Catlike Coding

Unity C# Tutorials

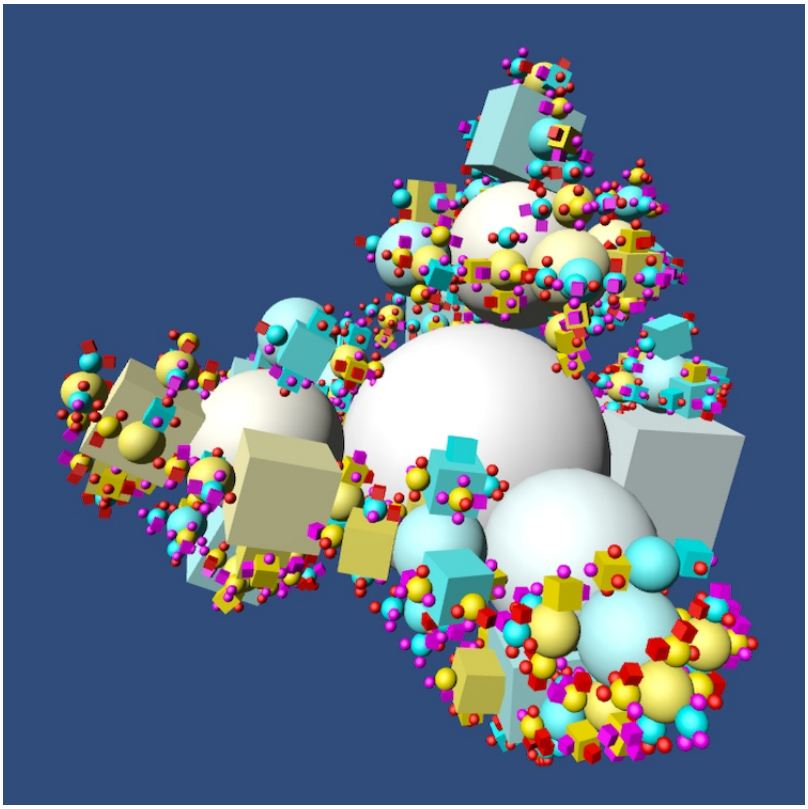
Constructing a Fractal Details Through Recursion

Instantiate game objects.
Work with recursion.
Use coroutines.
Add randomness.

Fractals are intriguing and often beautiful. In this tutorial we'll write a small C# script that manifests some fractal-like behavior.

You're assumed to know your way around Unity's editor and know the basics of creating C# scripts. If you've completed the Clock tutorial then you're good to go.

This is an old tutorial and will be upgraded, but it's still fun to do. Mentions of diffuse and specular materials no longer apply to Unity 2017, so you can ignore that.



You'll create randomized 3D fractals.

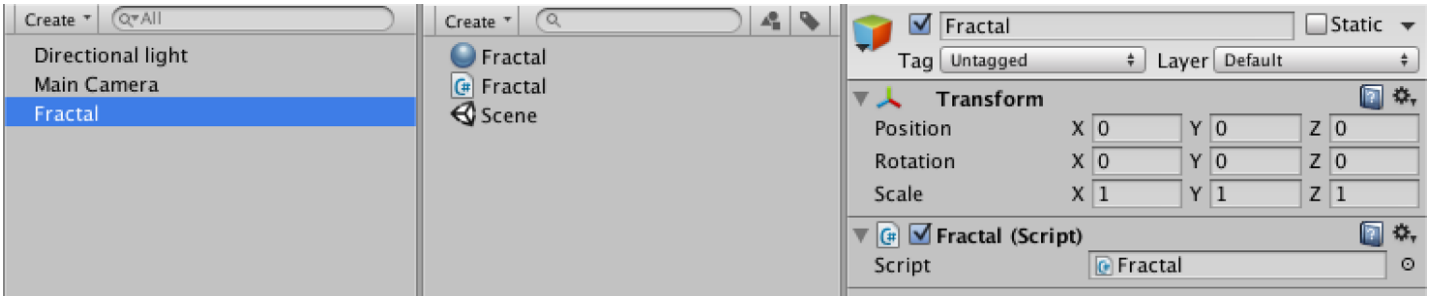
1 How to Make a Fractal

We are going to create a 3D fractal. We use the concept that the details of a fractal can look exactly like the whole thing. We can apply this to an object hierarchy in Unity. Start with some root object, then add children to it that are smaller but otherwise identical. Doing this by hand would be cumbersome, so we'll create a script to do it for us.

Start with a new project and a new scene. I put a directional light in there and moved the camera to a more interesting angle, but you can set it up as you please. I also went ahead and created a material to use for the fractal later. It simply uses the specular shader with default settings, which is just a bit more pleasing to look at than the default diffuse one.

Create a new empty game object and place it at the origin. This will be the base of our fractal. Then create a new C# script named *Fractal* and add it to the object.

```
using UnityEngine;
using System.Collections;

public class Fractal : MonoBehaviour {

}
```



Project setup.

2 Showing Something

What will our fractal look like? Let's make it configurable by adding a public mesh and material variable to our `Fractal` component script. We then insert a `Start` method in which we add a new `MeshFilter` component and a new `MeshRenderer` component. At the same time, we directly assign our mesh and material to them.

```
using UnityEngine;
using System.Collections;

public class Fractal : MonoBehaviour {

	public Mesh mesh;
	public Material material;

	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
	}
}
```

What's a mesh?

Mechanically, a `Mesh` is a construct used by the graphics hardware to draw complex stuff. It's a 3D object that's either imported into Unity, one of Unity's default shapes, or generated by code.

A mesh contains at least a collection of points in 3D space plus a set of triangles – the most basic 2D shapes – defined by these points. The triangles constitute the surface of whatever the mesh represents. Often, you won't realize that you're looking at a bunch of triangles instead of a real object.

What's a material?

Materials are used to define the visual properties of objects. They can range from very simple, like a constant color, to very complex.

Materials consist of a shader and whatever data the shader needs. Shaders are basically scripts that tell the graphics card how an object's polygons should be drawn.

The standard diffuse shader uses a single color and optionally a texture, along with the light sources in the scene, to determine the appearance of polygons. I'm using the slightly more complex specular shader here, which simulates a highlight as well.

When is `Start` invoked?

The **`Start`** method is called by Unity after the component is created, once it's active, and just before the first time its **`Update`** method would be called, if it had one. It's only called once.

How does `AddComponent` work?

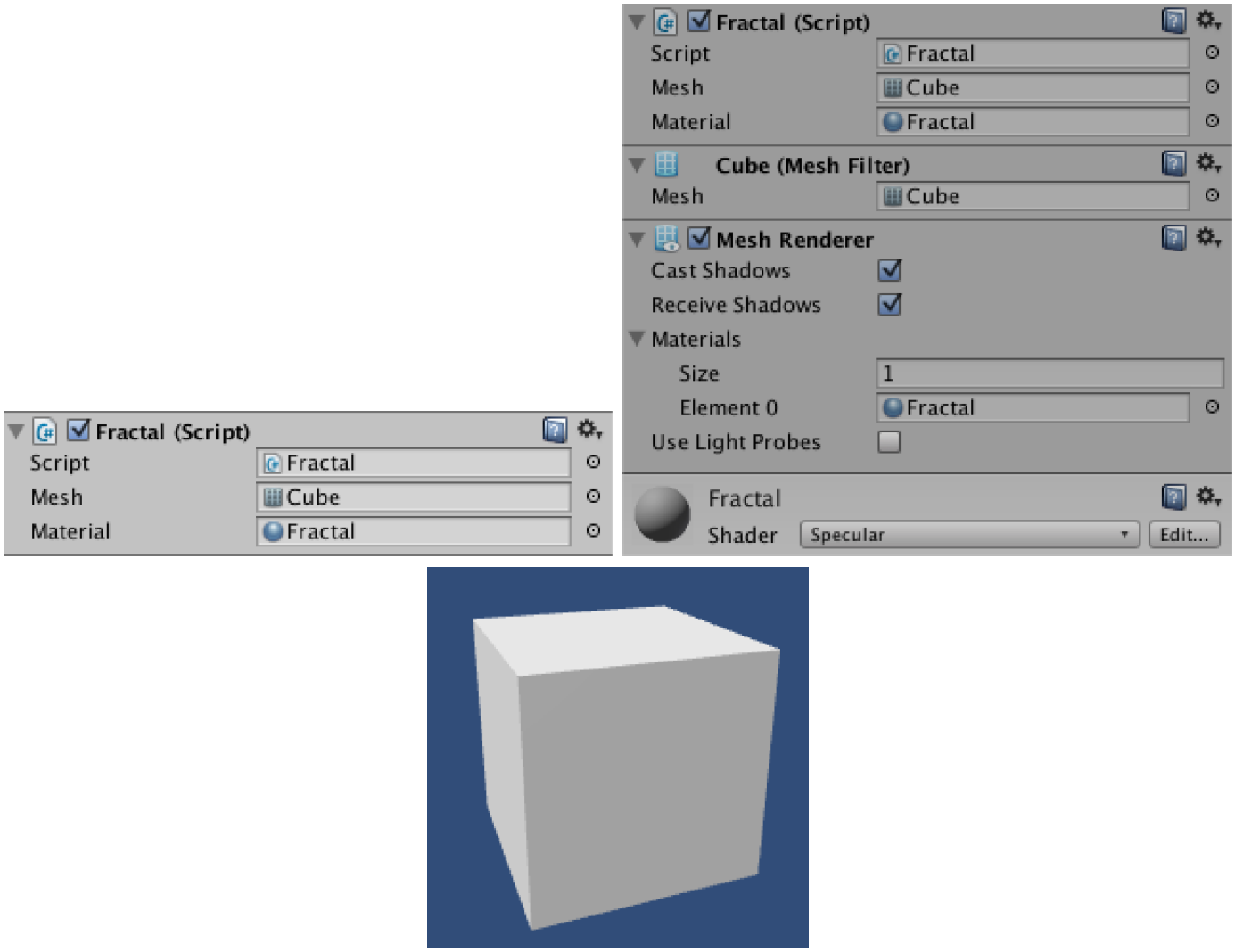
The `AddComponent` method creates a new component of a certain type, attaches it to the game object, and returns a reference to it. That's why we can immediately access the component's values. You could also use an intermediate variable.

```
		MeshFilter filter = gameObject.AddComponent<MeshFilter>();
		filter.mesh = mesh;
```

