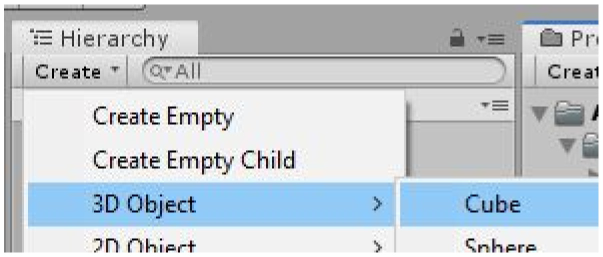
How to make a procedural tree in Unity

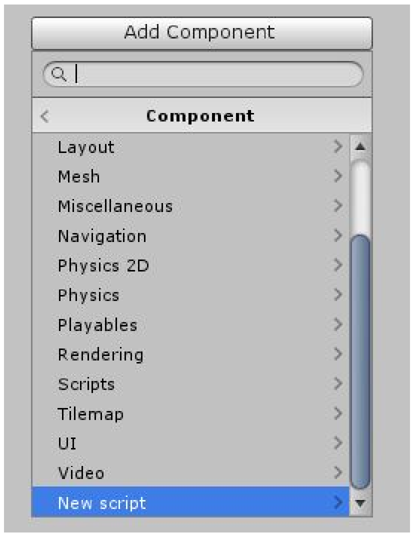
Creating a Unity project
Unity might be on your machine's desktop, but if it's not you can find it on AppsAnywhere. Open Unity, click "New", choose a name and save location and select the "3D" template if it's not already selected.

Creating a Branch prefab
For the branches we'll be using a Prefab. Unity Prefabs are basically templates for objects. You can place instances of a Prefab into the game world, and when you modify the Prefab all the instances will update too.
We'll just use a basic cube for the branch model. Click Create > 3D Object > Cube at the top of the Hierarchy panel. This will create a cube in the scene. Right click to rename it to "Branch".



The Hierarchy panel shows the objects in the scene and the Project panel shows the files that make up the project. Click and drag the cube from the Hierarchy panel to the Project panel. This will make it into a Prefab. You can then delete the cube from the Hierarchy by right clicking it.

Creating a TreeGenerator script component
Unity scenes are made up of GameObjects with components attached, such as meshes, lights and scripts. We want an object that only has a script attached. Click Create > Create Empty in the Hierarchy panel. Select the GameObject and click "Add Component" in the Inspector panel. Select "New script" and call it "TreeGenerator". Double click the script to open it in Visual Studio.

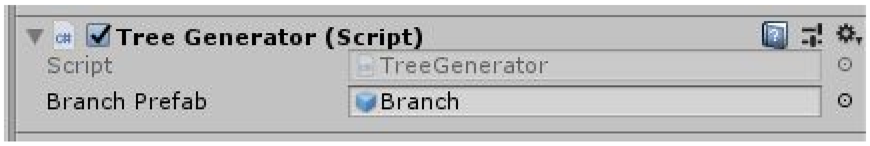


Generating the root branch
We'll make a public GameObject for the branch prefab and instantiate it in the Start function that was generated when we created the script. The Start function will be called once when we press Play. You can delete the Update function because we won't be using it.

```
public class TreeGenerator : MonoBehaviour
{
	public GameObject branchPrefab;

	void Start()
	{
		GameObject rootBranch = Instantiate(branchPrefab);
	}
}
```

There should now be a field in the script's inspector called "Branch Prefab". Click and drag the prefab from the Project panel to the field.


Tree Generator (Script)
Script: TreeGenerator
Branch Prefab: Branch

When you press Play, a cube should appear in your scene.

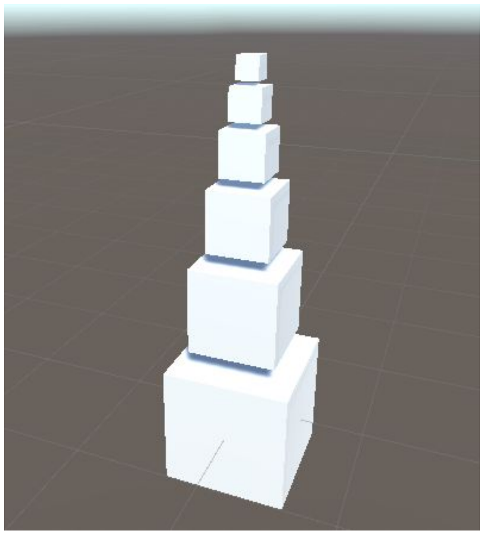
Generating child branches

We’re going to use a recursive function to generate the tree’s branches. We don’t want to call it endlessly, so when we call it we’ll tell it how many iterations to do.
We previously used Instantiate to create the root branch, but this time we’re also passing in a parent transform. This means the new branch will be a child of the old branch. We set the localPosition and localScale of the branch, which means the position and scale relative to the parent. For example, we set the local scale to 0.7, meaning each branch will be 0.7 times the size of its parent.

```
private void GenerateBranches(GameObject parentBranch, int iterations)
{
	GameObject branch1 = Instantiate(branchPrefab, parentBranch.transform);
	branch1.transform.localPosition = Vector3.up; // This gives you the vector (0, 1, 0)
	branch1.transform.localScale = Vector3.one * 0.7f; // This gives you the vector (0.7, 0.7,
0.7)

	// If iterations equals 1, we’ve reached the smallest branch and no more branches should
be generated.
	if (iterations > 1)
	{
		GenerateBranches(branch1, iterations - 1);
	}
}
```

