Procedural Generation

Objective

While procedural generation is a complex subject, I get asked a lot about it so this activity is designed to go over basics. These are all designed to be for prototyping purposes and are far from efficient or even the correct way to do it. They are designed to be simple to use and to give a pathway of understanding to students.

Basic 1D scene

Using loops to create a level

Setup

- Create a new scene.
- Make a default cube with a material and save it as a prefab.
- Delete the cube in the scene
- Create a script called ‘WorldCreator’
- Place the script on an empty gameobjects called ‘world’

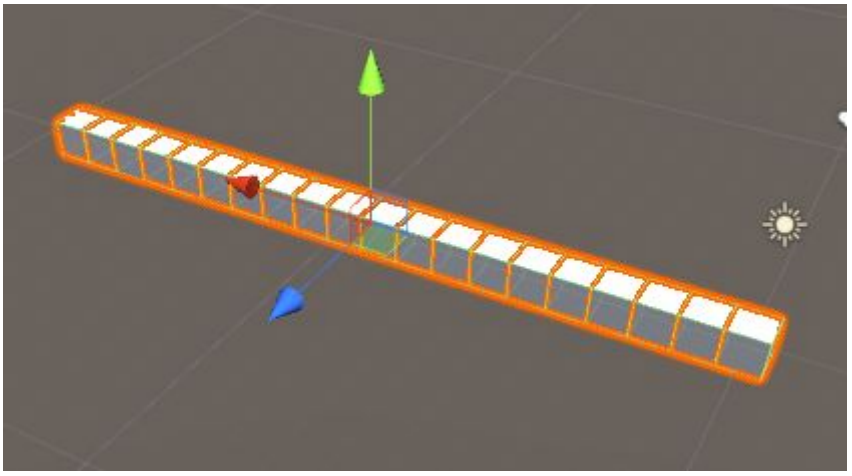
Lesson

In the ‘WorldCreator’ script

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

public class WorldCreator : MonoBehaviour
{

    public int worldSizeX = 20;
    public GameObject cubePrefab;

    // Use this for initialization
    void Start()
    {

        for (int x = 0; x < worldSizeX; x++)
        {
            Vector3 cubePosition = new Vector3(x, 0, 0);
            GameObject GO = Instantiate(cubePrefab, cubePosition, Quaternion.identity);
        }

    }
}
```

Make sure you drag the prefab to the script

Now you will see that we have made 20 cubes next to each other in a line.



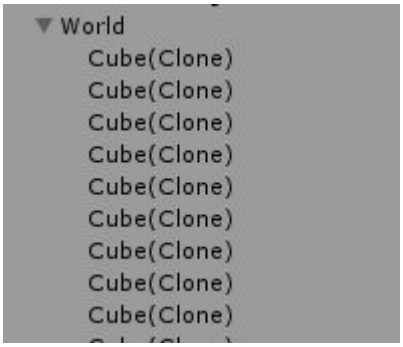
This is one of the powers of a for loop, each time we loop around we move the position of the next cube.

Because we are generating so many cubes (and we will make a lot more) we should work at cleaning up our hierarchy

Add this line

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

public class WorldCreator : MonoBehaviour
{

    public int worldSizeX = 20;
    public GameObject cubePrefab;

    // Use this for initialization
    void Start()
    {

        for (int x = 0; x < worldSizeX; x++)
        {
            Vector3 cubePosition = new Vector3(x, 0, 0);
            GameObject GO = Instantiate(cubePrefab, cubePosition, Quaternion.identity);

            GO.transform.SetParent(this.transform);
        }

    }
}
```

You should now see that our hierarchy is cleaner

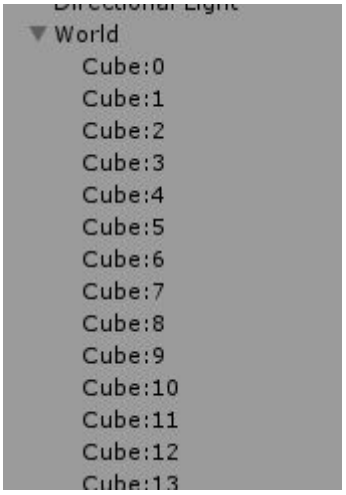


But the next issue we have is all our prefabs are named poorly.