The special syntax is because it's a generic method. It's effectively a method template that can work with a range of types. You tell it what type to use by mentioning it between angle brackets.

Now we can assign our custom material to the fractal component. Also assign a mesh by clicking on the dot next to the property and selecting Unity's default cube from the popup. After doing so, a cube will show up when we enter play mode.

Of course we could have also added the components manually, but we're creating a fractal here.



Components appear in play mode.

3 Making Children

How do we create the children of this fractal? The easiest way is to just go ahead and make one in `Start` by creating a new game object and adding a `Fractal` component to it. Try it, hit play, and quickly exit play mode again.

```
	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		new GameObject("Fractal Child").AddComponent<Fractal>();
	}
```

What does `new` do?

The `new` keyword is used to construct a new instance of an object or a struct. It's followed by calling a special constructor method, which has the same name as the class or struct it belongs to.

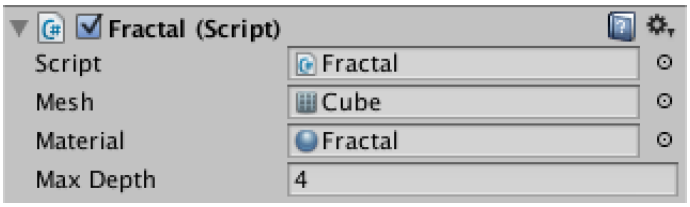
The problem is that every new fractal instance will create yet another one. This happens each frame, without end. Let it run for a while and your computer will get into trouble as it runs out of memory. Typically recursive algorithms that don't stop will consume your machine's resources almost instantly and result in either a stack overflow exception or a crash. In this case it is a rather benign explosion, because it happens slowly.

To prevent this from happening, we introduce the concept of a maximum depth. Our initial fractal instance will have a depth of zero. It's child will have a depth of one. The child of this child will have a depth of 2. And so on, until the maximum depth is reached.

Add a public `maxDepth` integer variable and set it to 4 in the inspector. Also add a private `depth` integer. Then only create a new child if we are below the maximum depth.

What will happen when we enter play mode now?

```
	public int maxDepth;

	private int depth;

	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		if (depth < maxDepth) {
			new GameObject("Fractal Child").AddComponent<Fractal>();
		}
	}
```



Maximum depth.

Exactly one child got created. Why? Because we never gave `depth` a value, it's always zero. Because zero is less than 4, our root fractal object created a child. The child's `depth` value is also zero. However, we never set the child's `maxDepth` either, so it is also zero. Therefore the child did not create another one.

Besides that, the child also lacks a material and a mesh. We need to copy these references from its parent. Let's add a new method that takes care of all the necessary initializations.

```
	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		if (depth < maxDepth) {
			new GameObject("Fractal Child").
				AddComponent<Fractal>().Initialize(this);
		}
	}

	private void Initialize (Fractal parent) {
		mesh = parent.mesh;
		material = parent.material;
		maxDepth = parent.maxDepth;
		depth = parent.depth + 1;
	}
```

What's `this`?

The `this` keyword refers to the current object or struct whose method is being called. It's being used implicitly all the time when referring to stuff from the same class. For example, whenever we've accessed `depth` we could have also done it via `this.depth`.

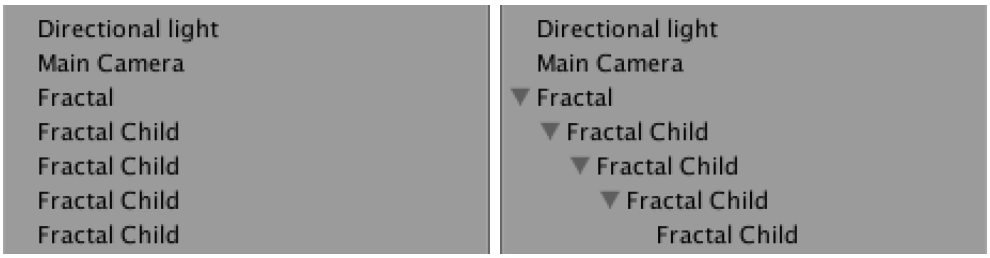
You typically only use this when you need to pass along a reference to the object itself, like we do for `Initialize`. Why? Because we're calling the `Initialize` method of the new child object, not of the parent object.

Is `Initialize` invoked before `Start`?

Yes, it is. First the new game object is created. Then a new `Fractal` component is created and added to it. At this point its `Awake` and `OnEnable` methods would be invoked, if they had existed. Then the `AddComponent` method finishes. Directly after that we invoke `Initialize`. The call to `Start` won't happen until the next frame.

When we enter play mode this time, four children will be created, as was expected. But they're not really children, as they all appear in the hierarchy root. The parent-child relationship between game objects is defined by their transformation hierarchy. So a child needs to make the parent of it's transform component equal to it's fractal parent's transform.

```
	private void Initialize (Fractal parent) {
		mesh = parent.mesh;
		material = parent.material;
		maxDepth = parent.maxDepth;
		depth = parent.depth + 1;
		transform.parent = parent.transform;
	}
```



Children, without and with nesting.