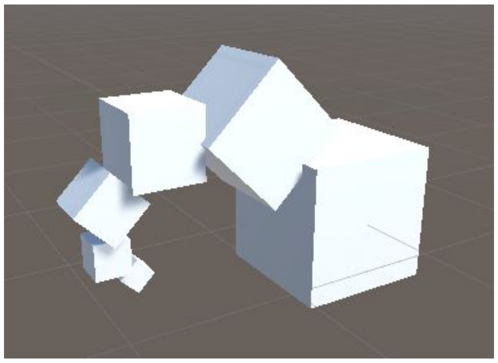
Call the function in Start and pass it the root branch and an iterations value of 5. When you press Play you should now get a weird tower thing!



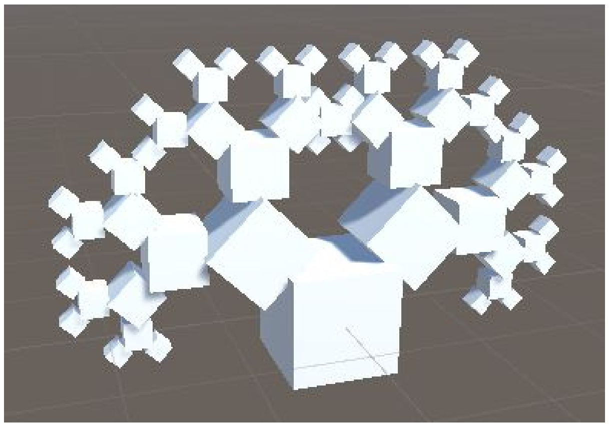
Changing the branch direction
So far child branches always face the same direction as their parents. In order to rotate them, we'll need to use a Quaternion. Quaternions are a way of storing 3D rotations. The maths behind them is complex, but in Unity there are functions to help us. The Quaternion.Euler function creates a Quaternion with the specified rotations in degrees around the x, y and z axes. We create a Quaternion and assign it to localRotation. We also rotate the localPosition using the Quaternion with the * operator.

```
Quaternion branch1Rotation = Quaternion.Euler(0, 0, 45);
branch1.transform.localRotation = branch1Rotation;
branch1.transform.localPosition = branch1Rotation * Vector3.up;
```

Your tower should now curl over to the left.



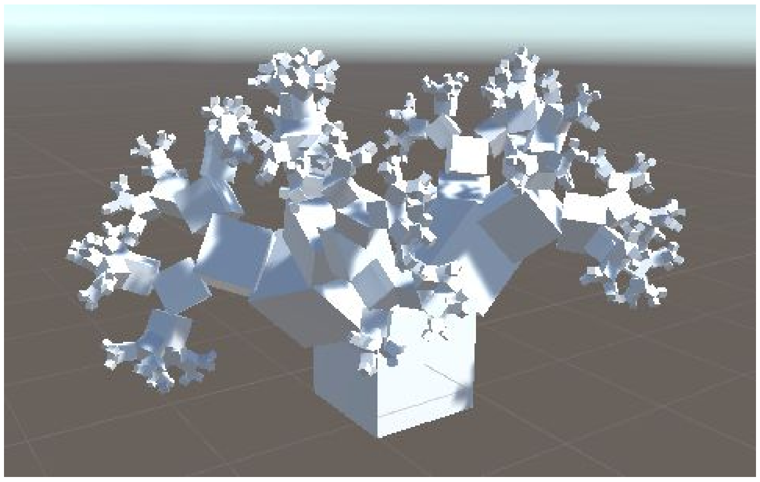
Creating more branches
Duplicate the code for the child branch and modify it to make a second child branch, this time using a branch rotation of Quaternion.Euler(0, 0, -45). You should now have a branching fractal.



Making it look more organic
When trying to make procedural objects look more organic, a common approach is to use randomisation. We're going to randomly twist the branches.
The twist rotation needs to be applied after our original rotation. Our original rotation was in the z axis and the twist needs to be in the y axis. However, Quaternion.Euler applies the rotations in the order z, x, y. As a workaround, we'll create two Quaternions and combine them in the order we want, using the * operator. Random.Range(0, 360) gives a random float in the range 0 to 360, representing a full rotation in degrees.

```
    Quaternion branch1Rotation =
        Quaternion.Euler(0, 0, 45) *
        Quaternion.Euler(0, Random.Range(0, 360), 0);
```

You should now have something that looks a bit like a tree!



If you've got time left over...

There's a lot of ways in which the program could be improved. Try making the tree more realistic or interesting! For example:

- Add more fields in the inspector - Instead of always using 45 degrees, you could let the user choose how spread out the branches are.
- Add leaves - Make a second prefab called "Leaf" and instantiate it whenever iterations reaches 1. You could make a material so it looks different to the branches.
- Re-generate the tree when a key is pressed - If you call Destroy(rootBranch) it'll destroy all the children too. Use Input.GetKeyDown() inside the Update function. The Update function is called once per frame.
- Generate a whole forest! Hopefully that doesn't break the lab machines...

If you get stuck, ask us and we can try to help. We'd love to see what you come up with!