Catlike Coding

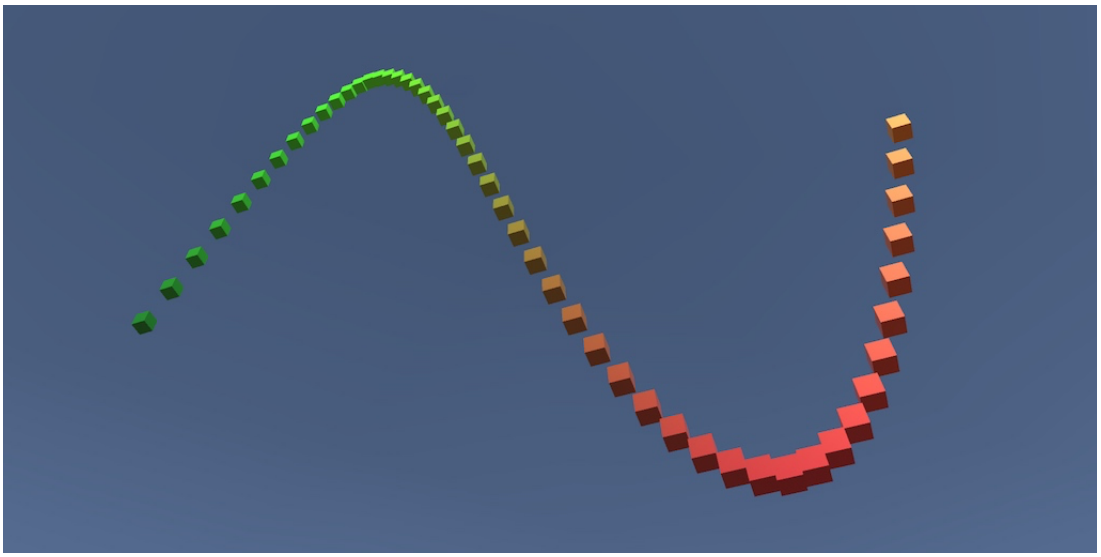
Unity C# Tutorials

Building a Graph Visualizing Math

Create a prefab.
Instantiate a line of cubes.
Show a mathematical function.
Create a custom shader.
Make the graph move.

In this tutorial we will use game objects to build a graph, so we can show mathematical formulas. We'll also make the function time-dependent, resulting in an animating graph.

This tutorial assumes you've done the Game Objects and Scripts tutorial and that you're using at least Unity 2017.1.0.



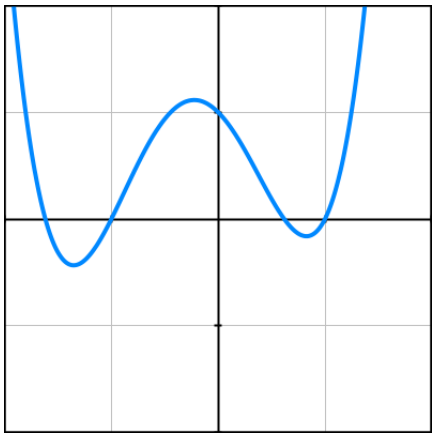
Using cubes to show a sine wave.

1 Creating a Line of Cubes

A good understanding of mathematics is essential when programming. At its most fundamental level math is the manipulation of symbols that represent numbers. Solving an equation boils down to rewriting one set of symbols so it becomes another—usually shorter—set of symbols. The rules of mathematics dictate how this rewriting can be done.

For example, we have the function $f(x) = x + 1$. We can substitute a number for x, say 3. That leads to $f(3) = 3 + 1 = 4$. We provided 3 as input and ended up with 4 as output. We can say that the function maps 3 to 4. A shorter way to write this would be as an input-output pair, like (3,4). We can create many pairs of the form $(x, f(x))$. For example (5,6) and (8,9) and (1,2) and (6,7). But it is easier to understand the function when we order the pairs by the input number. (1,2) and (2,3) and (3,4) and so on.

The function $f(x) = x + 1$ is easy to understand.
$f(x) = (x - 1)^4 + 5x^3 - 8x^2 + 3x$ is harder. We could write down a few input-output pairs, but that likely won't give us a good grasp of the mapping it represents. We're going to need many points, close together. That will end up as a sea of numbers, which are hard to parse. Instead, we could interpret the pairs as two-dimensional coordinates of the form $\begin{bmatrix} x \\ f(x) \end{bmatrix}$. This is a 2D vector where the top number represents the horizontal coordinate, on the X axis, and the bottom number represents the vertical coordinate, on the Y axis. In other words, $y = f(x)$. We can plot these points on a surface. If we use enough points, we end up with a line. The result is a graph.



Graph with x between −2 and 2.

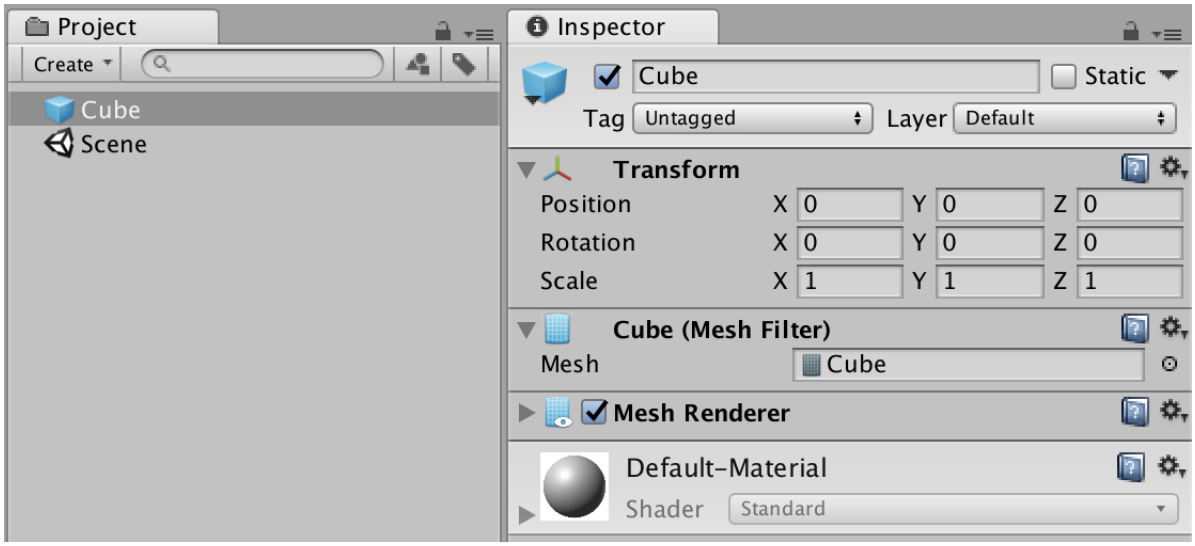
Looking at a graph can quickly give us an idea of how a function behaves. It's a handy tool, so let's create one in Unity. Start with a new scene via *File / New Scene* for this purpose, or use the default scene of a new project.