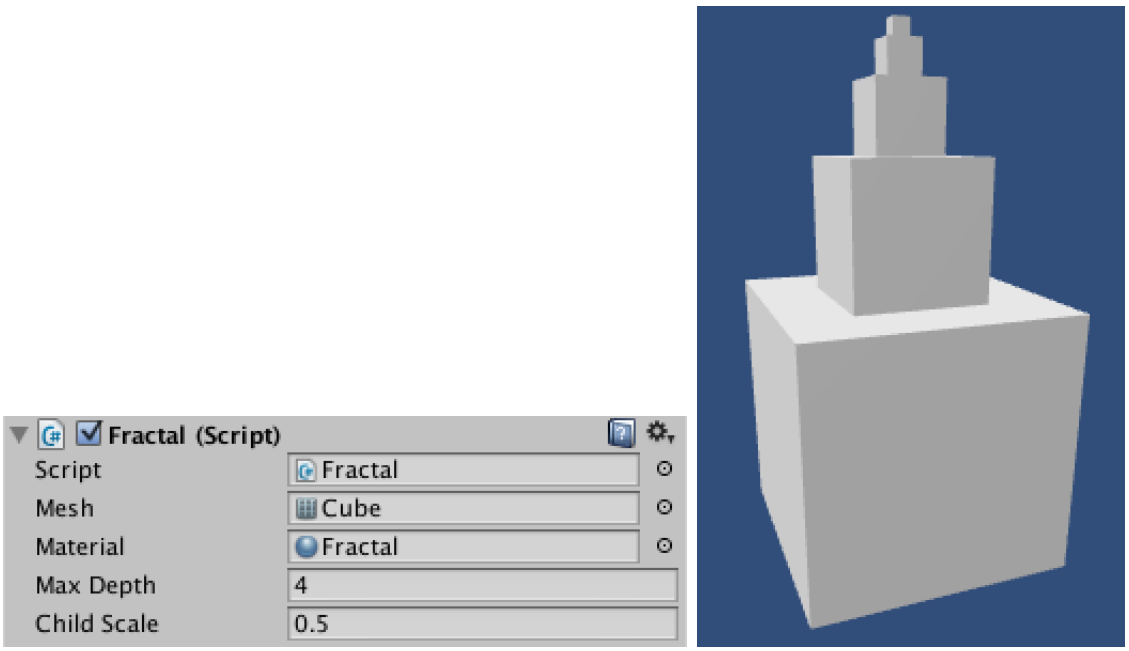
4 Shaping Children

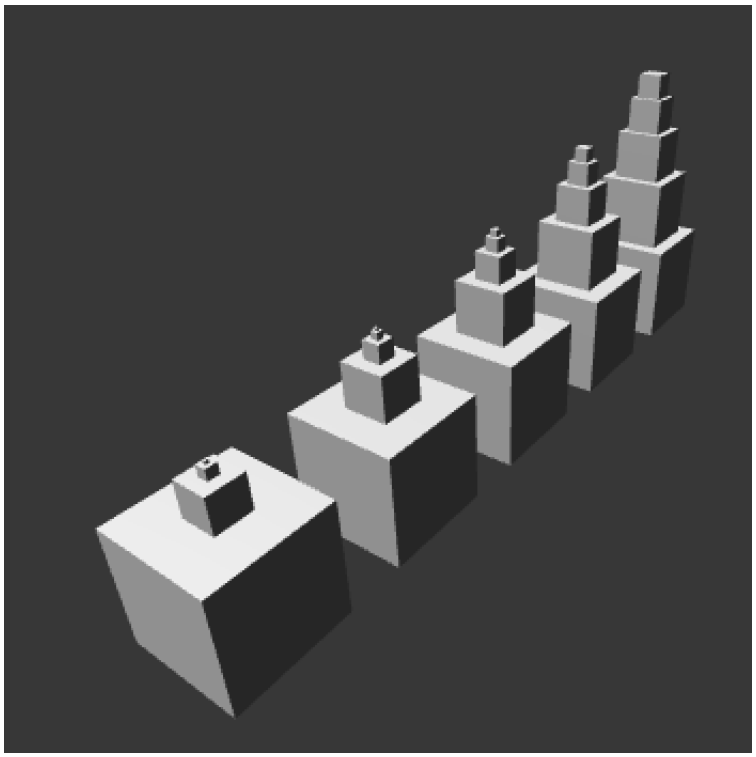
So far the children are superimposed on their parent, which means we still only see a single box. We need to move them inside their local space so that they become visible. And because they are supposed to be smaller than their parent, we have to scale them down too.

First the scaling. How much? Let's make it configurable with a new variable named `childScale` and assign it a value of 0.5 in the inspector. Don't forget to pass this value from parent to child as well. Then use it to set the child's local scale.

Next, where shall we move the children? Let's simply move them straight up, so that they touch their parent. We assume that the parent has a size of one in all directions, which is true for the cube that we're using. Moving up by a half puts us at the point where parent and child should touch. So we have to move an additional amount equal to half the size of the child.

```
	public float childScale;

	private void Initialize (Fractal parent) {
		mesh = parent.mesh;
		material = parent.material;
		maxDepth = parent.maxDepth;
		depth = parent.depth + 1;
		childScale = parent.childScale;
		transform.parent = parent.transform;
		transform.localScale = Vector3.one * childScale;
		transform.localPosition = Vector3.up * (0.5f + 0.5f * childScale);
	}
```





Child scale of 0.5, and from 0.3 to 0.7 for comparison.

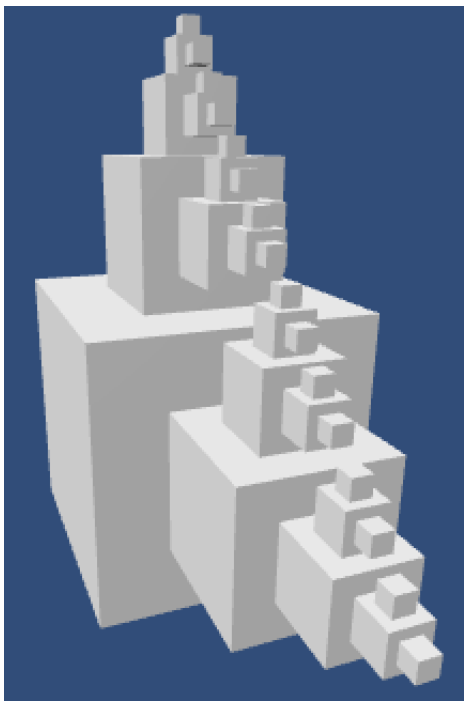
5 Making Multiple Children

What we are creating now looks like a tower, but not really a fractal. We need it to branch, by creating multiple children per parent. It is easy to just create a second object, but it also has to grow in a different direction. So let's add a direction parameter to our `Initialize` method and use that to position the second child to the right instead of up.

```
	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		if (depth < maxDepth) {
			new GameObject("Fractal Child").AddComponent<Fractal>().
				Initialize(this, Vector3.up);
			new GameObject("Fractal Child").AddComponent<Fractal>().
				Initialize(this, Vector3.right);
		}
	}

	private void Initialize (Fractal parent, Vector3 direction) {
		…
		transform.localPosition = direction * (0.5f + 0.5f * childScale);
	}
```

What does … mean?

It means that I have omitted a chunk of code that hasn't changed. It should be clear where the new or changed code is situated, or its exact placement doesn't matter.



Two children per parent.

This is starting to look more like it! Do you still understand in what order the cubes have been created? As they're all created in a few frames, it goes too fast for us to see it grow. I think it's fun to slow this process down so we can watch it happen. We can do this by using a coroutine to create the children.

Think of coroutines as methods in which you can insert pause statements. While the method invocation is paused, the rest of the program continues. Though this point of view is too simplistic, it is all we need to make use of it right now.

Move the two child creation lines to a new method named `CreateChildren`. This method needs to have `IEnumerator` as a return type, which exists in the `System.Collections` namespace. That's why Unity includes it in their default script template and why I included it in the beginning as well.

Instead of just calling this method in `Start`, we have to call it as an argument for Unity's `StartCoroutine` method.

Then add a pause directive before creating each child. Do this by creating a new `WaitForSeconds` object for half a second or so, then yielding it back to Unity.

```
	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		if (depth < maxDepth) {
			StartCoroutine(CreateChildren());
		}
	}

	private IEnumerator CreateChildren () {
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").
			AddComponent<Fractal>().Initialize(this, Vector3.up);
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").
			AddComponent<Fractal>().Initialize(this, Vector3.right);
	}
```

What's an enumerator?

Enumeration is the concept of going through some collection one item at a time, like looping over all elements in an array. An enumerator – or iterator – is an object that provides an interface for this functionality. `System.Collections.IEnumerator` describes such an interface.

Why do we need this? Because coroutines use them. This is also why Unity includes `System.Collections` in their default script template, and why I included it as well.

What does `return` do?

You use the `return` keyword to indicate that a method is finished and what its result is. What you return must match the type of the method. If it's a `void` method then you simply return nothing.

It's not needed to have a `return` statement at the end of a `void` or a special constructor method, for all other methods it's required.

It is possible to have multiple `return` statements inside a method. In that case there are multiple possible exit points. You'd typically use `if` statements to determine which `return` gets used.

What does `yield` do?

The `yield` statement is used by iterators to make life easy for them. To make enumeration possible, you'd need to keep track of your progress. This involves some boilerplate code that is essentially always the same. What you'd really want is to just write something like `return firstItem; return secondItem;` until you are done. The `yield` statement allows you to do exactly that.

So whenever you're using `yield`, an enumerator object is created behind the scenes to take care of the tedious bits. That's why our `CreateChildren` method has `IEnumerator` as its return type.

By the way, you can also yield another iterator. In that case this other iterator will be processed completely, so you can stitch them together in creative ways.

How do coroutines work?