Add this line under the instantiate line

```
    {
        Vector3 cubePosition = new Vector3(x, 0, 0
        GameObject GO = Instantiate(cubePrefab, cub

        GO.transform.SetParent(this.transform);
        GO.name = "Cube:" + x;
    }

}
```

See how much cleaner this looks



Results

When run the game should instantiate 20 blocks in a row that are correctly named.

Basic 2D & 3D scene

Using loops to create a 2D level

Setup

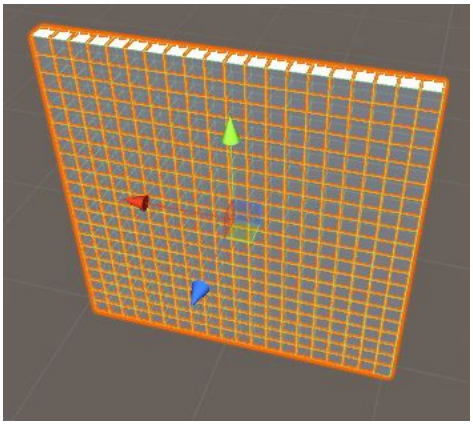
Continued from the previous scene

Lesson

Add these lines

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

public class WorldCreator : MonoBehaviour
{
    public int worldSizeY = 10;
    public int worldSizeX = 20;
    public GameObject cubePrefab;

    // Use this for initialization
    void Start()
    {

        for (int x = 0; x < worldSizeX; x++)
        {
            for (int y = 0; y < worldSizeX; y++)
            {
                Vector3 cubePosition = new Vector3(x, y, 0);
                GameObject GO = Instantiate(cubePrefab, cubePosition, Quaternion.identity);
                GO.name = "Cube: x-" + x + " y-" + y;
                GO.transform.SetParent(this.transform);
            }

        }

    }
}
```

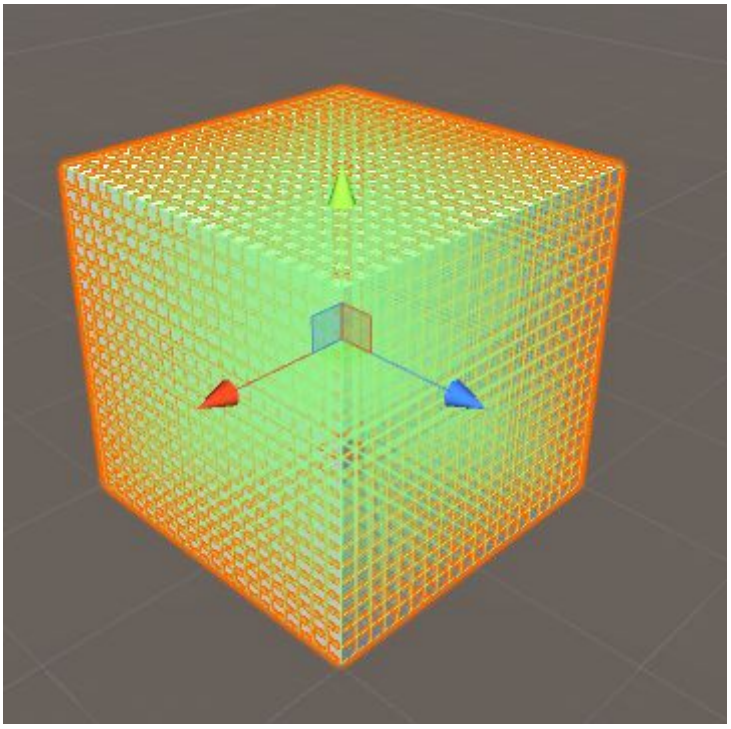
Now you should have 400 cubes



See if you can add the 3rd dimension without looking at the next bit of code

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

public class WorldCreator : MonoBehaviour
{
    public int worldSizeY = 10;
    public int worldSizeX = 20;
    public int worldSizeZ = 20;
    public GameObject cubePrefab;

    // Use this for initialization
    void Start()
    {

        for (int x = 0; x < worldSizeX; x++)
        {
            for (int y = 0; y < worldSizeX; y++)
            {
                for (int z = 0; z < worldSizeZ; z++)
                {
                    Vector3 cubePosition = new Vector3(x, y, z);
                    GameObject GO = Instantiate(cubePrefab, cubePosition, Quaternion.identity);
                    GO.name = "Cube: x-" + x + " y-" + y + " z-" + z;
                    GO.transform.SetParent(this.transform);
                }
            }
        }
    }
}
```