1.1 Prefabs

Graphs are created by placing points at the appropriate coordinates. To do this, we need a 3D visualization of a point. We'll simply use Unity's default cube game object for this. Add one to the scene and remove its collider component, as we won't use physics.

> **Are cubes the best way to visualize graphs?**
>
> You could also use a particle system or line segments, but individual cubes are the simplest to use.

We will be using a script to create many instances of this cube and position them correctly. In order to do this, we'll use the cube as a template. Drag the cube from the hierarchy window into the project window. This will create a new asset, with a blue cube icon, known as a prefab. It is a pre-fabricated game object that exists in the project, but not in a scene.



A cube prefab.

Prefabs are a handy way to configure game objects. If you change the prefab asset, all instances of it in any scene are changed in the same way. For example, changing the prefab's scale will also change the scale of the cube that's still in the scene. However, each instance uses its own position and rotation. Also, game objects can have their properties modified, which overrides the prefab's values. If large changes are made, like adding or removing a component, the relationship between prefab and instance will be broken.

We're going to use a script to create instances of the prefab, so we no longer need the cube instance that is currently in the scene. So delete it.

1.2 Graph Component

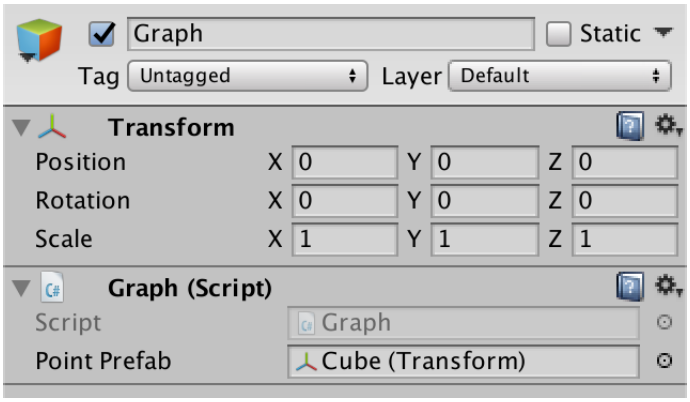
We need a C# script to generate our graph. Create one and name it *Graph*. We begin with a simple class that extends `MonoBehaviour` so it can be used as a component for game objects. Give it a public field to hold a reference to a prefab for creating points, named `pointPrefab`. As we'll need access to the `Transform` component to position the points, make that the field's type.

```
using UnityEngine;

public class Graph : MonoBehaviour {

	public Transform pointPrefab;
}
```

Add an empty game object to the scene, via *GameObject / Create Empty*, place it at the origin, and name it *Graph*. Add our `Graph` component to this object, via dragging or via its *Add Component* button. Then drag our prefab asset onto the *Point Prefab* field of the graph. It now holds a reference to the prefab's `Transform` component.



Graph object with reference to prefab.

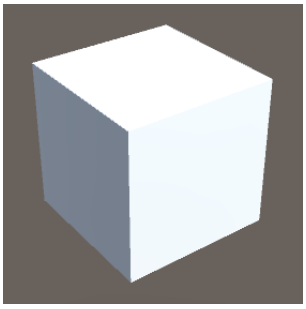
1.3 Instantiating Prefabs

Instantiating a game object is done via the `Instantiate` method. This is a publicly available method of Unity's `Object` type, which `Graph` indirectly inherited by extending `MonoBehaviour`. The `Instantiate` method clones whatever Unity object is provided as an argument. In the case of a prefab, it will result in an instance being added to the current scene. Let's do this when our `Graph` component awakens.

```
public class Graph : MonoBehaviour {

	public Transform pointPrefab;

	void Awake () {
		Instantiate(pointPrefab);
	}
}
```



Instantiated prefab.

At this point, entering play mode will produce a single cube at the origin, provided that the prefab asset's position is set to zero. To place the point somewhere else, we need to adjust the position of the instance. The `Instantiate` method gives us a reference to whatever it created. Because we gave it a reference to a `Transform` component, that's what we get in return. Let's keep track of it with a variable.

```
	void Awake () {
		Transform point = Instantiate(pointPrefab);
	}
```

Now we can adjust the point's position, by assigning a 3D vector to it. Like we adjusted the local rotation of the handles of the clock in Game Objects and Scripts, we'll adjust the local position of the point via its `localPosition` property, not `position`.

3D vectors are created with the `Vector3` struct. As it's a struct, it acts like a value, similar to a number, not an object. For example, let's set the X coordinate of our point to 1, leaving its Y and Z coordinates at zero. `Vector3` has a `right` property for this.

```
		Transform point = Instantiate(pointPrefab);
		point.localPosition = Vector3.right;
```

Shouldn't properties be capitalized?

The convention is to capitalize properties, yes, but Unity often doesn't do this.

When entering play mode now, we still get one cube, just at a slightly different position. Let's instantiate a second one and place it an additional step to the right. This can be done by multiplying the right vector by 2. Repeat the instantiation and positioning code, then add the multiplication to the new code.

```
	void Awake () {
		Transform point = Instantiate(pointPrefab);
		point.localPosition = Vector3.right;

		Transform point = Instantiate(pointPrefab);
		point.localPosition = Vector3.right * 2f;
	}
```

Can you multiply structs and numbers?