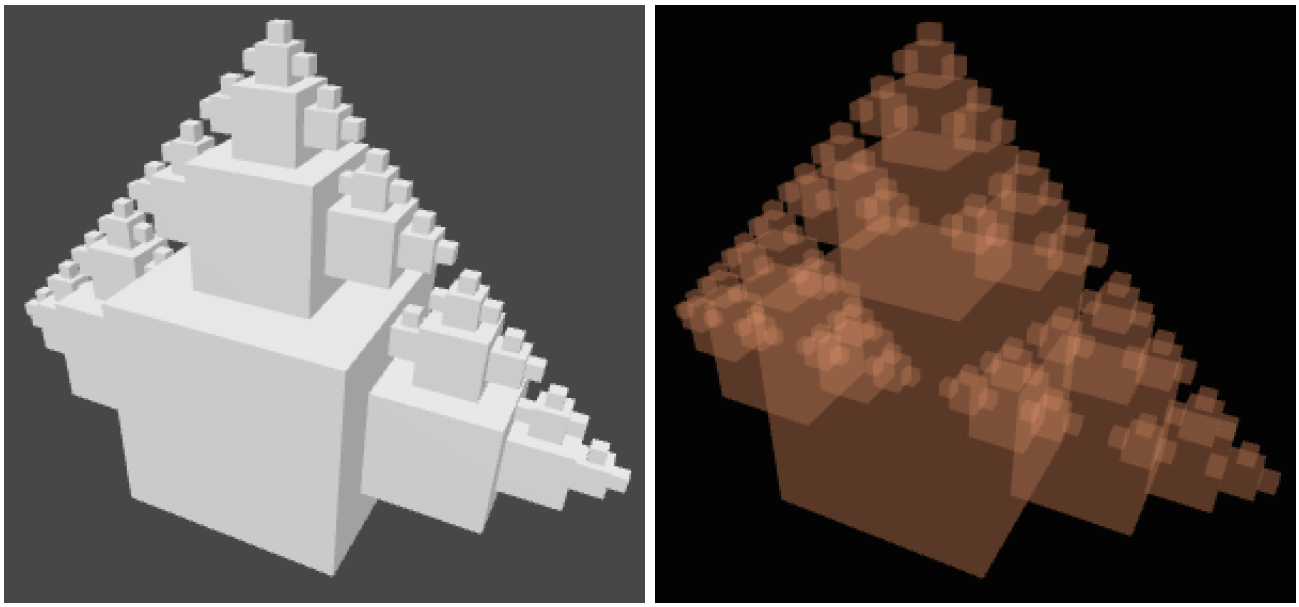
When you're creating a coroutine in Unity, what you're really doing is creating an iterator. When you pass it to the `StartCoroutine` method, it will get stored and gets asked for its next item every frame, until it is finished.

The yield statements produce the items. The statements in between – the stuff that you want to happen – are side-effects of the iterator doing its job.

You can yield special things like `WaitForSeconds` to have more control over when your own code continues, but the overall approach is simply that of an iterator.

Now we can watch it grow! Did you see a problem in the way it did? Let's add a third child per parent, this time on the left side.

```
	private IEnumerator CreateChildren () {
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").
			AddComponent<Fractal>().Initialize(this, Vector3.up);
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").
			AddComponent<Fractal>().Initialize(this, Vector3.right);
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").
			AddComponent<Fractal>().Initialize(this, Vector3.left);
	}
```



Three children per parent, normal and overdraw vision.

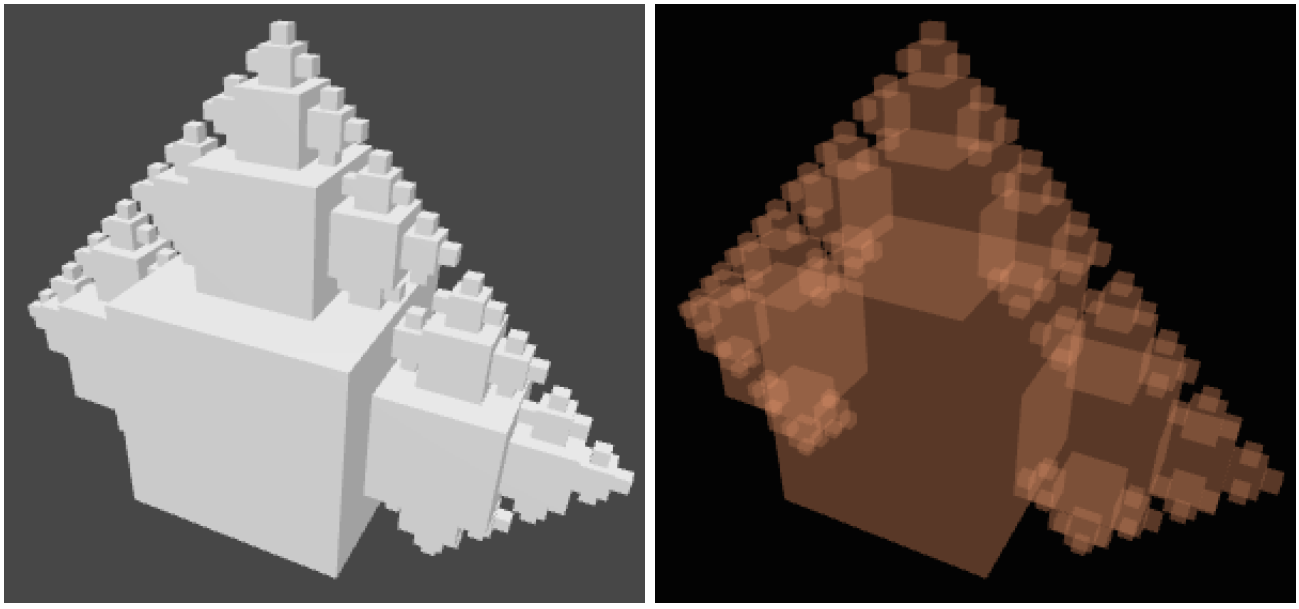
How can I use overdraw vision?

The toolbar of the scene view has a dropdown list that is set to *RGB* by default. One of its other options is *Overdraw*.

The problem is that the children have the same orientation as their parent. This means that a left child whose parent is itself a right child will find itself inside its grandparent, and vice versa. To solve this, we have to rotate the children so that their upward direction will point away from their parent.

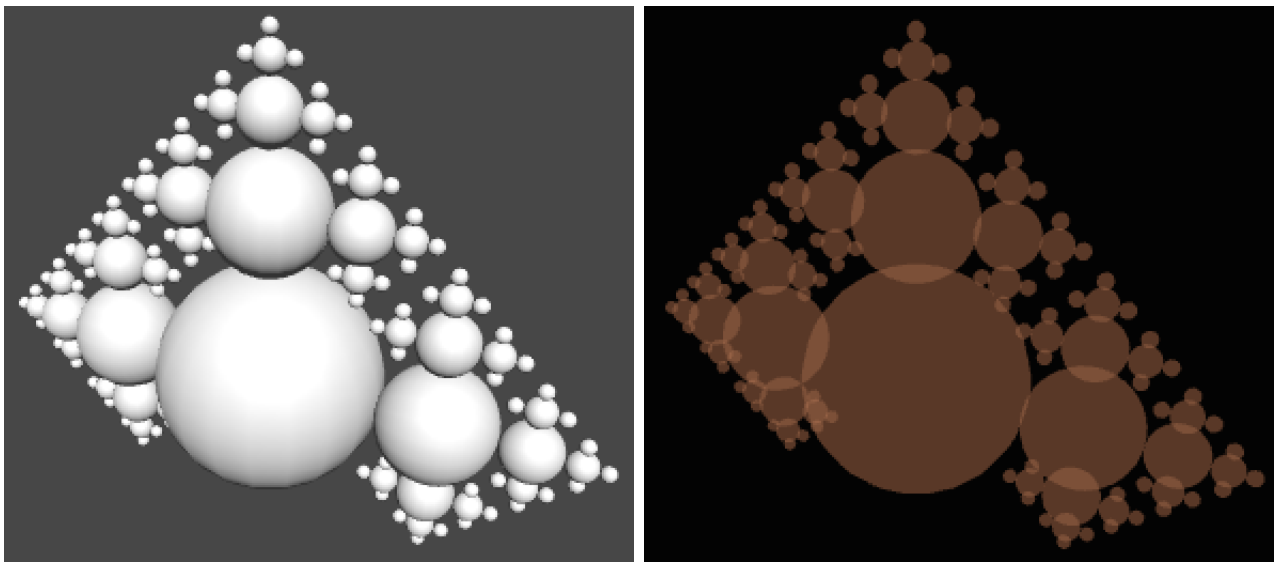
We will solve this by adding an orientation parameter to `Initialize`. It will be a quaternion used to set the local rotation of the new child. The upward child needs no rotation, the right child needs to rotate 90 degrees clockwise, and the left child needs to rotate in the opposite direction.

```
	private IEnumerator CreateChildren () {
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").AddComponent<Fractal>().
			Initialize(this, Vector3.up, Quaternion.identity);
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").AddComponent<Fractal>().
			Initialize(this, Vector3.right, Quaternion.Euler(0f, 0f, -90f));
		yield return new WaitForSeconds(0.5f);
		new GameObject("Fractal Child").AddComponent<Fractal>().
			Initialize(this, Vector3.left, Quaternion.Euler(0f, 0f, 90f));
	}

	private void Initialize (Fractal parent,
	                         Vector3 direction,
	                         Quaternion orientation) {
		…
		transform.localRotation = orientation;
	}
```



Three children per parent, now rotated.