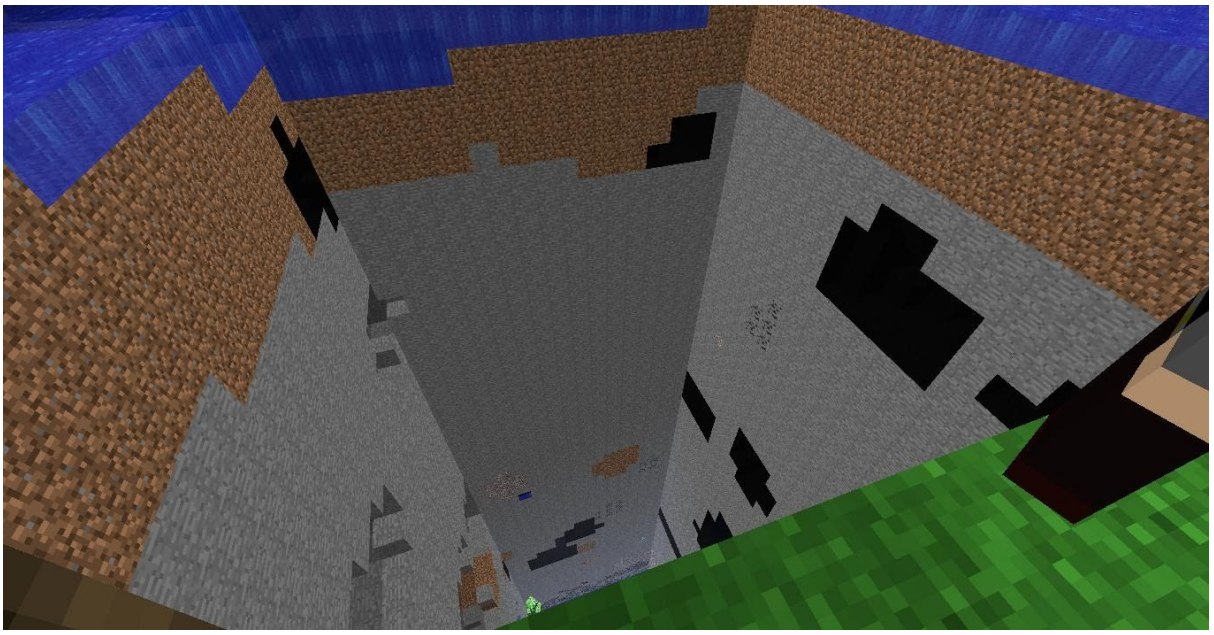
Now we have 3 dimensions with 8000 cube



Now, you might be thinking this is how Minecraft does its levels. And it is similar, but they do a lot of optimisation, Even through we have 8000 cubes, only 872 cubes can been seen so they don’t draw the other 7128 cubes.

And for each of those cubes that we can see, we can only see 1,2 or 3 of their sides. So they don't draw any sides that we cannot see.

It then stitches all the cubes into one object (called a chunk) to save on memory.
You can see this in minecraft if a server is lagging and a chunk doesn't load in correctly.



As you can imagine, this all takes a LOT of work to do and is far more complex than we cover in this course.

