VECTROSITY

Version 4.4

Thank you for purchasing Vectrosity! The goal is to make line-drawing easy, flexible, and fast. You may want to dive right in and consult this documentation later to clear up the details. If so, have a look at the scripts in the **Scripts** folder in the **Vectrosity4Demos** Unitypackage. (Make sure to import either the **Vectrosity4** or **Vectrosity4Source** package before importing the VectrosityDemos package.) You should probably start with the **_Simple2DLine** and **_Simple3DObject** scenes, which contain some of the most basic functionality.

Note that Vectrosity 4 requires Unity 4.6 or later. If you're using Unity 4.0 through 4.5, you'll need to use the Vectrosity 3 files instead. This documentation applies to Vectrosity 4 only. If you have any questions you can contact sales@starscenesoftware.com.

See the Reference Guide for a complete list of all Vector and VectorManager functions and their parameters.

Vectrosity is available as a .dll or as source code. **Make sure you only import ONE of these two packages!**

- **Vectrosity4:** The core Vectrosity package for Vectrosity 4.0, which contains all Vectrosity scripts in a .dll. Since it's a .NET .dll, it will work with any Unity license, Free or Pro. Vectrosity 4.0 requires Unity 4.6 or later.
- **Vectrosity4Source:** This contains the core Vectrosity 4.0 scripts as source code. You'd generally want to use the .dll package instead, since .dlls are more convenient and can make script compiling faster, but the source is provided if you want to make modifications.

The following should be imported after you've imported one of the above packages. They aren't required, but they can be useful.

- **Vectrosity4EditorScripts:** The LineMaker editor script, which allows you to easily make vector lines from 3D objects.
- **Vectrosity4Demos:** Many demonstrations of various Vectrosity functions in a number of scenes. The demo scenes are located in the "Vectrosity/Demos/_Scenes" folder after importing the VectrosityDemos package. Some scenes have several related scripts — check the Main Camera object and enable/disable the ones you want.

There are several documentation files:

- **Vectrosity Documentation:** You're reading it! This explains the concepts of Vectrosity and includes various programming examples.
- **Vectrosity Reference Guide:** This is useful for quickly looking up information about the VectorLine and VectorManager classes and variables. It's probably most useful when you have some familiarity with how Vectrosity works and need a reminder or some extra details.
- **Vectrosity 4 Upgrade Guide:** Have a look at this if you're upgrading from an older version of Vectrosity to 4.0. It describes any changes that you'll need to make in order for your scripts to continue working.
- **Vectrosity Changelog:** Briefly explains the changes and new stuff in each version of Vectrosity.

If you're using Unity 4.0 - Unity 4.5, you'll need to use the files in the Vectrosity 3.1.2 folder. This documentation only applies to Vectrosity 4, so if you use Vectrosity 3, you should stop reading now and open the Vectrosity 3 documentation instead.

In order to use any Vectrosity functions, you must import the Vectrosity namespace. This means putting

```
import Vectrosity; // Javascript or Boo
```

or

```
using Vectrosity; // C#
```

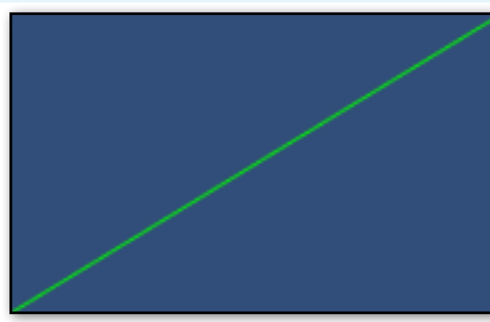
at the top of any script that uses Vectrosity. Note that most of the scripts in this documentation assume that you're importing the namespace. So if you get errors, make sure you've included this line first.

The simplest way to draw lines is with the **SetLine** command. This is similar to the Debug.DrawLine command, except it works in builds as well as the editor, and it's not limited to two points. SetLine takes a color, and two or more points. The points can be Vector2 for drawing lines in screen space, or Vector3 for drawing lines in world space. Create a new scene, then copy the following Unityscript or C# code into a script and save it:

```
import Vectrosity;    // Unityscript

function Start () {
   VectorLine.SetLine (Color.green, Vector2(0, 0), Vector2(Screen.width-1, Screen.height-1));
}
```

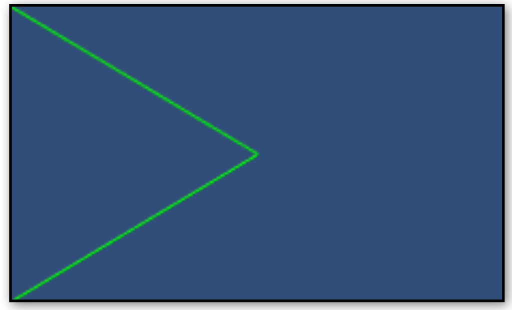
```
using UnityEngine;    // C#
using Vectrosity;

class LineTest : MonoBehaviour {
   void Start () {
      VectorLine.SetLine (Color.green, new Vector2(0, 0), new Vector2(Screen.width-1,
Screen.height-1));
   }
}
```

After attaching the script to the camera and clicking Play in Unity, this will result in a 1-pixel-thick green line that extends from the lower-left corner of the screen to the upper-right corner. Note that the script doesn't have to be attached to a camera; it can be attached to any object in the scene.



Every point you add in SetLine will create an additional line segment, so let's extend the above example:

```
VectorLine.SetLine (Color.green, new Vector2(0, 0), new Vector2(Screen.width/2,
Screen.height/2), new Vector2(0, Screen.height) );
```



This draws a line from the lower-left corner to the middle of the screen, and then to the upper-left corner, which results in the image on the right. You can keep adding more points if you like.

Another related command is **SetRay**. This essentially works like Debug.DrawRay, where you supply a color, a starting point, and a direction. Note that SetRay can only use Vector3 points for drawing in world space, unlike SetLine, which can either use Vector2 points for screen space or Vector3 points for world space. When attached to an object, this code will draw a line from its position to a point five units along its forward direction:

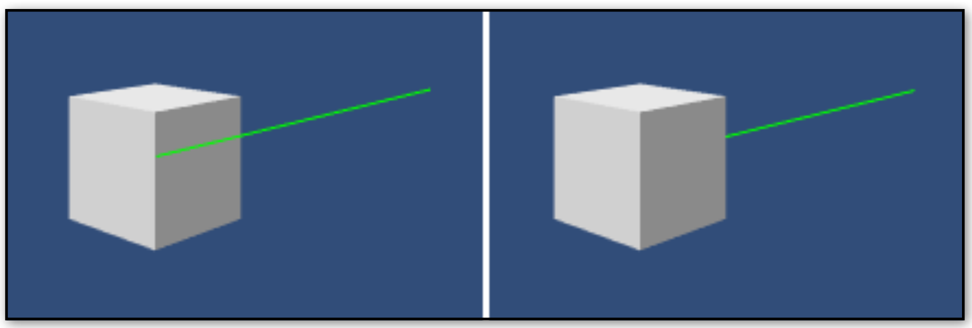
```
VectorLine.SetRay (Color.green, transform.position, transform.forward * 5);
```

Real 3D lines

If you have any other objects in the scene, you may notice that the lines are always drawn on top of everything. However, it's also possible to draw lines that actually exist in the scene and can be occluded by other 3D objects. To do that, you can use **SetLine3D** and **SetRay3D**:

```
VectorLine.SetRay3D (Color.green, transform.position, transform.forward * 5);
```

Here you can see the difference—on the left, the SetRay code is used on a cube, and on the right, the SetRay3D code is used, which means that the cube occludes the line.



Update and timing

Be careful about using these commands in Update! Every time you call SetLine or SetRay, it creates a new line, and unlike the standard behavior for Debug commands, these lines stick around permanently by default. If you do call SetLine or SetRay every frame in Update, your scene will quickly fill with hundreds of duplicate line objects.

One thing you can do instead is to pass in a time value. This will cause the line to be drawn for the given number of seconds and then be deleted. For example,

```
VectorLine.SetRay (Color.green, 3.0, transform.position, transform.forward * 5.0);
```

The above code makes the ray be drawn for 3 seconds, and then it disappears. This works for SetLine and the 3D variants as well:

```
VectorLine.SetLine (Color.green, 4.0, Vector2(0, 0), Vector2(250, 250));
```

VectorLine objects

But what if you want to change your lines later, or animate them in Update or coroutines? As it happens, SetLine returns a **VectorLine** object. A VectorLine is a type of object in Vectrosity that contains a bunch of information about lines. VectorLines can do many things besides drawing 1-pixel-thick lines, and they have a number of parameters. SetLine and SetRay are, in fact, really just shortcuts for creating basic VectorLine objects. The whole concept of Vectrosity actually involves creating VectorLines and then drawing them. Normally you create a VectorLine once, and if you want to change or animate it, you update the points that were used to create the VectorLine.

Since SetLine and SetRay return a VectorLine object when called, you can assign that to a variable, and change this variable wherever you need to. For example, the following script will first draw a line made from two random points, then every time you press the space key, it will add another point and re-draw the line:

```
import Vectrosity;

private var myLine : VectorLine;

function Start () {
   myLine = VectorLine.SetLine (Color.green, RandomPoint(), RandomPoint());
}
function RandomPoint () : Vector2 {
   return new Vector2(Random.Range(0, Screen.width), Random.Range(0, Screen.height));
}
function Update () {
   if (Input.GetKeyDown (KeyCode.Space)) {
      myLine.points2.Add (RandomPoint());
      myLine.Draw();
   }
}
```

The first thing we do is create a variable with a type of VectorLine as a global variable, so it can be referred to by other functions in the same script. Then, the Start function assigns the VectorLine returned by VectorLine.SetLine to the “myLine” variable. The RandomPoint function returns a Vector2 located at a random point on the screen; this is easier than writing it out every time. Finally, the Update function adds a new point and re-draws the line whenever the space key is pressed.

This VectorLine object contains a **points2** list, with one entry for each point that was defined when using SetLine. We used two points, so the points2 list has two entries initially. (Note that if you used Vector3 points instead of Vector2 when using SetLine, then you should use **points3** instead of points2.) In order to re-draw a line that already exists, you use **VectorLine.Draw**. If you used SetLine3D or SetRay3D and want to redraw those, then you can use **VectorLine.Draw3D** instead. Neither Draw nor Draw3D create new line objects — rather, they only re-draw existing VectorLine objects, so they are safe to use as often as you need.

And that’s it for the basics. There is much more you can do with line drawing; SetLine and SetRay are for very simple lines only. They’re good quick substitutes for Debug.DrawLine and Debug.DrawRay, but if you want do more of the advanced line effects that are possible with Vectrosity, you may prefer to create VectorLine objects directly. This is covered in detail in the next sections.

VectorLine

When creating a VectorLine object, in the simplest form you need to supply a name, an array or list of points, the material with which the line will be drawn (or null if you want Vectrosity to use its own material, as described in the **Material** section below), and the width of the line in pixels:

```
var myLine = VectorLine("MyLine", linePoints, lineMaterial, 2.0);       // Javascript
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f); // C#
```

That creates a line object called **MyLine**, which uses an array of points specified in **linePoints**, the material specified in **lineMaterial**, and is **2** pixels thick. By default, the line is white. This can be changed after the line is created by using the VectorLine.color property, described later.

This won't actually draw the line yet (see Drawing Lines for that); it just creates a VectorLine object that will be used to draw the line later. Remember that usually you create a line only once, and update the points later if you want to change the line in some way. The type of these objects is **VectorLine** (surprise!), so to declare a global VectorLine object, you can do this:

```
var myLine : VectorLine; // JS
VectorLine myLine;       // C#
```

You can use type inferencing when declaring VectorLine objects, such as the example code at the top of this page — in that case myLine is inferred as type VectorLine. Note that type inferencing is not the same as dynamic typing: it occurs at compile-time, and has no effect on code execution speed. However, if you're declaring the type explicitly for style reasons, code without type inferencing would be written like this:

```
var myLine : VectorLine = VectorLine("MyLine", linePoints, lineMaterial, 2.0); // JS
VectorLine myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f);  // C#
```

Name

The name is primarily a debugging aid, since VectorLine objects get added to the scene at runtime, and it would be confusing if every line was just called "GameObject". Also, the name is associated with bounds meshes created when using ObjectSetup (see the VectorManager section below), so it's a good idea to use different names for different VectorLine objects. Finally, you can access VectorLine.name if needed. For example, "Debug.Log (myLine.name);". If you change the name of a VectorLine, the object name also changes.

Line points

In order to set up a line, you need an array or a generic List. The array or list can be either Vector2 or Vector3. When using Vector2 points, Vectrosity uses **screen space** for line coordinates. In screen space, (0, 0) is the bottom-left corner, and (Screen.width-1, Screen.height-1) is the upper-right corner. Input.mousePosition, for example, uses screen space. When using Vector3 points, Vectrosity uses **world space** for line coordinates, like normal 3D objects do. It's also possible to use Vector2 for **viewport space** coordinates, where (0.0, 0.0) is the bottom-left corner, and (1.0, 1.0) is the upper-right corner, regardless of the screen resolution. Details about viewport coords are covered in **3D Lines and Viewport Lines**.

You can populate the array or List with points before using it to create a VectorLine, in which case the VectorLine will use the supplied points. You can also supply an empty array or a new list, and adds points later, after the VectorLine is created.

So, first let's create an array supplied with points:

```
var linePoints = [ Vector2(20, 30), Vector2(100, 50) ];            // JS
Vector2 linePoints = { new Vector2(20, 30), new Vector2(100, 50) }; // C#
```

You can have a maximum of 32,765 points per line when using discrete lines (the default), or 16,383 points per line when using continuous lines (see below for the difference between discrete and continuous lines). You'll get an error when declaring a VectorLine if you try to exceed the maximum. This code creates a VectorLine with the supplied points and draws it:

```
var myLine = VectorLine("MyLine", linePoints, null, 2.0);         // JS
var myLine = new VectorLine("MyLine", linePoints, null, 2.0f);    // C#
myLine.Draw();
```

You can use a generic List instead:

```
var linePoints = new List.<Vector2>(); // JS
linePoints.Add (Vector2(20, 30));
linePoints.Add (Vector2(100, 50));
var myLine = VectorLine("MyLine", linePoints, null, 2.0);

var linePoints = new List<Vector2>();  // C#
linePoints.Add (new Vector2(20, 30));
linePoints.Add (new Vector2(100, 50));
var myLine = new VectorLine("MyLine", linePoints, null, 2.0f);
```

If you're going to add points later and don't want to supply any first, you can declare the array or list in-line:

```
var myLine = VectorLine("MyLine", new Vector2[0], null, 2.0);          // JS
var myLine = VectorLine("MyLine", new List.<Vector2>(), null, 2.0);
var myLine = new VectorLine("MyLine", new Vector2[0], null, 2.0f);     // C#
var myLine = new VectorLine("MyLine", new List<Vector2>(), null, 2.0f);
```

In this case you'd refer to VectorLine.points2 (or VectorLine.points3 if using Vector3 points):

```
myLine.points2.Add (new Vector2(20, 30));
myLine.points2.Add (new Vector2(100, 50));
```

You can also use other generic List functions such as RemoveAt, AddRange, etc. Regardless of whether you use an array or a list when creating the VectorLine, line.points2 or line.points3 is always a generic List.

Note that if you supply a populated array instead of a List when creating the VectorLine, the points are copied to the points2 or points3 list, and the array is no longer referenced. If you supply a generic List instead of an array, the points2 or points3 list will be a reference to the list you supplied, so making changes in either your list or VectorLine.points2/points3 will have the same effect.

Material

Another thing you need is a material for the lines. If you just need a fairly basic line with no texture, you can pass in “null” instead of a material, and Vectrosity will create a suitable material for you and use that:

```
var myLine = VectorLine("MyLine", linePoints, null, 2.0);      // JS
var myLine = new VectorLine("MyLine", linePoints, null, 2.0f); // C#
```

Otherwise, if you need something fancier, you can supply your own material:

```
var myMaterial : Material;  // JS

function Start () {
    var myLine = VectorLine("MyLine", linePoints, myMaterial, 2.0);
}
```

```
public Material myMaterial; // C#

void Start () {
    var myLine = new VectorLine("MyLine", linePoints, myMaterial, 2.0f);
}
```

The material can use any kind of shader, but if you want to use line segment colors, the shader will need to use vertex colors. Many of the built-in particle shaders are good for this, such as Particles/Additive, or Particles/Additive (Soft), and the sprite and UI shaders use vertex colors as well.

If you want the line to react to lighting, so that it uses a shader such as the standard Diffuse shader, you will need to use the VectorLine.AddNormals function after the line is created. Similarly, if you want to use a shader that uses normal mapping, you can use the VectorLine.AddTangents function. Note that any lines that use lighting will need to be rendered by an OverlayCamera canvas instead of the default Overlay canvas; see Canvas and Camera for more details.

Width

The width is simply the number of pixels wide the line will be. Note that this is a float; it’s fine to have a line width of 2.5, for example.

And that’s it for the necessary parameters. There are two optional parameters, LineType and Joins, covered below.