Now that the children are rotated, they no longer immediately grow back into the fractal. But you might have noticed that some of the smallest children still end up disappearing into the root cube. This happens because with a scaling factor of 0.5, this fractal will self-intersect in four steps. You can reduce the scaling to solve this problem, or use spheres instead of cubes.



Spheres don't self-intersect with a child scale of 0.5.

6 Better Code For More Children

Our code has become a bit unwieldy. Let us generalize by moving the direction and orientation data to static arrays. Then we can reduce `CreateChildren` to a short loop and use the child index as a parameter for `Initialize`.

```
	private static Vector3[] childDirections = {
		Vector3.up,
		Vector3.right,
		Vector3.left
	};

	private static Quaternion[] childOrientations = {
		Quaternion.identity,
		Quaternion.Euler(0f, 0f, -90f),
		Quaternion.Euler(0f, 0f, 90f)
	};

	private IEnumerator CreateChildren () {
		for (int i = 0; i < childDirections.Length; i++) {
			yield return new WaitForSeconds(0.5f);
			new GameObject("Fractal Child").AddComponent<Fractal>().
				Initialize(this, i);
		}
	}

	private void Initialize (Fractal parent, int childIndex) {
		…
		transform.localPosition =
			childDirections[childIndex] * (0.5f + 0.5f * childScale);
		transform.localRotation = childOrientations[childIndex];
	}
```

How do arrays work?

Arrays are objects of fixed length that contain a linear sequence of variables. When declaring a variable, putting square brackets behind its type indicates that you want an array of that type. So `int myVariable;` gets you an integer, while `int[] myVariable;` get you an array of integers.

Accessing one of the entries inside an array is done by putting its array index – not its position – between square brackets behind the variable. So `myVariable[0]` gets you the first entry in the array, `myVariable[1]` gets you the second, and so on.

Actually creating an array and assigning it to the variable is done with `myVariable = new int[10];` which in this case creates a new array with room for ten entries. Alternatively, you can create one implicitly by listing its initial values between curly brackets, like `myVariable = {1, 2, 3};` does.

How does a `for` loop work?

A `for` loop is a compact way of writing a loop that iterates over something. In this case we use an integer named `i` as the iterator. The first part declares the iterator integer, the second part checks the loop's condition, and the third part increments the iterator.

You can use a `while` loop to get the exact same result, but it doesn't conveniently group the iterator code.

```
for(int i = 0; i < 10; i++) { DoStuff(i); }
```

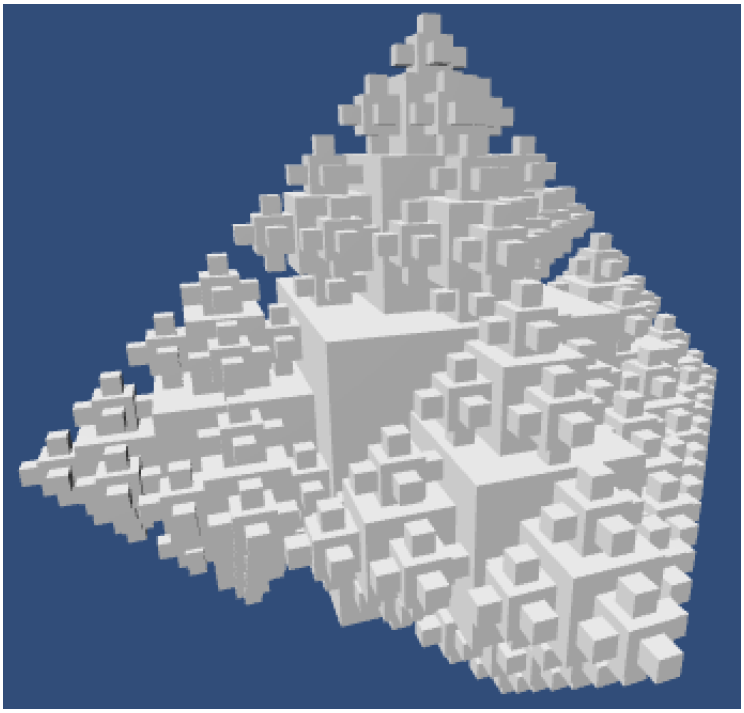
is the same as

```
int i = 0; while(i < 10) { DoStuff(i); i++; }
```

By the way, `i++` is shorthand for `i += 1`, which is shorthand for `i = i + 1`.

Now let's introduce two more children by simple adding their data to the arrays. One going forward, the other going backward.

```
    private static Vector3[] childDirections = {
        Vector3.up,
        Vector3.right,
        Vector3.left,
        Vector3.forward,
        Vector3.back
    };

    private static Quaternion[] childOrientations = {
        Quaternion.identity,
        Quaternion.Euler(0f, 0f, -90f),
        Quaternion.Euler(0f, 0f, 90f),
        Quaternion.Euler(90f, 0f, 0f),
        Quaternion.Euler(-90f, 0f, 0f)
    };
```



Five children per parent, the full fractal.

We now have the full fractal structure. But what about the underside of the root cube? The idea is that the fractal is growing out of something, like a plant. Although I won't, if you want to, you could add a special sixth child going downward that's only added to the root node. Adding it to all nodes would be a waste, because it would grow directly into its grandparent.

7 Explosive Growth

How many cubes are we actually creating? As we're always creating five children per parent, the total number of cubes when fully grown will depend on the maximum depth. A maximum depth of zero produces only one cube, the initial root. A maximum depth of one produces five additional children, for a total of six cubes. As it is fractal, this pattern repeats and we can write it as the function $f(0) = 1, f(n) = 5 \times f(n - 1) + 1$.

The above function produces the sequence 1, 6, 31, 156, 781, 3906, 19531, 97656, and so on. You will see these numbers show up as the amount of draw calls in the game view statistics of Unity. If you have dynamic batching enabled, it will be the sum of *Draw Calls* and *Saved by batching*.

Unity will probably be able to handle a maximum depth up to four or five. Any higher and your frame rate will become horrible.

Besides quantity, duration is also an issue. Right now we're pausing half a second before creating a new child. This produces synchronized bursts of growth for a few seconds. We could distribute the growth more evenly by randomizing the delays. This also results in a more unpredictable and organic pattern that I find more fun to watch.

So let's replace the fixed delay with a random range between 0.1 and 0.5. I also increased the maximum depth to 5 to make the effect more noticeable.

```
	private IEnumerator CreateChildren () {
		for (int i = 0; i < childDirections.Length; i++) {
			yield return new WaitForSeconds(Random.Range(0.1f, 0.5f));
			new GameObject("Fractal Child").AddComponent<Fractal>().
				Initialize(this, i);
		}
	}
```

How does the random range work?

`Random` is a utility class that contains some stuff to create random values. Its **`Range`** method can be used to generate a random value within some range.

There are two versions of the **`Range`** method. You can call it with two floats, in which case it returns a float between the minimum and maximum value, both inclusive.

Alternatively, you can call **`Range`** with two integers, in which case it returns an integer between the minimum, inclusive, and maximum, exclusive. The typical use case for this version is selecting an index at random, like `someArray[Random.Range(0, someArray.Length)]`.

