Catlike Coding

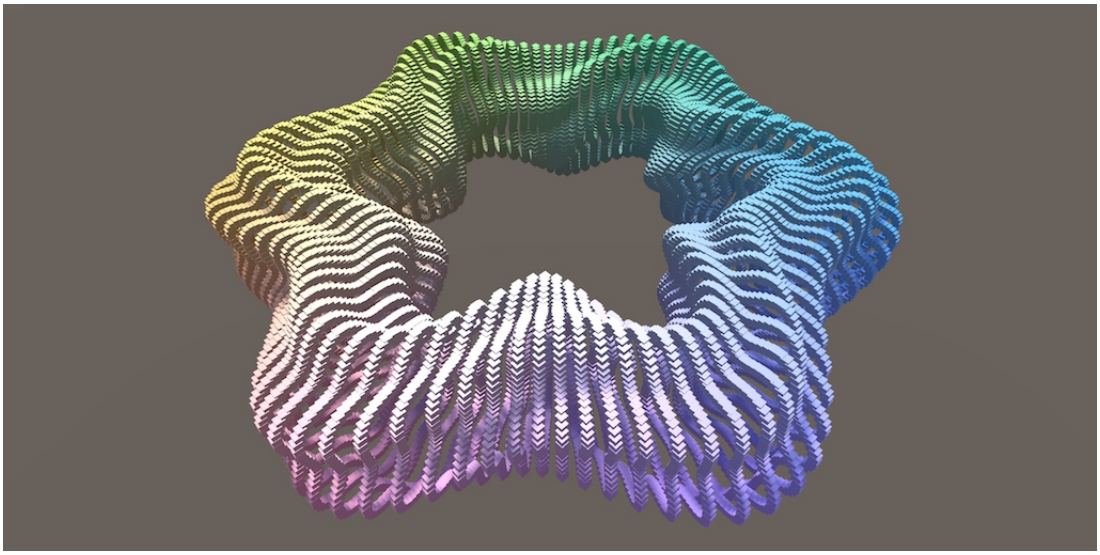
Unity C# Tutorials

Mathematical Surfaces Sculpting with Numbers

Support multiple function methods.
Use a delegate and enumeration.
Display 2D functions with a grid.
Define surfaces in 3D space.

This tutorial is a continuation of Building a Graph. We'll make it possible to display multiple and more complex functions.

This tutorial assumes that you're using at least Unity 2017.1.0.



Combining a few waves to create complex shapes.

1 Switching Between Functions

After finishing the previous tutorial, we have a line graph that shows an animated sine wave while in play mode. It is also possible to show other mathematical functions. You can change the code and the function will change along with it. You can even do this while the Unity editor is in play mode. Execution will be paused, the current game state saved, then the scripts are compiled again, and finally the game state is reloaded and play resumes. Not everything survives a recompile while in play mode, but our graph does. It will switch to animating the new function, without being aware that something changed.

While changing code during play mode can be convenient, it is not a handy way to switch back and forth between multiple functions. It would be much better if we could simply change a configuration option of the graph to do this. Let's make that possible.

1.1 Function Method

To have our graph support multiple functions at the same time, we have to program all functions into it. However, the code that loops through the graph's points doesn't care which function is used. We don't need to repeat that code for each individual function. Instead, we'll extract the mathematical function part and put it in its own method.

Add a new method to `Graph` to contain the code for our sine function. This works just like creating the `Awake` and `Update` methods, except we'll name this one `SineFunction`.

```
	void SineFunction () {}
```

This function is going to represent our mathematical function $f(x, t) = \sin(\pi(x + t))$. To do so, it has to produce an output, which is a floating-point number. So instead of `void`, the type of the function has to be `float`.

```
	float SineFunction () {}
```

The function also needs parameters. It currently has an empty parameter list. To add the x parameter, put it in between the parentheses after the method name. Just like for the method itself, its parameters much have their type written before them as well. As it are all floating-point numbers, we again have to use `float`.

```
	float SineFunction (float x) {}
```

Add the t parameter as well, also with its type. The parameter declarations have to separated with a comma.

```
	float SineFunction (float x, float t) {}
```

Now we can put the code that computes the function inside the method, using its `x` and `t` parameters.

```
	float SineFunction (float x, float t) {
		Mathf.Sin(Mathf.PI * (x + t));
	}
```

The last step is to explicitly indicate what the result of the method is. As this is a `float` method, it has to return a `float` when it's done. We indicate that by writing `return` followed by what the result is supposed to be, which is our mathematical computation.

```
	float SineFunction (float x, float t) {
		return Mathf.Sin(Mathf.PI * (x + t));
	}
```

It is now possible to invoke this method inside `Update`, using `position.x` and `Time.time` as arguments for its parameters. Its result can by used to set the point's Y coordinate, instead of using explicit math.

```
	void Update () {
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
//			position.y = Mathf.Sin(Mathf.PI * (position.x + Time.time));
			position.y = SineFunction(position.x, Time.time);
			point.localPosition = position;
		}
	}
```

Note that `Time.time` is the same each time we invoke that property inside the loop. We can make do with retrieving its value only once, before the loop, storing it in a variable.

```
	void Update () {
		float t = Time.time;
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
			position.y = SineFunction(position.x, t);
			point.localPosition = position;
		}
	}
```

1.2 A Second Function

Now that we have a function method, let's make another one. This time we'll make a slightly more complex function, using more than one sine. Begin by duplicating the `SineFunction` method and rename the new one to `MultiSineFunction`.

```
	float SineFunction (float x, float t) {
		return Mathf.Sin(Mathf.PI * (x + t));
	}

	float MultiSineFunction (float x, float t) {
		return Mathf.Sin(Mathf.PI * (x + t));
	}
```

We'll keep the sine function that we already have, but add something extra to it. To make that easy, assign the current result to an `y` variable before returning it.

```
	float MultiSineFunction (float x, float t) {
		float y = Mathf.Sin(Mathf.PI * (x + t));
		return y;
	}
```

The simplest way to add more complexity to a sine wave it to add another one that has double the frequence. This means that it changes twice as fast, which is done by multiplying the argument of the sine function by 2. At the same time, we'll halve the result of this function. That keeps the shape of the sine wave the same, just at half size.

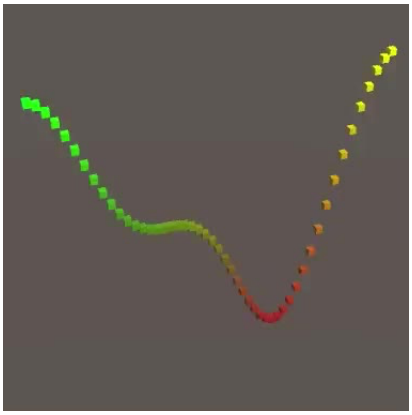
```
		float y = Mathf.Sin(Mathf.PI * (x + t));
		y += Mathf.Sin(2f * Mathf.PI * (x + t)) / 2f;
		return y;
```

This gives us the mathematical function $f(x,t) = \sin(\pi(x+t)) + \frac{\sin(2\pi(x+t))}{2}$. As both the positive and negative extremes of the sine function are 1 and −1, the maximum and minimum values of this new function will be 1.5 and −1.5. To guarantee that we stay in the −1–1 range, we should divide the entire thing by 1.5, which is the same as multiplying by $\frac{2}{3}$.

```
		float y = Mathf.Sin(Mathf.PI * (x + t));
		y += Mathf.Sin(2f * Mathf.PI * (x + t)) / 2f;
		y *= 2f / 3f;
		return y;
```

Let's use this function instead of `SineFunction` in **`Update`** and see what it looks like.

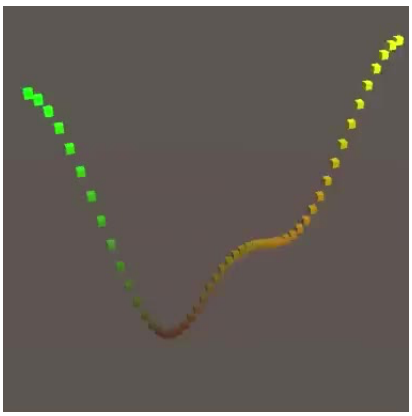
```
			position.y = MultiSineFunction(position.x, t);
```



Multi Sine.

