(MazeDirection.East);
		}
		else if (Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.DownArrow)) {
			Move(MazeDirection.South);
		}
		else if (Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.LeftArrow)) {
			Move(MazeDirection.West);
		}
		else if (Input.GetKeyDown(KeyCode.Q)) {
			Rotate(currentDirection.GetNextCounterclockwise());
		}
		else if (Input.GetKeyDown(KeyCode.E)) {
			Rotate(currentDirection.GetNextClockwise());
		}
	}
```

We also add two convenient methods to `MazeDirections` that gives us the next direction in clockwise and counterclockwise order.

```
	public static MazeDirection GetNextClockwise (this MazeDirection direction) {
		return (MazeDirection)(((int)direction + 1) % Count);
	}

	public static MazeDirection GetNextCounterclockwise (this MazeDirection direction) {
		return (MazeDirection)(((int)direction + Count - 1) % Count);
	}
```

Now we can both move and rotate, but the movement is absolute instead of relative to our orientation. Let's change that.

```
	private void Update () {
		if (Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.UpArrow)) {
			Move(currentDirection);
		}
		else if (Input.GetKeyDown(KeyCode.D) || Input.GetKeyDown(KeyCode.RightArrow)) {
			Move(currentDirection.GetNextClockwise());
		}
		else if (Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.DownArrow)) {
			Move(currentDirection.GetOpposite());
		}
		else if (Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.LeftArrow)) {
			Move(currentDirection.GetNextCounterclockwise());
		}
		else if (Input.GetKeyDown(KeyCode.Q)) {
			Look(currentDirection.GetNextCounterclockwise());
		}
		else if (Input.GetKeyDown(KeyCode.E)) {
			Look(currentDirection.GetNextClockwise());
		}
	}
```

Opening Doors

So far we've been walking straight through doors without opening them. We could use various approaches to opening doors, but a simple one is to just open all doors of a cell after the player enters it, and close them again when the player exited it. In fact, other stuff might happen as well, so we'll use a generic approach.

Have `Player` notify cells when it enters and exits them in `SetLocation`, by calling two new methods. Check whether there's a cell to exit, because the first time a location is set this won't be the case.

```
	public void SetLocation (MazeCell cell) {
		if (currentCell != null) {
			currentCell.OnPlayerExited();
		}
		currentCell = cell;
		transform.localPosition = cell.transform.localPosition;
		currentCell.OnPlayerEntered();
	}
```

`MazeCell` doesn't do anything with those events itself, but passes them along to its edges.

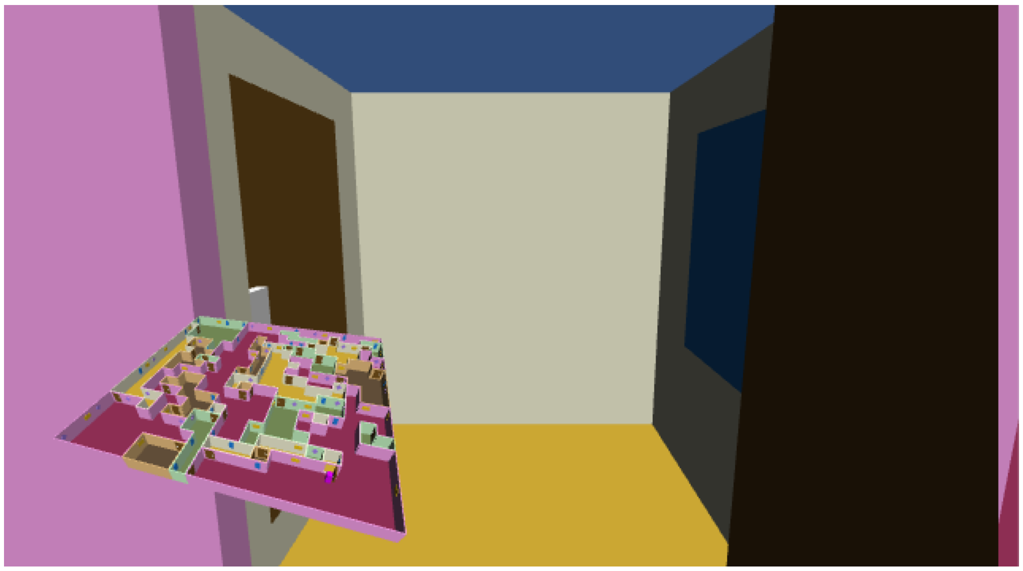
```
	public void OnPlayerEntered () {
		for (int i = 0; i < edges.Length; i++) {
			edges[i].OnPlayerEntered();
		}
	}