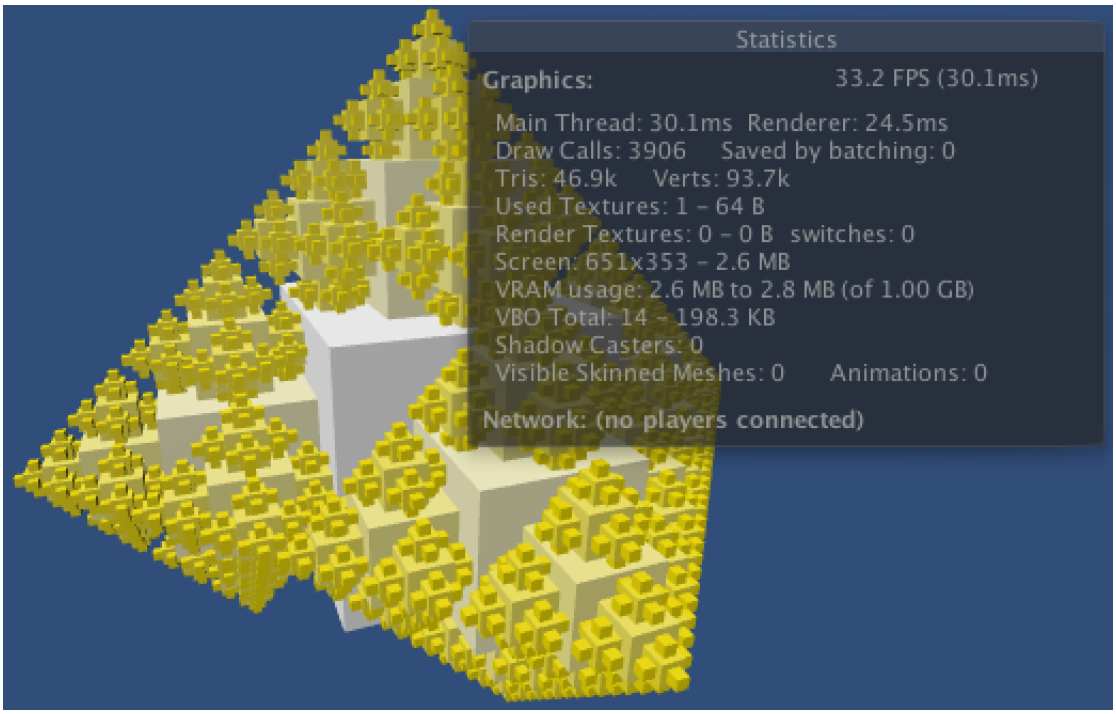
8 Adding Color

The fractal is a little dull. Let's liven it up by adding some color variation. We do this by interpolating from white at the root to yellow at the smallest children. The static `Color.Lerp` method is a handy way to do this. The interpolator goes from zero to one, which we do by dividing our current depth by the maximum depth. Because we don't want an integer division here, we convert depth to a float first.

```
	private void Start () {
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = material;
		GetComponent<MeshRenderer>().material.color =
			Color.Lerp(Color.white, Color.yellow, (float)depth / maxDepth);
		if (depth < maxDepth) {
			StartCoroutine(CreateChildren());
		}
	}
```

What does `Lerp` do?

Lerp is shorthand for linear interpolation. Its typical signature is `Lerp(a, b, t)` and it computes `a + (b - a) * t`, with `t` clamped to the 0–1 range. Multiple versions exist for various types, including floats, vectors, and colors.



Colored, but no dynamic batching.

This already looks a lot more interesting! But another thing happened as well. Dynamic batching used to work, but now it doesn't. How do we fix this?

What is dynamic batching?

Dynamic batching is a form of draw call batching performed by Unity. In short, it combines meshes that share the same material into larger meshes. Doing so reduces the amount of communication between the CPU and the GPU. You can enable or disable it via *Edit / Projects Settings / Player*, in the *Other Settings* group.

It only works for small meshes. For example, you'll find that it works with Unity's default cube, but not with the default sphere.

The problem is that by adjusting the color of a child's material we silently created a duplicate of that material. This is necessary because otherwise everything using that material would end up with the same color. However, batching only works when the exact same material is used for multiple items. Equivalence is not checked – that would be too expensive to do – it must really be the same material.

Let's create only one material duplicate per depth instead of per cube. Add a new array field to hold the materials. Then check whether an array exists in `Start` and if not call a new `InitializeMaterials` method. In this method we will explicitly duplicate our material and change its color per depth.

```
	private Material[] materials;

	private void InitializeMaterials () {
		materials = new Material[maxDepth + 1];
		for (int i = 0; i <= maxDepth; i++) {
			materials[i] = new Material(material);
			materials[i].color =
				Color.Lerp(Color.white, Color.yellow, (float)i / maxDepth);
		}
	}

	private void Start () {
		if (materials == null) {
			InitializeMaterials();
		}
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material = materials[depth];
		if (depth < maxDepth) {
			StartCoroutine(CreateChildren());
		}
	}
```

What's `null`?

The default value of a variable that's not a simple value is `null`. This means that the variable doesn't reference anything. Trying to invoke or access anything from a variable that is `null` results in an error. You can test for this value to make sure that doesn't happen. You can also set such a variable to `null` yourself, in case you no longer need whatever it was referencing.

Note that objects don't automatically cease to exist when setting a reference to them to `null`. Only when there's nobody left with a reference to them will they be become candidates for removal by the garbage collector.

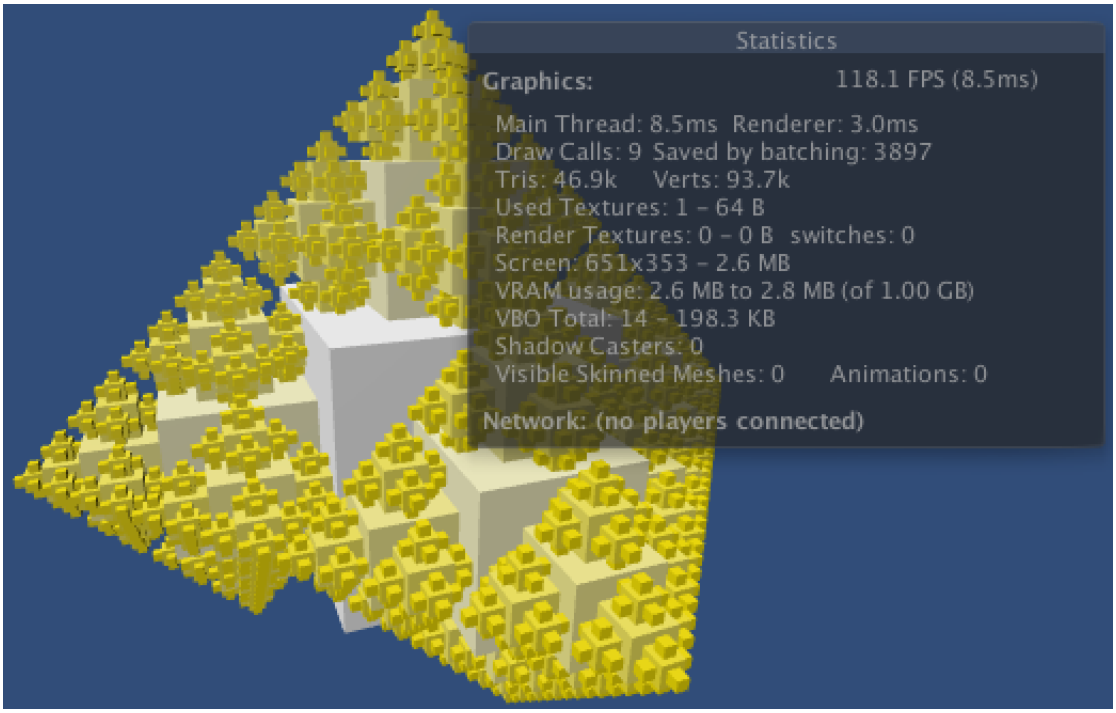
Also note that this approach works with private component fields, but not public ones. That's because Unity's serialization system would create an empty array for it, so it won't be `null`.

Now instead of passing the material reference from parent to child, pass the materials array reference instead. If we didn't, each child would be forced to create its own materials array and we wouldn't have solved the problem.

```
	private void Initialize (Fractal parent, int childIndex) {
		mesh = parent.mesh;
		materials = parent.materials;
		…
	}
```

Why not make `materials` static?

We're not making the materials array static because it depends on the maximum depth, which can be different from fractal to fractal. Why yes, you can have multiple fractals at the same time, and of course they can have different maximum depths.