You could say that a smaller sine wave is now following a larger sine wave. We could even make the smaller one slide along the larger one, for example by doubling its time factor. The result will be a function that doesn't just slide as time progresses, it changes its shape. Because the sine waves repeat, it will loop back to the same shape every two seconds.

```
	float MultiSineFunction (float x, float t) {
		float y = Mathf.Sin(Mathf.PI * (x + t));
		y += Mathf.Sin(2f * Mathf.PI * (x + 2f * t)) / 2f;
		y *= 2f / 3f;
		return y;
	}
```

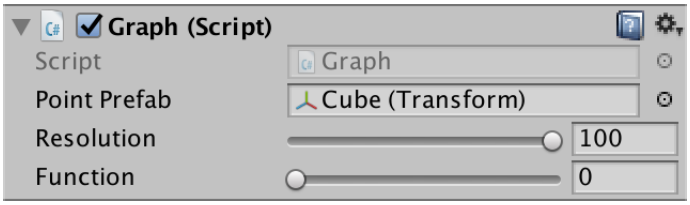


Morphing Multi Sine.

1.3 Selecing Functions in the Editor

The next thing we can do is add some code that makes it possible to control which method is used by the graph. We could do this with a slider, just like for the graph's resolution. As we have two function to choose from, we'll need a public integer field with a range of 0–1. Name it `function` so it's obvious what it controls.

```
	[Range(0, 1)]
	public int function;
```



Function slider.

We can use an `if-else` block pair inside `Update` to control which function is invoked. If the slider is set to 0, we'll use `SineFunction`. Otherwise, we'll use `MultiSineFunction`.

```
	void Update () {
		float t = Time.time;
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
			if (function == 0) {
				position.y = SineFunction(position.x, t);
			}
			else {
				position.y = MultiSineFunction(position.x, t);
			}
			point.localPosition = position;
		}
	}
```

This makes it possible to control the function via the graph's inspector, also while we're in play mode.

1.4 Static Methods

Although the `SineFunction` and `MultiSineFunction` are part of `Graph`, they are effectively self-contained. They only rely on their parameters and math to do their job. They do rely on `Mathf`, but we can see that simply as math. Besides that, they don't need to access any other methods or fields of `Graph`. This suggests that we could put them in another class or struct and they'd still work. So we could create a separate class for function methods and put them all in there. However, because `Graph` is the only one using these methods, there's not much of a reason to do that.

By default, methods and field are associated with actual object or value instances of a class or struct type. But this need not be the case. We can indicate that this association doesn't exist. That's done by putting the `static` keyword in front of the method or field definition. Let's do that for our two methods.

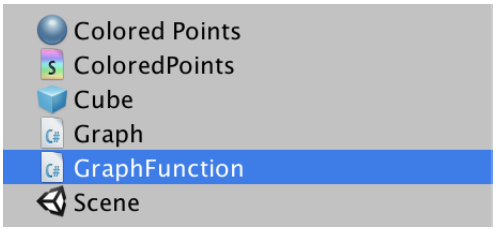
```
	static float SineFunction (float x, float t) {
		return Mathf.Sin(Mathf.PI * (x + t));
	}

	static float MultiSineFunction (float x, float t) {
		float y = Mathf.Sin(Mathf.PI * (x + t));
		y += Mathf.Sin(2f * Mathf.PI * (x + 2f * t)) / 2f;
		y *= 2f / 3f;
		return y;
	}
```

The methods are still part of `Graph`, but they're now directly associated with the class type and no longer bound to object instances. Had we made them public, we could've invoked them from anywhere, like `Graph.SineFunction(0f, 0f)`, just like `Mathf.Sin(0f)`. Inside the `Graph` class itself, we don't have to explicitly add the type prefix, so our existing code still works.

> **What's the point of making our methods static?**
>
> Right now there isn't really a point, but we'll make use of it shortly. Currently it's just an indication that the methods are self-contained.
>
> Because static methods aren't associated with object instances, the compiled code doesn't have to keep track of which object you're invoking the method on. This means that static method invocations are a bit faster, but it's usually not significant enough to worry about.

1.5 Delegates

A simple `if-else` block works for two functions, but it gets unwieldy fast when trying to support more functions. It would be much more convenient if we could use a variable to store a reference to the method that we want to invoke. This is possible, by using a delegate type. A delegate is a special type that defines what kind of method something can reference. There isn't a standard type for our mathematical function methods, but we can define it ourselves. To do so, create a new C# script asset and name it `GraphFunction`.



GraphFunction script asset.

> **Do we have to use a new script?**
>
> It's actually possible to define the delegate type inside `Graph`, but putting each type in its own script makes it explicit that they are separate things. In bigger projects, small types that are only used in the context of another type are often nested inside those types.

Get rid of the default code in this script. Instead, we'll use the `UnityEngine` namespace and then define a public delegate type named `GraphFunction`. This isn't the same as a class or struct definition, it has to be followed by a semicolon.

```
using UnityEngine;

public delegate GraphFunction;
```

The delegate type defines the form of the methods that it can be used for. This form is a method's signature, which is defined by its return type and parameter list. In our case, the return type of the methods is `float` and there are two parameters, both `float` as well. Apply this signature to `GraphFunction` delegate type. The actual names used for the parameters doesn't matter, but their types must be correct.

```
public delegate float GraphFunction (float x, float t);
```

Now we can declare a `GraphFunction` variable inside `Graph.Update`, before the loop. After that, it's possible to invoke this variable as if it were a method. That allows us to get rid of the `if-else` code inside the loop.

```
	void Update () {
		float t = Time.time;
		GraphFunction f;
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
//			if (function == 0) {
//				position.y = SineFunction(position.x, t);
//			}
//			else {
//				position.y = MultiSineFunction(position.x, t);
//			}
			position.y = f(position.x, t);
			point.localPosition = position;
		}
	}
```

Instead, we now have to put an `if-else` block before the loop, assigning a reference to the appropriate method to our variable.

```
		GraphFunction f;
		if (function == 0) {
			f = SineFunction;
		}
		else {
			f = MultiSineFunction;
		}
		for (int i = 0; i < points.Length; i++) {
			…
		}
```

1.6 An Array of Delegates

Although we've moved the `if-else` block out of the loop, we still haven't eliminated it. We can get rid of it completely by replacing it with indexing an array. Now that we have a `GraphFunction` type, we can add a `functions` array field of this type to `Graph`.

```
	GraphFunction[] functions;
```

We're always going to put the same elements in this array, so we can explicitly define its contents as part of its declaration. This is done by assigning an array element sequence, between curly brackets. The simplest is an empty sequence.

```
	GraphFunction[] functions = {};
```

This means that we immediately get an array instance, but it is empty. Change this so it will contain reference both function methods, first `SineFunction`, followed by `MultiSineFunction`.

```
	GraphFunction[] functions = {
		SineFunction, MultiSineFunction
	};
```

Because this array is always the same, there's no point to create one per graph instance. Instead, let's define it once for the `Graph` type itself, making it static like our function methods.

```
	static GraphFunction[] functions = {
		SineFunction, MultiSineFunction
	};
```

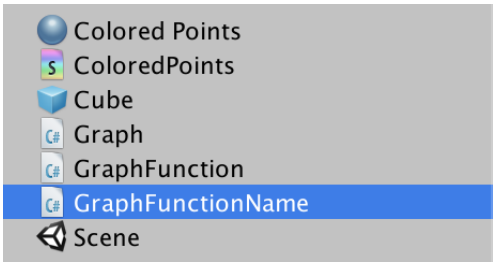
Next, use the array in `Update`, using the `function` instance field to index it. After that, we can finally remove the `if-else` code.

```
		GraphFunction f = functions[function];
//		if (function == 0) {
//			f = SineFunction;
//		}
//		else{
//			f = MultiSineFunction;
//		}
```

1.7 Enumerations

An integer slider works, but it is not obvious that 0 represents the sine function and 1 represents the multi-sine function. It would be clearer if we had a dropdown list containing meaningful names. We can use an enumeration to achieve this.