Results

When run the game should instantiate 20x10x20 block of cubes all correctly named and put in a parent game ibject;

Add some Color scene

Shows that we can use the numbers for more than just position.
This is a diversion to the lesson, but interesting to do.

Setup

Continued from the previous scene

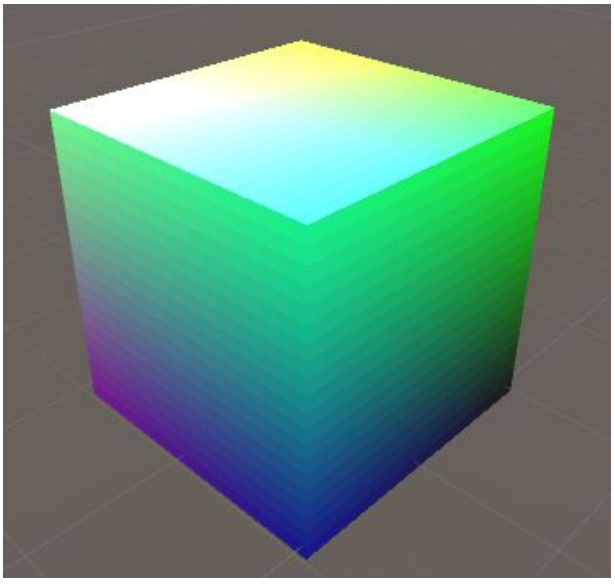
Lesson

New that we are using 3 loops to create a box, we can do different things to each box. Here is a little side thing to show this. It's REALLY memory inefficient as we are making 8000 different materials. But it does look cool.

Insert these lines below the SetParent line

```
for (int x = 0; x < worldSizeX; x++)
{
    for (int y = 0; y < worldSizeX; y++)
    {
        for (int z = 0; z < worldSizeZ; z++)
        {
            Vector3 cubePosition = new Vector3(x, y, z);
            GameObject GO = Instantiate(cubePrefab, cubePosition, Quaternion.identity);
            GO.name = "Cube: x-" + x + " y-" + y + " z-" + z;
            GO.transform.SetParent(this.transform);

            float red = (float)x / (worldSizeX - 1);
            float green = (float)y / (worldSizeY - 1);
            float blue = (float)z / (worldSizeZ - 1);
            GO.GetComponent<Renderer>().material.color = new Color(red, green, blue);

        }
    }
```

Run the script



Now comment out these lines as they will get in the way of the next part

```
//float red = (float)x / (worldSizeX - 1);
//float green = (float)y / (worldSizeY - 1);
//float blue = (float)z / (worldSizeZ - 1);
//GO.GetComponent<Renderer>().material.color = new Color(red, green, blue);
```

Results

When run all the cube will be a different colour based on its position;

Use Perlin

From here we are going to get very mathy. We are going to use a mathematical function to create noise in a certain way that we will use later to tell our world where we should put blocks.

Firsts the basics

Setup

Save the last scene (we will come back to it) and create a new scene
Create a plane and a script called ‘PerlinNoise’
The below script is taken from the Unity website and doesn’t need to be talked through, It’s a little complex.
I have also left it as text so it can be copy and pasted.

Lesson

```
using UnityEngine;
using System.Collections;

public class PerlinNoise : MonoBehaviour {
	private int pixWidth = 100;
	private int pixHeight = 100;
	public float xOrg;
	public float yOrg;
	public float scale = 10.0f;
	private Texture2D noiseTex;
	private Color[] pix;
	private Renderer rend;
	void Start() {
		rend = GetComponent<Renderer>();
		noiseTex = new Texture2D(pixWidth, pixHeight);
		noiseTex.filterMode = FilterMode.Point;
		pix = new Color[noiseTex.width * noiseTex.height];
		rend.material.mainTexture = noiseTex;
	}
	void CalcNoise() {
		float y = 0.0F;
		while (y < noiseTex.height) {
			float x = 0.0F;
			while (x < noiseTex.width) {
				float xCoord = xOrg + x / noiseTex.width
* scale;
```

```
                        float yCoord = yOrg + y / noiseTex.height
* scale;
                        float sample = Mathf.PerlinNoise(xCoord,
yCoord);
                        pix[(int)y * noiseTex.width + (int)x] =
new Color(sample, sample, sample);
                        x++;
                    }
                    y++;
                }
                noiseTex.SetPixels(pix);
                noiseTex.Apply();
            }
            void Update() {
                CalcNoise();
            }
}
```