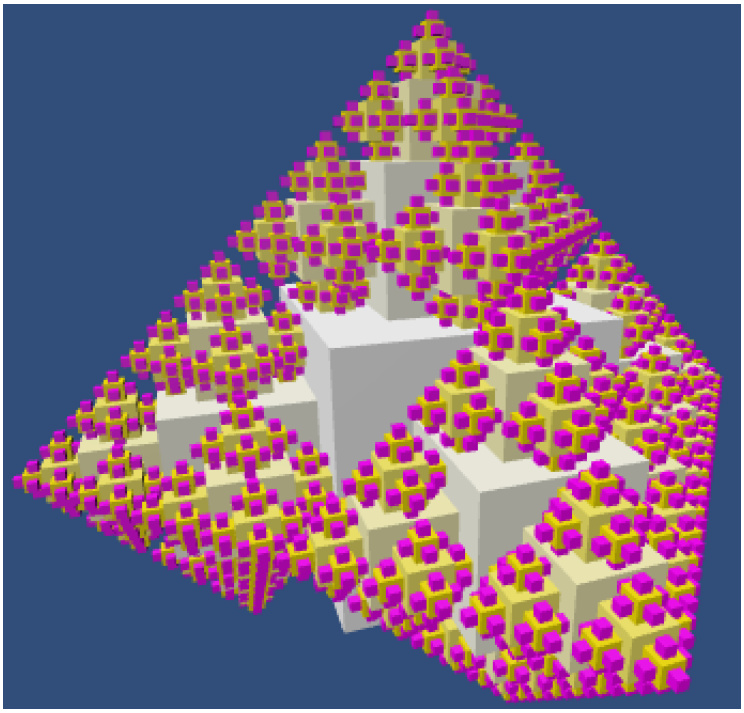


Colored, now with dynamic batching.

We have our batching back, and it makes quite a difference in this case! The color is still not that interesting though. A nice tweak is to give the deepest level a radically different color. This can reveal patterns in the fractal that you might not have noticed before.

Simply change the last color to magenta afterwards. Also adjust the interpolator so that we still see the full transition to yellow. And while we're at it, squaring it results in a slightly nicer transition.

```
	private void InitializeMaterials () {
		materials = new Material[maxDepth + 1];
		for (int i = 0; i <= maxDepth; i++) {
			float t = i / (maxDepth - 1f);
			t *= t;
			materials[i] = new Material(material);
			materials[i].color = Color.Lerp(Color.white, Color.yellow, t);
		}
		materials[maxDepth].color = Color.magenta;
	}
```



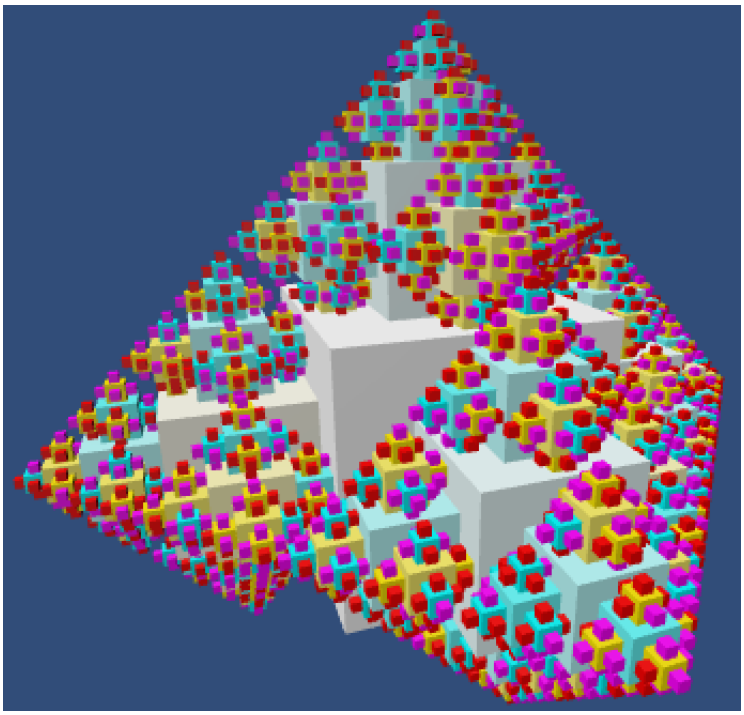
Now with magenta tips.

Let's add a second color progression, say from white to cyan with red tips. We'll use a single two-dimensional array to hold them both, then select one at random when we need a material. This way our fractal will look different each time we enter play mode. Feel free to add a third progression if you like.

How do two-dimensional arrays work?

You can add a second dimension to an array by inserting a comma inside its brackets. You then also need to provide two indexes whenever you want to access one of the array's elements. This approach extends to higher dimensions as well.

```
	private Material[,] materials;

	private void InitializeMaterials () {
		materials = new Material[maxDepth + 1, 2];
		for (int i = 0; i <= maxDepth; i++) {
			float t = i / (maxDepth - 1f);
			t *= t;
			materials[i, 0] = new Material(material);
			materials[i, 0].color = Color.Lerp(Color.white, Color.yellow, t);
			materials[i, 1] = new Material(material);
			materials[i, 1].color = Color.Lerp(Color.white, Color.cyan, t);
		}
		materials[maxDepth, 0].color = Color.magenta;
		materials[maxDepth, 1].color = Color.red;
	}

	private void Start () {
		if (materials == null) {
			InitializeMaterials();
		}
		gameObject.AddComponent<MeshFilter>().mesh = mesh;
		gameObject.AddComponent<MeshRenderer>().material =
			materials[depth, Random.Range(0, 2)];
		if (depth < maxDepth) {
			StartCoroutine(CreateChildren());
		}
	}
```



Randomized colors.

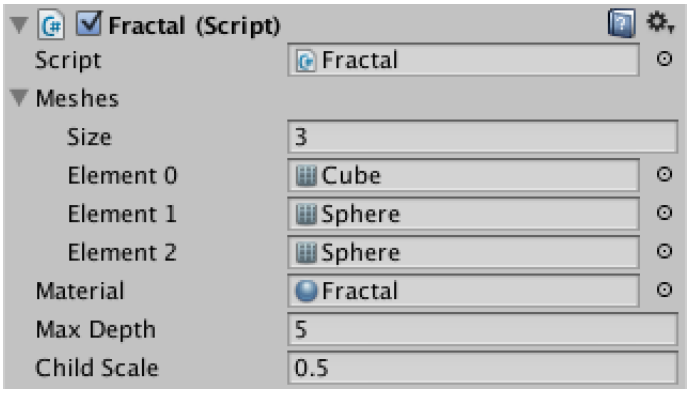
9 Randomizing Meshes

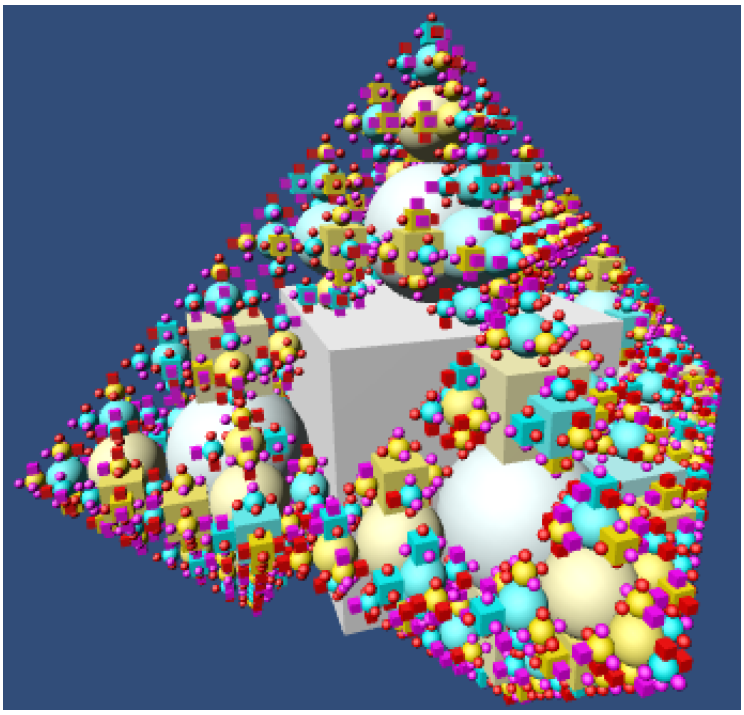
Besides colors, we also randomly choose which mesh to use. Let's replace our public mesh variable with an array and pick one out of it at random in `Start`.

```
	public Mesh[] meshes;

	private void Start () {
		if (materials == null) {
			InitializeMaterials();
		}
		gameObject.AddComponent<MeshFilter>().mesh =
			meshes[Random.Range(0, meshes.Length)];
		gameObject.AddComponent<MeshRenderer>().material =
			materials[depth, Random.Range(0, 2)];
		if (depth < maxDepth) {
			StartCoroutine(CreateChildren());
		}
	}

	private void Initialize (Fractal parent, int childIndex) {
		meshes = parent.meshes;
		…
	}
```

If we were to put only a single cube in our new array property in the inspector, then the result would be the same as before. But add a sphere and you suddenly get a 50% chance of spawning either a cube or a sphere per element of the fractal.