Enumerations can be created by defining an `enum` type. Create a new C# script asset to contain this type, named `GraphFunctionName`.



GraphFunctionName script asset.

The minimal definition of an enumeration works the same as a class, except that `enum` is used instead of `class`.

```
public enum GraphFunctionName {}
```

The block after the enumeration's name contains a comma-separated list of labels. These are strings that follow the same rules and conventions as type names. As names for our functions, use `Sine` and `MultiSine`.

```
public enum GraphFunctionName {
	Sine,
	MultiSine
}
```

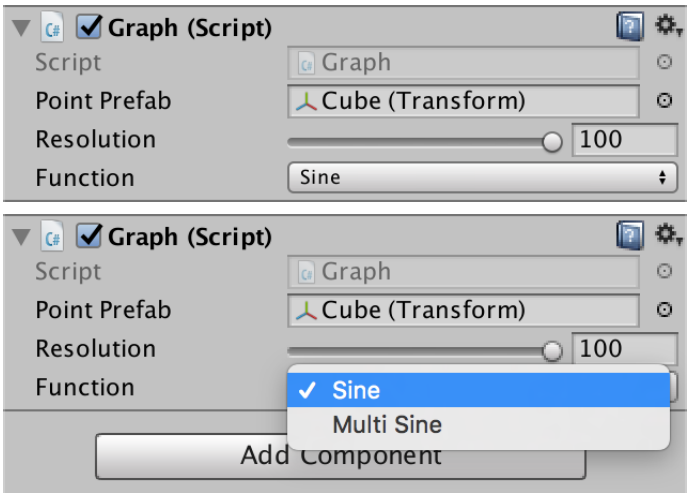
Next, replace the `function` integer field of `Graph` with another `function` field that has the new `GraphFunctionName` type.

```
//	[Range(0, 1)]
//	public int function;
	public GraphFunctionName function;
```

Enumerations can be considered syntactical sugar. By default, each label of the enumeration represents an integer. The first label corresponds to 0, the second label to 1, and so on. So we can keep using the enumeration field to index our array. However, the compiler will complain that an enumeration cannot be implicitly cast to an integer. We have to explicitly perform this cast when using it as an index in `Update`.

```
		GraphFunction f = functions[(int)function];
```

We're now using an enumeration to select which function to use. When the inspector displays an enumeration, it will create a dropdown list containing all labels of that enumeration type. This makes it clear which function we select, as long as we make sure that the labels of `GraphFunctionName` and the contents of `Graph.functions` match.



Function dropdown list.

2 Adding Another Dimension

So far we've worked with traditional line graphs. They map 1-dimensional values to other 1D values, though if you take time into account it's actually mapping 2D values to 1D values. So we're already mapping higher-dimensional input to a 1D value. Like we added time, we can add additional spatial dimensions too.

Currently, we're using the X dimension as the spatial input for our functions. The Y dimension is used to display the output. That leaves Z as a second spatial dimension to use for input. To visualize it, upgrade our shader so it uses the Z coordinate to set the blue color channel. This can be done by replacing the usage of `rg` and `xy` with `rgb` and `xyz` when calculating the albedo.

```
			o.Albedo.rgb = IN.worldPos.xyz * 0.5 + 0.5;
```

2.1 Adjusting the Functions

To support a second non-time input for our function, add a `z` parameter after the `x` parameter of the `GraphFunction` delegate type.

```
public delegate float GraphFunction (float x, float z, float t);
```

This requires us to also add the parameter to our two function methods in `Graph`, even though they don't actually use the extra dimension.

```
	static float SineFunction (float x, float z, float t) {
		return Mathf.Sin(Mathf.PI * (x + t));
	}

	static float MultiSineFunction (float x, float z, float t) {
		float y = Mathf.Sin(Mathf.PI * (x + t));
		y += Mathf.Sin(2f * Mathf.PI * (x + 2f * t)) / 2f;
		y *= 2f / 3f;
		return y;
	}
```

To make this work, we have to provide the position's Z coordinate as the second argument when invoking the function method in `Update`.

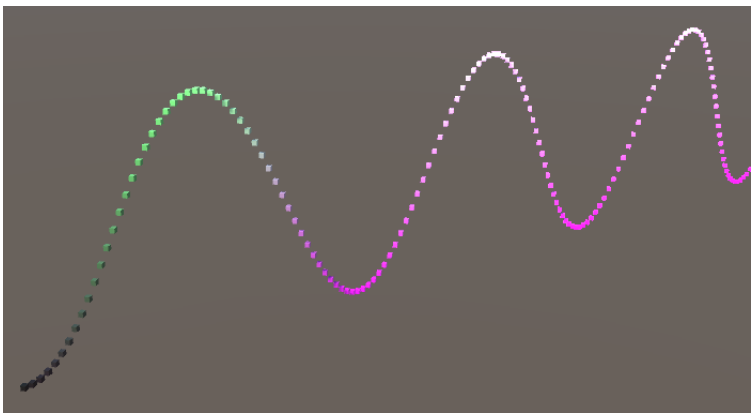
```
			position.y = f(position.x, position.z, t);
```

2.2 Creating a Grid of Points

To show the Z dimension, we have to turn our line of points into a grid of points. We can do this by creating multiple lines, each offset one step along Z. We'll use the same range for Z as we use for X, so we'll create as many lines as we currently have points. This means that we have to square the amount of points. Adjust the creation of the `points` array in `Awake` so it's big enough to contain all the points.

```
		points = new Transform[resolution * resolution];
```

As we increase the X coordinate each iteration based on the resolution, simply creating more points will result in a single long line. We have to adjust the initialization loop to take the second dimension into account.



A line that is too long.

First, let's keep track of the X coordinate explicitly. Do this by declaring and incrementing an `x` variable inside the `for` loop, just like the `i` iterator variable. The declaration and increment sections of the loop definition can be turned into comma-separated lists for this purpose.

```
		for (int i = 0, x = 0; i < points.Length; i++, x++) {
			Transform point = Instantiate(pointPrefab);
			position.x = (i + 0.5f) * step - 1f;
			point.localPosition = position;
			point.localScale = scale;
			point.SetParent(transform, false);
			points[i] = point;
		}
```

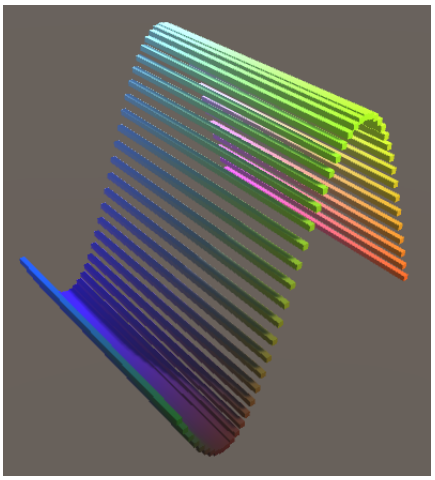
Each time we finish a row, we have to reset `x` back to zero. A row is finished when `x` has become equal to `resolution`, so we can use an `if` block at the top of the loop to take care of this. Then use `x` instead of `i` to calculate the X coordinate.

```
		for (int i = 0, x = 0; i < points.Length; i++, x++) {
			if (x == resolution) {
				x = 0;
			}
			Transform point = Instantiate(pointPrefab);
			position.x = (x + 0.5f) * step - 1f;
			point.localPosition = position;
			point.localScale = scale;
			point.SetParent(transform, false);
			points[i] = point;
		}
```

Next, each row has to be offset along the Z dimension. This can be done by adding a `z` variable to the `for` loop as well. This variable must not be incremented each iteration. Instead, it only increments when we move on to the next row, for which we already have an `if` block. Then set the position's Z coordinate just like its X coordinate, using `z` instead of `x`.