Once attached to the plane you can run the game and play with it's numbers to change the values of the texture to show the noise.

These are just numbers between 0 and 1 (represented as a shade between black and white) and we can use these to cut out cubes from our world.
So far this is shown dimensions, we can use 2 of these perlin noise generators to make it in 3 dimensions.

Let's demonstrate this on our cubes

Save this scene and go back to the original

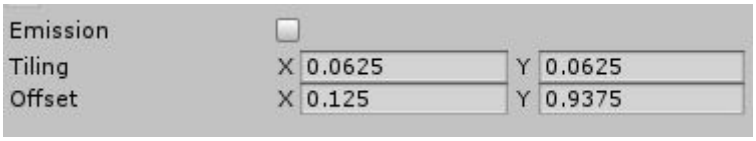
First we need to make a new script called **'Cube'** and place it on the cube prefab.

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

public class Cube : MonoBehaviour
{

    public Vector3 position;
}
```

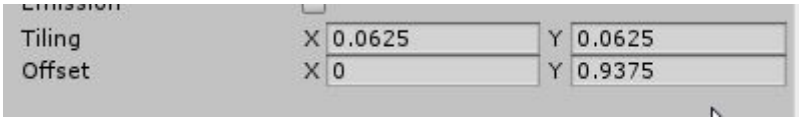
Also duplicate the cube prefab call one '**Stone'** and one '**Dirt'**
We have provided a textures for you to place on these prefabs

Here are the material settings for each texture

Dirt



Stone



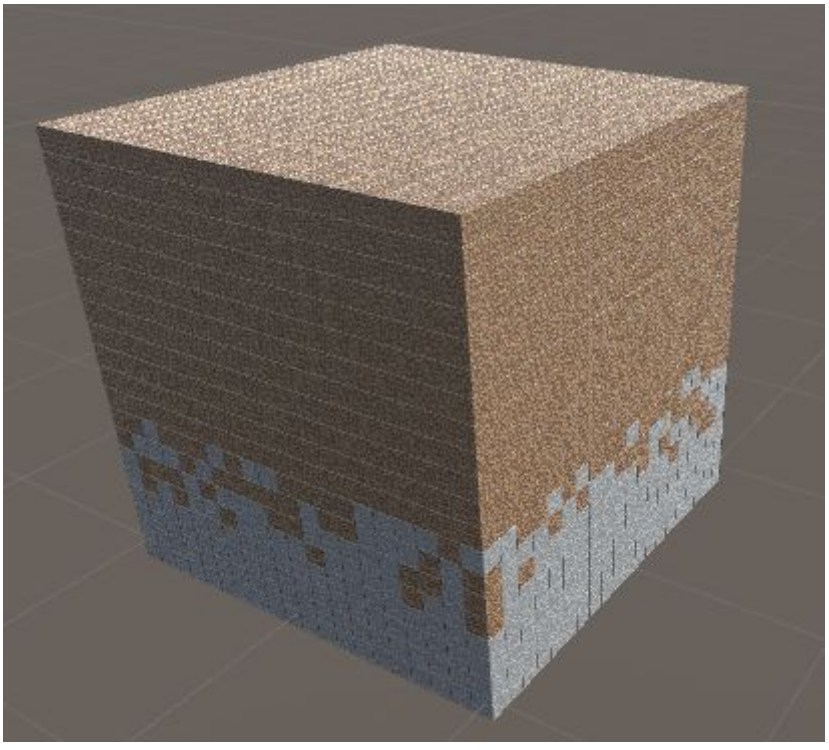
Then update the **WorldCreator** Script

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

public class WorldCreator : MonoBehaviour {
```