Normally you cannot, but it is possible to define such functionality. This is done by creating a method with a special syntax, so it can be invoked as if it were a multiplication. In this case, what appears to be a simple multiplication is actually a method invocation, something like `Vector3.Multiply(Vector3.right, 2f)`.

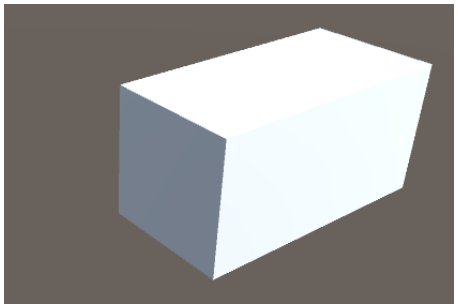
Being able to use methods as if they were simple operations makes writing code faster and easier to read. It is not essential, but nice to have, just like being able to implicitly use namespaces. Such convenient syntax is known as syntactic sugar.

Having said that, methods should only be used as operators if they strictly match the original meaning of that operator. In the case of vectors, some mathematical operators are well-defined, so it's fine for those.

This code will lead to a compile error, because we attempt to define the `point` variable twice. If we want to use another variable, we have to give it a different name. Alternatively, we reuse the variable that we already have. We don't need to hold on to the reference to the first point anyway. Simply assign the new point to the same variable.

```
		Transform point = Instantiate(pointPrefab);
		point.localPosition = Vector3.right;

//		Transform point = Instantiate(pointPrefab);
		point = Instantiate(pointPrefab);
		point.localPosition = Vector3.right * 2f;
```



Two instances, with X coordinates 1 and 2.

1.4 Code Loops

Let's create more point, until we have ten. We could repeat the same code eight more times, but this is very inefficient programming. Ideally, we only write the code for one point and instruct the program to execute it multiple times, with slight variation.

The `while` statement can be used to cause a block of code to repeat. Apply it to the first two lines of our method and remove the other lines.

```
	void Awake () {
		while (
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right;
		)
//		point = Instantiate(pointPrefab);
//		point.localPosition = Vector3.right * 2f;
	}
```

Like `if` statements, `while` must be followed by an expression within round brackets. Like with `if`, the code block following `while` will only be executed if the expression evaluates as true. Afterwards, the program will loop back to the `while` statement. If at this point the expression again evaluates as true, the code block will be executed again. This repeats until the expression evaluates as false.

So we have to add an expression after `while`. We must be careful to make sure that the loop doesn't repeat forever. Infinite loops cause programs to get stuck, requiring manual termination by the user. The safest possible expression that compiles is simply `false`.

```
        while (false) {
            Transform point = Instantiate(pointPrefab);
            point.localPosition = Vector3.right;
        }
```

Can we define `point` inside the loop?

Yes. Although the code gets repeated, we've defined the variable only once. It gets reused each iteration of the loop, like we manually did earlier.

You could also define `point` before the loop. That allows you to use the variable outside the loop as well. Otherwise, its scope is limited to the block of the `while` loop.

Limiting the loop can be done by keeping track of how many times we've repeated the code. We can use an integer variable to keep track of this. It will contain the iteration number of the loop, so let's name it `i`. To be able to use it in the `while` expression, it must be defined earlier.

```
    void Awake () {
        int i;
        while (false) {
            Transform point = Instantiate(pointPrefab);
            point.localPosition = Vector3.right;
        }
    }
```

Each iteration, increase the number by one.

```
        int i;
        while (false) {
            i = i + 1;
            Transform point = Instantiate(pointPrefab);
            point.localPosition = Vector3.right;
        }
```

This produces a compile error, because we're trying to use `i` before we've assigned a value to it. We have to explicitly assign zero to `i` when we define it to make this work.

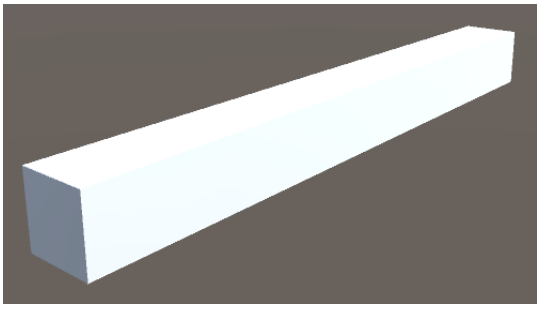
```
        int i = 0;
```

Now `i` becomes 1 at the start of the first iteration, 2 at the start of the second iteration, and so on. But the `while` expression is evaluated before each iteration. So right before the first iteration `i` is zero, it's 1 before the second, and so on. So after the tenth iteration `i` is ten. At this point we want to stop the loop, so its expression should evaluate as false. In other words, we should continue as long as `i` is less than ten. Mathematically, that's expressed as $i < 10$. It is written the same in code, also with the less-than operator.

```
		int i = 0;
		while (i < 10) {
			i = i + 1;
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right;
		}
```

Now we'll get ten cubes after entering play mode. But they all end up at the same position. To put them in a row along the X axis, multiply the `right` vector by `i`.

```
			point.localPosition = Vector3.right * i;
```



Ten cubes in a row.

Note that currently the first cube ends up with an X coordinate of 1 and the last cube end up with 10. Ideally, we start at 0, positioning the first cube at the origin. We can shift all points one unit to the left by multiplying `right` by `(i - 1)` instead of `i`. However, we could skip that extra subtraction by increasing `i` at the end of the block, instead of at the beginning.

```
		while (i < 10) {
//			i = i + 1;
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right * i;
			i = i + 1;
		}
```

1.5 Concise Syntax

Because looping a certain amount of times is so common, it is convenient to use as short a syntax as possible. Some syntactic sugar can help us with that.

First, let's considering incrementing the iteration number. When an operation of the form `x = x * y` is performed, it can be shortened to `x *= y`. This works for all operators that act on two operands of the same type.

```
//            i = i + 1;
            i += 1;
```

Going even further, when incrementing or decrementing a number by 1, this can be shortened to `++x` or `--x`.

```
//            i += 1;
            ++i;
```

One property of assignment statements is that they can also be used as expressions. This means that you could write something like `y = (x += 3)`. That would increase `x` by three and assign the result of that to `y` as well. This suggests that we could increment `i` inside the `while` expression, shortening the code block.

```
        while (++i < 10) {
            Transform point = Instantiate(pointPrefab);
            point.localPosition = Vector3.right * i;
//            ++i;
        }
```

However, now we're incrementing `i` before the comparison, instead of afterwards, which would lead to one less iteration. Specifically for situations like this, the increment and decrement operators can also be placed after a variable, instead of before it. The result of that expression is the original value, before it was changed.

```
//        while (++i < 10) {
        while (i++ < 10) {
            Transform point = Instantiate(pointPrefab);
            point.localPosition = Vector3.right * i;
        }
```

Although the `while` statement works for all kinds of loops, there is an alternative syntax particularly suited for iterating over ranges. It is the `for` loop. It works like `while`, except that both the iterator variable declaration and its comparison are contained within round brackets, separated by a semicolon.

```
//		int i = 0;
//		while (i++ < 10) {
		for (int i = 0; i++ < 10) {
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right * i;
		}
```

That would produce a compile error, because there are actually three parts. The third is for incrementing the iterator, keeping it separate from the comparison.

```
//		for (int i = 0; i++ < 10) {
		for (int i = 0; i < 10; i++) {
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right * i;
		}
```

Why use `i++` and not `++i` in the `for` loop?

As the increment expression is not used for anything else, it doesn't matter which version we use. We could've also used `i += 1` or `i = i + 1`.

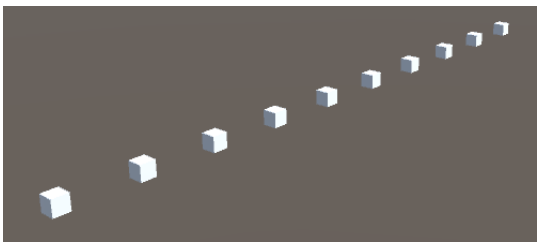
The classical for loop has the form `for (int i = 0; i < someLimit; i++)`. You will encounter that code fragment in a lot of programs and scripts.

1.6 Changing the Domain

Currently, our cubes are given X coordinates 0 through 9. This isn't a convenient range when working with functions. Often, a range of 0–1 is used for X. Or when working with functions that are centered around zero, a range of −1–1. Let's reposition our cubes accordingly.

Positioning our ten cubes along a line segment two units long will cause them to overlap. To prevent this, we're going to reduce their scale. Each cube has size 1 in each dimension by default, so to make them fit we have to reduce their scale to $\frac{2}{10} = \frac{1}{5}$. We can do this by setting each point's local scale to the `Vector3.one` property divided by five.

```
		for (int i = 0; i < 10; i++) {
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right * i;
			point.localScale = Vector3.one / 5f;
		}
```



Small cubes.

To bring the cubes back together again, divide their positions by five as well.

```
			point.localPosition = Vector3.right * i / 5f;
```

This makes them cover the 0–2 range. To turn that into the −1–1 range, subtract 1 before scaling the vector.

```
			point.localPosition = Vector3.right * (i / 5f - 1f);
```

Now the first cube has X coordinate −1, while the last has coordinate 0.8. However, the cube size is 0.2. As the cube is centered on its position, the left side of the first cube is at −1.1, while the right side of the last cube is at 0.9. To neatly fill the −1–1 range with our cubes, we have to shift them half a cube to the right. This can be done by adding 0.5 to `i` before dividing it.

```
			point.localPosition = Vector3.right * ((i + 0.5f) / 5f - 1f);
```

1.7 Hoisting the Vectors out of the Loop

Although all the cubes have the same scale, we calculate it in every iteration of the loop. We don't have to do this. Instead, we could calculate it once before the loop, store it in a `Vector3` variable, and use that in the loop.

```
	void Awake () {
		Vector3 scale = Vector3.one / 5f;
		for (int i = 0; i < 10; i++) {
			Transform point = Instantiate(pointPrefab);
			point.localPosition = Vector3.right * ((i + 0.5f) / 5f - 1f);
			point.localScale = scale;
		}
	}
```

We could also define a variable for the position before the loop. As we're creating a line along the X axis, we only need to adjust the X coordinate of the position inside the loop. So we no longer have to multiply by `Vector3.right`.

```
		Vector3 scale = Vector3.one / 5f;
		Vector3 position;
		for (int i = 0; i < 10; i++) {
			Transform point = Instantiate(pointPrefab);
//			point.localPosition = Vector3.right * ((i + 0.5f) / 5f - 1f);
			position.x = (i + 0.5f) / 5f - 1f;
			point.localPosition = position;
			point.localScale = scale;
		}
```

Can we change a vector's components individually?

The `Vector3` struct has three floating-point fields, `x`, `y`, and `z`. These fields are public, so we can change them.

Because structs behave like simple values, the idea is that they should be immutable. Once constructed, they should't change. If you want to use a different value, assign a new struct to the field or variable, like we do with numbers. If we say that $x = 3$ and later that $x = 5$, we've assigned a different number to x. We didn't modify the number 3 itself to become a 5. However, the vector types of Unity are mutable. This is done both for convenience and performance, because individual vector components are often manipulated independently.

To get an idea of how to work with mutable vectors, you can consider the use of `Vector3` a convenient substitute for using three separate `float` values. You can access them independently, yet also copy and assign them as a group.

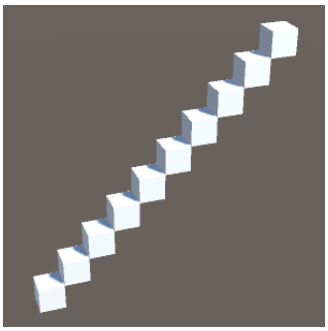
This will result in a compile error, complaining about the use of an unassigned variable. This happens because we're assigning `position` to something while we haven't set its Y and Z coordinates yet. Explicitly set them to zero before the loop.

```
		Vector3 position;
		position.y = 0f;
		position.z = 0f;
		for (int i = 0; i < 10; i++) {
			…
		}
```

1.8 Using X to Define Y

The idea is that the positions of our cubes are defined as $\begin{bmatrix} x \\ f(x) \\ 0 \end{bmatrix}$, so we can use them to display a function. At this point, the Y coordinates are always zero, which represents the trivial function $f(x) = 0$. To show a different function, we have to determine the Y coordinate inside the loop, instead of before it. Let's make it equal to X, representing the function $f(x) = x$.

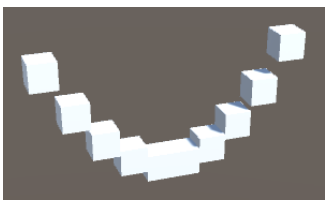
```
		Vector3 position;
//		position.y = 0f;
		position.z = 0f;
		for (int i = 0; i < 10; i++) {
			Transform point = Instantiate(pointPrefab);
			position.x = (i + 0.5f) / 5f - 1f;
			position.y = position.x;
			point.localPosition = position;
			point.localScale = scale;
		}
```



Y equals X.

A slightly less obvious function would be $f(x) = x^2$, which defines a parabola with its minimum at zero.

```
			position.y = position.x * position.x;
```



Y equals X squared.

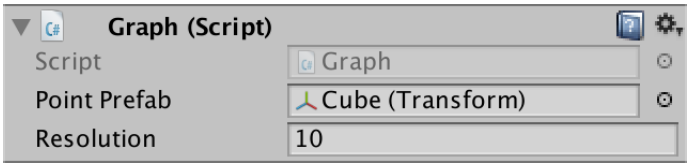
2 Creating More Cubes

Although we have a functional graph at this point, it is ugly. Because we're only using ten cubes, the suggested line looks very blocky and discrete. It would look better if we used more and smaller cubes.

2.1 Variable Resolution

Instead of using a fixed amount of cubes, we can make it configurable. To make this possible, add a public integer field for the resolution to `Graph`. Give it a default of 10, which is what we're using now.

```
	public int resolution = 10;
```



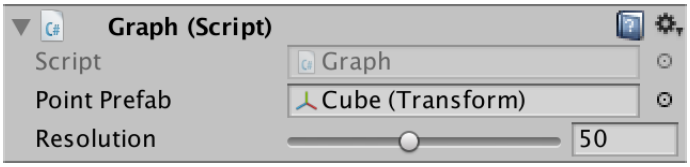
Resolution number.

The idea is that we can adjust the graph's resolution by changing this value, which can be done via the inspector. However, not all integers are valid resolutions. At minimum, they have to be positive. We can instruct the inspector to enforce a range for our resolution. This is done by writing `Range` in between square brackets before the definition of the field.

```
	[Range] public int resolution = 10;
```

`Range` is an attribute type defined by Unity. An attribute is a way to attach metadata to code structures, in this case a field. Unity's inspector checks whether a field has a `Range` attribute attached to it. If so, it will use a slider instead of the default input field for numbers. However, to do this it needs to know the allowed range. So `Range` has two parameters, for the minimum and maximum value. Let's use 10 and 100. Also, attributes are typically written above instead of in front of what they're attached to.

```
	[Range(10, 100)]
	public int resolution = 10;
```



Resolution slider.

Does this guarantee that `resolution` is limited to 10–100?

No, all it does is instruct the inspector to use a slider with that range. It doesn't affect `resolution` in any other way. So you could write code to assign it any integer you like. In the case of this tutorial, we assume that the resolution is only adjusted via the inspector, nowhere else.

2.2 Variable Instantiation

To actually use the resolution, we have to change how many cubes we instantiate. Instead of looping a fixed amount of times in `Awake`, the amount of iterations is now constrained by the resolution. So if the resolution is set to 50, we'll create 50 cubes after entering play mode.

```
		for (int i = 0; i < resolution; i++) {
			…
		}
```

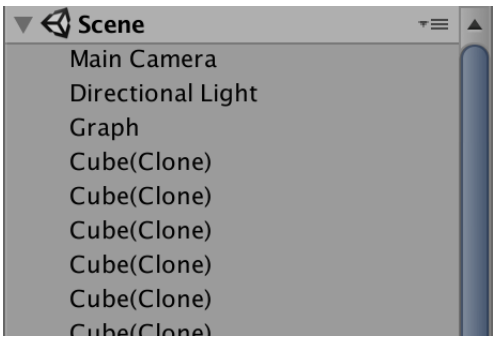
We also have to adjust the scale and positions of the cubes to keep them inside the −1–1 domain. The size of each step that we have to make per iteration is now $\frac{2}{resolution}$ instead of always $\frac{1}{5}$. Store this value in a variable and use it to calculate the scale of the cubes and their X coordinates.

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

Using resolution 50.

2.3 Setting the Parent

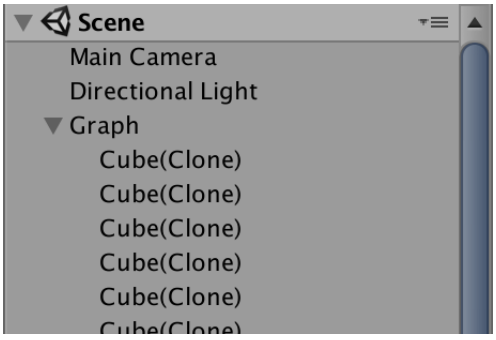
After entering mode with resolution 50, a lot of instantiated cubes show up in the scene, and thus also the scene view.



Many root objects.

These cubes are currently root objects, but it makes sense for them to be children of the graph object. We can set up this relationship after instantiating a cube, by invoking the `SetParent` method of the cube's **`Transform`** component. We have to supply it the **`Transform`** component of its new parent. We can directly access the graph object's **`Transform`** component via its `transform` property, which **`Graph`** inherited.

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



Child objects of Graph.