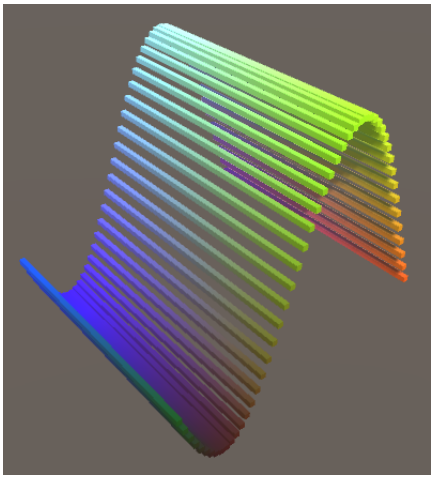
```
		for (int i = 0, x = 0, z = 0; i < points.Length; i++, x++) {
			if (x == resolution) {
				x = 0;
				z += 1;
			}
			Transform point = Instantiate(pointPrefab);
			position.x = (x + 0.5f) * step - 1f;
			position.z = (z + 0.5f) * step - 1f;
			point.localPosition = position;
			point.localScale = scale;
			point.SetParent(transform, false);
			points[i] = point;
		}
```

We now create a square grid of points instead of a single line. Because our functions still only rely on the X dimension, it will look like the original points have been extruded into lines.



Sine on a grid.

Because there are now a lot of points placed in a small space, it's likely that points will cast shadows on each other. The Y rotation of the default directional light is set to −30°, which results in a lot of visible shadows when looking at the graph in positive direction. To better see the colors, you can rotate the light to get more pleasing shadows, like using a positive Y rotation of 30°, or simply disable shadows.



Light with Y rotation of 30°.

Why has the frame rate dropped a lot?

Compare to the previous line, the grid contains a lot more points. A resolution 50, it has 2,500 points. At resolution 100, it has 10,000 points. Your computer should be able to deal with that just fine, but it might struggle when both the game and scene window are visible at the same time, because that can double the amount of work. This is especially true when you have the graph selected and the scene view draws an outline around all the cubes.

2.3 Double Looping

Although our current approach to create a grid layout works, the usage of the `if` block is awkward. A more readable way to loop over two dimensions is to use a separate loop per dimension. To do this, remove the old `for` loop declaration and `if` block, replacing them with a `for` loop that loops over Z. Inside that loop, create another loop that does the same for X. The points are created inside that second nested loop. The effect of this that we loop over X multiple times, increasing Z after each line, just like before.

The `i` iterator variable is no longer needed to end the loop, but it's still needed to index the `points` array. Define it in the outer loop, but increment it in the inner loop. That way it is known throughout the entire process and gets incremented for each point.

```
//		for (int i = 0, x = 0, z = 0; i < points.Length; i++, x++) {
//			if (x == resolution) {
//				x = 0;
//				z += 1;
//			}
		for (int i = 0, z = 0; z < resolution; z++) {
			for (int x = 0; x < resolution; x++, i++) {
				Transform point = Instantiate(pointPrefab);
				position.x = (x + 0.5f) * step - 1f;
				position.z = (z + 0.5f) * step - 1f;
				point.localPosition = position;
				point.localScale = scale;
				point.SetParent(transform, false);
				points[i] = point;
			}
		}
```

Note that the Z coordinate only changes per iteration of the outer loop. This means that we don't have to compute it inside of the inner loop. We can hoist it up one level and eliminate duplicate work.

```
		for (int i = 0, z = 0; z < resolution; z++) {
			position.z = (z + 0.5f) * step - 1f;
			for (int x = 0; x < resolution; x++, i++) {
				Transform point = Instantiate(pointPrefab);
				position.x = (x + 0.5f) * step - 1f;
//				position.z = (z + 0.5f) * step - 1f;
				point.localPosition = position;
				point.localScale = scale;
				point.SetParent(transform, false);
				points[i] = point;
			}
		}
```

Does it matter which dimension is used for the outer loop?

I've used Z for the outer loop and X for the inner loop. This is equivalent to our earlier approach. It means that the grid is constructed by creating rows of points along X, and the rows are offset along Z. You could also do it the other way around, using X for the outer loop and Z for the inner loop. In that case the grid is constructed by creating rows of points along Z, offset along X. Only the order in which the points have been created differs, everything else is the same.

2.4 Incorporating Z

We have a 2D grid of input points, so let's make use of that new second dimension. But first, let's define a local constant for π, so we don't have to write `Mathf.PI` all the time. That's simply convenient because we'll be using it more often.

```
	const float pi = Mathf.PI;

	static float SineFunction (float x, float z, float t) {
		return Mathf.Sin(pi * (x + t));
	}

	static float MultiSineFunction (float x, float z, float t) {
		float y = Mathf.Sin(pi * (x + t));
		y += Mathf.Sin(2f * pi * (x + 2f * t)) / 2f;
		y *= 2f / 3f;
		return y;
	}
```

Instead of adjusting the existing two functions, we're going to create a new function that uses both X and Z as input. Create a method for it, named `Sine2DFunction`. Have it represent the function $f(x, z, t) = \sin(\pi(x + z + t))$, which is the most straightforward way to make a sine wave based on both x and z.

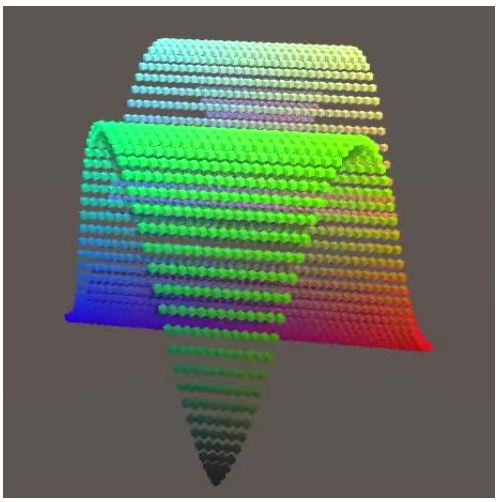
```
	static float Sine2DFunction (float x, float z, float t) {
		return Mathf.Sin(pi * (x + z + t));
	}
```

Add this method to the `functions` array, placing it directly after `SineFunction`.

```
	static GraphFunction[] functions = {
		SineFunction, Sine2DFunction, MultiSineFunction
	};
```

Add a name for it to `GraphFunctionName` too, using `Sine2D`.

```
public enum GraphFunctionName {
	Sine, Sine2D, MultiSine
}
```

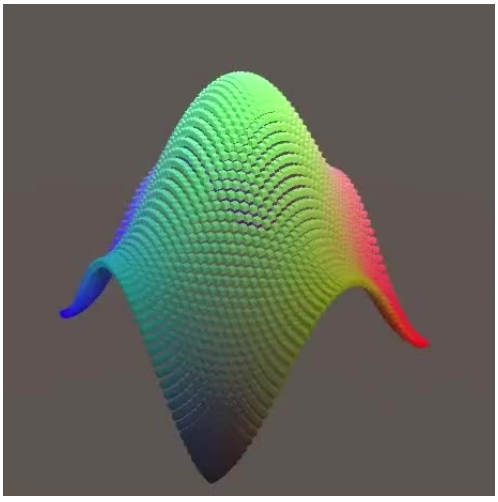


A diagonal sine wave.

When using this function in play mode, you'll see the familiar sine wave, except it's oriented along the XZ diagonal instead of straight along X. That's because we're using $x + z$ instead of just x as input for the sine function.

An alternative and more interesting way to use both dimensions is to combine two independent sine waves, one for each dimension. Simply add them together, then halve the result so the output stays inside the −1–1 range. This gives us the function $f(x, z, t) = \frac{\sin(\pi(x + t)) + \sin(\pi(z + t))}{2}$. To make the code easy to read, we'll use an `y` variable and split it into three lines.

```
	static float Sine2DFunction (float x, float z, float t) {
//		return Mathf.Sin(pi * (x + z + t));
		float y = Mathf.Sin(pi * (x + t));
		y += Mathf.Sin(pi * (z + t));
		y *= 0.5f;
		return y;
	}
```



One sine per dimension.

Why use `*= 0.5f` instead of `/= 2f`?

