

VECTROSITY

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 **VectrosityDemos** Unitypackage. (Make sure to import either the **Vectrosity** or **VectrositySource** 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 is written for Unity 4.0 or later, and will not work with earlier versions.

[What's Included \(page 2\)](#): What you get in the Vectrosity package.

[Basic Line Drawing \(page 3\)](#): Just draw a line!

[Setting Up Lines \(page 6\)](#): Information about lines, including textures, materials, and various line parameters.

[Drawing Lines \(page 12\)](#): Putting lines on the screen after they've been set up.

[Setting the Camera \(page 14\)](#): Information about the vector camera.

[Moving Lines Around \(page 15\)](#): Moving, rotating, scaling entire lines at once.

[Removing VectorLines \(page 17\)](#): What to do when you just don't want a line anymore.

[Resizing VectorLines \(page 18\)](#): When the original size isn't enough.

[Line Extras \(page 19\)](#): Miscellaneous things you can do with lines, including partial lines, layers, and more.

[Uniform-Scaled Textures \(page 24\)](#): How to make things like dotted and dashed lines.

[End Caps \(page 26\)](#): Arrow heads, rounded ends, and other things you can put at the ends of lines.

[Line Widths \(page 28\)](#): Different widths for different line segments.

[Line Colors \(page 29\)](#): Assigning colors to line segments.

[Line Colliders \(page 30\)](#): Have lines be interactive with 2D colliders.

[Drawing Points \(page 32\)](#): Making dots instead of lines.

[Single-Pixel Lines and Points \(page 33\)](#): If your lines are 1 pixel thick, read this for extra efficiency.

[Vector Utilities \(page 34\)](#): Various things to make line creation easier, such as boxes, curves, selection, etc.

[3D Lines and Viewport Lines \(page 47\)](#): Lines that exist in the scene, and lines made with viewport coords.

[Vector Manager \(page 49\)](#): Utilities for working with 3D vector objects.

[LineMaker \(page 52\)](#): An editor utility to help create 3D vector objects.

[Tips and Troubleshooting \(page 54\)](#): Q & A for common problems.

[Appendix \(Project Settings for Tank Zone\) \(page 56\)](#): Use these to set up the Tank Zone example.

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

There are several Unity packages:

- **Vectrosity_Unity4_0:** The core Vectrosity package, which contains all Vectrosity scripts in a .dll. Since it's a .NET .dll, it will work with any Unity license, Free or Pro. This is normally what you should use if you're using Unity 4.0 through Unity 4.2. If you use the .dll, make sure you don't import the source code.
- **Vectrosity_Unity4_3:** This is the same as above, but contains some features that will only work with Unity 4.3 or later. Make sure you import only one .dll.
- **VectrositySource:** This contains the core Vectrosity scripts as source code. It will work with any version of Unity as long as it's at least Unity 4.0. You'd generally want to use the .dll package instead, since it can potentially reduce script compilation time, and any errors you might make while using Vectrosity functions will point to the relevant line in your scripts (when double-clicking the error in the Unity console), rather than the source code. However, you can still use the source if you want to make modifications. If you're using the source, make sure you're not using either of the .dlls.
- **VectrosityEditorScripts:** The [LineMaker](#) editor script, which allows you to easily make vector lines from 3D objects. This is optional, but in order to work, one of the above packages should be imported first.
- **VectrosityDemos:** Many demonstrations of various Vectrosity functions in a number of scenes. You should import the .dll or the source code before importing this package. The demo scenes are located in the "VectrosityDemos/_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.
- **TankZone:** A complete game project that uses Vectrosity for all of the graphics. As with the demos, you should import the .dll or source first. You can load the _TankZone scene and play the game, read the scripts, and generally mess around. (But please note that the Tank Zone assets are ©2014 Starscene Software. Feel free to do whatever you like with them for your own use, but redistribution isn't allowed. Also, it's intended for the desktop license of Unity and won't run on iOS or Android without some work.) See the [Appendix](#) for some settings that you'll need for Tank Zone to work correctly.

There are some additional 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 Upgrade Guide:** Have a look at this if you're upgrading from an older version of Vectrosity. 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.

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#
```

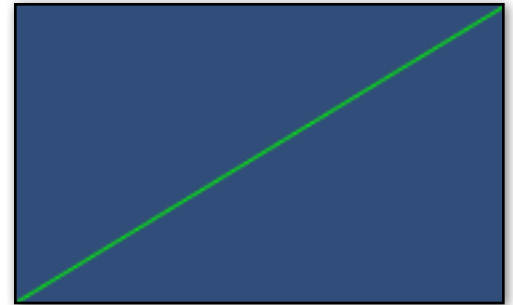
at the top of any script that uses Vectrosity. Note that most of the scripts in this documentation assume that you're importing the namespace. So if you get errors, make sure you've included this line first.

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 this Javascript code into a script and save it:

```
import Vectrosity;

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

(You can use C# or Boo as well; Javascript is frequently used in code examples in this documentation for the sake of simplicity. If you're using C#, you need to add "new" in front of each `Vector2`, and "function Start" would be "void Start".) 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.