LineType

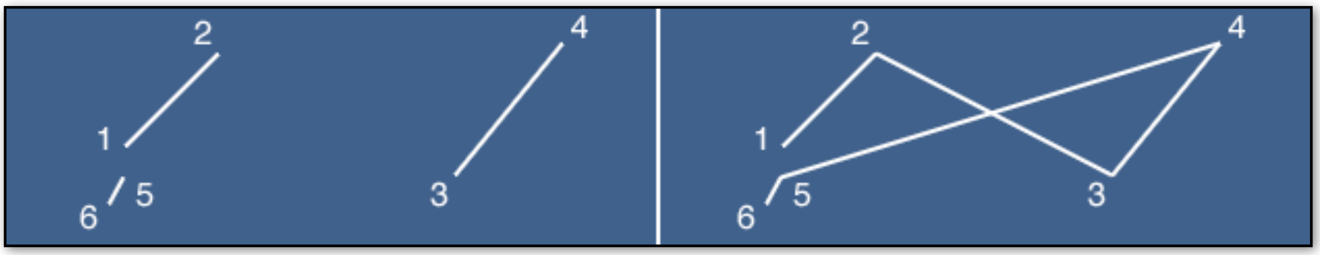
Lines in Vectrosity can be drawn in two different ways: discrete and continuous. The default is discrete, though it can be supplied explicitly by using **LineType.Discrete**:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f, LineType.Discrete);
```

To make a continuous line, use **LineType.Continuous**:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f, LineType.Continuous);
```

A discrete line means each line segment is drawn individually, and is defined by two points. By contrast, continuous lines are drawn with all the points connected sequentially: the second point of any line segment is always the first point of the next line segment.



Left: a discrete line made of 6 points. Right: the same line drawn as continuous.

So with discrete lines, it's possible to have many "separate" lines contained within a single VectorLine. Which type you use depends largely on what you intend to do with the line...a script that allows users to draw lines on the screen, for example, would make more sense as a continuous line. Some Vectrosity functions, such as MakeWireframe, require discrete lines, since not all line segments would be necessarily connected.

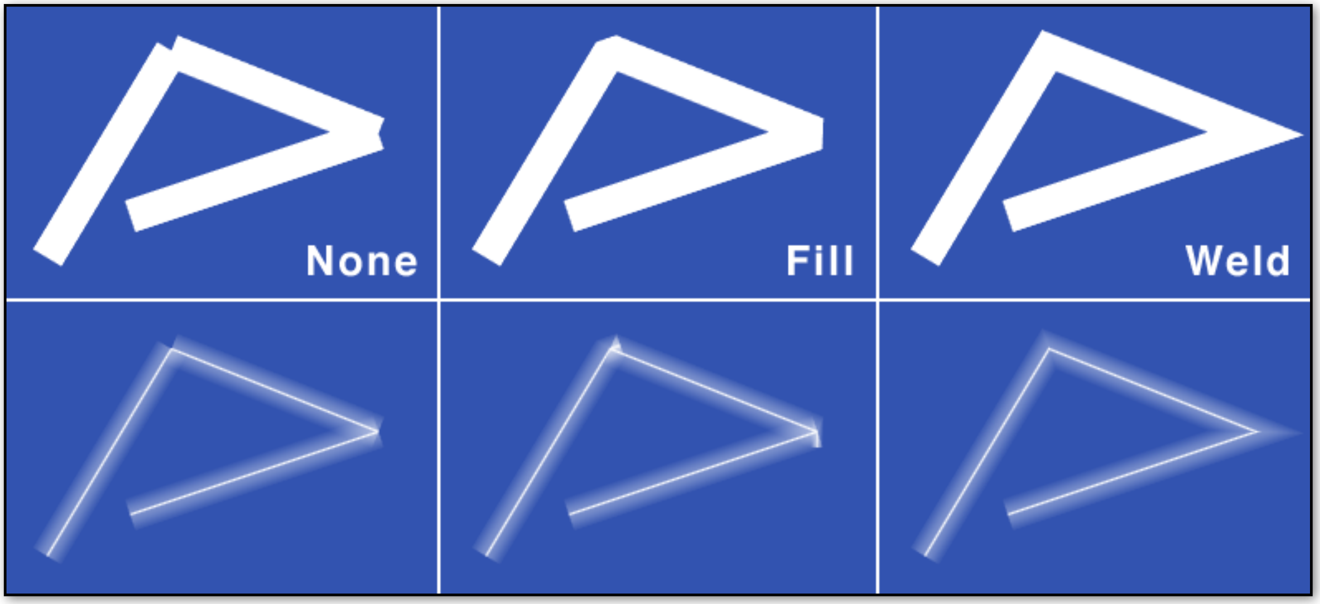
Since discrete lines take two Vector2 or Vector3 points to describe each line segment, they must use a points array with an even length. Continuous lines can have a points array of any size. They can also use Joins.Fill (see below), whereas discrete lines can't.

Joins

Each line segment is, by default, drawn as a simple rectangle. If you're using thin VectorLines — up to about 3 pixels thick — this usually works well. Thicker lines, however, can pose a bit of a problem, since the rectangular nature of each line segment becomes apparent when segments are joined at an angle, resulting in ugly gaps. Therefore, there are several options for the Joins parameter, which affects how segments are joined together. The options are **Joins.Fill**, **Joins.Weld**, and **Joins.None** (which is the default, so it doesn't have to be explicitly supplied).

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f,
                             LineType.Continuous, Joins.Fill);
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f,
                             LineType.Continuous, Joins.Weld);
```

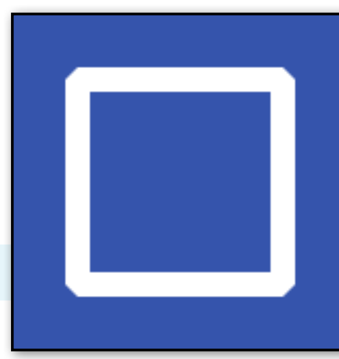
The effects of different types of joins, both with and without a texture, are shown below:



Clearly, Joins.None isn't usually appropriate for thick lines.

Next up, Joins.Fill is a good choice for solid-colored lines. It fills in gaps nicely, although it's somewhat slower than Joins.None since additional quads have to be used. Also, you can see artifacts at the joins when used with textures, and it only works with continuous lines, not discrete.

Finally, Joins.Weld is often a good choice for lines with textures. The vertices of sequential line segments are welded, which prevents texture artifacts. Also, it works with discrete lines as well as continuous. (With the limitation that only sequential line segments are affected — for example, if the ending point of line segment 4 is exactly the same as the starting point of line segment 5, then those two segments will be welded. If, however, the end of line segment 4 is the same as the start of line segment 7, those segments won't be welded, even though they are visually connected on-screen.) The drawback is that it's slower to draw compared to Joins.None or Joins.Fill, since there are some extra calculations that have to be done. Also, in certain cases the weld for a certain line segment can be cancelled; see the next page for info on that.



Both Joins.Fill and Joins.None will connect the first and last points if they are the same. In other words, if the first entry in the line points array and the last entry are identical. This allows you to make a closed shape (circle, square, etc.), and all the joins will be filled appropriately, as shown on the right.

Note that the type of joins can be changed at any time after a line is created:

```
myLine.joins = Joins.Weld;
```

Remember that discrete lines can't use Joins.Fill, so nothing will happen if you try to set a line to Joins.Fill in that case. See the DrawLines example scene in the VectrosityDemos package for an interactive illustration of the different types of joins. You can typically expect Joins.Fill to be about 50% slower than Joins.None, and Joins.Weld to be about half the speed (100% slower) compared to Joins.None.

Using maxWeldDistance

As mentioned above, the weld for certain line segments when using Joins.Weld can be cancelled. Namely, as line segments get closer to being parallel, the weld points extend farther and farther from the actual line segment joint:

This can lead to unwanted artifacts in some of the more extreme cases. To prevent that, VectorLines have a maxWeldDistance property, which looks at the weld point and cancels the weld operation for a particular line segment if the weld distance is too far:

By default, this distance is twice the pixel width that the VectorLine was created with. So a line with a pixel width of 20, for example, will have a default maxWeldDistance of 40. This can be changed at any time after the line is created, like so:

```
myLine.maxWeldDistance = 100;
```

The smaller the maxWeldDistance, the smaller the angle at which the weld will be cancelled.

Additional options

In addition to the name, points, material, width, LineType, and Joins, there are a number of other options that you can specify after the VectorLine object is declared. These are less frequently used than the parameters you supply when declaring a VectorLine, so for the sake of simplicity they aren't included in the VectorLine constructor. The additional options include: color, active, capLength, depth, layer, drawStart, drawEnd, and more. These are detailed in the Line Extras section below. But first, let's draw the line.

At last, we'll actually draw a line! This is pretty simple: **VectorLine.Draw**. Just add Draw() to the VectorLine object you've set up:

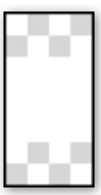
```
myLine.Draw();
```

Once a line is drawn, it's persistent, so VectorLine.Draw doesn't have to be called again unless the line changes in some way. So Draw can generally be called from pretty much anywhere — in Update if you're updating the line every frame, otherwise just where needed.

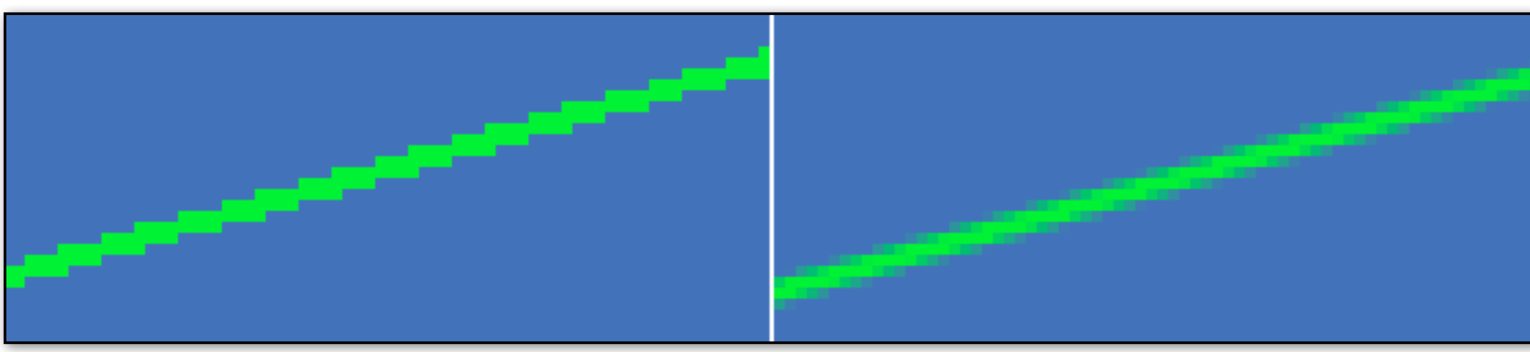
If you need to draw lines that exist in 3D space, then you can use **Draw3D**. See 3D Lines for more information on 3D line drawing. It's also possible to draw points instead of line segments. See Drawing Points for more information about that.

"Free" anti-aliasing

It's possible to get anti-aliased lines even if you don't have FSAA set in Unity. This is especially helpful on mobile devices, since FSAA might be too expensive there, depending on the device. To do this, you need to use a material that uses a shader that has alpha texture support, such as Particles/Additive, or Unlit/Transparent. Then you need a texture for this material that has transparent pixels at the top and bottom. The VectrosityDemos package has several sample line textures; in this case ThinLine and ThickLine are good for plain anti-aliased lines. ThinLine is simply a 2X4 pixel texture, with transparent pixels on the top and bottom and solid white in the middle:



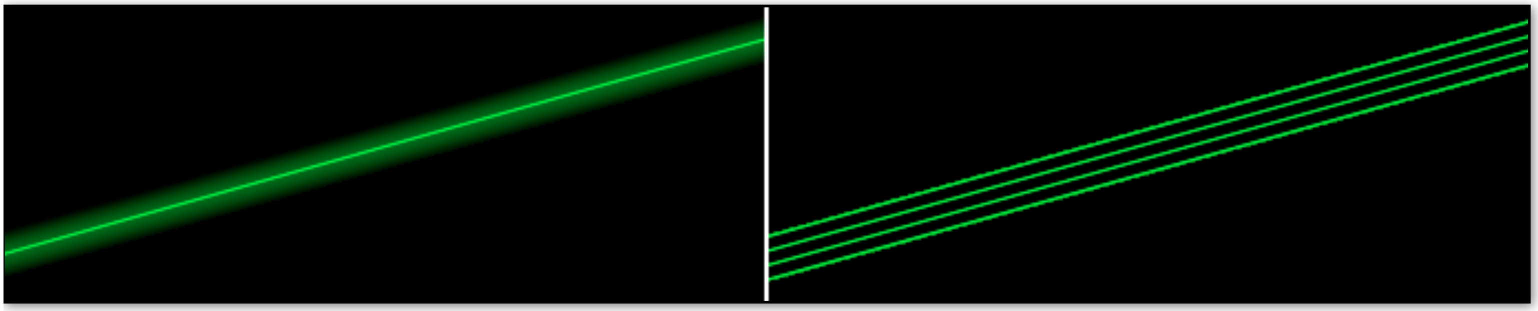
Make sure the texture is set to bilinear filtering and not point. The line anti-aliasing works by taking advantage of the inherent anti-aliasing you get when bilinear filtering is used, as illustrated below:



Left: a close-up of a line using the default material and no FSAA. Right: the same line using the Unlit/Transparent shader and the ThinLine texture, and still no FSAA. The AA comes from the texture itself as the result of bilinear filtering.

The catch here is that, when used with thicker lines, the transparent areas in the ThinLine texture will get stretched out and blurry, because the texture gets scaled up. Hence the ThickLine texture, which is taller and more appropriate for thicker lines. Also, you need to specify a thicker line width to account for the texture size. With the ThinLine texture, for example, the line width needs to be 4 in order to have the same apparent width that a normal, non-texture line would have with a width of 2.

You can use different textures to get different effects. For example, below are two lines using the GlowBig texture and the Bars texture respectively. Experiment with different textures of your own, too. Also see the Uniform-Scaled Textures section, which describes making dotted, dashed, and similar lines.



Vectrosity normally creates two canvases at runtime: one for standard lines drawn as an overlay (anything that uses Draw), and one for 3D lines that are drawn in the scene (anything that uses Draw3D). If needed, you can refer to these by using VectorLine.canvas and VectorLine.canvas3D, respectively.

For example, you might want to change the drawing order of the standard vector canvas compared to other canvases you have in your scene. This can be done by changing the sorting order of the canvas, where higher values are drawn on top of lower values. By default, the vector canvas has a sorting order of 1. So let's change it to -1, which would make a canvas with a sorting order of 0 be drawn on top of the vector canvas:

```
VectorLine.canvas.sortingOrder = -1;
```

Another thing you might want to do, depending on what you're doing with your scene, is change the canvas rendering mode from the default of Overlay to OverlayCamera. This involves changing canvas.renderMode and canvas.worldCamera. A shortcut is to use VectorLine.SetCanvasCamera:

```
VectorLine.SetCanvasCamera (Camera.main);
```

Now the vector canvas renders as OverlayCamera, with the rendering camera being Camera.main. Note that if you're using a line material that reacts to lighting, the camera should be set to OverlayCamera or else the lines may be invisible.

If you want to make VectorLines render to a texture, you can set up a camera using a rendertexture, and use that camera with SetCanvasCamera.

If you've used VectorLine.canvasID to create multiple canvases, then you can use VectorLine.canvases and VectorLine.canvases3D to reference lists of the canvases. In both cases the type is List<Canvas>, so for example VectorLine.canvases.Count would tell you how many vector canvases there are. Note that VectorLine.canvas is the same as VectorLine.canvases[0], and VectorLine.canvas3D is the same as VectorLine.canvases3D[0]. You can use this to change properties of individual vector canvases:

```
VectorLine.canvases[1].sortingOrder = -2;
```

The camera

Vectrosity needs a reference to the camera you're using in order to properly draw lines made with Vector3 points. (If you're only using Vector2 points, then you don't have to worry about this.) By default it uses Camera.main, which is the first camera found tagged MainCamera, so if this matches your setup, which is common, then you won't actually have to do anything.

In other cases, where you have a particular camera that you want to use with Vectrosity, you can use SetCamera3D:

```
VectorLine.SetCamera3D (myCamera);
```

Where "myCamera" is a Camera variable. This should be done before using Draw or Draw3D. It will cause any Vector3 lines to be oriented to that camera, and VectorLine.canvas3D (which uses RenderMode.World) will have its worldCamera set to your specified camera.

Note that if the camera is ever destroyed (such as through a scene change), you should call SetCamera3D again with the appropriate camera.

Updating lines

After you've created a VectorLine, you may want to alter some or all of the points to create animation or other effects. To do this, simply use the entries in the VectorLine.points2 or VectorLine.points3 lists, depending on whether you created the line with Vector2 or Vector3 points. In this example, a diagonal line is drawn, then flipped if you hit the space bar:

```
import Vectrosity;

private var myLine : VectorLine;

function Start () {
	myLine = new VectorLine ("Line", [Vector2(0, 0), Vector2(400, 400)], null, 2.0);
	myLine.Draw();
}

function Update () {
	if (Input.GetKeyDown (KeyCode.Space)) {
		myLine.points2[0] = Vector2(0, 400);
		myLine.points2[1] = Vector2(400, 0);
		myLine.Draw();
	}
}
```

Note that if the initial points used for the line are made with a generic List rather than an array, you can also refer to the List rather than VectorLine.points2. This is because points2 will be a reference to the List, so they are the same thing, but if you use an array, then the array is copied to points2 and no longer referred to. Since points2 and points3 are Lists, you can also add and remove points at will.

Drawing lines with a transform

Sometimes you might want to move an entire line around the screen. One possibility is to loop through all the elements in the points array and change each one, but that's kind of tedious. A better way is to specify the transform of an object when drawing lines.

You can move or rotate the entire line just by moving a transform—think of it as a proxy, or as a sort of parent, where altering the transform causes the line to be affected as well. Frequently an empty game object is useful for this. When you move the transform, the line mirrors the movement. This applies to the rotation and scale as well as the position. For example, if the transform is moved to position (5, 0, 0) and rotated to (0, 0, 45), then the associated line will also be offset 5 units on the X axis and rotated 45° around the Z axis. (What the units refer to depends on whether you're using 2D points or 3D points...2D points use pixels, whereas 3D points use world units.)

See the DrawLines scene in the VectrosityDemos package for an example of rotating a line left and right by using a transform. Another example of using a transform for special effects can be found in the TextDemo.js script.

To use a transform, add it using the drawTransform property after creating a line:

```
var myLine = new VectorLine("Line", linePoints, lineColors, null, lineWidth);
myLine.drawTransform = transform;
myLine.Draw();
```

The above example would use the transform component of whatever object the script it attached to. Note that if the transform is moved or changed in some way, you should call Draw again to update the line. You can use the transform of any object:

```
myLine.drawTransform = GameObject.Find("Some Object").transform;
```

If you're continuously updating a line with a transform every frame, it's often best to call VectorLine.Draw in LateUpdate rather than Update to make sure it's updated correctly, which is to say after the transform has moved.

Drawing lines with a matrix

If you have your own Matrix4x4 and don't want to make an empty game object just to get the transform, you can supply your own matrix. This works in the same way as a transform; it's just that you supply the matrix yourself directly.