Both approaches are mathematically equivalent, but multiplication instructions are quicker than division instructions. If you're performing a lot of calculations inside loops, it's a simple optimization to make. It's not necessary for this tutorial, but it's a fine habit. You can go ahead and replace the division in `MultiSineFunction` too.

Let's create a 2D variant for the multi-sine function as well. In this case, we'll again use a single main wave but with two secondary waves, one per dimension, so we get a function of the form $f(x, z, t) = M + S_x + S_z$, where M stands for the main wave, S_x represents the secondary wave based on x, and S_z is the secondary wave based on z.

We'll use $M = \sin\left(\pi\left(x + z + \frac{t}{2}\right)\right)$, so the main wave is a slow-moving diagonal wave. The first secondary wave is $S_x = \sin(\pi(x + t))$, so it's a normal wave along X. And the third wave is $S_z = \sin(2\pi(z + 2t))$, which is double-frequency and fast-moving along Z.

We'll make the main wave M big, four times the amplitude of S_x. Because S_z has double the frequency and speed of the other secondary wave, we'll give it half the amplitude. This leads to the function $f(x, z, t) = 4M + S_x + \frac{S_z}{2}$, which has to be divided by 5.5 to normalize it to the −1–1 range. Create a `MultiSine2DFunction` method for this.

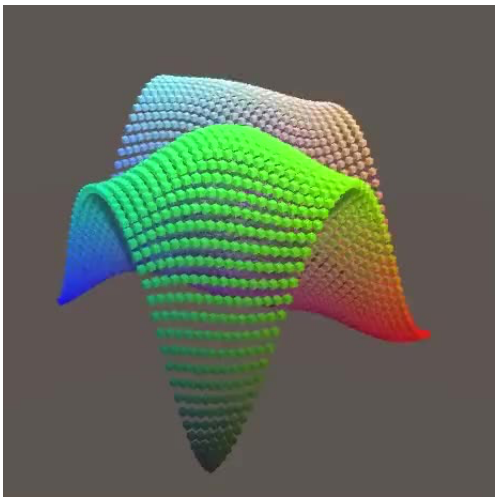
```
	static float MultiSine2DFunction (float x, float z, float t) {
		float y = 4f * Mathf.Sin(pi * (x + z + t * 0.5f));
		y += Mathf.Sin(pi * (x + t));
		y += Mathf.Sin(2f * pi * (z + 2f * t)) * 0.5f;
		y *= 1f / 5.5f;
		return y;
	}
```

Add it to the `functions` array.

```
	static GraphFunction[] functions = {
		SineFunction, Sine2DFunction, MultiSineFunction, MultiSine2DFunction
	};
```

And give it the `MultiSine2D` name.

```
public enum GraphFunctionName {
	Sine, Sine2D, MultiSine, MultiSine2D
}
```

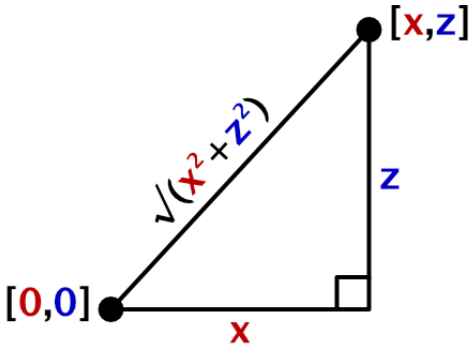


2D multi sine, combining three waves.

2.5 Creating a Ripple

We're going to create one more 2D function, this time it's one that represent an animated ripple on a surface. We'll let the ripple spread in all directions so we get a circular pattern. To do this, we have to create a sine wave based on the distance from the origin. This distance can be found using the Pythagorean theorem, which states that $a^2 + b^2 = c^2$ where c is the length of the hypotenuse of a right triangle and a and b are the lengths of its other two sides.

In the case of 2D points in the XZ plane, the hypotenuse of such a triangle corresponds to the line between the origin and that point, with it's X and Z coordinates as the lengths of the other two sides. Hence, the distance between each of our points and the origin is $\sqrt{x^2 + z^2}$.



Using the Pythagorean theorem.

Add a `Ripple` function method and have it calculate the distance, using `Mathf.Sqrt` to compute the square root. For now, just use that as the output.

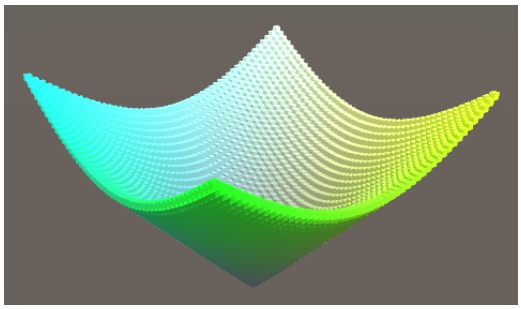
```
	static float Ripple (float x, float z, float t) {
		float d = Mathf.Sqrt(x * x + z * z);
		float y = d;
		return y;
	}
```

Append this method to the `functions` array.

```
	static GraphFunction[] functions = {
		SineFunction, Sine2DFunction, MultiSineFunction, MultiSine2DFunction,
		Ripple
	};
```

And also adds its name to the enumeration.

```
public enum GraphFunctionName {
	Sine, Sine2D, MultiSine, MultiSine2D,
	Ripple
}
```

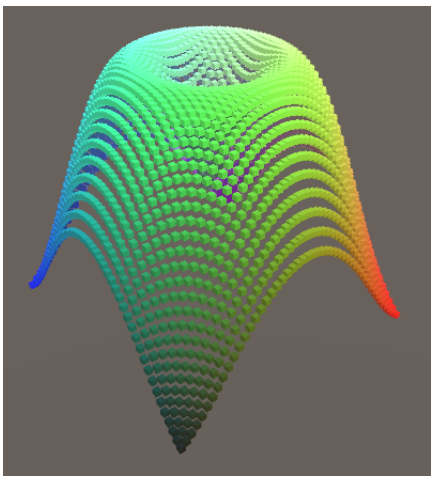


Distance from the origin.

What we get is a cone shape that's at zero in the middle and increases linearly with the distance from the origin. It ends up highest near the corners of the grid, because those points are furthest away from the origin. Exactly at the corners, the distance would be $\sqrt{2}$, which is roughly 1.4142.

To create our ripple, we'll have to use $f(x, z, t) = \sin(\pi D)$ where D is the distance.

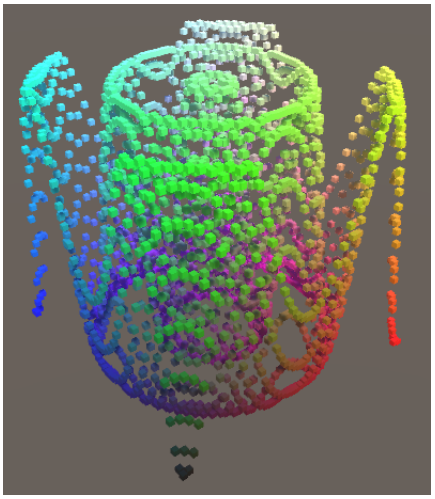
```
		float d = Mathf.Sqrt(x * x + z * z);
		float y = Mathf.Sin(pi * d);
		return y;
```



Sine of distance to origin.

This doesn't show us much of the wave pattern, so let's increase its frequency fourfold.

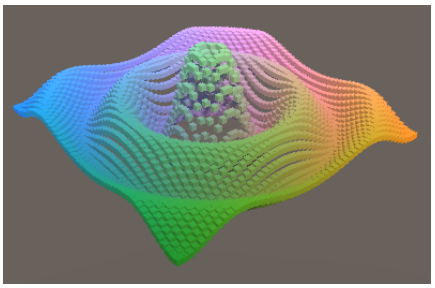
```
		float y = Mathf.Sin(4f * pi * d);
```



Higher frequency.

We're getting closer to a ripple shape, but the undulation is far too extreme. We can take care of that by reducing the amplitude of the wave. Instead of doing this uniformly, we can make it depend on the distance as well. For example, we could use $\frac{1}{10D}$ as the amplitude. This causes the ripple to get weaker further away from its origin, which mimics how ripples behave, although we won't bother with physically correct proportions. However, simply dividing by the distance will result in a division by zero at the origin, and cause the amplitude to become extreme near the origin. To prevent this, we'll use $\frac{1}{1 + 10D}$ instead.