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

We add those methods to `MazeCellEdge` as empty and virtual. That way nothing happens by default, but subclasses can override this.

```
	public virtual void OnPlayerEntered () {}

	public virtual void OnPlayerExited () {}
```

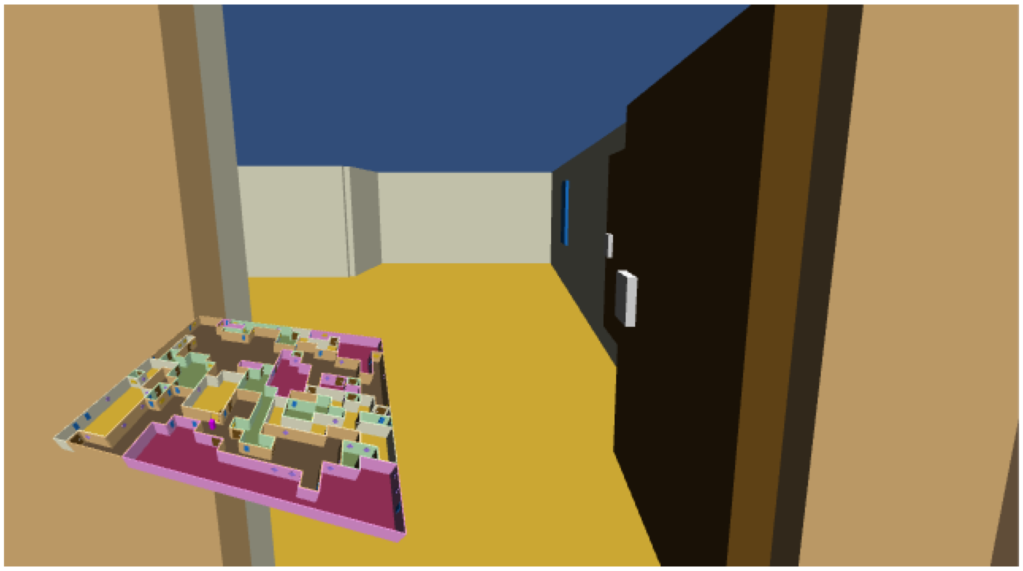
Now we can add overrides for `MazeDoor` to rotate its hinge. Because there are two sides of a door, we have to rotate both of them.

```
	public override void OnPlayerEntered () {
		OtherSideOfDoor.hinge.localRotation = hinge.localRotation = Quaternion.Euler(0f, -90f, 0f);
	}

	public override void OnPlayerExited () {
		OtherSideOfDoor.hinge.localRotation = hinge.localRotation = Quaternion.identity;
	}
```


A door opened in our face.

It works! Doors are now open when we stand next to them. Unfortunately the doors always rotate in the same direction. This means that about half the time the door will cut through our view in an ugly and obscuring way. We can solve this by always rotating doors away from where the player is currently standing. This can be done by remembering whether a door is mirrored and rotating based on that.

```
	private static Quaternion
		normalRotation = Quaternion.Euler(0f, -90f, 0f),
		mirroredRotation = Quaternion.Euler(0f, 90f, 0f);

	private bool isMirrored;

	public override void Initialize (MazeCell primary, MazeCell other, MazeDirection direction) {
		base.Initialize(primary, other, direction);
		if (OtherSideOfDoor != null) {
			isMirrored = true;
			hinge.localScale = new Vector3(-1f, 1f, 1f);
			Vector3 p = hinge.localPosition;
			p.x = -p.x;
			hinge.localPosition = p;
		}
		for (int i = 0; i < transform.childCount; i++) {
			Transform child = transform.GetChild(i);
			if (child != hinge) {
				child.GetComponent<Renderer>().material = cell.room.settings.wallMaterial;
			}
		}
	}

	public override void OnPlayerEntered () {
		OtherSideOfDoor.hinge.localRotation = hinge.localRotation =
			isMirrored ? mirroredRotation : normalRotation;
	}
```