```
var myLine = new VectorLine("Line", linePoints, lineColors, null, lineWidth);
var myMatrix = Matrix4x4.identity;
myLine.matrix = myMatrix;
myLine.Draw();
```

Advanced movement with rectTransform

Warning! In most cases it's not necessary to use this and doing so can cause Vectrosity to break. The information is included here for advanced users. If you have any doubts, don't use rectTransform; most things can be accomplished by using drawTransform instead, as described above.

That said, it's possible to move the line itself directly rather than changing line points or passing in a Transform. In this case you can use **VectorLine.rectTransform** to refer to the actual line GameObject's RectTransform component. The main reason you might want to use this is if you're using 1-pixel thick lines, which, since they are computed "between" pixels, can be rendered somewhat blurry if using FSAA. So you might want to offset the rectTransform by half a pixel to compensate:

```
myLine.rectTransform.anchoredPosition = new Vector2(0.5f, 0.5f);
```

Altering other properties such as eulerAngles can skew VectorLines and is generally not recommended.

Another good use for rectTransform is to parent lines to other objects, since line drawing order is controlled by the order in which lines are listed under the canvas. (See drawDepth in the Line Extras section.) For example, let's say you have two groups of lines and you want one group to always draw on top of another group. So you'd create two empty GameObjects and parent them to the vector canvas by using object1.transform.parent = VectorLine.canvas.transform, and the same for object2. Then you can parent VectorLines to these objects by using myLine.rectTransform.parent = object1.transform or object2.transform as desired.

Since some Unity objects are made when creating a VectorLine object, simply setting it to null won't remove those objects. Instead, you should use **VectorLine.Destroy**:

```
VectorLine.Destroy (myLine);     // JS
VectorLine.Destroy (ref myLine); // C#
```

This will properly dispose of the VectorLine and related objects. Null VectorLine objects are ignored, so if a line doesn't exist, it won't generate any null reference exception errors. As a convenience, you can also destroy a GameObject at the same time, which can be useful if you've been using drawTransform:

```
VectorLine.Destroy (myLine, gameObject);     // JS
VectorLine.Destroy (ref myLine, gameObject);// C#
```

This is almost the same as writing:

```
VectorLine.Destroy (myLine);
Destroy (gameObject);
```

The difference being that, as with null VectorLine objects, null GameObjects are ignored too.

If you have an array or generic List of VectorLines, you can pass that into Destroy and all the lines in the array or List will be destroyed:

```
var myLines = new VectorLine[10];
var myLines2 = new List.<VectorLine>();
for (var i = 0; i < myLines.Length; i++) {
    myLines[i] = new VectorLine("Line", new Vector2[2], null, 2.0);
    myLines2.Add (new VectorLine("Line", new Vector2[2], null, 2.0));
}
VectorLine.Destroy (myLines);
VecrorLine.Destroy (myLines2);
```

In the case of destroying arrays or Lists, you don't need "ref" if you're using C#.

Note: you should not use VectorLine.Destroy if you've used ObjectSetup (see the VectorManager section below). Instead, destroying the GameObject passed into ObjectSetup will also destroy the respective VectorLine automatically, so it's not necessary to manage VectorLines yourself.

As mentioned in the Setting Up Lines section, there are a number of additional options for lines, which you can set after the VectorLine is created.

Active

This used when you want to turn a line off, rather than destroying it, so you can turn it back on again later. Inactive lines aren't visible, and calling Draw or Draw3D with an inactive line won't do anything.

```
myLine.active = false; // Turns line off
myLine.active = true; // Turns line on
```

CanvasID

In some cases you may want to use more than one canvas for VectorLines. For example, a single canvas can have a maximum of approximately 65,000 vertices, so if you have several VectorLines with many points, it may not be possible to have them all use the same canvas. Assigning a number to VectorLine.canvasID will therefore cause the VectorLine to use the specified canvas (starting at 0). If the canvas doesn't exist for a particular ID, it will be created automatically. By default the canvasID is 0.

```
myLine.canvasID = 3;  // Puts the line on vector canvas 3 (which is the fourth canvas)
```

Note that there are separate sets of canvases for overlay lines made with Draw, and 3D lines drawn in the scene with Draw3D. So if you had some lines made with Draw using canvasIDs of 0 and 1, and other lines made with Draw3D that also used canvasIDs of 0 and 1, you would have a total of four vector canvases.

CapLength

This adds a given number of pixels to either end of each line segment. Primarily this is used for things like squaring off rectangular shapes:

Left: a 14-pixel-thick line with a segment cap length of 0, using Joins.None. Middle: the same line with a segment cap length of 7. Right: a segment cap length of 14.

Usually in this case you'd want to use exactly half the line width, but you can use different numbers for different effects. This uses floats, so if your line width is 3, you can use 1.5 for the segment cap. The effect in the middle panel in the above illustration could be achieved by using Joins.Weld instead, but using a segment cap is more efficient. Note that this only works with lines made with Vector2 points, and has no effect on lines made with Vector3 points. You set the segment cap length by using VectorLine.capLength after the VectorLine is created:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 14.0f);
myLine.capLength = 7.0f;
```

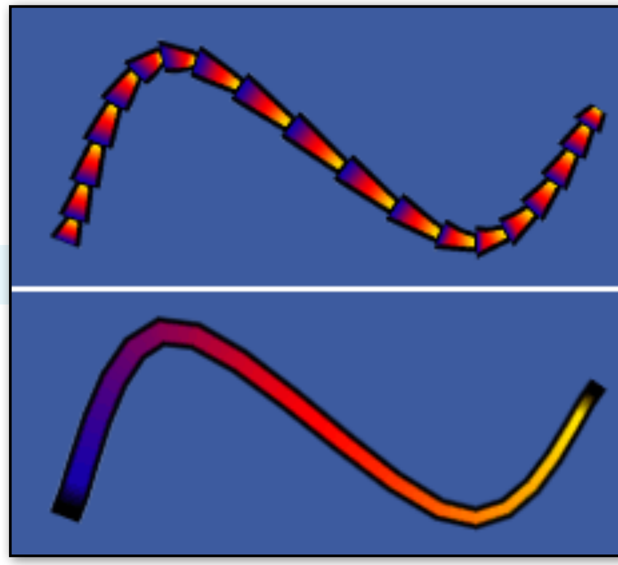
Color

A VectorLine is white by default. To change the color, you can set the color property:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f);
myLine.color = Color.red;
```

Besides changing the color of existing line segments, if you add more line points later, the additional line segments will use this color. See the Line Colors section for information about setting individual line segment colors.

ContinuousTexture



If you use a material with a texture, the default behavior for the texture is to repeat once for each line segment. By setting continuousTexture to true, you can make the texture stretch the entire length of the line.

```
myLine.continuousTexture = true;
```

In the image on the right, the line on the top shows the default behavior, while the line on the bottom has a continuous texture.

DrawDepth

This changes the order in which lines are drawn. By default, lines that are created later are drawn on top of lines that are created earlier. If you want to change the order, you can use **drawDepth**, where higher numbers are drawn on top of lower numbers. Note that changing the depth for one line may change the depth of others, since the numbers are always unique and in sequential order, and the highest depth is determined by the number of lines. In other words, if you had three lines, the possible numbers to use for drawDepth are 0, 1, and 2. Anything higher would be clamped to 2.

Let's say you have two lines, line1 and line2, where line1 was created first. This means line2 will draw on top of line 1—the default drawDepth values are 0 for line1 and 1 for line2. We'll change the order:

```
line1.drawDepth = 1;
Debug.Log (line2.drawDepth);
```

Now line1 is drawn on top, and the line2.drawDepth has changed to 0.

DrawStart and DrawEnd

If you want to draw only part of a line, you can set **drawStart** and **drawEnd**. This affects how the line is drawn on-screen, but leaves the points array untouched. You can use this for various effects such as animation. For example, if you had a line with 10 points, this code would cause only points 3-6 to be drawn, and the rest will be erased:

```
line.drawStart = 3;
line.drawEnd = 6;
line.Draw();
```

The drawStart and drawEnd properties work with both continuous and discrete lines. In the latter case, since it always takes two points to define a line segment, if you use an odd number for drawStart, it will be set to the next highest even number. Likewise, if you use an even number for drawEnd, it will be set to the next highest odd number. In any case, the start and end values are clamped between 0 and the maximum array index. See the PartialLine scene in the VectrosityDemos package for an illustration of using drawStart and drawEnd to animate a curved line without updating the points.

EndPointsUpdate

In some cases, you may be drawing lines where you add new points to the end, but the original points always stay the same. For example, a script where the user draws a line on-screen with the mouse. As an optimization, **VectorLine.endPointsUpdate** allows you to specify how many points from the end are actually updated when you call Draw or Draw3D. This allows Vectrosity to skip calculations for the rest of the points, which can result in improved performance. Once you set endPointsUpdate after declaring a VectorLine, only the specified number of points from the end are computed. In this example, just the last point will be updated:

```
myLine.endPointsUpdate = 1;
```

You can see an example of this in the DrawLinesMouse and DrawLinesTouch scripts included in the VectrosityDemos package.

LineWidth

You can change all segments in the line to a particular width at once, at any time after the line has been created, using **VectorLine.lineWidth**:

```
myLine.lineWidth = 4.0f;
```

It's also possible to change the widths of individual line segments using the SetWidth or SetWidths functions, described in [Vector Utilities](). Note that if you had been using multiple segment widths, using VectorLine.lineWidth will overwrite that and set all line segments to the specified number. You should call Draw or Draw3D after setting lineWidth in order to see the effect.

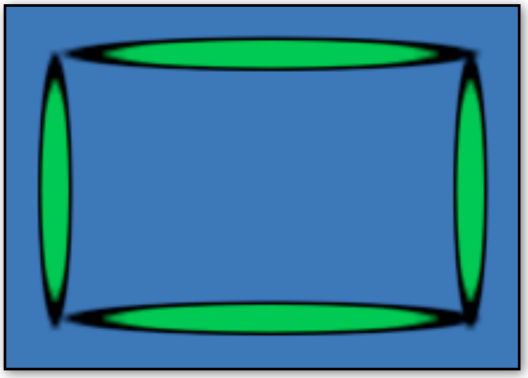
Material

If you want to change the material of a line at any point after it's been created, use VectorLine.material. This is pretty straightforward:

```
myLine.material = anotherMaterial;
```

You can also read VectorLine.material to get the material currently being used by the VectorLine.

So far you've seen various sorts of lines, where the textures used are stretched the length of each line segment (or the entire line, if you use VectorLine.continuousTexture). For standard solid-colored lines, this is exactly what you want. Sometimes, though, you'd prefer more flexibility, where a line has a repeating texture that's always scaled the same, regardless of how long an individual line segment might be. Picture dotted and dashed lines, for example. Consider this texture: . When used in a material to draw lines as usual, it will look like this:



That's interesting, but not what we want in this case. Here's where **VectorLine.textureScale** comes in. You set this property after creating a VectorLine. The basic format is this:

```
VectorLine.textureScale = myTextureScale;
```

Note that "myTextureScale" is a float. This is most commonly 1.0, but it can be anything:

```
VectorLine.textureScale = 2.5;
```

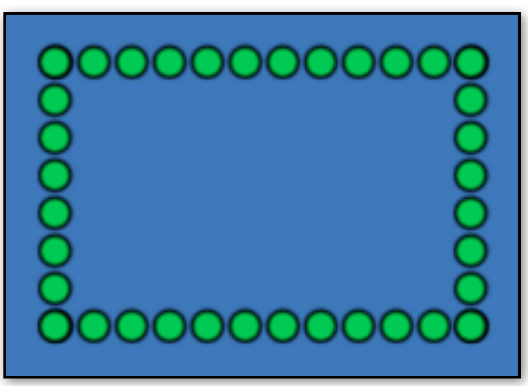
Let's set up a rectangle:

```
var lineMaterial : Material;

function Start () {
   var rectLine = new VectorLine("Rectangle", new Vector2[8], lineMaterial, 16.0);
   rectLine.capLength = 8.0;
   rectLine.MakeRect (Rect(100, 300, 176, 112));
   rectLine.Draw();
}
```

This code results in the above image, assuming a Material is used that contains the green dot texture. Now add another line of code, after rectLine is created but before calling Draw:

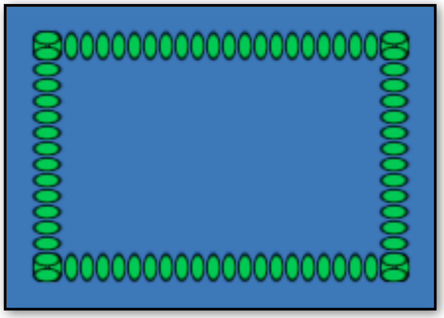
```
    rectLine.textureScale = 1.0;
```

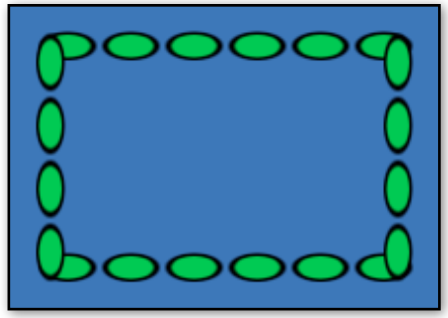
And we get this result instead:



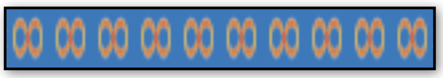
If the results aren't what you'd expect, make sure the texture is set to Repeat and not Clamp. If you change the textureScale property after the line is drawn, you'll need to call Draw or Draw3D again in order for the change to show up. (So yes, textureScale works with 3D lines as well as 2D lines.)

Using a textureScale of 1.0 means the texture is scaled horizontally so that its width is 1 times its height. If we used .5, it would be scaled to half its height, and 2.0 would scale it to twice its height:





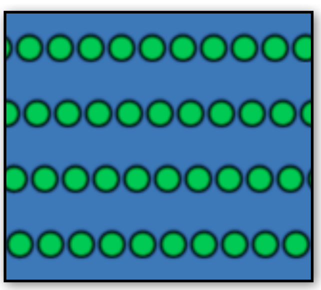
This is particularly useful for dashed lines, where you might want to make longer or shorter dashes, and it's also useful for non-square textures. This 32x16 texture, for example:  . A textureScale of 1.0 results in the left image, and using 2.0 results in the right image:



You can also optionally supply an offset:

```
myLine.textureOffset = .5;
```

This gets essentially the same results as you would get by specifying an x offset using renderer.material.mainTextureScale. However, it can be more convenient to specify it like this, and the offset is actually built into the line itself, rather than altering the material. You can even animate the offset by repeatedly setting VectorLine.textureOffset in the same way that you would using renderer.material.mainTextureScale. (Calling Draw or Draw3D after setting textureOffset isn't necessary). However, it's faster to set mainTextureScale, so normally you'd prefer to use that for animation where feasible. Using offsets of .25, .5, .75, and 1.0 (same as 0.0) for the offset would result in this:



For some more code examples of VectorLine.textureScale, see the "SelectionBox2" script (which also uses VectorLine.textureOffset) in the SelectionBox scene in the VectrosityDemos package, and the "DrawCurve" script in the Curve scene.

If you ever want to reset the texture scale of a line back to the default, you can set textureScale to 0 and redraw the line. This removes all custom texture scaling information from the VectorLine object:

```
myLine.textureScale = 0;
myLine.Draw();
```

iOS note: the GPU used in iOS devices tends not to render textures well if texture UVs are too far away from the 0.0 to 1.0 range. VectorLine.textureScale tries to maintain this automatically as much as possible, but if you see textures distorting when they're repeated many times over a long line segment, you may have to break the line segment up into shorter parts.

You can use end caps to make arrows, rounded ends, or otherwise differentiate the ends of the lines from the middle. To do this, you first need to call the **VectorLine.SetEndCap** function. You can have any number of different end caps, so think of SetEndCap as adding to a library. (This library only exists at runtime, not in your project.) Each end cap in the library has a different name, so make sure to use a unique name for each set. Note that you only need to set up each end cap once—after you've done so, that particular end cap will be available for all lines in any script. To actually make a particular line use an end cap, use VectorLine.endCap = "NameOfEndCap". (Using, of course, the actual name that you used with SetEndCap.)

End caps can be added to the front of the line, or the back, or both (or neither, in which case it's just a regular line). Additionally, the end cap at the front can appear at the back, but mirrored, which uses one texture instead of two. The "front" of the line is defined as the first point in the points array that makes up the line, and the "back" is the last point. In order to specify one of these options, you'd use the EndCap enum, which consists of: EndCap.Front, EndCap.Back, EndCap.Both, EndCap.Mirror, and EndCap.None. Note that when using EndCap.Both, the front and back textures must have the same width and height.

You'd also need to specify a material, which typically would be the same material used when setting up the VectorLine that you're planning on using with the end caps. This material should include the texture that will be used for the middle of the line. (Incidentally, you may want to use end caps in combination with VectorLine.continuousTexture, in case the front and end caps have different styles.)

Finally, you need to specify either one texture (if you're using EndCap.Front, Back, or Mirror), or two (if you're using EndCap.Both). To actually set the end caps for a line, use the **VectorLine.endCap** property. Here's an example that will result in a line that looks like an arrow:

```
var lineMaterial : Material;
var frontTex : Texture2D;
var backTex : Texture2D;

function Start () {
	VectorLine.SetEndCap ("Arrow", EndCap.Both, lineMaterial, frontTex, backTex);
	var points = [Vector2(100, 100), Vector2(200, 100)];
	var arrowLine = new VectorLine("ArrowLine", points, lineMaterial, 20.0);
	arrowLine.endCap = "Arrow";
	arrowLine.Draw();
}
```

You can try this out by using the "Arrow" material from the VectrosityDemos package, and the "arrowStart" and "arrowEnd" textures from the Textures/VectorTextures folder also in that package.

Or, if you change the material to the "ThickLine" material, change frontTex to the "endCap" texture, and change the code as shown below, then you'll get a line with rounded ends:

```
	VectorLine.SetEndCap ("RoundedEnd", EndCap.Mirror,
lineMaterial, frontTex);
```

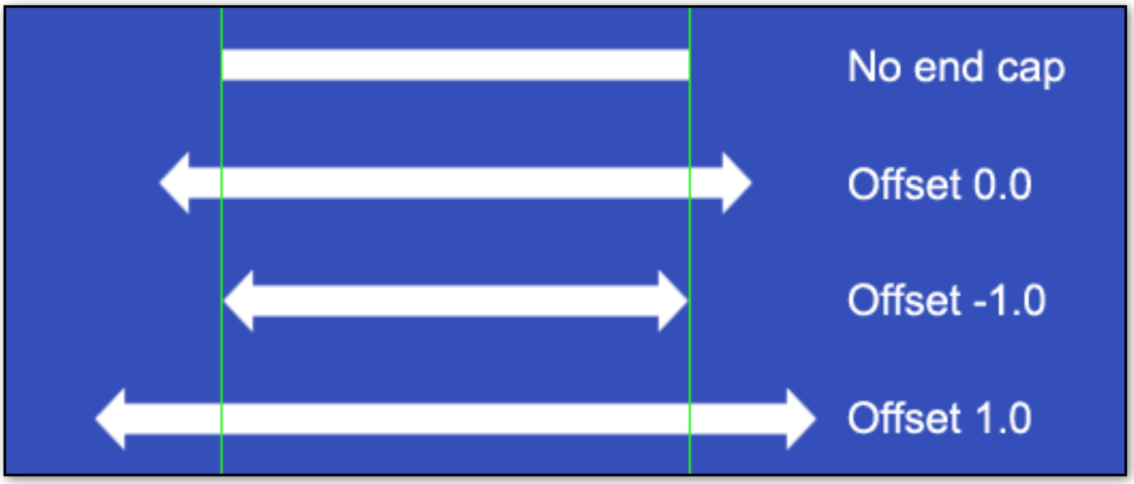
For another example of this, check out the DrawLinesTouch demo scene. Click on the Main Camera, then check "Use End Cap" on one of the scripts, and change the line width to something larger like 15.0. The EndCap demo scene also has several examples of lines with end caps.

End cap offset

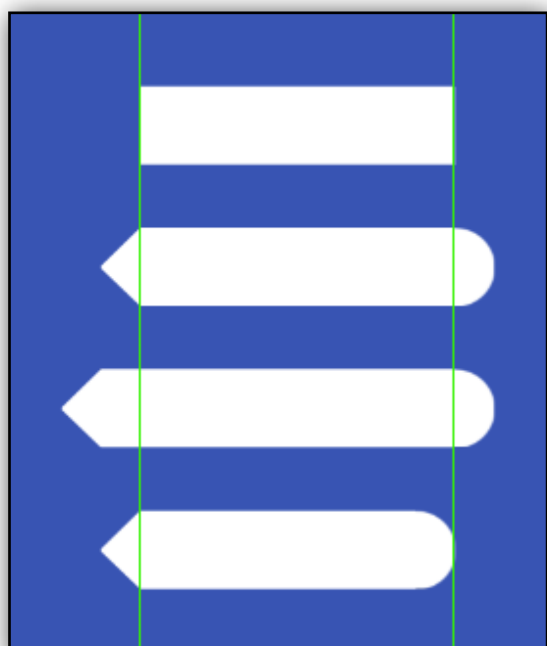
In certain cases you might want to move the end caps in or out further from the ends of the line. By default, the end caps are drawn extending from the ends of the line, or in other words they're added on. When you use SetEndCap, you can optionally specify a distance to move the end caps in or out. This offset distance is relative to the size of the end cap, so for example "1.0" means "the size of the end cap" and "0.5" means "half the size of the end cap". The offset, if desired, is added after the material:

```
VectorLine.SetEndCap ("Arrow2", EndCap.Mirror, lineMaterial, -1.0f, frontTex);
```

If left out, the default is 0. This illustration shows a line without end caps, followed by a line drawn with the default offset of 0, followed by an offset of -1 (so the line is as long as it would have been without the end cap), and finally an offset of 1 (so the line is additionally extended by the length of the end cap on each end):



Note that moving the offset inward on lines with many very short segments (such as a circle) can result in unexpected behavior. As long as the segment just before the end cap is at least as long as the end cap, though, you're good.



Sometimes you may want to have the front and back end caps offset by different amounts. In this case, supply two floats instead of one, where the first float is for the front cap and the second float is for the back cap. For example, this code uses an offset of 1.0 for the front and -1.0 for the back:

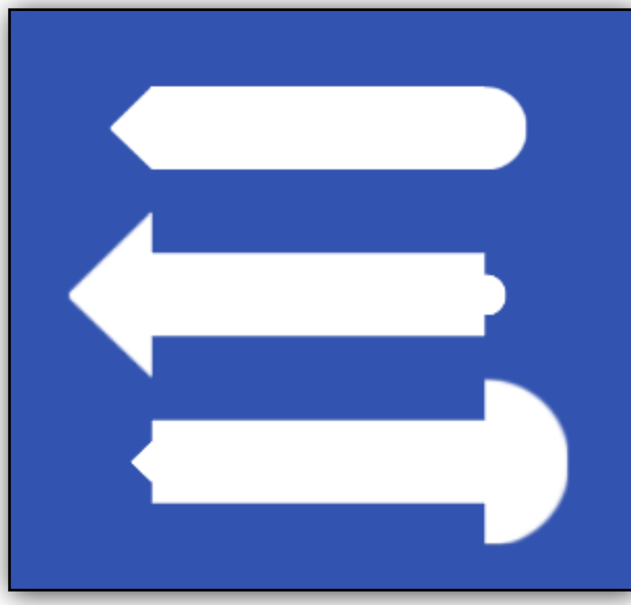
```
VectorLine.SetEndCap ("Arrow3", EndCap.Both,
lineMaterial, 1.0f, -1.0f, frontTex);
```

In the illustration on the left, the first line has no end cap, the second line has default offsets (0.0 for both caps), the third line has offsets of 1.0 and 0.0 (so the front is extended), and the fourth line has offsets of 0.0 and -1.0 (so the back is pulled in).

Scaling end caps

If you want either of the end caps to be larger or smaller than the line itself, you can supply an additional two floats, which indicate the relative size of the front and back caps, respectively. A scale of 1.0 means normal size, so for example 2.0 is twice normal and 0.5 is half normal. Note that when using scale values, you must supply both offsets, and both scales, even if you're using EndCap.Front or EndCap.Back. Here the offsets are both 0, and we set the front cap to be twice the size it would normally be, and the back cap to regular size:

```
VectorLine.SetEndCap ("Arrow4", EndCap.Both, lineMaterial, 0, 0, 2.0f, 1.0f, frontTex);
```



Top: front scale of 1.0 and back scale of 1.0. Middle: front scale of 2.0 and back scale of 0.5. Bottom: front scale of 0.5 and back scale of 2.0.

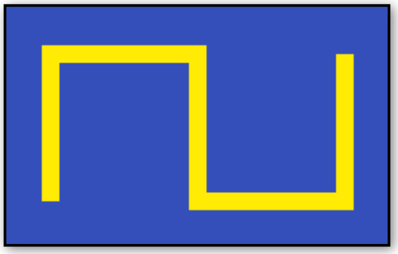
Removing end caps

To remove an end cap from a line, set VectorLine.endCap to null or "" (an empty string). If you want to remove an end cap from the library, you can either use VectorLine.RemoveEndCap, or else use SetEndCap with the appropriate name and EndCap.None.

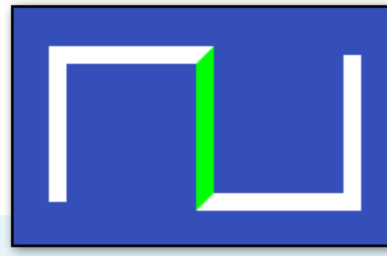
```
myLine.endCap = "";    // Removes the end cap from this line
```

Aside from VectorLine.color, you can use the **VectorLine.SetColor** or **VectorLine.SetColors** functions. These allow more flexibility since you can use them to set different colors on different line segments within the same line. If you just specify a color using SetColor, the entire line will change to that color:

```
myLine.SetColor (Color.yellow); // Change all line segments to yellow
```



So here we have a line made of 5 segments, which has been changed to yellow. However, unlike VectorLine.color, this only applies to existing line segments. Any additional line segments you add later will use VectorLine.color.

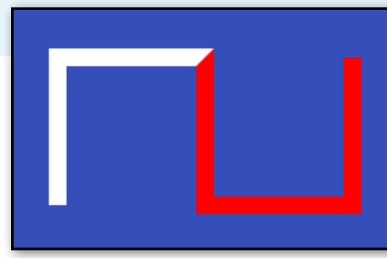


If you want to change just part of the line, you can supply an index which corresponds to the desired line segment. For example, to change the third segment to green (remember that you start at 0 when counting segments):

```
myLine.SetColor (Color.green, 2);
```

You can also change a range of segments by specifying the first and last indices:

```
myLine.SetColor (Color.red, 2, 4);
```

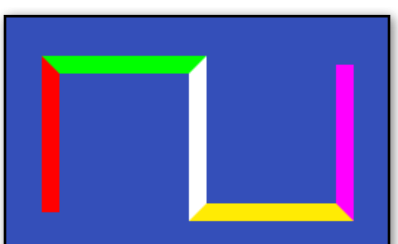


In this case, the segments starting with 2 up to and including 4 will be changed to red. Any indices that are out of bounds will be clamped to the maximum possible. So in the above example, if the maximum segment number for the line was 3, then segments 2-3 would be changed.

Note that SetColor actually uses Color32, but using Color.green and so on will work because Color implicitly converts to Color32.

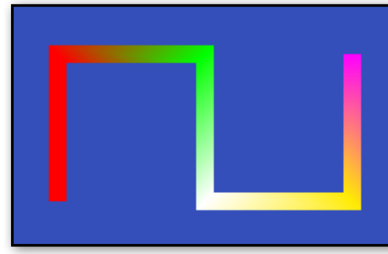
By comparison, **SetColors** takes a Color32 array or a List<Color32>. Each entry in the array is a color that corresponds to a line segment in the VectorLine. So the number of colors in the array or List should match the number of line *segments*—not the number of line *points*. If you're drawing VectorPoints, the length of the Color array or List must match the number of points, since each point can be a different color. Let's make a bunch of different colors for the line:

```
var myColors = new Color32[5];
myColors[0] = Color.red;
myColors[1] = Color.green;
myColors[2] = Color.white;
myColors[3] = Color.yellow;
myColors[4] = Color.magenta];
myLine.SetColors (myColors);
```



When SetColor or SetColors is called, the line colors are changed immediately, and you don't have to redraw the line using VectorLine.Draw or Draw3D. Note that these functions require a shader that uses vertex colors to have any visible effect, such as the built-in default shader, or one of the particle shaders or UI shaders.

Smooth colors



Normally lines use a specific color for each line segment. In some cases you may prefer that colors blend smoothly together instead. This is useful for things like lines that gradually fade out along their length, rather than having visible "steps" for each line segment. You can use the VectorLine.smoothColor property to do this—if it's set to true, then any usage of SetColor or SetColors will cause colors to be blended:

```
var myColors = new Color32[5];
myColors[0] = Color.red;
myColors[1] = Color.green;
myColors[2] = Color.white;
myColors[3] = Color.yellow;
myColors[4] = Color.magenta];
myLine.smoothColor = true;
myLine.SetColors (myColors);
```

GetColor

If you want to see what the color is for a specific line segment, you can specify that segment in VectorLine.GetColor. Here we print the color of line segment 3 (keep in mind segments start at 0), which using the above example would result in Color.yellow:

```
Debug.Log (myLine.GetColor (3));
```

Even if smoothColor has been used, each segment is considered to have a specific color.

In much the same way as you can have different colors for each line segment, you can also have different widths. One way this can accomplished is with an array of floats or ints, plus **VectorLine.SetWidths**:

```
var myWidths = [1, 2, 3, 10, 20]; // Unityscript
var myWidths = new int[] {1, 2, 3, 10, 20}; // C#
myLine.SetWidths (myWidths);
```

In addition to arrays, you can also use generic Lists, namely List<float> or List<int>.

As with colors, each entry in the widths array corresponds to a line segment, so the widths array must be half the length of the points array when using a discrete line, or the length of the points array minus one when using a continuous line. Here's a script that makes a line with two segments, then sets the first segment to 2 pixels, and the second segment to 6 pixels:

```
var linePoints = [Vector2(100, 100), Vector2(200, 100), Vector2(300, 100)];
var myLine = new VectorLine("MyLine", linePoints, null, 2.0, LineType.Continuous);
var widths = [2.0, 6.0];
myLine.SetWidths (widths);
myLine.Draw();
```

This results is a line which looks like the image to the right:

You can also make segment widths be interpolated smoothly from one line segment to another, rather than being distinct. To do this, set **VectorLine.smoothWidth** to "true" after a VectorLine is declared:

```
myLine.smoothWidth = true;
```

If that line is used in the above script (after the VectorLine is declared), you get this result instead:

Note that very short line segments with widely varying widths can cause issues when used with Joins.Weld. In this case, use longer line segments, or make sure that adjacent line segments don't vary in width too much, or both.

Another way to set line segment widths is by using **VectorLine.SetWidth**. This sets the width for a particular segment, which you specify by supplying a width and an index value that corresponds to that segment. This example sets line segment 5 to a width of 10.0:

```
myLine.SetWidth (10.0, 5);
```

You can also specify a range of segments. Here we set the widths for line segments 5 through 15 to 10.0:

```
myLine.SetWidth (10.0, 5, 15);
```

To get the width of a particular segment, use **VectorLine.GetWidth**:

```
Debug.Log (myLine.GetWidth (7));
```

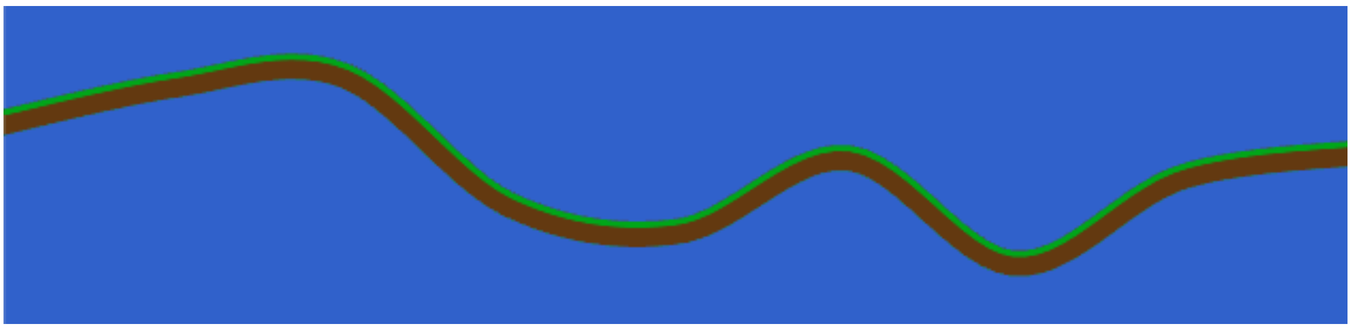
If used after the previous example, this example would print "10.0".

You can make lines interact with other objects using physics by making use of the **VectorLine.collider** property. Just set the collider property to true, and the line will automatically have a collider when drawn:

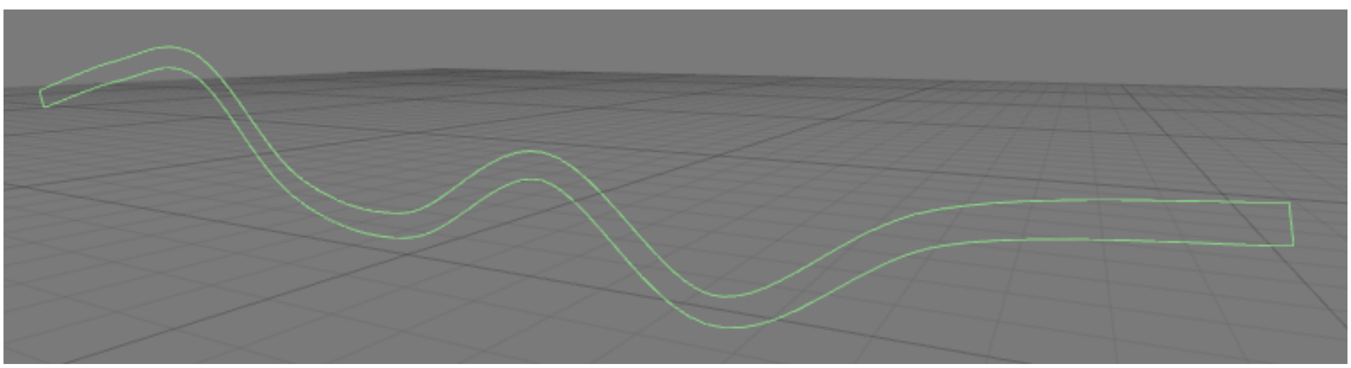
```
myLine.collider = true;
```

Note that this is for 2D physics only. This means the collider won't interact with 3D physics, and is on the X/Y plane. The camera shouldn't be rotated, or else a warning is printed and the collider won't match the line.

You can set the collider property to false in order to deactivate an existing collider. The collider won't be created until VectorLine.Draw is used, but once it's there, then setting VectorLine.collider to true or false doesn't require re-drawing the line. The collider works with both continuous lines and discrete lines. Here we have a continuous line made with a spline:



With collider = true, a matching edge collider is automatically created in the scene:



When used with discrete lines, a polygon collider is made instead of an edge collider, but they work essentially the same, and there are no differences in your code.

Note that creating colliders is relatively slow, so you'd want to be careful with updating complex lines every frame if you're using a collider, since that means the collider will have to be updated every frame too.

Lines made with Vector3 points can also use colliders. However, the collider itself is always 2D and uses 2D physics even for 3D lines, so again the camera shouldn't be rotated, or else the collider won't match the line.

See the RandomHills scene in the VectrosityDemos package for an example of the collider property in action.

Setting the material

If you want to use something other than the default physics material, assign a PhysicsMaterial2D to the **VectorLine.physicsMaterial** property:

```
var linePhysicsMaterial : PhysicsMaterial2D;      // JS
var lineMaterial : Material;

function Start () {
    var myLine = new VectorLine("ColliderLine", new Vector2[100], lineMaterial, 20.0);
    myLine.physicsMaterial = linePhysicsMaterial;
    myLine.collider = true;
}
```

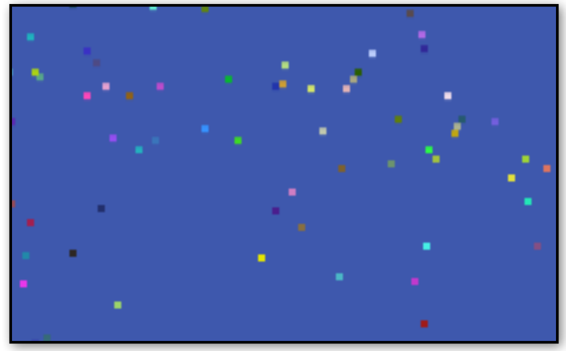
```
public PhysicsMaterial2D linePhysicsMaterial;     // C#
public Material lineMaterial;

void Start () {
    var myLine = new VectorLine("ColliderLine", new Vector2[100], lineMaterial, 20.0f);
    myLine.physicsMaterial = linePhysicsMaterial;
    myLine.collider = true;
}
```

Making colliders be a trigger

In some cases you may want to use a trigger instead of a standard collider. In this case, use VectorLine.trigger:

```
    myLine.collider = true;
    myLine.trigger = true;
```

This can be toggled at any time.



In addition to drawing lines, you can also draw points. This is useful for making single-pixel dots, though the size can be any number of pixels, and you can use a texture too. In fact it's somewhat similar to a particle system. You can use either Vector2 or Vector3 coordinates. If you use Vector2 coords, they are normally screen space, but can be viewport space with useViewportCoords, as described in 3D Lines and Viewport Lines.

To set up your points, use **VectorPoints**. This is basically the same as VectorLine, except it returns type VectorPoints rather than VectorLine, and you can't use LineType or Joins.

```
var points = new VectorPoints("Points", linePoints, lineMaterial, 2.0f);
```

Almost all functions that work with VectorLines will also work with VectorPoints, such as SetColors, MakeCircle, and so on. Like VectorLines, you can add other options after the VectorPoints object is declared, including color, drawDepth, etc. Some options that wouldn't make sense with points, such as capLength and endCap, are not available. Like with VectorLines, you can use a generic List instead of an array for the points. The maximum number of points in a single VectorPoints is 16383.

To draw points once they've been created, just use **VectorLine.Draw**, the same as with VectorLines:

```
myPoints.Draw();
```

If you're using a Vector3 array, you can use **Draw3D** to make the points appear in the scene itself, just like with VectorLines. You can specify a transform too, which works like it does with VectorLines:

```
myPoints.drawTransform = transform;
```

That way you can use a transform to move or rotate points on the screen without having to update the actual points themselves.

Here's a script that draws random points with random colors on the screen:

```
import Vectrosity;
var dotSize = 2.0;
var numberOfDots = 50;

function Start () {
   var dotPoints = new Vector2[numberOfDots];
   for (p in dotPoints)
      p = Vector2(Random.Range(0, Screen.width), Random.Range(0, Screen.height));
   var dotColors = new Color32[numberOfDots];
   for (c in dotColors)
      c = Color(Random.value, Random.value, Random.value);
   var dots = new VectorPoints("Dots", dotPoints, null, dotSize);
   dots.SetColors (dotColors);
   dots.Draw();
}
```

There are a number of utilities in the VectorLine class that help with constructing and working with lines.

AddNormals and AddTangents

By default, VectorLines don't use normals, and attempting to use a shader that requires normals will result in incorrect lighting. If you want to use a shader that requires normals, however, you can call AddNormals to make lighting effective:

```
myLine.AddNormals();
```

Along those same lines, if you use a shader that has normal mapping (such as the Bumped Diffuse shader and similar), the mesh will need tangents in order for the shader to work. Just call AddTangents:

```
myLine.AddTangents();
```

Tangents require normals, so if you're calling AddTangents, AddNormals is called automatically.

Note that if you need the normals or tangents to be re-computed, you can call AddNormals or AddTangents again. You should draw the line again after using these functions so the normals or tangents will be applied.

GetSegmentNumber

This is a small convenience utility which tells you how many segments are possible in a given VectorLine. You might use this to automate the segments or index parameters when calling MakeCircle, etc., or for determining the length of a Color array that you're planning on using with VectorLine.SetColors. The following code will print "49, 25":

```
var line1 = new VectorLine("Line1", new Vector2[50], null, 1.0f, LineType.Continuous);
var line2 = new VectorLine("Line2", new Vector2[50], null, 1.0f, LineType.Discrete);
Debug.Log (line1.GetSegmentNumber() + ", " + line2.GetSegmentNumber());
```

Resize

While you can add or remove points from VectorLines using standard generic List functions, at times it may be convenient to set the number of points to a particular value rather than use functions like AddRange or RemoveRange. You might do this, for example, if you were planning on adding a curve using MakeCurve (see below) and wanted to give the line more room for the extra curve. This code will add an extra 50 points to the existing points:

```
myLine.Resize (myLine.points2.Count + 50);
```

This code would make the line have 100 points, regardless of how many it has currently:

```
myLine.Resize (100);
```

You can use Resize to remove points; just make sure you don't go below 0:

```
myLine.Resize (myLine.points2.Count - 50);
```

GetPoint

Sometimes you might want to get a point a certain distance along a VectorLine. For example, you would need this in order to make an object travel the length of a line that you've drawn using MakeSpline, MakeEllipse, or any other function, including freehand drawing. To use VectorLine.GetPoint, first create a 2D line, then call GetPoint with the VectorLine, and the distance along the line (measured in pixels). GetPoint returns a Vector2 which is the point in screen space coordinates at that distance. (For Vector3 lines, see **GetPoint3D**, below.) This works with any line, regardless of whether it's continuous or discrete. For example, the script below will position a GUIText object 150 pixels along the line:

```
var textObject : GUIText;

function Start () {
	var curveLine = new VectorLine("Curve", new Vector2[100], null, 1.0,
LineType.Continuous);
	curveLine.MakeCurve (Vector2(100, 100), Vector2(300, 75), Vector2(300, 300),
Vector2(450, 375));
	curveLine.Draw();
	textObject.transform.position = Vector3.zero;
	textObject.pixelOffset = curveLine.GetPoint (150);
}
```

The distance is clamped between 0 and the line's length (see **GetLength** below). That is, if you specify a distance below 0, the result will be the same as if you used 0 (namely, the first point on the line), and if you use a distance greater than the line's total length, it will return the same result as if you used the line's length (namely, the last point on the line). Also, a line's .drawStart and .drawEnd variables will clamp the point appropriately.

If you want to get the line segment index that corresponds to a given length, then you can optionally pass in a variable as an out parameter, and after the GetPoint function the variable will contain the line segment index:

```
var index : int;  // JS
var myPoint = curveLine.GetPoint (150, index);
int index; // C#
var myPoint = curveLine.GetPoint (150, out index);
Debug.Log (index);
```

GetPoint01

This works the same as GetPoint, above, except it uses normalized coordinates from 0.0 through 1.0 for the distance. In other words, a percentage rather than an absolute value. This is useful if you don't really care how long the line is, but need a point at a certain percentage of a line's length. You could use this code in the above script instead of GetPoint, and it will position the GUIText at the halfway point along the line:

```
	textObject.pixelOffset = curveLine.GetPoint01 (0.5);
```

See the **SplineFollow** example scene in the VectrosityDemos package for examples of scripts that move an object along a line at a constant rate using GetPoint01.

As with GetPoint, the distance is clamped, in this case between 0.0 and 1.0. Anything below 0.0 will return the same result as 0.0, and anything above 1.0 will return the same result as 1.0.

GetPoint3D

If you need a point on a 3D line, then you can use VectorLine.GetPoint3D. It works the same as GetPoint, except it can only use lines made with Vector3 points. It returns a Vector3 world-space coordinate instead of a Vector2 screen-space coordinate, and the distance is measured in world units rather than pixels.

GetPoint3D01

Again, this is the same as GetPoint01, except it only works with lines made with Vector3 points, and it returns a Vector3 and uses world units for the distance.

GetLength

If you're not using GetPoint01 or GetPoint3D01, you may need to know how long a line is. As you might imagine, that's just what GetLength does.

```
var myLineLength = myLine.GetLength();
```

It returns a float, which is the length of the line in pixels for lines made with Vector2 points, or the length of the line in world units for lines made with Vector3 points.

SetDistances

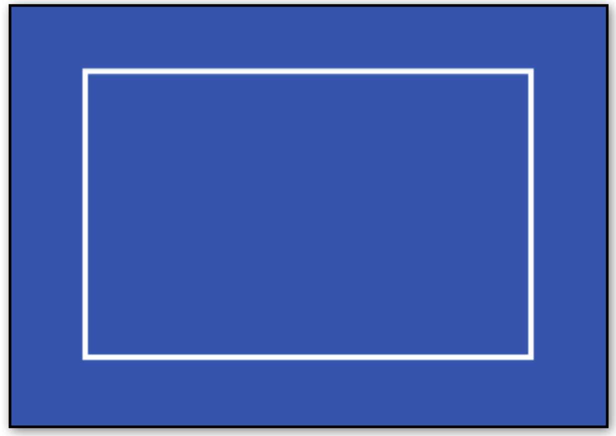
Let's say you've used GetPoint. Then, you change some points that make up the line and redraw it. Now you use GetPoint again, but the results aren't correct! What's wrong?

What happened is that the line segment distances in a VectorLine are computed the first time you use GetLength or any of the GetPoint functions. However, these line segment distances are not recomputed automatically if you later change the points that make up a VectorLine. This is for performance reasons — you're not necessarily going to use any of the GetPoint functions every time you update line points, so it's not the best use of CPU cycles to recompute them all the time.

Instead, you can call SetDistances after you update a line's points, but before you use GetLength or one of the GetPoint functions. This way the line segment distances are recomputed only when they're actually needed.

```
function Start () {
	var points = [Vector2.zero, Vector2(100, 100)];
	var line = new VectorLine("Line", points, null, 1.0);
	print (line.GetLength());

	points[1] = Vector2(200, 200);
	line.SetDistances();
	print (line.GetLength());
}
```

MakeRect



This is for quickly setting up squares or rectangles, since this is a pretty common thing to do with line drawing (think selection boxes and that sort of thing). You can do this by supplying either a Rect, or two Vector2s that describe the bottom-left corner and the top-right corner, where the coordinates are in screen pixels. This works for either continuous or discrete lines — with continuous lines, you need at least 5 points in the Vector2 array, and with discrete lines, you need at least 8.

MakeRect works with Vector3 arrays as well as Vector2 arrays. In the case of Vector3 arrays, you can pass in two Vector3s instead of two Vector2s, and the .z element of the Vector3s will be the depth in world space. Rects have no depth value, so using them with a 3D line will result in 0 being used for the depth.

By default, the rect is drawn starting at index 0 in the Vector2 array, though you can optionally specify a starting index. This way you can draw any number of rects in a single line, although this works best with discrete lines, since multiple rects in a continuous line would all be connected together. For example, if you had a discrete line with a points2 list of size 24, you could make three rects in this line, one starting at index 0, one starting at index 8, and one starting at index 16. If you try to specify a starting index for the list that wouldn't leave enough room for the rect, you'll get an error informing you of this. (For example, trying to use a starting index of 16 in a list with only 20 entries.)

MakeRect requires an already set-up VectorLine. It only calculates the lines, and doesn't draw them; for that you need to use VectorLine.Draw or Draw3D as usual. The format is:

```
myLine.MakeRect (Rect, index);
```

or:

```
myLine.MakeRect (Vector2, Vector2, index);
```

where "index" is optional (if not specified, it's 0). As mentioned above, you can use Vector3 for 3D lines. Here's an example of making a 100 pixel square starting at 300 pixels from the left and 200 pixels from the bottom:

```
var squareLine = new VectorLine ("Square", new Vector2[8], lineMaterial, 1.0f);
squareLine.MakeRect (new Rect(300, 200, 100, 100));
squareLine.Draw();
```

Making a selection box might look like this, assuming "selectionLine" is a VectorLine, and "originalPos" is the position where the mouse was originally clicked:

```
selectionLine.MakeRect (originalPos, Input.mousePosition);
```

See the **SelectionBox** scene in the **VectrosityDemos** package for a couple of examples. The **Main Camera** object has two scripts attached; enable or disable **SelectionBox** and **SelectionBox2** as desired to see the different effects.

MakeRoundedRect

If you want to have a box with rounded corners, this is the function to use. It works like MakeRect, but with additional parameters for defining the corner radius and number of segments to use.

Like MakeRect, you can use either a Rect:

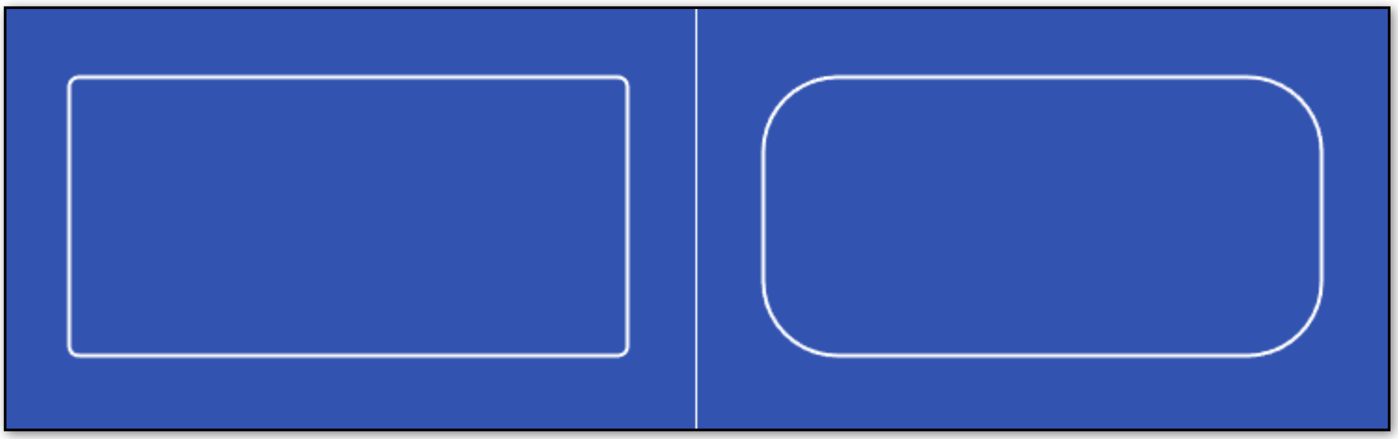
```
myLine.MakeRoundedRect (Rect, cornerRadius, cornerSegments, index);
```

or two points:

```
myLine.MakeRoundedRect (Vector2, Vector2, cornerRadius, cornerSegments, index);
```

You can use also two Vector3s for lines made with Vector3 points. The index is optional, and 0 by default.

The cornerRadius value is a measurement of the number of pixels (for Vector2 points) or units (for Vector3 points) used for the corners. Negative values work, but will have a funky inverted corner effect.



Left: a rounded rect with a cornerRadius of 5. Right: the same rect with a cornerRadius of 40.

The cornerSegments value is how many line segments are used for each corner. This must be at least 1 (and using 1 will look like a beveled corner rather than being rounded, which might be the desired effect in some cases). For small radius values, it's generally a good idea to use only a few segments, and large radius values typically require more segments to look smooth.

In most cases you don't need to know exactly how many points are used for creating a rounded rect, since if there aren't enough in the line's points2 or points3 list, more points will be added as necessary. But it can be useful sometimes, such as using the index to make multiple rounded rects in a single VectorLine. Therefore: the number of segments used for a rounded rect is 4 * cornerSegments + 4. How this translates to points depends on the type of line; for LineType.Continuous, it's the number of segments + 1, and for LineType.Discrete, it's the number of segments * 2.

Here's an example of making a rounded rect that starts at (50, 50) at one corner and goes to (450, 250) at the opposite corner, so it's 400 pixels wide and 200 pixels tall, with a radius of 15 pixels, using 5 line segments per corner:

```
var rectLine = new VectorLine ("RoundRect", new Vector2[2], lineMaterial, 2.0f);
rectLine.MakeRoundedRect (new Vector2(50, 50), new Vector2(450, 250), 15.0f, 5);
rectLine.Draw();
```

MakeCircle

This is for easily creating circles or other round-ish shapes like octagons. As with MakeRect, it calculates the appropriate values in a VectorLine's Vector2 or Vector3 array; you still use VectorLine.Draw or Draw3D to actually draw the circle after using MakeCircle.

You specify the line, origin, radius, and number of segments, where more segments make for smoother-looking circles. If you use just a few segments, it can be used for shapes like octagons, or even triangles if you use just 3 segments. You can optionally specify point rotation, which is generally useful for setting the orientation of low-segment shapes (the effect is pretty much invisible when using lots of segments). Also, as with MakeRect, you can specify the index, so you can create multiple circles in one VectorLine object. Again, this is primarily useful for discrete lines, since multiple circles in a single continuous line will of course all be connected together. The format is:

```
myLine.MakeCircle (origin, up, radius, segments, pointRotation, index);
```

Note that some of these parameters are optional and have default values. At minimum, you only need the origin and radius:

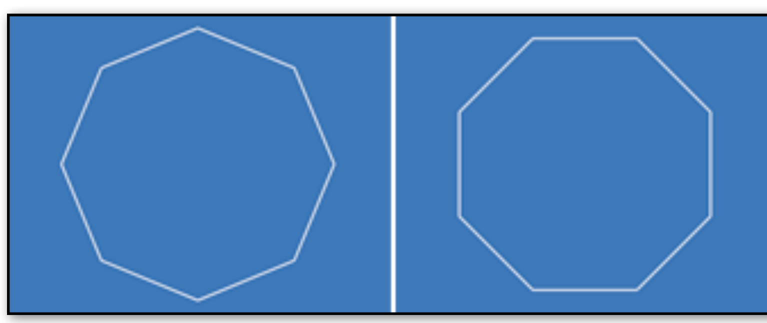
```
myLine.MakeCircle (origin, radius);
```

In this case, the circle will use all available segments in the VectorLine, so you don't have to specify the number of segments to use.

The size of the points2 or points3 list that's used for this VectorLine must be at least the number of segments plus one for a continuous line, or twice the number of segments for a discrete line. For example, if you're using 30 segments for a continuous line, the points list must have at least 31 elements. Using 30 segments for a discrete line would require 60 elements in the points list.

The origin is either a Vector2 for 2D lines, where x and y are screen pixels, or a Vector3 for 3D lines, where x, y, and z are in world space. The radius is a float, which describes half the total width of the circle in screen pixels (for Vector2 lines) or world units (for Vector3 lines). So a circle with a origin of Vector2(100.0, 100.0) and a radius of 35.0 would be 70 pixels wide, centered around the screen coordinate (100, 100). The number of segments is an integer, with a minimum of 3. Since a circle in this case is actually composed of a number of straight line segments, the more segments that are used, the more it resembles a true circle.

"PointRotation" is optional, with a default of 0.0. It's a float, specifying the degrees that the points are rotated clockwise around the circle. (Negative values mean counter-clockwise.) This is generally useful for making low-segment circles be oriented in a desired way:



Left: a point rotation of 0.0 used with 8 segments. Right: a point rotation of 22.5.

“Index” is also optional, with a default of 0. It’s used just like the index value in MakeRect. For example, a discrete line with a points2 list of 120 entries could contain two circles of 30 segments, one at index 0 and one at index 60. (Remember, discrete lines need twice the number of points as there are segments in the circle, since two points are used for each segment.)

“Up” is an optionally-specified up vector, which is only useful if you’re using MakeCircle with a Vector3 array. By default, circles drawn in Vector3 lines are oriented in the X/Y plane. You might prefer that circles have a different orientation, such as the X/Z plane, so they are parallel to the “ground”. The up vector is a Vector3, and is a direction specifying which way is up according to the circle. For example, to make a circle in the X/Z plane, you’d want the Y axis pointing up, so you’d use Vector3(0, 1, 0), or Vector.up:

```
myLine.MakeCircle (Vector3.zero, Vector3.up, 15.0);
```

The up vector can be any arbitrary Vector3. It doesn’t have to be normalized.

Note that MakeCircle is actually an alias for MakeEllipse (see below), so any error messages generated when using MakeCircle will reference MakeEllipse instead.

MakeEllipse

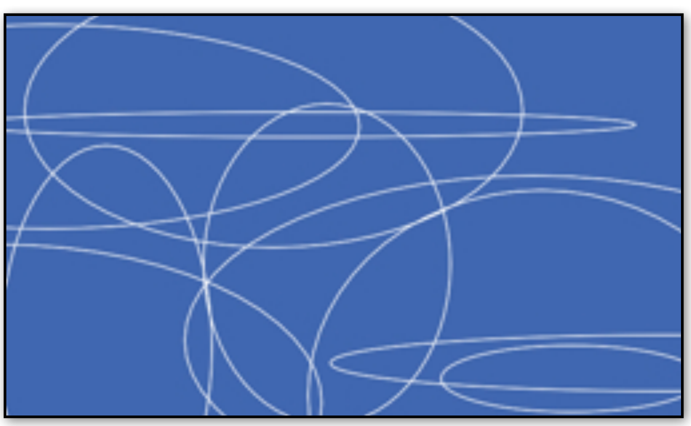
This is nearly identical to MakeCircle, with the exception that two radius values are used instead of just one. You can specify the x and y radius values to make ellipses of different widths/heights. MakeCircle actually uses this routine, but it passes one radius value for both x and y. The complete format is:

```
myLine.MakeEllipse (origin, up, xRadius, yRadius, segments, pointRotation, index);
```

As with MakeCircle, you can leave out some parameters, so the minimum is:

```
myLine.MakeEllipse (origin, xRadius, yRadius);
```

Both “xRadius” and “yRadius” are floats that specify the number of screen pixels for the respective radii (or world units if you’re using a Vector3 line). The usage otherwise is the same as MakeCircle. Note that “pointRotation” only rotates the points clockwise or counterclockwise within the ellipse shape; it doesn’t rotate the shape itself. So an ellipse elongated horizontally with a pointRotation value of 45.0 will not be tilted 45°, for example — it will still have the same basic orientation, and again is primarily useful for low-segment shapes, where the results are actually visible.



See the **Ellipse** scene in the **VectrosityDemos** package for a couple of example scripts. The **Main Camera** object in that scene has two scripts attached, **Ellipse1** and **Ellipse2**, which you can enable/disable to see the different effects. Ellipse1 creates a single ellipse using a continuous line, where you can adjust the xRadius, yRadius, number of segments, and point rotation in the inspector to see the effects of different values. Ellipse2 creates a number of random ellipses in a single VectorLine using a discrete line, where you can adjust the number of segments and total number of ellipses in the inspector.

MakeArc

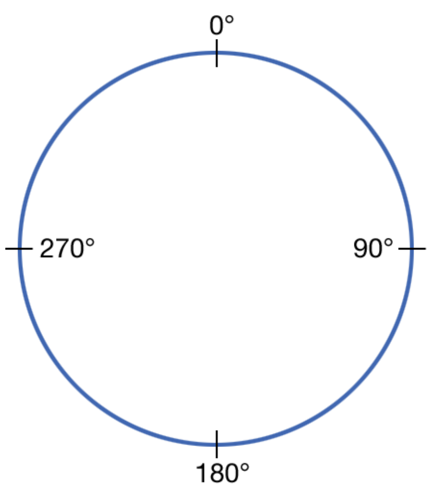
You can draw part of a circle or ellipse easily with this function. It's very similar to MakeEllipse, except you specify the start and end points of the arc in degrees. Also, the pointRotation parameter isn't used with MakeArc.

```
myLine.MakeArc (origin, up, xRadius, yRadius, startDegrees, endDegrees, segments, index);
```

At the minimum, you need to define the origin, radii, and the start and end degrees:

```
myLine.MakeArc (origin, xRadius, yRadius, startDegrees, endDegrees);
```

As with MakeCircle/MakeEllipse, the up vector is only used with Vector3 points. If the segments parameter is left out, the arc uses all available points in the line. Along with segments, the index is useful for creating multiple arcs in the same VectorLine (which should use LineType.Discrete to avoid having all the arcs connected to each other).



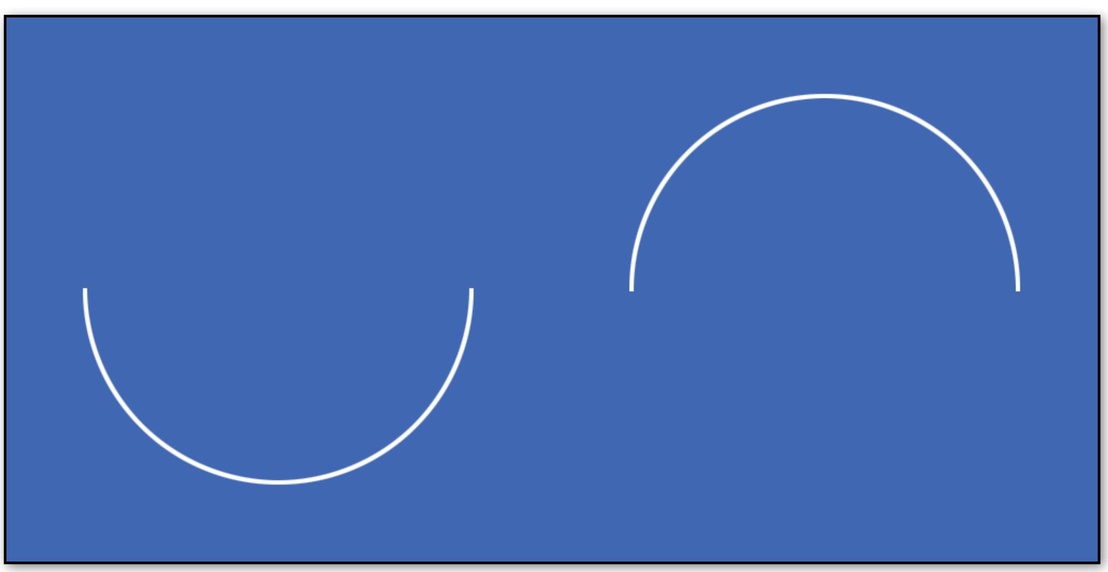
When used with Vector2 points, the degrees start at 0 at the top and go clockwise around the circle to 360.

When used with Vector3 points, where the "top" is depends on the up vector and the camera orientation.

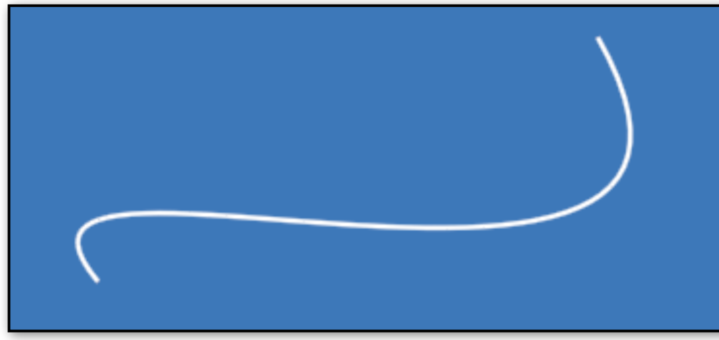
If the supplied degrees go over 360, they wrap around, so for example 370° is the same as 10°. The same is true for negative numbers: -10° is the same as 350°.

You can have the start degrees be greater than the end degrees, which is useful for defining which part of the circle is drawn in the case where you want to wrap around the top.

For example, here we have two arcs, where the first is defined using 90 for the startDegrees and 270 for the endDegrees. The second is defined using 270 for the start and 90 for the end. It would also work to use 450 (270 + 180) for the end.



MakeCurve



This allows the creation of bezier curves in existing VectorLine objects. These are curves made from two anchor points and two control points. You probably already get the general usage idea by now, after the MakeRect/Circle/Ellipse sections. It results in curves that might look like this, depending on how the anchor and control points are positioned:

The format is either:

```
myLine.MakeCurve (curvePoints, segments, index);
```

or

```
myLine.MakeCurve (anchor1, control1, anchor2, control2, segments, index);
```

In the first case, “curvePoints” is a Vector2 or Vector3 array of 4 elements, where element 0 is the first anchor point, element 1 is the first control point, element 2 is the second anchor point, and element 3 is the second control point. In the second case, the anchor and control points are written as individual Vector2s or Vector3s. These all use screen pixels as coordinates, or world coordinates for Vector3 lines.

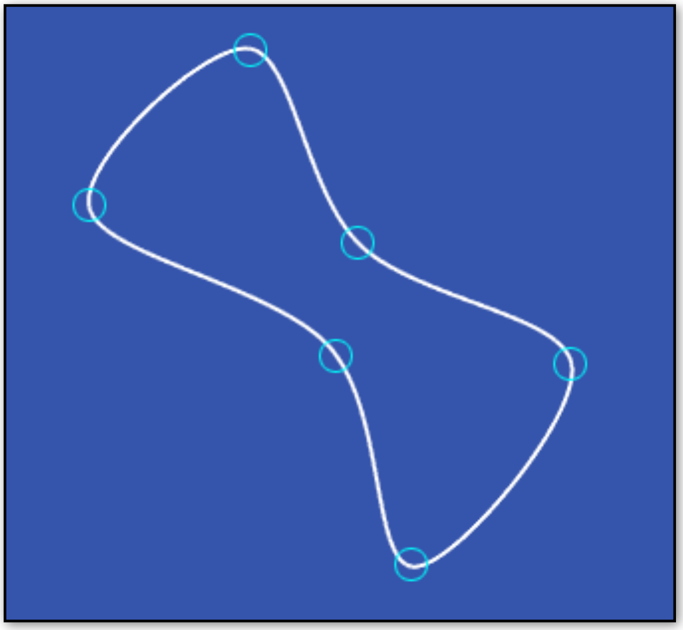
“segments” is an int, and works like it does in MakeCircle/MakeEllipse: the more segments, the smoother-looking the curve. Again, the number of elements in the Vector2 array should be the number of segments plus one for continuous lines, or twice the number of segments for discrete lines. Alternatively, you can leave out segments entirely, and MakeCurve will use as many segments as can fit in the VectorLine, so the shortest form is:

```
myLine.MakeCurve (curvePoints);
```

“index” is optional as usual, and is 0 by default. Again, multiple separate curves in a single VectorLine makes more sense using a discrete line, since the curves would be connected together when using a continuous line. If using a continuous line, you probably want separate VectorLine objects instead.

If you’re unfamiliar with the concept of how bezier curves work, open the **Curve** scene in the VectrosityDemos package. With the **DrawCurve** script active, you can hit Play, and interactively create curves by dragging anchor and control points around the screen. Basically, the anchor points behave just like the end points of a straight line segment, while the control points influence the shape of the curve. You can also disable the DrawCurve script and enable the **SimpleCurve** script instead, which draws a single curve using a Vector2 array of 4 points, which you can specify in the inspector.

MakeSpline



This is somewhat similar to MakeCurve, in that it makes curves in existing VectorLines. The main difference is that you can pass in an array with any number of points, and MakeSpline will create a curve that passes through all the points in that array. (If you're familiar with Catmull-Rom splines, you've probably guessed that this is what MakeSpline uses.) The spline can be open, like with MakeCurve, or it can be a closed loop.

See the **Spline** scene in the VectrosityDemos package for an example of a spline in action. You can move the spheres in the scene around as you like, and when you hit Play in the editor, a spline is created that touches all the spheres. On the **_Main Camera** object, you can set the number of segments, as well as toggle whether the curve is a closed loop, and whether to use a line or points.

The format is:

```
myLine.MakeSpline (splinePoints, segments, index, loop);
```

Only the first parameter is actually required, so the shortest form is:

```
myLine.MakeSpline (splinePoints);
```

"SplinePoints" is either a Vector2 or Vector3 array, with any number of elements. The resulting curve will pass directly through all the supplied points, so unlike the bezier curves used with MakeCurve, there are no control points.

"Segments" and "index" are optional as usual; see MakeCurve, etc. for details if you don't already know. If you leave out the number of segments, MakeSpline will make as many segments as the VectorLine allows. So if you use a VectorLine with 100 points using LineType.Continuous, that would result in 99 segments, or 50 segments if you use LineType.Discrete.

"Loop" is whether the spline is an open or closed shape. By default this is false, so if you want a closed loop, you have to specify true.

MakeText

You can even make text out of line segments. It's definitely not a substitute for TTF fonts normally used in Unity, but has some uses, such as in HUDs, since the text can be set to any size easily, and can be scaled and rotated by passing in a transform (see the TextDemo script in the Vectrosity demos). And, of course, any self-respecting vector graphics game, like Tank Zone, will need characters made out of vector lines.

The basic way to do this is to call VectorLine.MakeText after a VectorLine has been created, where you pass in the line, the string you want to display, a position (Vector2 or Vector3), and a size:


```
myLine.MakeText ("Vectrosity!", new Vector2(100, 100), 30.0f);
```

You can use "\n" in the string for a new line. You don't have to worry about how many line segments are needed for the text…if the line's points list isn't large enough to hold them all, it's resized.

The position is in screen space coordinates for Vector2 lines and world coordinates for Vector3 lines, and likewise the size is pixels for Vector2 lines and world units for Vector3 lines. The character and line spacings are respectively 1.0 and 1.5 by default, but you can override this by specifying them yourself:

```
myLine.MakeText ("Hello world!", new Vector2(100, 100), 30.0f, 0.8f, 1.2f);
```

These values are relative to the size, with 1.0 for character spacing being the full width of a character (text is always monospaced), and for line spacing, 1.0 likewise is the full height of a character. You can also specify whether text should be printed in upper-case only by adding "true" or "false"; by default this is true:

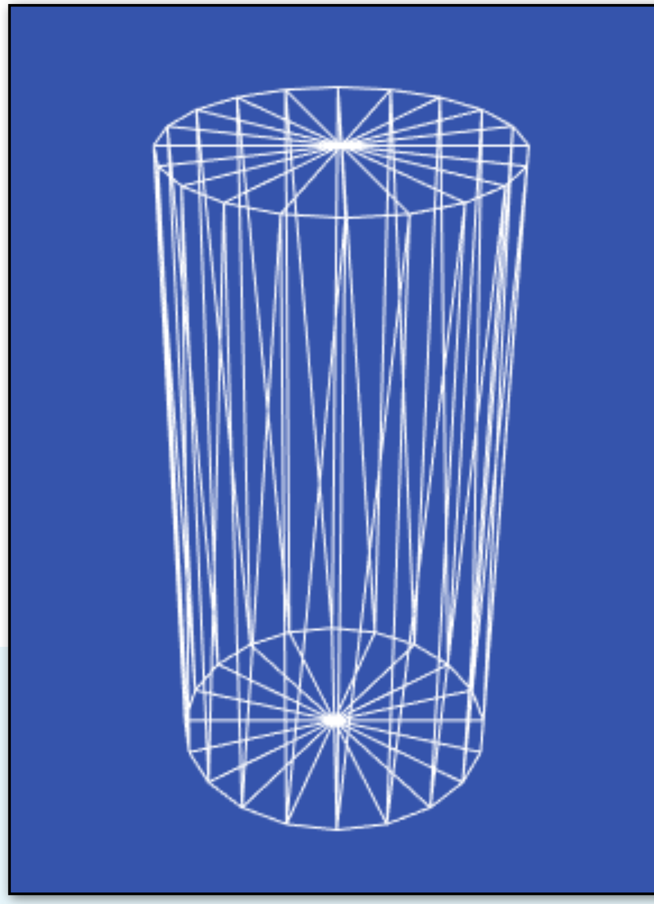
```
myLine.MakeText (“Hello world!", new Vector2(100, 100), 30.0f, false);
```

Any characters in the string which don't exist in the Vectrosity "font" are ignored. Currently, most of the standard ASCII set is included.

You can, however, add to or modify the characters as you like, as long as you're using the source code. The relevant file is VectorChar in Standard Assets/VectorScripts. If you open this, you'll see a list of all characters, with each one, as indicated by Unicode value, represented by a Vector2 array. For example, "points[65]" is an upper-case letter A. The standard coordinates to use range from (0, 0) for the upper-left of the square containing a character, to (0, -1) for the lower-right. Normally you wouldn't use the entire width of 1.0, or else the characters would run together with the default character spacing (the included characters are no wider than 0.6).

A convenient way to create characters is to use the **LineMaker** utility. For full details of how to use this, see the LineMaker section below. Briefly, you can drag the LetterGrid mesh from the Meshes folder into the scene, then select the Assets -> LineMaker menu item. This grid object is pretty small, so turn down the point and line size so you can see what you're doing, and construct a character as you like. When done, click on "Vector2" next to "Generate Complete Line", and paste the results into the VectorChar script as appropriate. You'll need to set "useCSharp" in the LineMaker script (in the Editor folder) to "true" if it's not already. You're not restricted to the grid points as-is; you can move them around in the scene if you'd like.

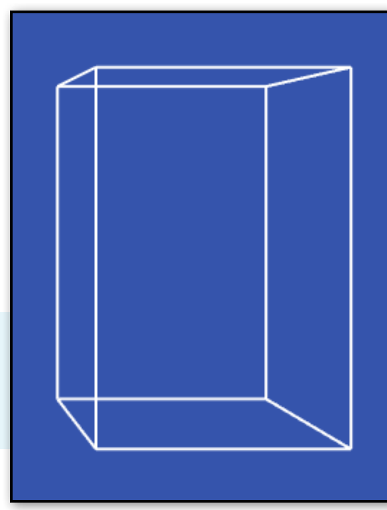
MakeWireframe



This has essentially the same effect as if you were using the LineMaker utility with an object and clicked on the “Connect all points” button, except it works at runtime with arbitrary meshes. To use it, first set up a line, then call VectorLine.MakeWireframe with the line and a mesh. The line must use Vector3 points, and must be a discrete line. It doesn’t matter how large points3 is — the list will be resized if necessary in order to fit all the line segments for the mesh. With the following example, you would attach the script to an object, select a mesh of some kind for the “myMesh” variable, and when run, the mesh will be displayed as a wireframe. You may want to use this in combination with the VectorManager functions.

```
var myMesh : Mesh;

function Start () {
	var line = new VectorLine("Wireframe", new Vector3[0], null, 1.0, LineType.Discrete);
	line.MakeWireframe (myMesh);
	line.Draw();
}
```

MakeCube



You can make arbitrary cubes easily using this function. Like MakeWireframe, it requires a discrete line made with Vector3 points. Unlike MakeWireframe, the size of the list matters — it must contain at least 24 points. When calling MakeCube, you specify the position as a Vector3, and the x, y and z dimensions as three floats:

```
var line = new VectorLine(“Cube”, new Vector3[24], null, 2.0);
line.MakeCube (Vector3(4.0, 0.0, 5.0), 3.0, 4.5, 3.0);
line.Draw();
```

You can also specify the index number of the array where the cube is drawn (the default is 0). You can use this to have multiple cubes in one VectorLine, or to combine cubes with other line segments. Just remember that the cube requires 24 points counting from the specified index, so if you started at index 24, for example, the total size of the array would need to be at least 48. This would make two nested cubes in one VectorLine:

```
line.MakeCube (Vector3.zero, 1.0, 1.0, 1.0);
line.MakeCube (Vector3.zero, 2.0, 2.0, 2.0, 24);
```

BytesToVector2Array and BytesToVector3Array

An alternative to specifying line points in code is to use a TextAsset file that contains Vector2 or Vector3 array data as binary data. You can create these files by using the **LineMaker** editor script (see the LineMaker section below). You can create specific shapes for lines and store them as assets in your Unity project, and use drag'n'drop as usual. You then use BytesToVector2Array or BytesToVector3Array to convert those assets to Vector2 arrays or Vector3 arrays respectively.

This is useful if you have complex pre-made shapes, where the alternative is long strings of Vector2 or Vector3 data. It also allows the flexibility of connecting assets in Unity's inspector instead of hard-coding data into scripts.

To use these functions, first you need a TextAsset variable. Then pass the bytes from the TextAsset into the function, which converts it to the appropriate array:

```
var lineData : TextAsset;   // JS

function Start () {
    var linePoints = VectorLine.BytesToVector2Array (lineData.bytes);
    var line = VectorLine("Vector Shape", linePoints, null, 2.0);
}
```

```
public TextAsset lineData;  // C#

void Start () {
    var linePoints = VectorLine.BytesToVector2Array (lineData.bytes);
    var line = new VectorLine("Vector Shape", linePoints, null, 2.0f);
}
```

BytesToVector3Array works exactly the same way, but naturally returns a Vector3 array.

The **_Simple3DObject** scene in the **VectrosityDemos** package has an example of this. On the **Cube** object, you can either use the **Simple3D** script (which has the vector cube data hard-coded into the script) or the **Simple3D 2** script (which uses the **CubeVector** TextAsset file). You can try dragging different files from the Vectors folder onto the VectorCube slot to get different shapes.

Version

In certain cases it may be useful to know, though code, what version of Vectrosity you're using, particularly if you're using the DLL and can't look at the source code. The Version function simply returns a string with version information.

```
Debug.Log (VectorLine.Version());
```

Selected

Sometimes you might want to have users be able to select a line. The Selected function makes this easy, where you pass in input coordinates (such as from Input.mousePosition) and get back true or false, depending on whether the input coordinates are currently over the line in question or not. For example, assuming a VectorLine called “line”, this will cause the line to turn red if the user clicks on it:

```
function Update () {
	if (Input.GetMouseButtonDown(0) && line.Selected (Input.mousePosition)) {
		line.SetColor (Color.red);
	}
}
```

The input coordinates should be in screen space, where (0, 0) is the bottom-left. Note that Selected works with VectorPoints as well as VectorLines.

If you’d like to know exactly what line segment (or point) was selected, you can pass in an integer variable, which will then contain the segment or point index after Selected is called. The index will contain -1 if Selected returns false. For example:

```
private var index : int;

function Update () {
	if (line.Selected (Input.mousePosition), index) {
		Debug.Log ("Selected line index: " + index);
	}
}
```

If you’re using C#, you must specify “out” for the index parameter:

```
	if (line.Selected (Input.mousePosition), out index) {
```

It’s also possible to pass in an extra integer parameter, which essentially extends a line’s width by that many pixels for the purposes of the Selected function. This can make it easier to click on lines (particularly thin ones), since the input doesn’t have to be precisely over the line. Note that the index parameter is required when using the extra distance parameter, even if you’re not planning to use the index. For example, this will extend the selection area of a line by 10 pixels:

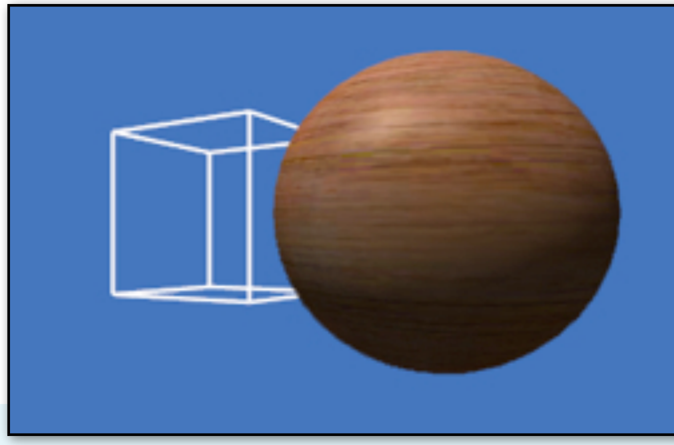
```
if (line.Selected (Input.mousePosition, 10, index)) {
	Debug.Log ("Selected!")
}
```

In some cases you may want to add extra selection space to the ends of line segments as well as the width. To do this, add another integer after the first; this will add 10 pixels to the width and 2 pixels to the length:

```
if (line.Selected (Input.mousePosition, 10, 2, index))
```

Note that by default Selected uses the camera set by SetCamera3D, or the first camera tagged “MainCamera” if SetCamera3D hasn’t been used. You can supply a different camera if desired, using any of the Selected overloads:

```
if (line.Selected (Input.mousePosition), index, myCamera) {
```



Draw3D

Normally, all lines (even lines made from Vector3 arrays) are rendered in an overlay canvas. There are times, however, when you might want vector lines to actually be a part of a scene, where they can be occluded by standard 3D objects. This is possible by using **VectorLine.Draw3D**. This can only be used with VectorLine objects created with Vector3 arrays or lists (Vector2 arrays aren't allowed), but otherwise works the same as VectorLine.Draw:

```
myLine.Draw3D();
```

As with Draw, you can assign a transform, and Draw3D will use that transform to modify the line:

```
myLine.drawTransform = transform;
```

VectorPoints can also use Draw3D. One thing to be aware of with 3D lines or 3D points is that, unlike lines or points that use VectorLine.Draw, they need to be updated whenever the camera moves, in order to preserve the correct perspective. See **Draw3DAuto** below for more details.

Since Draw3D uses Vector3 lines, that means a camera is required, as covered in Canvas and Camera.

3D lines with VectorManager

If you want VectorManager (see the next section) to use 3D lines instead of standard lines, this can be done by setting **useDraw3D**, which is false by default:

```
VectorManager.useDraw3D = true;
```

Draw3DAuto

Lines drawn with VectorLine.Draw normally don't need to be updated unless something changes. This is useful for 2D lines, but since 3D lines exist in the scene, they will appear distorted as the camera moves around, unless you call VectorLine.Draw3D every frame (usually in Update or LateUpdate).

To make this simpler, you can use **Draw3DAuto** instead. This will cause a VectorLine to automatically re-draw every frame. This means you can set up a VectorLine, draw it once using Draw3DAuto, and then forget about it. From then on, that line will always look right no matter where you move the camera, or if you change any line properties that would normally require a re-draw. For example:

```
function Start () {
	GameObject.CreatePrimitive (PrimitiveType.Cube);
	Camera.main.transform.position = Vector3(0,0,-10);
	Camera.main.gameObject.AddComponent(Light).range = 30;
	var line = new VectorLine("3DLine", [Vector3(0,-2,-4), Vector3(0,4,3)], null, 2.0);
	line.Draw3DAuto();
}

function Update () {
	Camera.main.transform.RotateAround (Vector3.zero, Vector3.up, 45.0*Time.deltaTime);
}
```

StopDrawing3DAuto

In case you want the 3D line to stop automatically updating, just call **StopDrawing3DAuto** with the appropriate VectorLine. From then on, that VectorLine will only update if you call Draw3D yourself.

```
line.StopDrawing3DAuto();
```

Viewport lines

Going back to 2D for a moment, viewport coordinates can be useful, rather than the default screen space coordinates. With viewport coords, (0.0, 0.0) is always the lower-left corner of the screen, and (1.0, 1.0) is always the upper-right, regardless of screen resolution. This means that, for example, using the points (0.5, 1.0) and (0.5, 0.0) will always draw a line down the middle of the screen, without having to do any calculations with Screen.width or Screen.height. Lines used with viewport coords must be made with Vector2 points. After creating the line, specify **VectorLine.useViewportCoords**:

```
var midline = new VectorLine("Midline", [Vector2(.5, 0), Vector2(.5, 1)], null, 2.0);
midline.useViewportCoords = true;
midline.Draw();
```

So whenever the line is drawn, Vectrosity will treat the points as viewport rather than screen coordinates. The only drawback is that different aspect ratios can cause different results (a circle might get stretched or squashed, for example), so using screen space coords can still be the way to go, depending on what you’re doing.

There is an additional class that makes 3D vector shapes behave almost exactly like regular GameObjects. See the **_Simple3DObject** scene in the VectrosityDemos package for an example of making a 3D vector cube using VectorManager.

Note: the scene view camera will cause OnBecameVisible and OnBecameInvisible functions to fire. Since these functions are used by most of the Visibility scripts that work with VectorManager, you may find that 3D vector shapes don't display properly in some cases. To avoid this, make sure the scene view isn't visible when you enter play mode in the editor, and therefore doesn't interfere with things when you're testing your project. One easy way to do that is to always use Maximize On Play.

ObjectSetup

To make a GameObject into a 3D vector object, you should first set up a Vector3 array or list describing the shape you want, and create a VectorLine object using this array. (See the section about LineMaker below for an easy way to create 3D vector shapes.) Then, call VectorManager.ObjectSetup, where you pass in the GameObject, the VectorLine object, the type of visibility control it should have, and the type of brightness control:

```
VectorManager.ObjectSetup (gameObject, vectorLine, visibility, brightness, makeBounds);
```

Depending on the parameters, this adds a couple of script components to the GameObject at runtime. You then have a 3D vector object that behaves just like the GameObject. Using a VectorPoints object instead of a VectorLine object is fine.

Note that from now on, everything is completely automated. All you have to do is move the supplied GameObject around as you normally would. It can be under physics control or direct control — whatever you like. You can think of it as a standard GameObject with the renderer replaced by a VectorLine object. If you don't want it around any more, just destroy the GameObject, and the VectorLine will be properly destroyed too. Don't use VectorLine.Destroy—the VectorManager will handle line destroying automatically when the GameObject is destroyed. If you have multiple GameObjects that you want to make into VectorLine objects, then each GameObject should call ObjectSetup.

There are several types of visibility control:

Visibility.Dynamic: The 3D vector object will always be drawn every frame when the GameObject is visible, and won't be drawn when it's not seen by any camera, just like a normal GameObject. This saves having to compute lines that are not in view, and is accomplished by using the renderer of the normal GameObject — if you disable the GameObject's renderer, then the vector object will be disabled too. Use this for moving objects.

Visibility.Static: Like Dynamic, the 3D vector object will only be drawn when visible. Unlike Dynamic, it will only be drawn when the camera moves. Also, the drawing routine is a little faster since it doesn't take the object's Transform into account. You would use this for objects which never move. For example, in the Tank Zone demo package, the tanks, saucers, and shells use Visibility.Dynamic, and the obstacles use Visibility.Static.

Visibility.Always: The 3D vector object will always be drawn every frame, with none of the optimizations from Dynamic or Static. You might use this if you have an object that's always going to be in front of the camera anyway. If the GameObject you're using has a mesh renderer, you should disable it if you only want to see the vector object and not the GameObject. (This doesn't apply to Visibility.Dynamic or Visibility.Static.)

Visibility.None: None of the VisibilityControl scripts will be added. If any of the other Visibility options have been used with this object previously, the visibility scripts will be removed. Usually there's not much reason to use this, unless you're updating the line yourself with VectorLine.Draw3D for some reason.

There are two types of brightness control:

Brightness.Fog: This simulates a fog effect for 3D vector objects. You can see this in the Tank Zone demo package, where objects fade to black in the distance. Control over the fog effect is done with VectorManager.SetBrightnessParameters (see below). Note that only single line colors are supported; that is, VectorLine.color. Any line segment colors set with SetColor or SetColors will be ignored.

Brightness.None: The line segment colors are left alone. If ObjectSetup had been used with Brightness.Fog for this object previously, the Brightness.Fog script will be removed.

An example of an object being set to static visibility with fog, using a VectorLine object called "myLine":