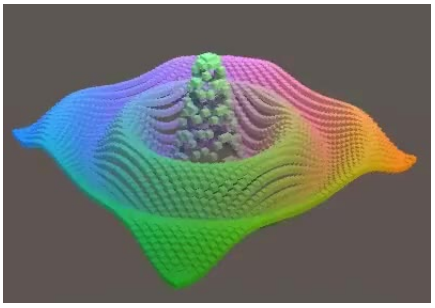
```
		float y = Mathf.Sin(4f * pi * d);
		y /= 1f + 10f * d;
		return y;
```



Scaled by distances.

Finally, add the time to the sine wave to animate it. Because the ripple should move outward, subtract t instead of adding it.

```
		float y = Mathf.Sin(pi * (4f * d - t));
```



Animated ripple.

3 Leaving the Grid

By using X and Z to define Y, we are able to create functions that describe a large variety of surfaces, but they're always linked to the XZ plane. No two points can have the same X and Z coordinates, while having a different Y coordinate. This means that the curvature of our surfaces is limited. Their slopes cannot become vertical and they cannot fold backward. To make this possible, our functions would have to not only output Y, but also X and Z.

3.1 Three-dimensional Functions

If our functions were to output 3D positions instead of 1D values, we could use them to create arbitrary surfaces. For example, the function $f(x, z) = \begin{bmatrix} x \\ 0 \\ z \end{bmatrix}$ describes the XZ plane, while the function $f(x, z) = \begin{bmatrix} x \\ z \\ 0 \end{bmatrix}$ describes the XY plane.

Because the input parameters for these functions no longer correspond to the final X and Z coordinates, it's no longer appropriate to name them x and z. Instead, they're often named u and v. So we'd get functions like $f(u, v) = \begin{bmatrix} u + v \\ uv \\ \frac{u}{v} \end{bmatrix}$.

Adjust our `GraphFunction` delegate to support this new approach. The only required change is replacing its `float` return type with `Vector3`, but let's also rename its parameters.

```
public delegate Vector3 GraphFunction (float u, float v, float t);
```

Our original sine function now has to be defined as $f(u, v, t) = \begin{bmatrix} u \\ \sin(\pi(u + t)) \\ v \end{bmatrix}$.

But because we're not adjusting X and Z we'll leave the parameter names of `SineFunction` unchanged. It now has to return a vector, directly using `x` and `z` for its X and Z coordinates, while calculating the Y coordinate.

```
	static Vector3 SineFunction (float x, float z, float t) {
//		return Mathf.Sin(pi * (x + t));

		Vector3 p;
		p.x = x;
		p.y = Mathf.Sin(pi * (x + t));
		p.z = z;
		return p;
	}
```

Make the same changes to `Sine2DFunction`.

```
	static Vector3 Sine2DFunction (float x, float z, float t) {
//		float y = Mathf.Sin(pi * (x + t));
//		y += Mathf.Sin(pi * (z + t));
//		y *= 0.5f;
//		return y;

		Vector3 p;
		p.x = x;
		p.y = Mathf.Sin(pi * (x + t));
		p.y += Mathf.Sin(pi * (z + t));
		p.y *= 0.5f;
		p.z = z;
		return p;
	}
```

Adjust the other three function methods too.

```
	static Vector3 MultiSineFunction (float x, float z, float t) {
		Vector3 p;
		p.x = x;
		p.y = Mathf.Sin(pi * (x + t));
		p.y += Mathf.Sin(2f * pi * (x + 2f * t)) / 2f;
		p.y *= 2f / 3f;
		p.z = z;
		return p;
	}
```

```
	static Vector3 MultiSine2DFunction (float x, float z, float t) {
		Vector3 p;
		p.x = x;
		p.y = 4f * Mathf.Sin(pi * (x + z + t / 2f));
		p.y += Mathf.Sin(pi * (x + t));
		p.y += Mathf.Sin(2f * pi * (z + 2f * t)) * 0.5f;
		p.y *= 1f / 5.5f;
		p.z = z;
		return p;
	}
```

```
	static Vector3 Ripple (float x, float z, float t) {
		Vector3 p;
		float d = Mathf.Sqrt(x * x + z * z);
		p.x = x;
		p.y = Mathf.Sin(pi * (4f * d - t));
		p.y /= 1f + 10f * d;
		p.z = z;
		return p;
	}
```

Because the X and Z coordinates of points are no longer constant, we can no longer rely on their initial values in `Update`. Instead, we have to supply fresh U and V inputs, which we can do with a double loop. We can then directly assign the result of the function method to the point's position, so we no longer need to retrieve it.

```
	void Update () {
		float t = Time.time;
		GraphFunction f = functions[(int)function];
//		for (int i = 0; i < points.Length; i++) {
//			Transform point = points[i];
//			Vector3 position = point.localPosition;
//			position.y = f(position.x, position.z, t);
//			point.localPosition = position;
//		}
		float step = 2f / resolution;
		for (int i = 0, z = 0; z < resolution; z++) {
			float v = (z + 0.5f) * step - 1f;
			for (int x = 0; x < resolution; x++, i++) {
				float u = (x + 0.5f) * step - 1f;
				points[i].localPosition = f(u, v, t);
			}
		}
	}
```

As this new approach no longer relies on the original positions, we no longer need to initialize them in `Awake`, making that method a lot simpler. We can make do with a single loop that initializes all points and leaving their positions unchanged.

```
	void Awake () {
		float step = 2f / resolution;
		Vector3 scale = Vector3.one * step;
		Vector3 position;
		position.y = 0f;
		position.z = 0f;
		points = new Transform[resolution * resolution];
//		for (int i = 0, z = 0; z < resolution; z++) {
//			position.z = (z + 0.5f) * step - 1f;
//			for (int x = 0; x < resolution; x++, i++) {
//				Transform point = Instantiate(pointPrefab);
//				position.x = (x + 0.5f) * step - 1f;
//				point.localPosition = position;
//				point.localScale = scale;
//				point.SetParent(transform, false);
//				points[i] = point;
//			}
//		}
		for (int i = 0; i < points.Length; i++) {
			Transform point = Instantiate(pointPrefab);
			point.localScale = scale;
			point.SetParent(transform, false);
			points[i] = point;
		}
	}
```

3.2 Creating a Cylinder

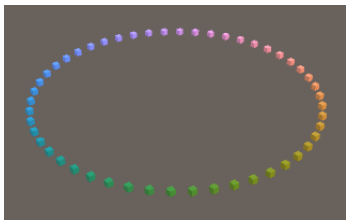
To demonstrate that we indeed are no longer limited to one point per (X, Z) coordinate pair, let's create a function that defines a cylinder. Add a `Cylinder` function method for this purpose, starting with always returning a point at the origin.

Also add this method to the `functions` array and add a name for it to `GraphFunctionName`, as usual. I'll no longer explicitly mention this step.

```
	static Vector3 Cylinder (float u, float v, float t) {
		Vector3 p;
		p.x = 0f;
		p.y = 0f;
		p.z = 0f;
		return p;
	}
```

A cylinder is an extruded circle, so we'll begin with just the circle part. As mention in the previous tutorial, all points on a 2D circle can be defined via $\begin{bmatrix} \sin(\theta) \\ \cos(\theta) \end{bmatrix}$ with θ going from 0 to 2π. We can use u instead, which in our case goes from −1 to 1. To create a circle in the XZ plane, we need the function $f(u) = \begin{bmatrix} \sin(\pi u) \\ 0 \\ \cos(\pi u) \end{bmatrix}$.