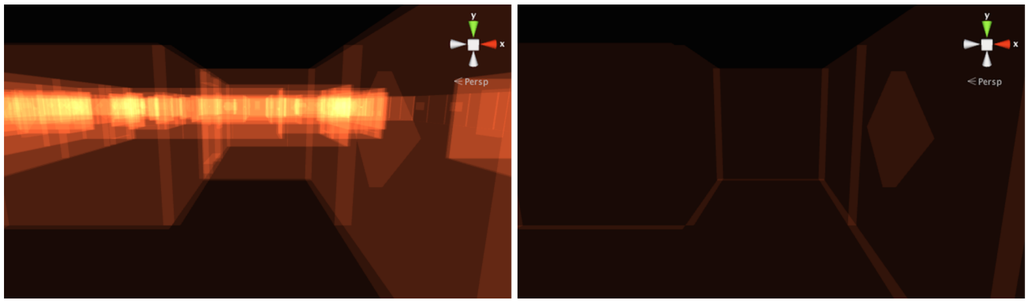


Always-push doors.

Hiding Rooms

An additional thing we could do is only show the room that the player is currently inside of. Besides reducing what is shown on the map, it can also eliminate lots of unnecessary draw calls for the player camera. As this functionality operates on entire rooms, let's add `Show` and `Hide` method to `MazeRoom`, which call the same methods on all their cells.

Why does obscured stuff get rendered?



Overdraw without hiding and with hiding rooms.

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

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

We let `MazeCell` implement this functionality by simply switching its game object on and off.

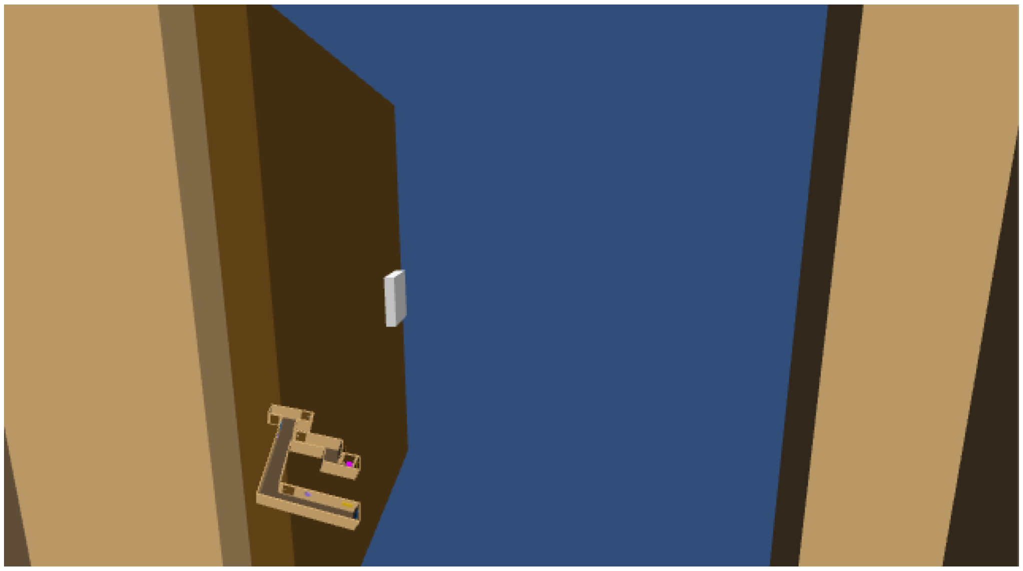
```
	public void Show () {
		gameObject.SetActive(true);
	}

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

And we also let `Maze` hide all rooms when it's done generating.