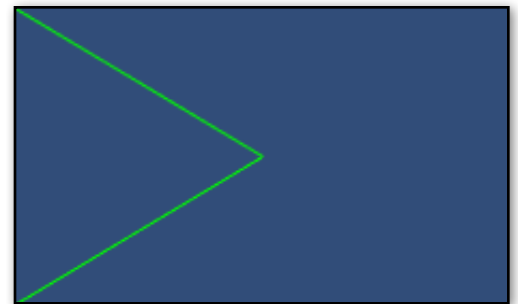


Every point you add in `SetLine` will create an additional line segment, so let's extend the above example:

```
function Start () {
    VectorLine.SetLine (Color.green, Vector2(0, 0), Vector2(Screen.width/2, Screen.height/2),
    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 this image:

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.



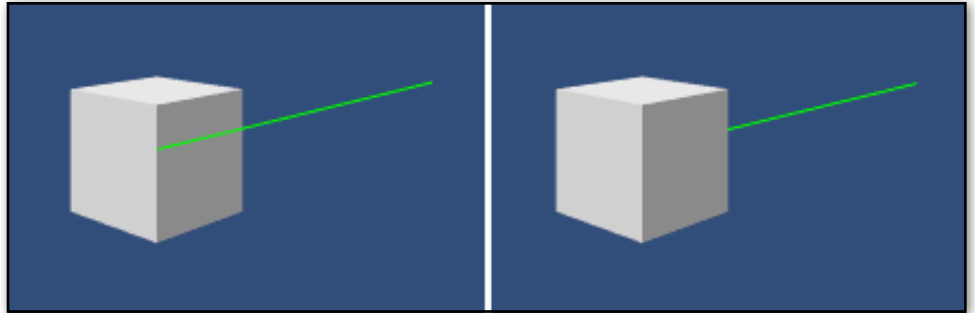
```
function Start () {
    VectorLine.SetRay (Color.green, transform.position, transform.forward * 5.0);
}
```

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**:

```
function Start () {  
    VectorLine.SetRay3D (Color.green, transform.position, transform.forward * 5.0);  
}
```

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,

```
function Start () {  
    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:

```
function Start () {  
    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 draw a diagonal line across the screen, then if you press the space key, it will flip the line:

```
import Vectrosity;

private var myLine : VectorLine;

function Start () {
    myLine = VectorLine.SetLine (Color.green, Vector2(0, 0), Vector2(Screen.width,
Screen.height));
}

function Update () {
    if (Input.GetKeyDown (KeyCode.Space)) {
        myLine.points2[0] = Vector2(0, Screen.height);
        myLine.points2[1] = Vector2(Screen.width, 0);
        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. Finally, the Update function alters and re-draws the line when a certain action is taken.

This VectorLine object contains a **points2** array, with one entry for each point that was defined when using SetLine. We used two points, so the points2 array has two entries. (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 of points that makes up the line, 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.

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". Instead they are called "Vector" + the supplied name, which is also used for the line mesh. 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, "print(myLine.name);". If you change the name of a VectorLine, the object and mesh names also change.

Line points

In order to set up a line, you need some points. Often this is done with a Vector2 array, but you can also use a Vector3 array. 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.

To set up some points for a line, declare a Vector2 or Vector3 array:

```
var linePoints = new Vector2[2]; // JS & C#
```

That code creates a Vector2 array with 2 points. You can have a maximum of 32,766 points per line when using discrete lines (the default), or 16,384 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. You can define the points later, or you can define them when you're declaring the array:

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

It's fine if you only use some of the points in the array and leave others zeroed out. You can change the points in the array at any time, and call `VectorLine.Draw` to update the line with the new points.

It's also possible to declare the array of points inline when declaring the `VectorLine`:

```
var myLine = VectorLine("MyLine", new Vector2[100], lineMaterial, 2.0); // JS
var myLine = new VectorLine("MyLine", new Vector2[100], lineMaterial, 2.0f); // C#
```

You might do this if you're going to be using utilities like `VectorLine.MakeCurve`, where you just need some points in the line and don't want to bother with a separate array.