```
VectorManager.ObjectSetup (gameObject, myLine, Visibility.Static, Brightness.Fog);
```

MakeBounds: this is true by default, so it only needs to be supplied if you don't want to use it. In this case, you'd add "false" at the end:

```
VectorManager.ObjectSetup (gameObject, myLine, Visibility.Always, Brightness.Fog, false);
```

What this does when true, is create an invisible bounds mesh for the GameObject that you're using to control the vector object. Why would you want this? Well, remember Visibility.Dynamic and Visibility.Static, and how they optimize things by not updating lines when the objects aren't visible—they do this by using OnBecameVisible and OnBecameInvisible. But these functions require mesh renderers to work. And you don't actually want to see a mesh renderer, you just want to see the VectorLines.

The invisible bounds mesh solves this by creating a "bounding cube mesh" that actually only consists of eight vertices that make up the object bounding box, and no triangles. But this is good enough for Unity! Even though you can't see this mesh, it still allows OnBecameVisible and OnBecameInvisible to work just fine.

This invisible bounds mesh is created automatically when you use Visibility.Dynamic or Visibility.Always. Whatever mesh renderer the GameObject might be using is replaced by the bounds mesh (don't worry, not permanently — only at runtime). **Note**: as an optimization, only one bounds mesh is created per VectorLine name. So all VectorLines called "Enemy", for example, will have the same bounds mesh. This means it's highly recommended to use different names for different types of VectorLine objects, and not just call every line "X" or something. Of course, if you have a bunch of objects that all look alike, such as the tanks in the Tank Zone demo, it's fine for them to use the same bounds mesh.

So, why might you use false for makeBounds, anyway? You might if you're using Visibility.Dynamic or Visibility.Always, and you actually *don't* want the GameObject's mesh to be replaced. For example, the Simple3D3 script in the VectrosityDemos package uses false for makeBounds, so that the cube mesh is still visible. This makes a solid-shaded cube with vector line highlights, for a nifty wireframe effect.

SetBrightnessParameters

When using Brightness.Fog, you need some way to control the look. There are five parameters: minimum brightness distance, maximum brightness distance, levels, distance check frequency, and fog color.

Total Fadeout Distance: The distance from the camera at which brightness will be faded out all the way. The default is 500. Anything beyond this distance will be drawn with only the fog color.

Full Brightness Distance: The distance from the camera at which brightness will be at the maximum. The default is 250. Anything closer than this distance will be drawn at max brightness. Anything between the full brightness and total fadeout distances will be proportionally lerped between the line color and the fog color.

Levels: The number of brightness levels, with the default being 32. This simulates limited color precision, where there are visible “steps” between each level. For a smoother fade, use a higher number.

Distance Check Frequency: How often the brightness control routine is run on objects that use Brightness.Fog. The default is .2, which is 5 times per second. You might want this to run more often if you have more brightness levels, or fast-moving objects.

Fog Color: The color which objects fade to as they approach the maximum brightness distance. This is black by default. Usually you want this to be the same as the background color.

An example where the minimum brightness distance is 600, the max is 200, there are 64 brightness levels, the routine runs 10 times per second, and fades to a dark blue:

```
VectorManager.SetBrightnessParameters (600.0, 200.0, 64, .1, Color(0, 0, .25));
```

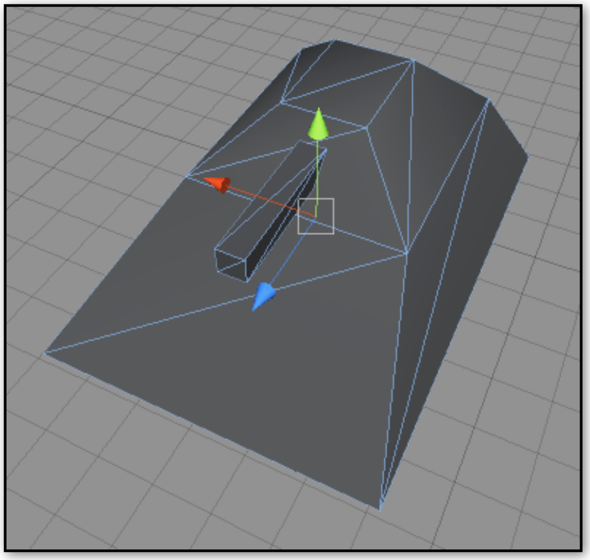
GetBrightnessValue

If you are doing some effects where it would be useful to know what brightness a 3D vector object should be at a certain distance, then you can use VectorManager.GetBrightnessValue (Tank Zone uses it in a couple of places). If you pass in a Vector3 in world space, typically from a GameObject’s transform.position, it returns a value between 0.0 and 1.0, where 1.0 would be 100% brightness and 0.0 would be 0% brightness.

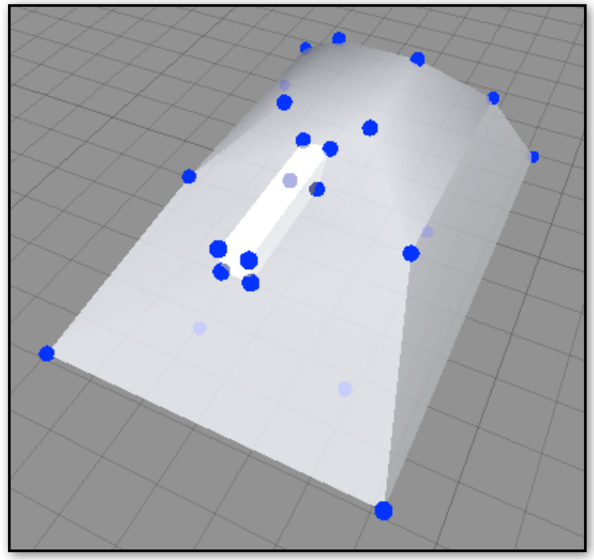
```
Debug.Log (VectorManager.GetBrightnessValue (transform.position));
```

If you have complex 3D or 2D vector shapes you want to use, you can use LineMaker to make the process quick and easy. Make sure you've imported the Vectrosity4EditorScripts package first.

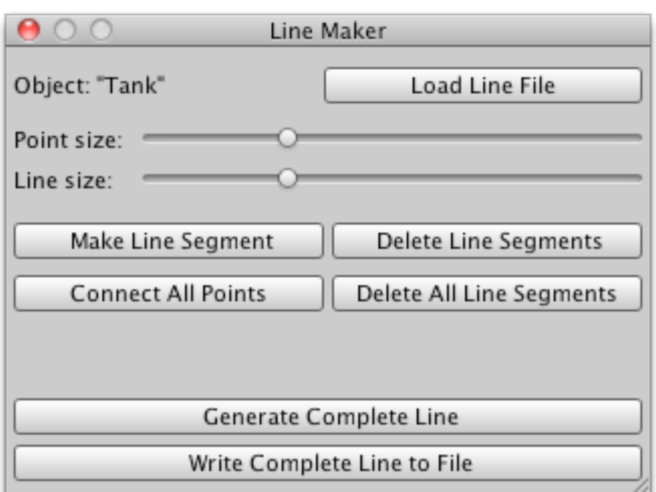
First, make a mesh in your 3D app of choice. Ideally this should be reasonably low-poly…LineMaker can get a little slow with high-poly objects. Drag the mesh into your scene.



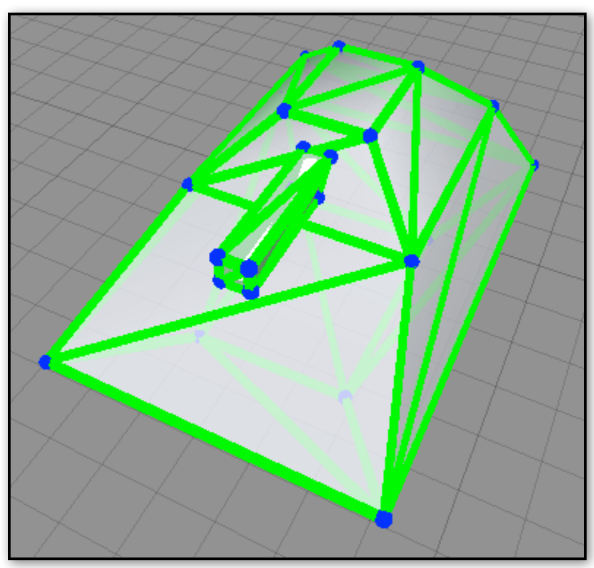
Then, with the object selected, choose **LineMaker…** from the Assets menu. Your mesh will become transparent, with blue dots placed at each vertex, and the LineMaker window will appear.



The LineMaker window has a number of controls. At the top, under the name of the object, are two sliders that control the size of the points and lines that make up the 3D vector object. You can adjust these depending on the size of your mesh, in order to make working with it easy. When done making the vector line, you can close this window, and the mesh will be restored to its normal state.



If you click on "Connect All Points", all points in the mesh will automatically be connected by green lines. These lines are what your 3D vector object will look like. You may find it easier to connect all points first, and then remove whatever line segments you don't want, rather than building it up from scratch.



To make a line segment, select two points in the scene, then click on “Make Line Segment” in the LineMaker window. Continue to do this for all line segments, rotating the view as necessary to get at all points (don’t rotate the object), until your shape is complete. Remember that only two points should be selected for each segment.

You can click “Delete All Line Segments” to delete everything and start over. To delete individual line segments, select one or more in the scene, then click “Delete Line Segments”. You can also delete selected line segments using the Command+Delete (or Control+Delete on Windows) key combination.

No matter how you end up making line segments, when you’re done, you have two options for saving the 3D vector shape. The first way is to click on “Generate Complete Line”. This creates a line of text that contains the points and copies it to the system clipboard. You should then paste this text into a script, inside a Vector array. An empty array looks like this:

```
var tankLines = [];
```

Paste the text between the brackets (this example uses just two points for brevity...the real thing would be quite a bit longer!):

```
var tankLines = [Vector3(1.748, -2, -2.513), Vector3(3.497, -.814, -5.0131)];
```

C# NOTE: If you’re using C# instead of Javascript, first open the LineMaker script and change the variable at the top from “useCsharp = false” to “useCsharp = true”. A C# Vector3 array looks like this:

```
Vector3[] tankLines = {new Vector3(1.748f, -2f, -2.513f), new Vector3(3.497f, -.814f,
-5.0131f)};
```

You may prefer to use TextAssets for the shapes instead of long strings of Vector3 array data. In this case you can click on “Write Complete Line to File”, and save the TextAsset somewhere in your project. Refer to **BytesToVector3Array** in the **Vector Utilities** section above for information on how to use these files.

If the shape you’re using exists only on the X/Y plane and all the Z coordinates are the same, then you’ll have the option of using a 2D array. There will be two additional “Vector2” buttons at the bottom, which do the same thing as the normal buttons to the left, but generate Vector2 arrays instead of Vector3 arrays.

Note that any meshes you use that have no triangles, but only edges, will be unable to use the “Connect All Points” button, and in this case you must connect all points manually. Also note that if you rotate/scale the object, you should do that before using LineMaker. Any rotation or scaling after you start LineMaker won’t be reflected in the line data that’s generated.

If you select one or more line segments in the scene and switch focus back to the LineMaker window, you’ll see a line of text that lists indices for those line segments. These are the array indices you would use for color and line width arrays. You might want to know this information if you plan to set specific line segments to certain colors or widths.

Finally, the “Load Line File” button does pretty much what it says: it loads in a previously-saved line file so you can make additional changes. If you load in a file, you’ll see that the “Connect All Points” button is replaced by a “Restore Loaded Lines” button. That’s because when you save a line file, only the line segment data is saved, and the actual mesh data is lost and can’t be reconstructed. So, after loading a file, LineMaker can no longer figure out how to connect all points. Instead, “Restore Loaded Lines” will restore the lines to the state they were in right after you loaded the file.

Q: I can't see any lines!

A: If you're supplying your own color, make sure the alpha value is non-zero, or else the line will be transparent.

If you're using VectorManager for 3D shapes, be aware that the scene view camera can interfere with the Visibility scripts. To avoid this, just make sure the scene view isn't active when running. The easiest way to do that is to use Maximize On Play.

If you're using a line material that reacts to lighting, you should set the vector canvas to OverlayCamera. The easiest way to do that is by using VectorLine.SetCanvasCamera with the appropriate camera.

Q: I get error messages when I try to import Vectrosity into my project.

A: Make sure you're using Unity 4.6 or later, since Vectrosity 4 uses some features not available in earlier versions of Unity. Also, make sure you haven't imported both the Vectrosity .dll and the source code—you can only import one or the other.

Q: How can I get the best speed?

A: Make sure you don't recreate lines unnecessarily. Don't destroy and remake lines every frame. For the most part, you should only create lines once in Start or Awake, and then manipulate the lines by changing already-existing points. For dynamic lines such as those used in touchscreen line-drawing routines, make use of VectorLine.endPointsUpdate to update only the last part of the line as needed, rather than constantly redrawing the entire line. See the DrawLinesTouch or DrawLinesMouse example scripts in the VectrosityDemos package for examples of this.

For the absolute best speed, stick to Draw rather than Draw3D. Remember that Draw can still draw lines made with Vector3 points, though they will be drawn on top of everything else. Only use Draw3D when the line really has to be drawn "inside" the scene along with other objects. Also, continuous lines are a little more efficient than discrete lines, so use continuous where possible, and if you're drawing thick lines, use Joins.Fill instead of Joins.Weld if it's feasible.

Finally, only call Draw when you actually need to. For example, there's no reason to put Draw in an Update function if the line only changes occasionally. However, note that 3D lines do need to be updated whenever the camera moves (unlike standard lines), so you can usually use Draw3DAuto for those.

Q: I get error messages when I try to build for mobile.

A: Make sure you haven't included any scripts from Tank Zone. The Tank Zone demo scripts use dynamic typing, which isn't available for iOS or Android builds.

Q: I get an error when compiling for iOS.

A: Make sure the Vectrosity .dll in Unity is named "Vectrosity.dll" exactly; if it's renamed it won't work.

Q: I'm using VectorLine.rectTransform, and it's messing up the lines.

A: In most cases you shouldn't use rectTransform unless you have a very good understanding of how Vectrosity works. Consider it an "advanced" feature, which can be used for optimization in certain circumstances, but it's usually not necessary, and most things in Vectrosity can be accomplished without using it. Typically you would pass in the transform of an object using .drawTransform, and then manipulate that transform, instead of using rectTransform.

Q: I get an error about an unknown identifier when I try to do anything.

A: Make sure you import the Vectrosity namespace in your scripts. That's "import Vectrosity;" for Unityscript and Boo, and "using Vectrosity;" for C#.

Q: My lines look weird and don't seem to be facing the camera.

A: Vectrosity is probably using a different camera than the one you want. It can happen that you accidentally have a duplicate camera somewhere, in which case getting rid of it will fix the problem (and make your project run faster too). If you have a multi-camera setup and are using 3D lines, be sure to use VectorLine.SetCamera3D with the correct camera.

Q: I tried using GetPoint but it's not working right.

A: Make sure you call SetDistances first, before calling GetPoint, if you've changed the VectorLine.

Q: How can I use Vectrosity with the Oculus Rift?

A: For each camera, you will need to call SetCamera3D followed by Draw3D for any lines you're using.

APPENDIX (Project settings for Tank Zone)

The Tank Zone demo game needs some specific project settings in order to run correctly, and these settings won’t import with the Vectrosity package. Therefore, you should add 4 entries to the Input Manager, and use the settings below.

▼ KeyLeft

Name	KeyLeft
Descriptive Name	Left stick up
Descriptive Negative Name	Left stick down
Negative Button	s
Positive Button	w
Alt Negative Button	
Alt Positive Button	
Gravity	3
Dead	0.001
Sensitivity	3
Snap	☑
Invert	☐
Type	Key or Mouse Button
Axis	Y axis
Joy Num	Get Motion from all Joysticks

▼ LeftStick

Name	LeftStick
Descriptive Name	Left stick up
Descriptive Negative Name	Left stick down
Negative Button	
Positive Button	
Alt Negative Button	
Alt Positive Button	
Gravity	3
Dead	0.2
Sensitivity	3
Snap	☑
Invert	☑
Type	Joystick Axis
Axis	Y axis
Joy Num	Get Motion from all Joysticks

▼ KeyRight

Name	KeyRight
Descriptive Name	Right stick up
Descriptive Negative Name	Right stick down
Negative Button	k
Positive Button	i
Alt Negative Button	
Alt Positive Button	
Gravity	3
Dead	0.001
Sensitivity	3
Snap	☑
Invert	☐
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

▼ RightStick

Name	RightStick
Descriptive Name	Right stick up
Descriptive Negative Name	Right stick down
Negative Button	
Positive Button	
Alt Negative Button	
Alt Positive Button	
Gravity	3
Dead	0.2
Sensitivity	3
Snap	☑
Invert	☑
Type	Joystick Axis
Axis	4th axis (Joysticks)
Joy Num	Get Motion from all Joysticks