```
	public IEnumerator Generate () {
		WaitForSeconds delay = new WaitForSeconds(generationStepDelay);
		cells = new MazeCell[size.x, size.z];
		List<MazeCell> activeCells = new List<MazeCell>();
		DoFirstGenerationStep(activeCells);
		while (activeCells.Count > 0) {
			yield return delay;
			DoNextGenerationStep(activeCells);
		}
		for (int i = 0; i < rooms.Count; i++) {
			rooms[i].Hide();
		}
	}
```

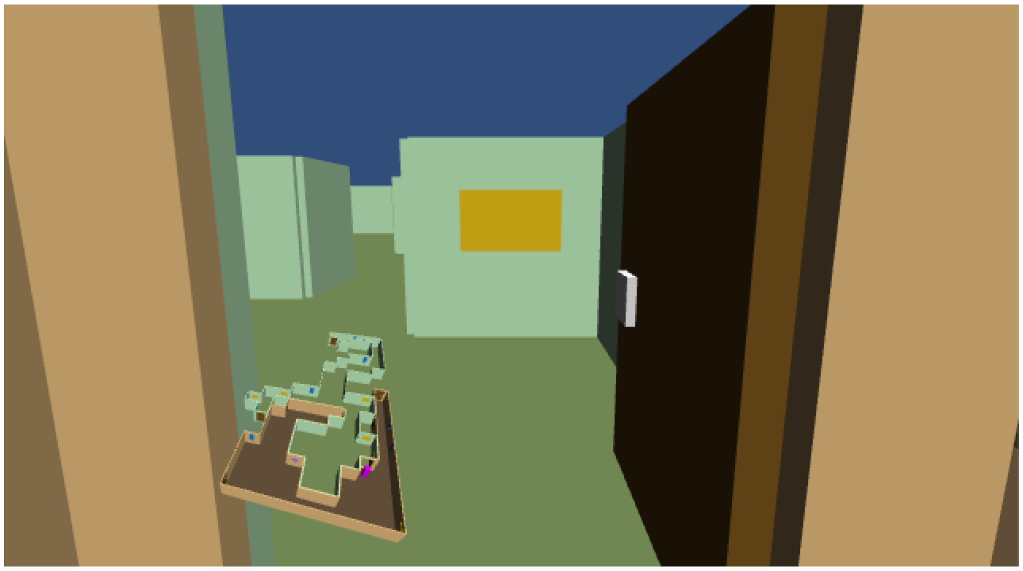
This will place the player in an invisible maze. To make the rooms appear and disappear, we let `MazeCell` show and hide its room when it is entered or exited.

```
	public void OnPlayerEntered () {
		room.Show();
		for (int i = 0; i < edges.Length; i++) {
			edges[i].OnPlayerEntered();
		}
	}

	public void OnPlayerExited () {
		room.Hide();
		for (int i = 0; i < edges.Length; i++) {
			edges[i].OnPlayerExited();
		}
	}
```


Looking into the void.

So all rooms except the one that we are currently in are now hidden. Unfortunately, this means that when we open a door we will look into an invisible room. Fortunately this is easy to solve. All we have to do is have `MazeDoor` show and hide the other cell's room when it is opened and closed.

```
	public override void OnPlayerEntered () {
		OtherSideOfDoor.hinge.localRotation = hinge.localRotation =
			isMirrored ? mirroredRotation : normalRotation;
		OtherSideOfDoor.cell.room.Show();
	}

	public override void OnPlayerExited () {
		OtherSideOfDoor.hinge.localRotation = hinge.localRotation = Quaternion.identity;
		OtherSideOfDoor.cell.room.Hide();
	}
```


Rooms shown and hidden at the right time.

And indeed now rooms also show up when we could see them through an open door.

We can keep adding and tweaking the maze, but I will end the tutorial here. Have fun giving the maze your own special touch!

Enjoyed the tutorial? Help me make more by becoming a patron!

Tweet 85
Share 13
Share 56
Share 81

Downloads

maze-01.unitypackage
The project after Game Flow.

maze-02.unitypackage
The project after Maze Fundamentals.

maze-03.unitypackage
The project after Cell Coordinates and Integer Vectors.

maze-04.unitypackage
The project after Random Cell Generation.

maze-05.unitypackage
The project after Backtracking.

maze-06.unitypackage
The project after Connecting the Cells.

maze-07.unitypackage
The project after Generating the Entire Maze.

maze-08.unitypackage
The project after Decorating the Maze.

maze-09.unitypackage
The project after Placing Doors.

maze-10.unitypackage
The project after Adding Rooms.

maze-11.unitypackage
The project after Expanding Rooms.

maze-12.unitypackage
The project after Walking Around.

maze-13.unitypackage
The project after What it Looks Like.

maze-14.unitypackage
The project after Opening Doors.

maze-finished.unitypackage
The finished project.

Questions & Answers

How does `GetKeyDown` work?

The `Input` class has a collection of static methods and properties to get information about user input. Its `GetKeyDown` method returns `true` in the frame that a physical button has been pressed down by the user. You pass it the code of the key you're interested in.

This method only returns `true` during the frame that the key became active. The `GetKey` method will also return `true` as long as the key stays pressed. The `GetKeyUp` method will return `true` in the frame that the buttom was released. `Input` has similar methods for mouse buttons and configurable input buttons.

What's a prefab?

A prefab is a Unity object – or hierarchy of objects – that doesn't exist in the scene and hasn't been activated. You use it as a template, creating clones of it and adding those to the scene.

What does `Instantiate` do?

Unity's `Object` class, which every MonoBehaviour inherits from, contains the static `Instantiate` method. This method creates a clone of whatever `Object` instance you pass to it. Optionally, you can supply a new position and rotation for the clone, otherwise it keeps the values of the original.

Note that `Instantiate` returns an `Object` reference. If you want to do something with the new clone, you have to cast it to its specific type, which in our case is `Maze`.

Typically, this method is used with prefabs, but you can also clone objects that already exist in the scene.

What does `as` do?

The `as` operator is for casting to a different type. We could have also written `(Maze)Instantiate`. The big difference is that casting in the latter way could perform custom type conversion and will result in an error when used on an incompatible type. The `as` operator doesn't convert anything, it only checks whether the object instance is of the correct type. If so, it passes along the reference, otherwise it will result in `null`, not an error. As such, it only works with reference types, not value types.

What does `Destroy` do?

`Destroy` is sort-of the counterpart of `Instantiate`. Pass it a component or a game object, and it will make sure it gets destroyed. This means that memory will be freed, which might activate the garbage collector at some point.

Note that we are destroying `mazeInstance.gameObject`, because we want the entire game object gone, not just its `Maze` component.

How does `WaitForSeconds` work?

The `WaitForSeconds` object will monitor Unity's time value and simply keep iterating until the specified amount of seconds have passed. As countines perform one iteration step per frame, its precision is only as good as the frame rate.

Note that we can reuse the `WaitForSeconds` instance. We don't need to create a new one every time. Just don't use it in two coroutines at the same time.

Where does a coroutine live?

A coroutine is attached to the `MonoBehaviour` object who's `StartCoroutine` was called. In our case `GameManager` starts the coroutine, so it is attached to our game manager instance and lives and dies with it. `Maze` just provided the iterator. That's why we need to stop the coroutine when we destroy our maze instance. If we wouldn't, Unity would complain that our coroutine is accessing a destroyed object.

We could have also attached the coroutine to `Maze` and it would work just fine without us having to manually stop the coroutine. Later on we'll still have a reason to stop a coroutine manually, though.

Why isn't `IntVector2` immutable?

Value types like `int` and `float` are immutable. This means that you cannot change the values themselves. They have no identity, they represent constant concepts. A 3 is always a 3. Executing 3 + 1 does not change the 3 nor the 1, it produces a new 4.

A `struct` is a custom value type. It is good design to make them immutable as well, because if we did then everything would be ideal. Value types are immutable. Passing them around copies them. They don't act like objects some of the time.

We could make our vector immutable by simply keeping its components private and not adding public functionality to change them.

However, Unity's various vector structs are not immutable. This is often convenient and fast, but also inconsistent. We can do `someVector.x = 3`, which changes an existing vector. But we cannot do `someTransform.localPostion.x = 3`, because of various reasons.

The Unity developers decided to make vectors mutable value types, and there are good practical reasons to do so. For the sake of consistency, let's use the same approach for own own vector.

Aren't we changing a here?

Yes, we are changing a inside the operator method. We could also make room for a new vector, but why should we? Our `IntVector2` is a struct so it's passed by value. We can mess around with the argument as much as we like, it won't change the caller's value at all. We could do this even if we had decided to make `IntVector2` immutable, because we're inside the struct's defintion and know what we're doing. Right?

Shouldn't cell coordinates be fixed?

A cell will get its coordinates when it is created and will never change position. So it would be good design if there would be no way for someone to change the coordinates of a cell. But because we make the coordinates public, everyone could change them at any time!

We could make the coordinates private and provide an initializer method to set them once. But then we would need to make sure that the initialize method is never called again. But once we've done that, anyone could still get to the game object and do lots of destructive stuff. Like remove the cell component, or destroy the entire game object! How could we stop that? We cannot.

I'm not saying you shouldn't protect the coordinates, but pointing out that no matter what design principles you apply, you're never safe when working with Unity objects. And then there's reflection. Don't assume to be safe, make sure programmers behave.

What's an attribute?

An attribute is a means to attach metadata to fields, methods, and types. For example, you can tell Unity how to display a variable in the editor, whether to save or not save data, indicate that a component requires other components, and lots of other stuff.

Atributes are added between brackets in front of whatever they're attached to and can have arguments, like `[Nice] int number` or `[Nice(42)] int number`. Multiple attributes are separated by commas, like `[Nice, Sweet] int number`.

How does serialization work?

Serialization is the process of converting a collection of data in memory into a stream of data that can be stored in a persistent state or transmitted over a network. It's what Unity does when it saves your scene and asset data. Deserialization is its complement, constructing data in memory from a stream.

This functionality is part of .NET and you can read more about it on MSDN, though it's not required.

Can a struct be Serializable?

Yes, since Unity 4.5 custom structs can be serialized. Anything saying they can't be serialized is outdated.

So don't change `IntVector2` into a class. If you do, the algorithm will fail unless you make sure you never adjust the values of a coordinate. For example, the plus operator method tweaks one of its arguments, which is fine for structs as they are copied, but not for classes.

How do properties work?

Properties are methods that pretend to be a variable. It's a form of syntactic sugar. Here is an example property.

```
int X { get { return x; } set { x = value; } }
```

And here is what is basically boils down to.

```
int GetX () { return x; } int SetX (int value) { x = value; }
```

You can leave out the set or the get part, in which case you have a read-only or a write-only property. You can also give get and set difference access modifiers.

What does && do?

The && operator is used for boolean logic and stands for *"and also"*. In other words, `x && y` is only true if both x and y are true.

Note that if x is found to be false, there's no point in checking y anymore. If y were a method call or property, it won't be invoked.

The companion of && is the `||` operator, which stands for *"or else"*. So `x || y` is true if at least one of them is. Also, if x is found to be true, then y will not be considered.

What's an `enum`?

You use `enum` to define an enumeration type, which is an ordered list of names. A variable of this type can have one of these names as its value. Each of these names corresponds to a number, by default starting at zero. They are useful whenever you need a limited list of named options.

Under the hood, enums are simply integers. This means that you can freely convert between an `enum` type and `int`, which we will use quite a bit. You could also declare them to be of a handfull of other types, but we stick to using integers.

Just to be sure, `enum` has nothing to do with enumerators or iterators.

What's an extension method?

An extension method is a static method inside a static class that behaves like an instance method of some type. That type could be anything, a class, an interface, a struct, a primitive value, or an enum. The first argument of an extension method needs to have the `this` keyword and defines the type and instance value that the method will operate on.

Does this allow us to add methods to everything? Yes, just like you could write any static method that has any type as its argument. Is this a good idea? When used in moderation, it can be. It's a specialized tool that has its uses, but wielded it with abandon will result in an unstructured mess.

What does it mean to be abstract?

When a class is abstract it does not allow objects instances of itself to be created. By itself that would be rather useless, but this does not prevent subclasses of itself to be instantiated. Basically, it acts like a foundation, to be extended by other classes but not usable on its own.

Abstract classes can have abstract methods as well. These are methods without a body. This enforces that all subclasses will have to provide their own implementation of that method, unless they are abstract as well.

What does `throw` do?

The `throw` keyword works like `return`, except that it passes along something throwable – an error or exception – instead of the appropriate return type. When something is thrown, invocations will fail upward until it is caught somewhere, which usually result in an error message. In case of a mobila app that's optimized with *Fast but no Exceptions*, it will cause a crash.

Where are the prefab hinge's children?

Unfortunately, Unity only shows the direct children of a prefab asset root. Even though they aren't shown, the children do exist. The prefab's preview shows their 3D models, and they will be visible in any instance added to the scene. You can edit the invisible parts by dragging the prefab into the scene, making changes to the instance, and applying it back to the prefab. This approach will be necessary until Unity fully supports nested prefabs.

What does the question mark do?

The question mark, combined with a colon, is a short version of an if-else statement. For example, you could write a full if-else block.