Fill this array however you like. I put the sphere in there twice so it is twice as likely to be used as the cube. You can also add other meshes, though capsules and cylinders don't work so well because they're elongated.





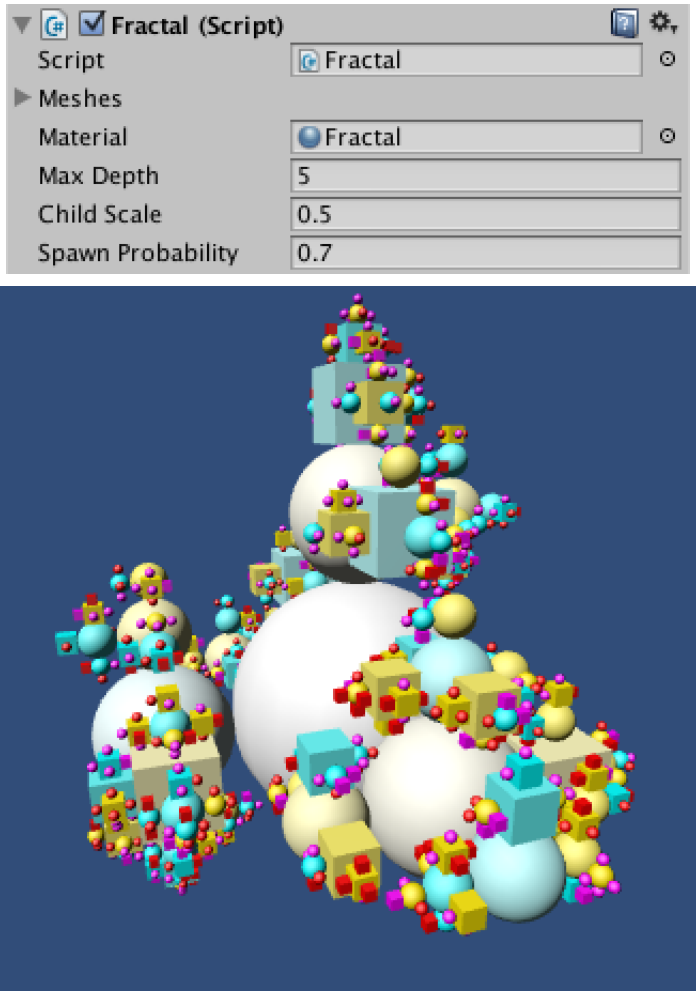
Randomly choosing between a cube and a sphere.

10 Making the Fractal Irregular

Our fractal is nice and complete, but we can make it a bit more organic by cutting off some of its branches. We do that by introducing a new public `spawnProbability` variable. We pass it along and use it to randomly determine whether we spawn a child or skip it. A probability of 0 means that no children will spawn at all, while a probability of 1 means that all children will spawn. Even values a little below one could drastically alter the shape of our fractal.

The static `Random.value` property produces a random value between zero and one. Comparing it to `spawnProbability` tells us whether we should create a new child or not.

```
	public float spawnProbability;

	private IEnumerator CreateChildren () {
		for (int i = 0; i < childDirections.Length; i++) {
			if (Random.value < spawnProbability) {
				yield return new WaitForSeconds(Random.Range(0.1f, 0.5f));
				new GameObject("Fractal Child").AddComponent<Fractal>().
					Initialize(this, i);
			}
		}
	}

	private void Initialize (Fractal parent, int childIndex) {
		…
		spawnProbability = parent.spawnProbability;
		…
	}
```



A 70% spawn chance produces irregular shapes.

11 Rotating the Fractal

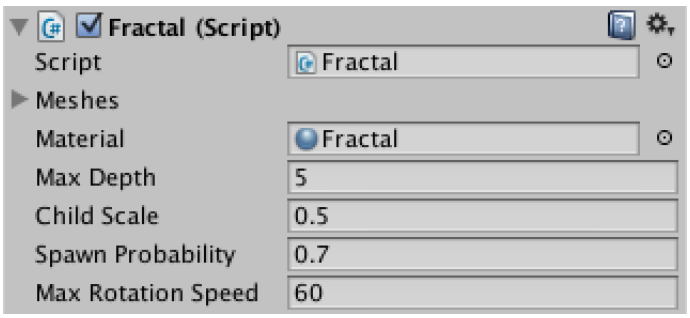
All this time our fractal has been a good boy and stayed motionless. But a little movement would make it a lot more interesting to watch. Let's add a very simple `Update` method that rotates around the current Y axis at a speed of 30 degrees per second.

```
	private void Update () {
		transform.Rotate(0f, 30f * Time.deltaTime, 0f);
	}
```

With this simple addition all the fractal's parts are now merrily spinning. All at the same speed. Yes, let's randomize it! And make the maximum speed configurable too.

Note that we have to initialize our rotation speed in `Start` - not in `Initialize` - because the root element should rotate too.

```
	public float maxRotationSpeed;

	private float rotationSpeed;

	private void Start () {
		rotationSpeed = Random.Range(-maxRotationSpeed, maxRotationSpeed);
		…
	}

	private void Initialize (Fractal parent, int childIndex) {
		…
		maxRotationSpeed = parent.maxRotationSpeed;
		…
	}

	private void Update () {
		transform.Rotate(0f, rotationSpeed * Time.deltaTime, 0f);
	}
```

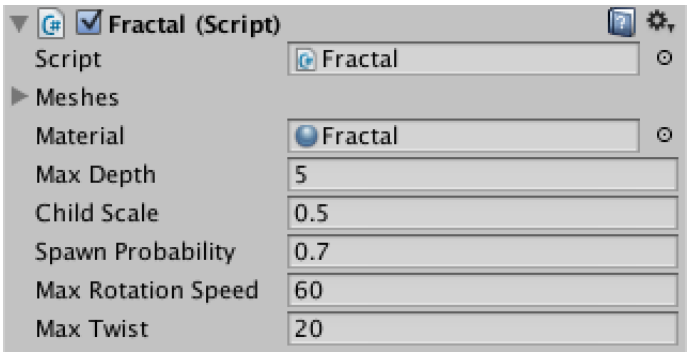


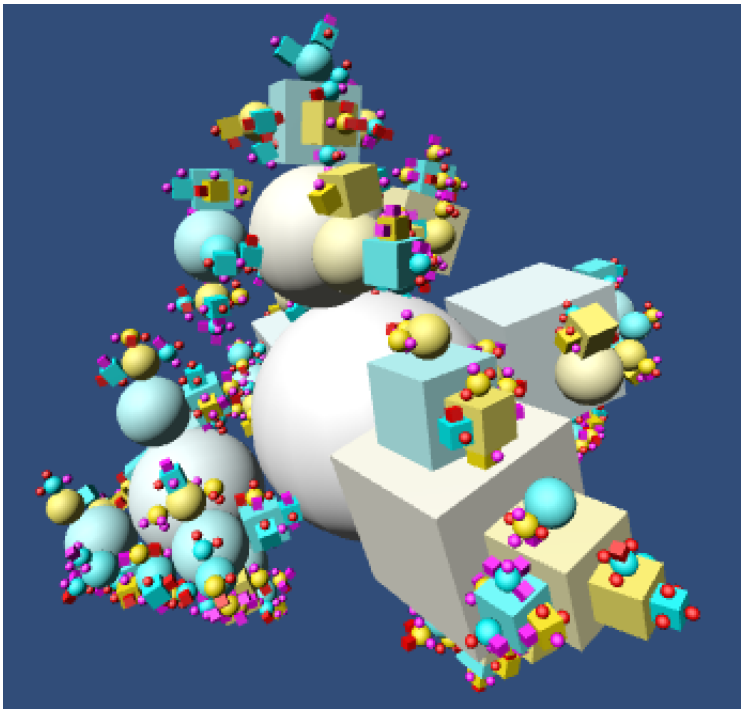
Configured for speed.

12 Adding More Chaos

Are there more adjustments that we could do to throw our fractal out of whack in subtle ways? There are many! One of them is to knock the fractal's elements out of alignment by adding a subtle rotation. Let's call it twist.

```
	public float maxTwist;

	private void Start () {
		rotationSpeed = Random.Range(-maxRotationSpeed, maxRotationSpeed);
		transform.Rotate(Random.Range(-maxTwist, maxTwist), 0f, 0f);
		…
	}

	private void Initialize (Fractal parent, int childIndex) {
		…
		maxTwist = parent.maxTwist;
		…
	}
```





Some nice twist.

Another option would be to mess with the child scale. Or maybe skip a depth sometimes. Fiddle with positions? From this point I leave it up to you!

Enjoying the tutorials? Are they useful?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick