# Unit 3 – Lesson 10. CatLike Maze Generation – Part 1

**Aim:**

* How do we build a Unity project with C# scripts to generate mazes?

**Objectives:** After the lesson, students should be able to:

* Obtain better understanding of maze generation algorithms

**CLASS PROCEDURE:**

***Do Now:***

Open the Maze game project in Unity. Create a new empty game object and name it GameManager, and move the GameManager script from the camera to the newly created game object. What have we set up for the maze yesterday?

***Class Activity / Discussion:***

1. What is the difference between a game object and 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.

1. What does the Instantiate() do?

* Unity's [**Object**](http://docs.unity3d.com/Documentation/ScriptReference/Object.html) class, which every MonoBehaviour inherits from, contains the static Instantiate method.
* This method creates a clone of whatever [**Object**](http://docs.unity3d.com/Documentation/ScriptReference/Object.html) 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**](http://docs.unity3d.com/Documentation/ScriptReference/Object.html) 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.

1. What does the 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.

1. 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.

***Pair – sharing / Class Activity:***

Today we will try to complete the Maze Fundamental in class. Once again we follow the tutorial online <http://catlikecoding.com/unity/tutorials/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;

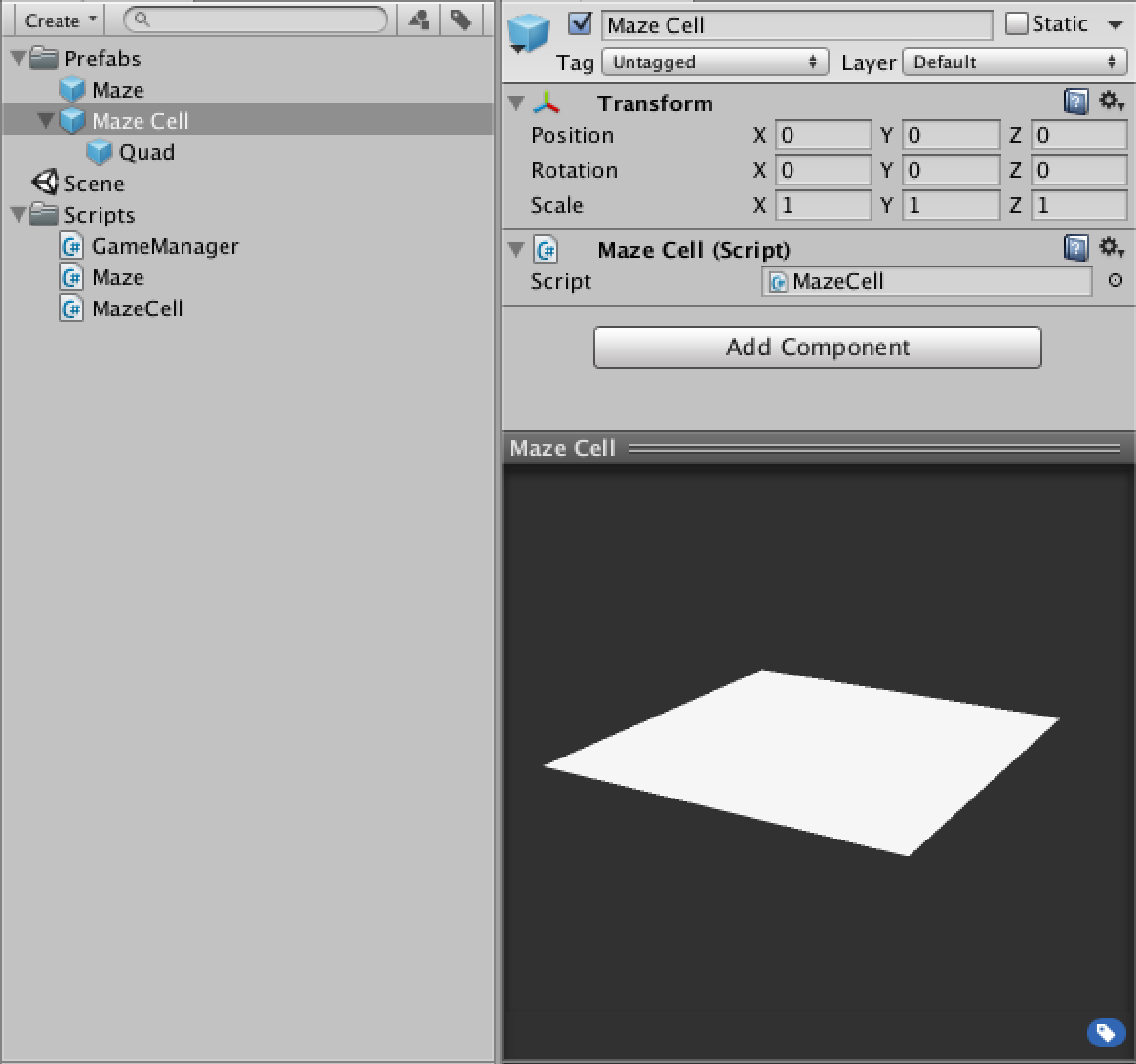
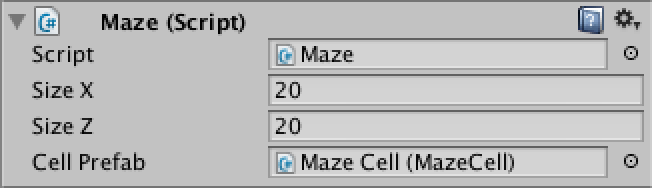
**public** **MazeCell** cellPrefab;

**private** **MazeCell**[,] cells;

**using** UnityEngine;

**public** **class** **MazeCell** : [**MonoBehaviour**](http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html) {}

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 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**](http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html)(x - sizeX \* 0.5f + 0.5f, 0f, z - sizeZ \* 0.5f + 0.5f);

}

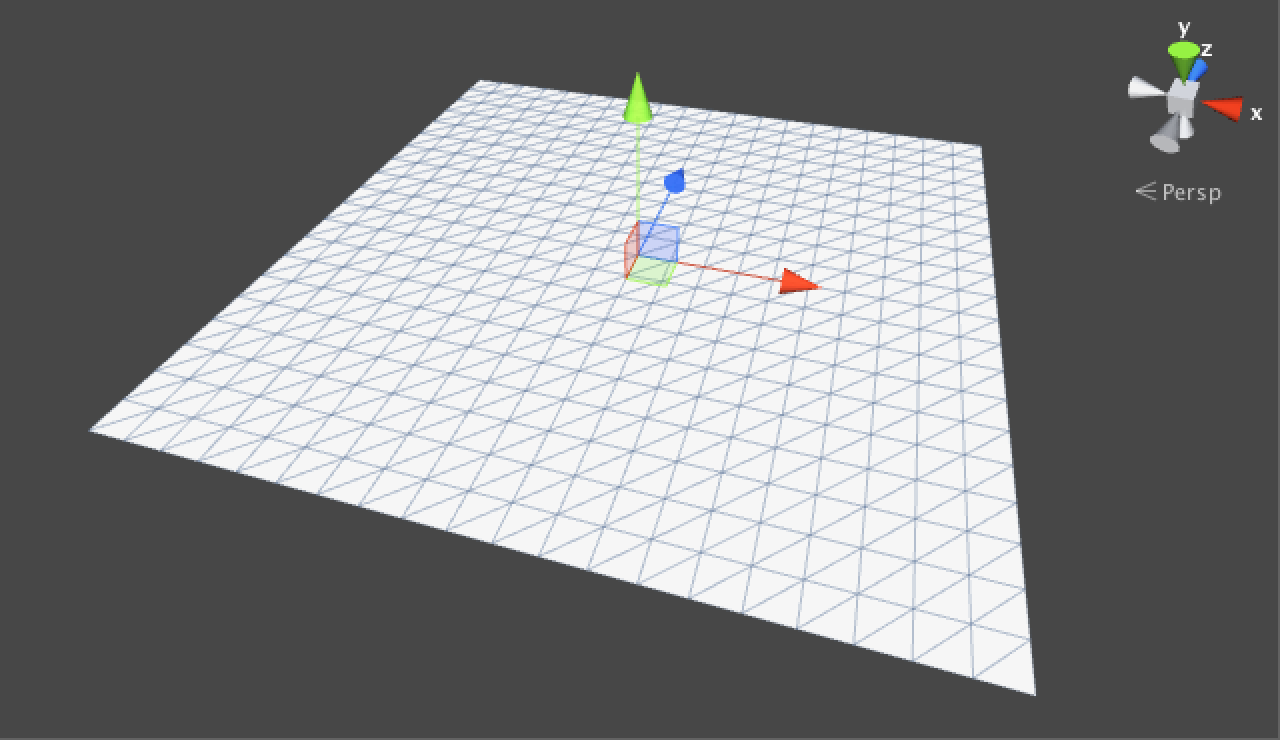
Now let **GameManager** call Generate and the maze should appear when you enter play mode.

**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](http://catlikecoding.com/unity/tutorials/maze/#a-wait-for-seconds)[**WaitForSeconds**](http://docs.unity3d.com/Documentation/ScriptReference/WaitForSeconds.html) work?

**public** **float** generationStepDelay;

**public** [**IEnumerator**](http://social.msdn.microsoft.com/search/en-us?query=IEnumerator) Generate () {

[**WaitForSeconds**](http://docs.unity3d.com/Documentation/ScriptReference/WaitForSeconds.html) delay = **new** [**WaitForSeconds**](http://docs.unity3d.com/Documentation/ScriptReference/WaitForSeconds.html)(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?](http://catlikecoding.com/unity/tutorials/maze/#a-coroutine-live)

**private** **void** BeginGame () {

mazeInstance = Instantiate(mazePrefab) **as** **Maze**;

StartCoroutine(mazeInstance.Generate());

}

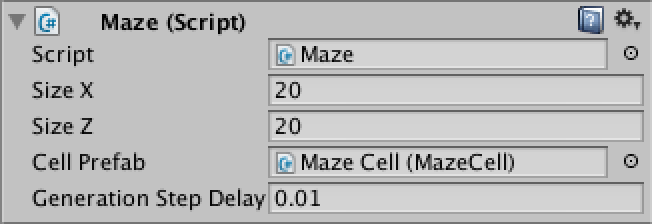
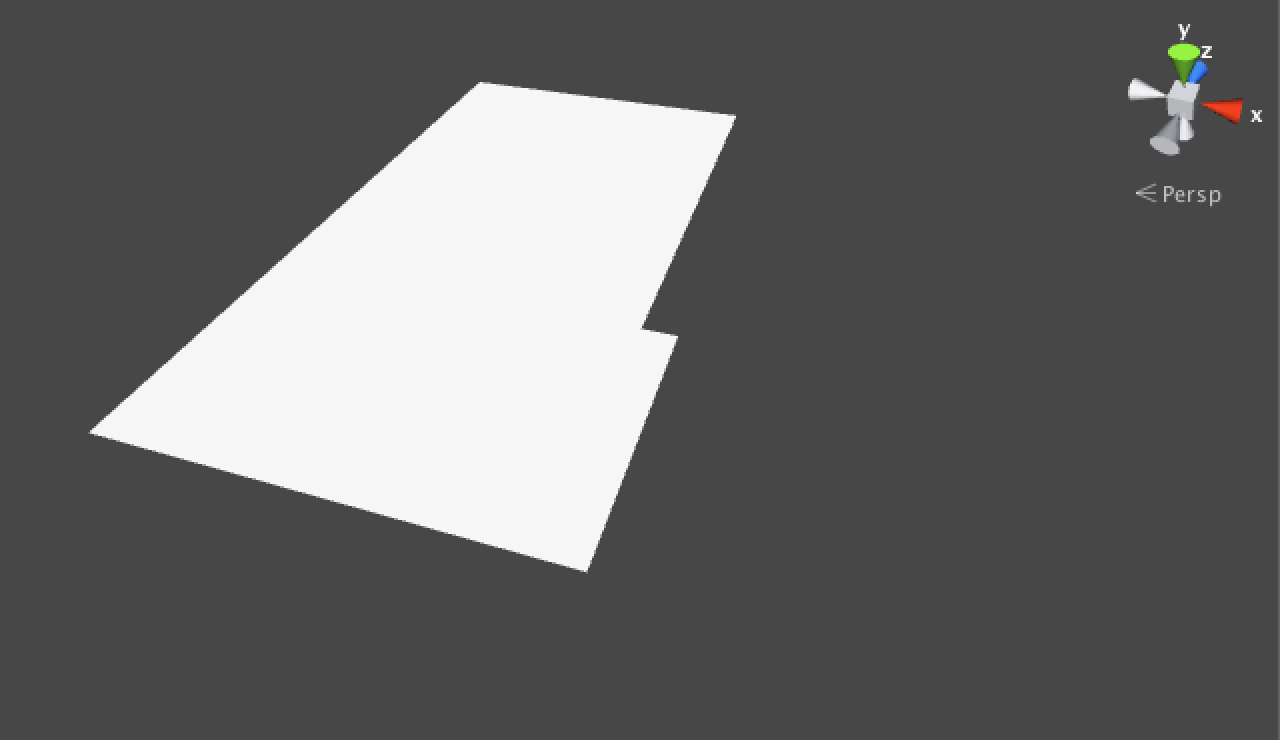
**private** **void** RestartGame () {

StopAllCoroutines();

Destroy(mazeInstance.gameObject);

BeginGame();

}



Maze generation with step delay.

If finish early, please continue to set up the cells. Have fun!