When a new parent is set, Unity will attempt to keep the object at its original world position, rotation, and scale. In our case, we don't need that. We can signal this by supplying **`false`** as a second argument to `SetParent`.

```
			point.SetParent(transform, false);
```

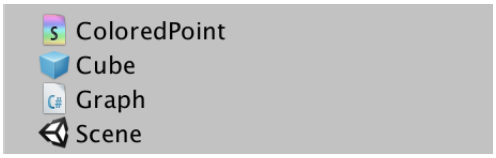
3 Coloring the Graph

A white graph isn't pretty to look at. We could use another solid color, but that's isn't very interesting either. What we could do is use a point's position to determine its color.

A straightforward way to adjust the color of each cube would be to set the color property of its material. We can do this in the loop. As each cube will get a different color, this means that we would end up with one unique material per object. While this works, it isn't very efficient. It would be must simpler if we could use a single material that directly uses the position as its color. Unfortunately, Unity doesn't have such a material. So let's make our own.

3.1 Creating a Custom Shader

The GPU runs shader programs to render 3D objects. Unity's material assets determine which shader is used, and allows its properties to be configured. We need to create a custom shader to get the functionality that we want. Create one via *Assets / Create / Shader / Standard Surface Shader* and name it *ColoredPoint*.



Custom shader asset.

We now have a shader asset, which you can open like a script. Our shader file contains code to define a surface shader, which uses different syntax than C#. Below are the contents of the file, with all comment lines removed for brevity.

```
Shader "Custom/ColoredPoint" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
		_Metallic ("Metallic", Range(0,1)) = 0.0
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		#pragma surface surf Standard fullforwardshadows

		#pragma target 3.0

		sampler2D _MainTex;

		struct Input {
			float2 uv_MainTex;
		};

		half _Glossiness;
		half _Metallic;
		fixed4 _Color;

		UNITY_INSTANCING_CBUFFER_START(Props)
		UNITY_INSTANCING_CBUFFER_END

		void surf (Input IN, inout SurfaceOutputStandard o) {
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
			o.Albedo = c.rgb;
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

How do surface shaders work?

Unity provides a framework to quickly generate shaders that perform default lighting calculations, which you can influence by adjusting certain values. Such shaders are known as surface shaders. If you want to know more about shaders, you can go through the Rendering tutorials series.

Our new shader has properties for a solid color, a texture, plus how glossy and metallic the surface should be. As we'll base the color on a point's position, we don't need the solid color nor the texture. The below code has all unneeded bits removed, leaving albedo as solid black and using an alpha of 1.

```
Shader "Custom/ColoredPoint" {
    Properties {
//        _Color ("Color", Color) = (1,1,1,1)
//        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Standard fullforwardshadows

        #pragma target 3.0

//        sampler2D _MainTex;

        struct Input {
//            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
//        fixed4 _Color;

        UNITY_INSTANCING_CBUFFER_START(Props)
        UNITY_INSTANCING_CBUFFER_END

        void surf (Input IN, inout SurfaceOutputStandard o) {
//            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
//            o.Albedo = c.rgb;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
//            o.Alpha = c.a;
            o.Alpha = 1;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

What's albedo and alpha?

The color of the diffuse reflectivity of a material is known as its albedo. Albedo is Latin for whiteness. It describes how much of the red, green, and blue color channels are diffusely reflected. The rest is absorbed.

Alpha is used as a measure of opacity. At alpha 0 a surface is fully transparent, while at alpha 1 it is fully opaque.

At this point the shader doesn't compile, because surface shaders cannot work with an empty input structure. This is where we determine what custom data is needed to color pixels. In our case, we need the position of the point. We can access the world position by adding `float3 worldPos;` to the input.

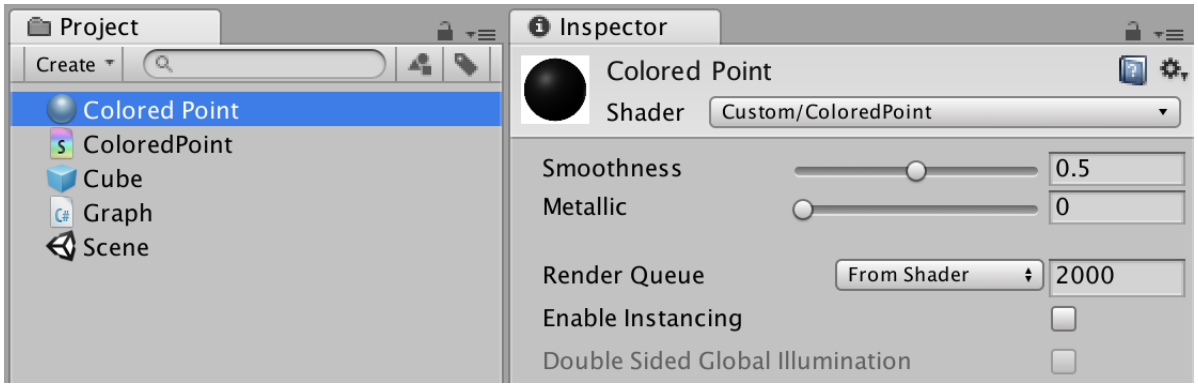
```
		struct Input {
			float3 worldPos;
		};
```

Does this mean that moving the graph would affect its color?

Yes. With this approach, the coloring will only be correct as long as the graph sits at the origin.

Also note that this position is determined per vertex. In our case, that is for each corner of a cube. The color will be interpolated across the cube's faces. The larger the cubes, the more obvious this color transition will be.

Now that we have a functioning shader, create a material for it, named *Colored Point*. Set it to use our shader, by selecting *Custom / Colored Point* via its *Shader* dropdown list.



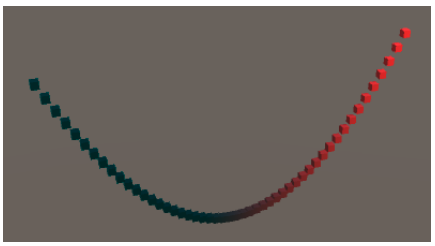
Material for colored points.