```
int y; if(x > 1) { y = 10; } else { y = 20; }
```

But you can also use this condensed syntax.

```
int y = x > 1 ? 10 : 20;
```

What does `Range` do?

The `Range` attribute tells the Unity editor to use a slider to display an int or float variable. Its arguments determine the minimum and maximum values allowed by the slider.

What's `base`?

The `base` keyword is a reference to the object instance itself, just like the `this` keyword. The important difference is that it also represents the base type that the class extends. This allows you to access functionality that you override. A typical use case is to override a method by first calling the base method and then adding your own stuff, which is exactly what we do here. Had we omitted the keyword, we would've created an infinite recursion and the method would keep invoking itself until we got a stack overflow exception.

Why is `virtual` needed?

Not marking a method as virtual is useful for preventing subclasses from overriding your own implementation. But the technical reason is because there's an important difference between a non-virtual and a virtual method. Looking at it from a very low level, a method call requires a jump to some place where the instructions for that method are stored. If this method were always the same, the jump will always be to the same place. Easy. But when a method can be overridden by a subclass, where we need to jump depends on the type of the object it's being invoked on. This problem is solved by looking up the method's location in a table. This table is known as a *virtual method table*, hence the name of the keyword. If you didn't know about that table, *overridable* would probably make more sense.

What's a `ScriptableObject`?

You typically extend `ScriptableObject` when creating your own asset type, because Unity can store these objects just like game objects and components. An additional benefit is that such objects are correctly stored and retrieved when scripts are recompiled while Unity is in play mode. This means we could tweaks scripts while walking around a maze and our rooms will be preserved.

Couldn't we just add the `Serializable` attribute to `MazeRoom`? No, because Unity copies such data by value and does not maintain the relationships between objects. We will give cells a reference to their room, and this would mean that after a recompile each cell will have its own copy of the room. While things will initially look fine, it will quickly lead to problems.

An alternative is to extend `MonoBehaviour` instead, in which case each room would be a component and we'd have to attach them to a game object, the maze being the most logical choice.

What does % do?

The % operator computes the remainder of a division. For example, `3 % 4` is 3, `4 % 4` is zero, and `5 % 4` is 1. When used on a non-negative integer and the length of an array that has at least one element, the result will always be a valid index for that array.

What does `is` do?

The `is` operator is the companion of the `as` operation. Instead of casting, it tells you whether an object is of some type.

Why does obscured stuff get rendered?

You only need to render what you can see, right? But how do you know what you can and cannot see?

Unity only renders stuff that is active. Also, it only renders stuff that the camera could theoretically see, everything that lies outside of its view is not queued for rendering. The graphics card will detect when it's about to fill a pixel that's behind something else, so it won't need to bother to shade it. However, we had to go all the way to the pixel level before this discovery was made, which means that we have to process lots of stuff that we end up not seeing.

To improve performance, some trick must be used to quickly determine what's really visible, and only send that to the graphics card. This is known as occlusion culling and is nontrivial. Unity Pro has a solution for it, but it is not suited for randomly generated mazes. Fortunately, our simple trick of hiding rooms already helps a lot.