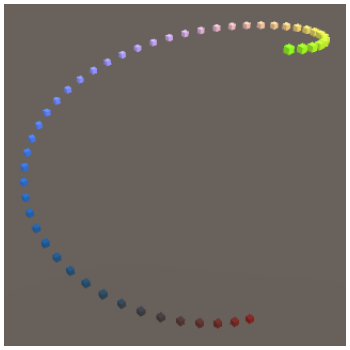
```
	static Vector3 Cylinder (float u, float v, float t) {
		Vector3 p;
		p.x = Mathf.Sin(pi * u);
		p.y = 0f;
		p.z = Mathf.Cos(pi * u);
		return p;
	}
```



A circle.

Because the function doesn't use v, all points that use the same v input end up at the exact same position. So we're effectively reduced to a single line. To see how this line wraps around to form a circle, make Y equal to u.

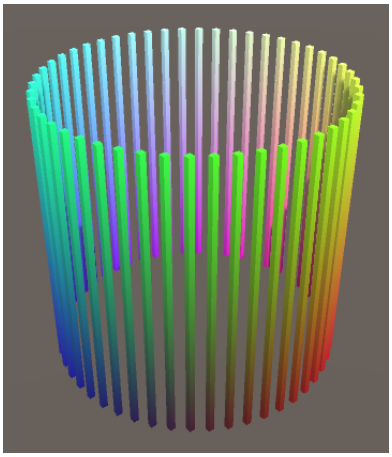
```
		p.x = Mathf.Sin(pi * u);
		p.y = u;
		p.z = Mathf.Cos(pi * u);
```



Increasing Y along the circle.

This shows us that the line starts at $\begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix}$ and curves around the origin in a clockwise direction, consistent with the function's input. To turn it into an actual cylinder, make Y equal to v instead. That way we end up building a cylinder by stacking flat circles along Y.

```
		p.x = Mathf.Sin(pi * u);
		p.y = v;
		p.z = Mathf.Cos(pi * u);
```



A cylinder.

We're currently using the unit circle as the basis for our cylinder, but we don't have to. The circle's radius can be adjusted by scaling the amplitude of the sine and cosine by the same amount. In general, the function becomes $f(u, v) = \begin{bmatrix} R\sin(\pi u) \\ v \\ R\cos(\pi v) \end{bmatrix}$ where R is the circle's radius. Adjust our method so it uses an explicit radius of 1.

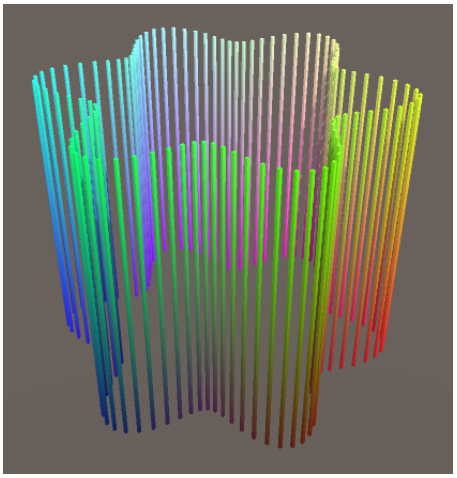
```
		float r = 1f;
		p.x = r * Mathf.Sin(pi * u);
		p.y = v;
		p.z = r * Mathf.Cos(pi * u);
```

What would happen if we used different amplitudes?

When you make the amplitudes of the sine and cosine different, you get an ellipse.

We could use any other radius, it doesn't even have to be constant. For example, we could vary the radius along u. Let's use another sine wave for that, like $R = 1 + \frac{\sin(6\pi u)}{5}$.

```
		float r = 1f + Mathf.Sin(6f * pi * u) * 0.2f;
```

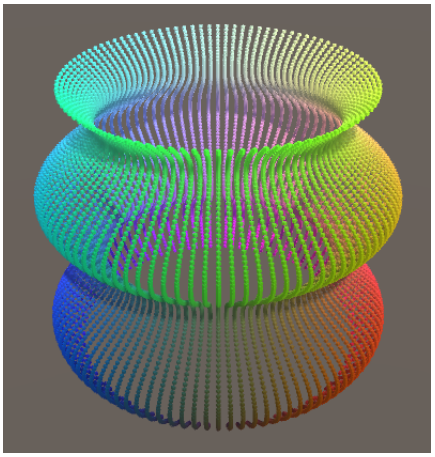


Wobbly cylinder, resolution 100.

The result is that the cylinder becomes wobbly. The circle has become a rounded star shape. The surface follows a wave pattern around the circle, moving in and out six times.

We can also make the radius depend on v, for example $R = 1 + \frac{\sin(2\pi v)}{5}$. In this case, each ring of the cylinder has a constant radius, but the radius varies along the cylinder's length.

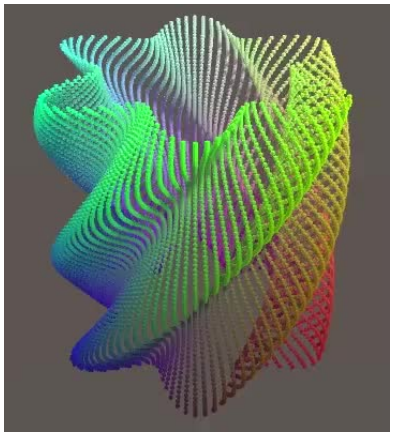
```
		float r = 1f + Mathf.Sin(2f * pi * v) * 0.2f;
```



Using V instead of U.

Even more interesting would be to use both u and v to create a diagonal wave, which will end up twisting around the cylinder. Let's add t as well, to animate it. Finally, to make sure that the radius doesn't exceed 1, reduce its baseline to $\frac{4}{5}$.

```
		float r = 0.8f + Mathf.Sin(pi * (6f * u + 2f * v + t)) * 0.2f;
```

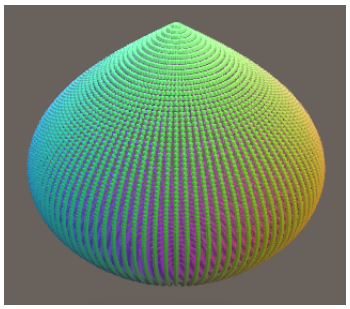


Twisting Cylinder.

3.3 Creating a Sphere

Now that we have a cylinder, let's move on to a sphere. To do this, duplicate the `Cylinder` method and rename it to `Sphere`. Let's see it we can turn the cylinder into a sphere by reducing its radius at its top and bottom to zero with another wave, using $R = \cos\left(\frac{\pi v}{2}\right)$.

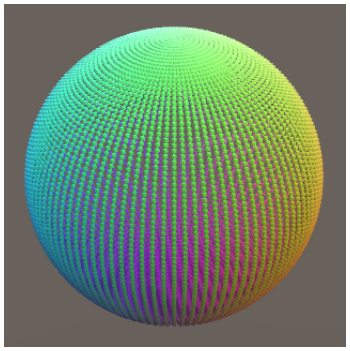
```
	static Vector3 Sphere (float u, float v, float t) {
		Vector3 p;
		float r = Mathf.Cos(pi * 0.5f * v);
		p.x = r * Mathf.Sin(pi * u);
		p.y = v;
		p.z = r * Mathf.Cos(pi * u);
		return p;
	}
```



Almost a sphere.

This gets us close, but the reduction of the cylinder's radius isn't circular yet. That's because a circle is made with both a sine and a cosine, and we're only using the cosine at this point. The other part of the equation is Y, which is currently still equal to v. To complete the circle, Y has to become equal to $\sin\left(\frac{\pi v}{2}\right)$.

```
		p.y = Mathf.Sin(pi * 0.5f * v);
```



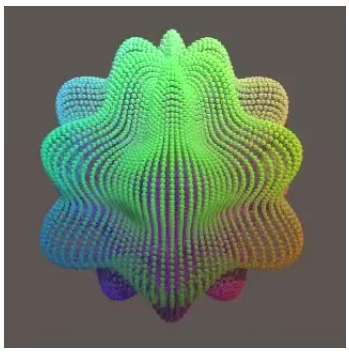
A sphere.

We end up with a sphere, created with a pattern that's usually known as a UV-sphere. While this approach creates a correct sphere, note that the distribution of points isn't uniform, because the sphere is created by stacking circles with varying radius. At the poles of the sphere, their radius becomes zero.