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

public class WorldCreator : MonoBehaviour
{
    public int worldSizeY = 10;
    public int worldSizeX = 20;
    public int worldSizeZ = 20;
    public GameObject cubePrefab;

    public GameObject dirtPrefab;
    public GameObject stonePrefab;

    private List<Cube> cubeList = new List<Cube>();

    //Surface
    [Range(1f, 100f)]
    public float surfaceScale = 2;

    [Range(0f, 1f)]
    public float surfaceCutoff = 0.7f;

    [Range(0f, 100f)]
    public float surfaceOffsetX = 0;

    [Range(0f, 100f)]
    public float surfaceOffsetZ = 0;
```

```
for (int y = 0; y < worldSizeX; y++)
{
    for (int z = 0; z < worldSizeZ; z++)
    {
        GameObject cubeToPlace = null;
        if (Random.Range(0, y + 5) > worldSizeY - y || y > worldSizeY - 3)
        {
            cubeToPlace = dirtPrefab;
        }
        else
        {
            cubeToPlace = stonePrefab;
        }

        Vector3 cubePosition = new Vector3(x, y, z);
        GameObject GO = Instantiate(cubeToPlace, cubePosition, Quaternion.identity);
        GO.name = "Cube: x-" + x + " y-" + y + " z-" + z;
        GO.transform.SetParent(this.transform);

        GO.GetComponent<Cube>().position = new Vector3(x, y, z);
        cubeList.Add(GO.GetComponent<Cube>());

        //float red = (float)x / (worldSizeX - 1);
        //float green = (float)y / (worldSizeY - 1);
        //float blue = (float)z / (worldSizeZ - 1);
        //GO.GetComponent<Renderer>().material.color = new Color(red, green, blue);

    }
}
```

When run we can make a layer of dirt and Stone



Now we add the cave system

We are going to add a lot, so I will break it into different parts

First add these variables to the top of the script

```
//Cave System
[Range(0.01f, 20)]
public float caveScale = 10;

[Range(0f, 1f)]
public float caveCutoff = 0.7f;

[Range(0f, 100f)]
public float caveOffsetX = 0;
[Range(0f, 100f)]
public float caveOffsetY = 0;
[Range(0f, 100f)]
public float caveOffsetZ = 0;
```

Add this function (don't worry about missing reference to the Perlin3D function, we will be adding it soon (*see next page*))

```
public void UpdateCube()
{
    foreach (Cube cube in cubeList)
    {
        if (Perlin3D((cube.position.x) / caveScale, (cube.position.y) /
            caveScale, (cube.position.z) / caveScale) > caveCutoff)
        {
            //Check if part of the surface
            if (Mathf.PerlinNoise((cube.position.x + surfaceOffsetX) /
                surfaceScale, (cube.position.z + surfaceOffsetZ) / surfaceScale) >
                (cube.position.y / (float)worldSizeY) - surfaceCutoff)
            {
                cube.gameObject.SetActive(true);
            }
            else {
                cube.gameObject.SetActive(false);
            }
        }
        else {
            cube.gameObject.SetActive(false);
        }
    }
}
```

This function

```
void Update()
{
    UpdateCube();
}
```

And this function

```
private float Perlin3D(float x, float y, float z)
{
    x += caveOffsetX;
    y += caveOffsetY;
    z += caveOffsetZ;

    float AB = Mathf.PerlinNoise(x, y);
    float BC = Mathf.PerlinNoise(y, z);
    float AC = Mathf.PerlinNoise(x, z);

    float BA = Mathf.PerlinNoise(y, x);
    float CB = Mathf.PerlinNoise(z, y);
    float CA = Mathf.PerlinNoise(z, x);

    float ABC = AB + BC + AC + BA + CB + CA;

    float average = ABC / 6;
    return average;
}
```

Once run you should have the ability to create a world