Have our *Cube* prefab asset use this material, instead of the default one. You can do this by dragging the material directly onto the prefab asset.

3.2 Coloring Based on World Position

When entering play mode, our graph will now instantiate black cubes. To give them a color, we have to change the `o.Albedo` variable. Instead of setting it to zero, make its red component equal to the X coordinate.

```
//			o.Albedo = 0;
			o.Albedo.r = IN.worldPos.x;
```

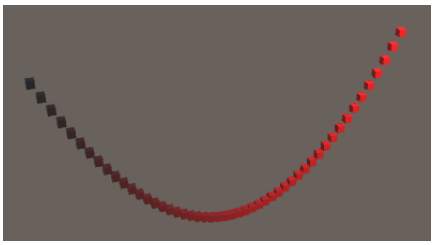


Colored graph based on X.

> **Shouldn't we initialize `o.Albedo` completely?**
>
> We don't need to set its green or blue components, because they've already been set to zero before the `surf` code gets invoked.

Cubes with positive X coordinates now become increasingly red. Those with negative X coordinates remain black, because colors cannot be negative. To get a red transition from −1 to 1, we have to halve the X coordinates and add 0.5.

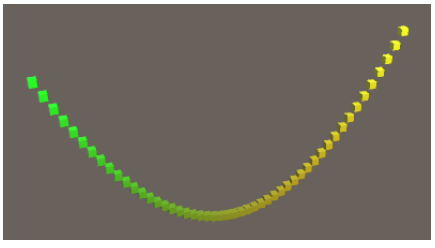
```
		o.Albedo.r = IN.worldPos.x * 0.5 + 0.5;
```



Full transition.

Let's also use the Y coordinate for the green color component, in the same way as X. In the shader, we can do this in a single line by using `IN.worldPos.xy` and assigning to `o.Albedo.rg`.

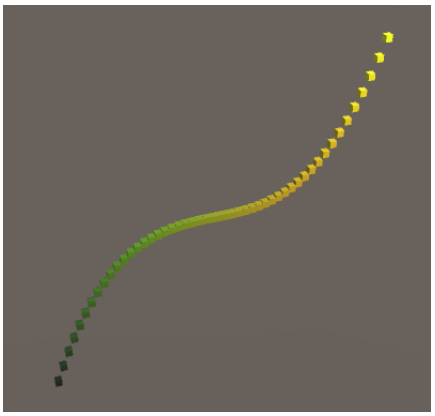
```
		o.Albedo.rg = IN.worldPos.xy * 0.5 + 0.5;
```



Coloring with both X and Y.

Red plus green becomes yellow, so our graph currently goes from light green to yellow. Had the Y coordinates started at -1, we would've gotten dark green colors as well. To see this, change the code in `Graph.Awake` so it displays the function $f(x) = x^3$.

```
		position.y = position.x * position.x * position.x;
```



Y going from −1 to 1.

4 Animating the Graph

Displaying a static graph is useful, but a moving graph is more interesting to look at. So let's add support for animating functions. This is done by including time as an additional function parameter, using functions of the form $f(x, t)$ instead of just $f(x)$, where t is the time.

4.1 Keeping Track of the Points

To animate the graph, we'll have to adjust its points as time progresses. We could do this by deleting all points and creating new ones each update, but that's an inefficient way to do this. It's much better to keep using the same points, adjusting their positions each update. To make this easy, we're going to use a field to keep a reference to our points. Add a `points` field to `Graph` of type `Transform`.

```
	Transform points;
```

This field allows us to reference a single point, but we need access to all of them. To make this possible, we need to use an array of points. We can turn our field into an array by putting empty square brackets behind its type.

```
	Transform[] points;
```

The `points` field can now refer to an array, whose elements are of type `Transform`. Arrays are objects, not simple values. We have to explicitly create such an object and make our field reference it. This is done by writing `new` followed by the array type, so `new Transform[]` in our case. Create the array in `Awake`, before our loop, and assign it to `points`.

```
		points = new Transform[];
		for (int i = 0; i < resolution; i++) {
			…
		}
```

When creating an array, you have to specify its size. This defines how many elements it has, which cannot be changed after it has been created. This length is written inside the square brackets when constructing the array. In our case, its length is equal to the resolution.

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

Now we can fill the array with references to our points. Accessing an array element is done by writing its index between square brackets behind the array field or variable. Array indices start at zero for the first element, just like the iteration counter of our loop. So we can use that to access the appropriate array element.

```
		points = new Transform[resolution];
		for (int i = 0; i < resolution; i++) {
			…
			points[i] = point;
		}
```

4.2 Updating the Points

To actually animate the graph, we need to set the Y coordinate of the points in the component's `Update` method. So we no longer need to calculate them in `Awake`, though we still need to use some explicit value, which should be zero.

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

Add an `Update` method with a `for` loop just like `Awake` has, but without any code in its block yet.

```
	void Update () {
		for (int i = 0; i < resolution; i++) {}
	}
```

We're going to loop through our array of points and set their Y coordinates. Because the array's length is the same as the resolution, we could also use that to constrain our loop. Each array has a `Length` property for this purpose, so let's use that.

```
		for (int i = 0; i < points.Length; i++) {}
```

Each iteration, we begin by getting a reference to the current array element. Then we retrieve that point's position.

```
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
		}
```

Now we can derive the position's Y coordinate, as we did earlier.

```
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
			position.y = position.x * position.x * position.x;
		}
```

Because vector's aren't objects, we only adjusted the local variable. To apply it to the point, we have to set its position again.

```
		for (int i = 0; i < points.Length; i++) {
			Transform point = points[i];
			Vector3 position = point.localPosition;
			position.y = position.x * position.x * position.x;
			point.localPosition = position;
		}
```

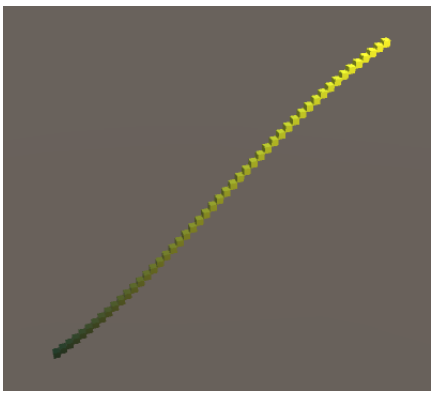
Couldn't we directly assign to `point.localPosition.y`?