To be able to control the radius of the sphere, we have to adjust our formula somewhat. We'll use $f(u, v) = \begin{bmatrix} S\sin(\pi u) \\ R\sin\left(\frac{\pi v}{2}\right) \\ S\cos(\pi u) \end{bmatrix}$, where $S = R\cos\left(\frac{\pi v}{2}\right)$, and R is the radius.

That approach makes it possible to animate the sphere's radius. Let's use separate sine waves for u and v this time, $R = \frac{4}{5} + \frac{\sin(\pi(6u + t))}{10} + \frac{\sin(\pi(4v + t))}{10}$.

```
		float r = 0.8f + Mathf.Sin(pi * (6f * u + t)) * 0.1f;
		r += Mathf.Sin(pi * (4f * v + t)) * 0.1f;
		float s = r * Mathf.Cos(pi * 0.5f * v);
		p.x = s * Mathf.Sin(pi * u);
		p.y = r * Mathf.Sin(pi * 0.5f * v);
		p.z = s * Mathf.Cos(pi * u);
```



Pulsing sphere.

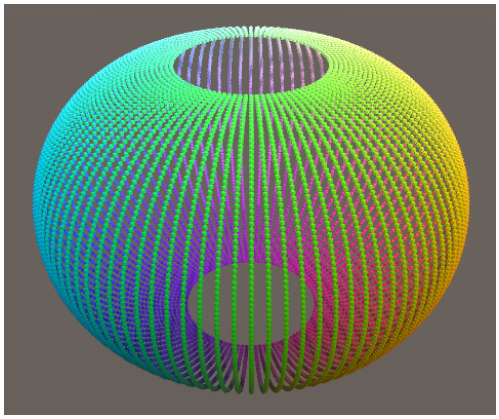
3.4 Creating a Torus

We're going to end this tutorial by turning a sphere into a torus. Copy `Sphere` and rename it to `Torus`, then eliminate the code for the sphere's radius.

```
	static Vector3 Torus (float u, float v, float t) {
		Vector3 p;
		float s = Mathf.Cos(pi * 0.5f * v);
		p.x = s * Mathf.Sin(pi * u);
		p.y = Mathf.Sin(pi * 0.5f * v);
		p.z = s * Mathf.Cos(pi * u);
		return p;
	}
```

We'll create the torus by pulling the sphere apart, like we're grabbing it by its poles an pulling it in all directions, in the XZ plane. We can do that by adding a constant value to S, for example ½.

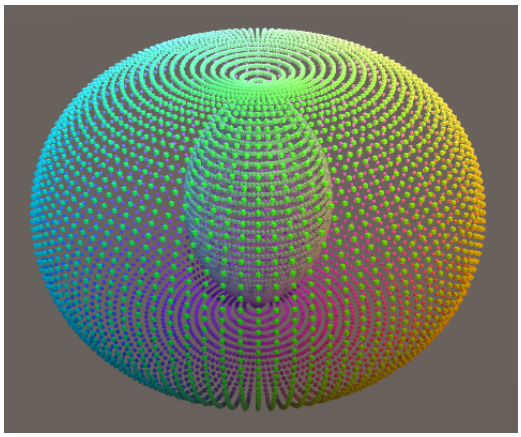
```
		float s = Mathf.Cos(pi * 0.5f * v) + 0.5f;
```



Pulling apart a sphere.

This gives us half a torus, with only the outer portion of its ring accounted for. To complete the torus, we have to use v to describe an entire circle instead of half a circle. That can be done by using πv instead of $\frac{\pi v}{2}$.

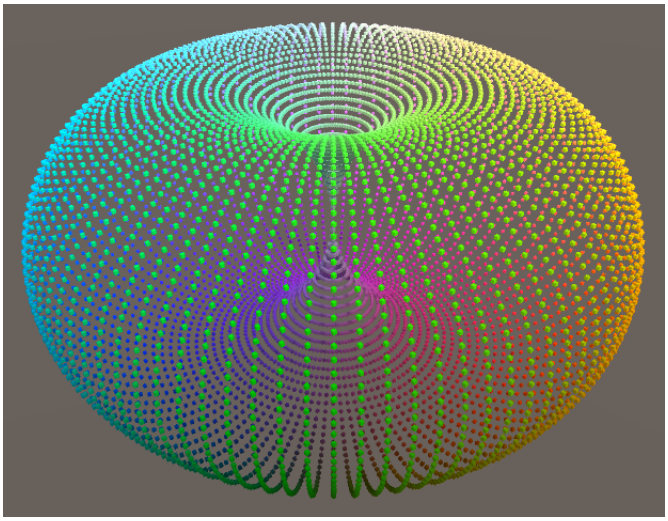
```
		float s = Mathf.Cos(pi * v) + 0.5f;
		p.x = s * Mathf.Sin(pi * u);
		p.y = Mathf.Sin(pi * v);
		p.z = s * Mathf.Cos(pi * u);
```



A self-intersecting spindle torus.

Because we've pulled the sphere apart by half a unit, this produces a self-intersecting shape, which is known as a spindle torus. Had we pulled it apart by one unit instead, we would've gotten a torus that doesn't self-intersect, but doesn't have a hole either, which is known as a horn torus. So how far we pull the sphere apart influences the shape of the torus. Specifically, it defines the major radius of the torus, which we'll designate with R_1. So our function becomes $f(u, v) = \begin{bmatrix} S\sin(\pi u) \\ \sin(\pi v) \\ S\cos(\pi u) \end{bmatrix}$, where $S = \cos(\pi v) + R_1$.

```
		float r1 = 1f;
		float s = Mathf.Cos(pi * v) + r1;
```

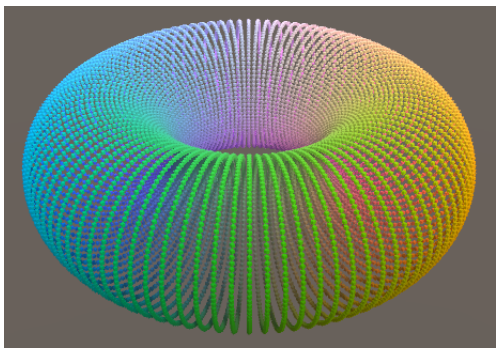


A horn torus.

Making R_1 greater than 1 will open up a hole in the middle of the torus, which would make it a ring torus. But this assumes that the circles that we wrap around the ring always have radius 1. That is the secondary radius of the torus, R_2, and we can change that one as well, if we use the function $f(u, v) = \begin{bmatrix} S\sin(\pi u) \\ R_2\sin(\pi v) \\ S\cos(\pi u) \end{bmatrix}$, where $S = R_2\cos(\pi v) + R_1$.

Let's keep R_1 at 1 and reduce R_2 to ½.

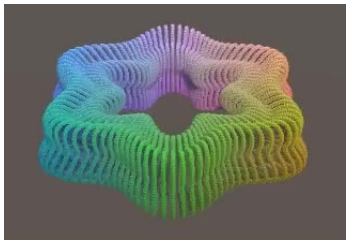
```
float r1 = 1f;
float r2 = 0.5f;
float s = r2 * Mathf.Cos(pi * v) + r1;
p.x = s * Mathf.Sin(pi * u);
p.y = r2 * Mathf.Sin(pi * v);
p.z = s * Mathf.Cos(pi * u);
```



A ring torus.

Now we have two radiuses to play with to make a more interesting torus. A fairly simple but still interesting approach is to add a u wave to R_1 and a v wave to R_2, both animated, while making sure that the torus fits inside the −1–1 range.

```
		float r1 = 0.65f + Mathf.Sin(pi * (6f * u + t)) * 0.1f;
		float r2 = 0.2f + Mathf.Sin(pi * (4f * v + t)) * 0.05f;
```



A more interesting torus.

You now have some experience working with nontrivial functions that describe 3D surfaces, plus how to visualize them. You can experiment with your own functions to get a better grasp of how it works. There are many seemingly complex shapes that can be created with just a few sine waves. After you're done, you can move on to Constructing a Fractal.

made by Jasper Flick