If `localPosition` were a field, then this would be possible. We could directly set the Y coordinate of the point's position. However, `localPosition` is a property. It passes a vector to us, or accepts one from us. So we'd end up adjusting a local vector value, which doesn't affect the point's position at all. As we haven't explicitly stored it in a variable first, the operation is meaningless and will produce a compiler error.

4.3 Showing a Sine Wave

From now on, while in play mode, the points of our graph get positioned every frame. We just don't notice this, because they always end up at the same positions. We have to incorporate the time into the function in order to make it change. However, simply adding the time will cause the function to rise and quickly disappear out of view. To prevent this from happening, we have to use a function that changes but remains within a fixed range. The sine function is ideal for this, so we'll use $f(x) = \sin(x)$. We can use the `Sin` function of Unity's `Mathf` type to compute it.

```
			position.y = Mathf.Sin(position.x);
```

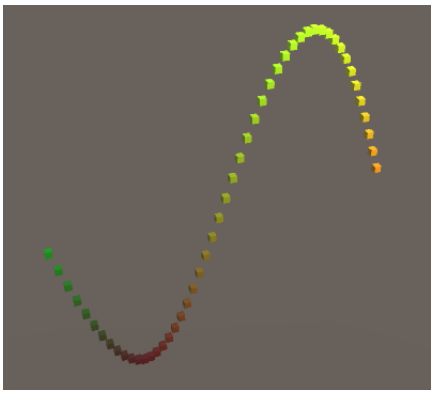


The sine of X.

> **What's `Mathf`?**
>
> It is a struct that contains a collection of mathematical functions and constants for working with numbers and vectors. As it works with floating-point numbers, it was given the *f* suffix.

The sine wave oscillates between −1 and 1. It repeats every 2π (pi) units, which is roughly 6.28. As our graph's X coordinates are between −1 and 1, we current see less than a third of the repeating pattern. To see it in its entirety, scale X by π so we end up with $f(x) = \sin(\pi x)$. We can use the `Mathf.PI` constant as an approximation of π.

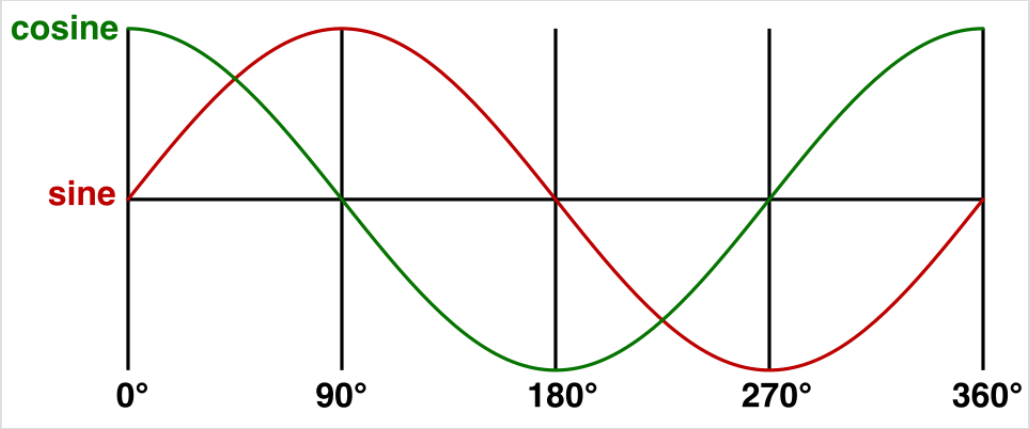
```
		position.y = Mathf.Sin(Mathf.PI * position.x);
```



The sine of πX.

What's a sine wave and π?

The sine is a trigonometric function, operating on an angle. In our case, the most useful example is a circle with radius 1, the unit circle. Each point on the circle has an angle θ (theta) associated with it, as well as a 2D position. One way to define the coordinates of those positions is $\begin{bmatrix} \sin(\theta) \\ \sin\left(\theta + \frac{\pi}{2}\right) \end{bmatrix}$. This represents starting at the top of the circle and going around it in a clockwise direction. Instead of $\sin\left(\theta + \frac{\pi}{2}\right)$ you can also use the cosine, leading to $\begin{bmatrix} \sin(\theta) \\ \cos(\theta) \end{bmatrix}$.

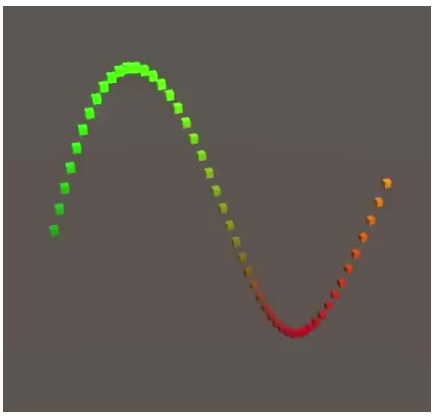


Sine and cosine.

The angle θ is expressed in radians, which corresponds to the distance traveled along the circumference of the unit circle. At the halfway point, 180°, you've traveled a distance equal to π, which is roughly 3.14. So the entire circumference has a length of 2π. In other words, π is the ratio between a circle's circumference and its diameter.

To animate this function, add the current game time to X before calculating the sine function. If we scale the time by π as well, the function will repeat every two seconds. So use $f(x, t) = \sin(\pi(x + t))$, were t is the elapsed game time. This will advance the sine wave as time progresses, shifting it in the negative X direction.

```
position.y = Mathf.Sin(Mathf.PI * (position.x + Time.time));
```



Animated function.

The next tutorial is Mathematical Surfaces.

Enjoying the tutorials? Are they useful? Want more?

Please support me on Patreon!

Or make a direct donation!

made by Jasper Flick