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, the material should use an unlit shader, and it should use vertex colors if you want to specify line segment colors using Vectrosity. Many of the built-in particle shaders are good for this, such as `Particles/Additive`, or `Particles/Additive (Soft)`. With some shaders you can also have a material color that's applied on top of vertex colors, such as with the `Particles/Alpha Blended` shader. It's generally better to use a shader that doesn't require normals, but if you need to, you can 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, however, that if you want to use the depth property of lines, so that lines draw on top of or underneath each other in the correct order, the shader also needs to write to the zbuffer. See the `ParticleZwrite` shader in the `Shaders` folder in the `VectrosityDemos` project, since the standard particle shaders don't do this.

Several other shaders in the `VectrosityDemos` project are appropriate for different types of lines. `SolidColor` is an extremely simple shader that's appropriate for single-color lines that don't use the depth property. `Unlit` is for single-color lines that do use the depth property, and can use a texture. `UnlitAlpha` is the same, except it supports textures with alpha, and doesn't have zbuffer writing (this can potentially look ugly with alpha when lines overlap at different depths). In some cases you may need to experiment a little to get something that works well with your setup.

Also note that some shaders don't appear to work with deferred rendering with Vectrosity's camera setup, such as the built-in particle shaders. If your lines don't show up, try a different shader.

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. Read on for some additional optional parameters you can use....

Color

By default, lines are white. If you're using a material that allows you to set the color in the inspector, you can use the material color to change this. An alternative is to supply the color when declaring the `VectorLine`. This only works if the material uses a shader that allows vertex colors, such as most of the built-in particle shaders, or if you use the default Vectrosity-generated material.

```
var lineColor = Color(.5, .25, .75);  
var myLine = VectorLine("MyLine", linePoints, lineColor, lineMaterial, 2.0);
```

You can, of course, use one of Unity's default colors instead of a `Color` variable if you want, such as `Color.red` and so on:

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

It's also possible to pass in an array of colors. In this case, each entry in the `Color` array corresponds to a line segment in the `VectorLine`:

```
var lineColors = new Color[linePoints.Length/2];  
var myLine = VectorLine("MyLine", linePoints, lineColors, lineMaterial, 2.0);
```

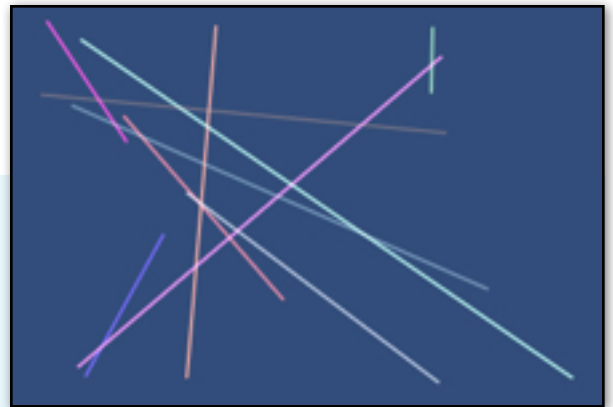
How many line segments are in a `VectorLine` depends on the number of points in the array, and the `LineType` (which is described below). By default, the number of segments is half the number of entries in the points array, which is why the example above uses `"linePoints.Length/2"`. If you use a continuous line, then the number of line segments is the number of entries in the points array minus one. So in this case, you'd do:

```
var lineColors = new Color[linePoints.Length-1];
```

If you want to apply different colors to lines after they're created, you can use `VectorLine.SetColor` or `VectorLine.SetColors` (see [Line Colors](#) below).

Here's a complete script that will create a line made of random line segments with random colors. You can use a new empty scene that has only a camera in it (which should be tagged "MainCamera") and attach this script to the camera.

```
var lineWidth = 2.0;  
var numberOfPoints = 50; // Should be an even  
    number, since we use a discrete line  
  
function Start () {  
    var linePoints = new Vector2[numberOfPoints];  
    for (p in linePoints)  
        p = Vector2(Random.Range(0, Screen.width), Random.Range(0, Screen.height));  
    var lineColors = new Color[numberOfPoints/2];  
    for (c in lineColors)  
        c = Color(Random.value, Random.value, Random.value);  
    var line = new VectorLine("Line", linePoints, lineColors, null, lineWidth);  
    line.Draw();  
}
```



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, 2.0, LineType.Discrete);
```

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

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

A discrete line means each line segment is drawn individually, and is defined by two points. In many cases this is the most efficient way of drawing several lines (especially since each line segment can have its own width and color), since it uses just one draw call no matter how many line segments you use in a single VectorLine object.

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. This means only a single line is possible in a VectorLine object that's drawn this way, but on the other hand it's a little faster to draw. In some cases it can be simpler to use several continuous VectorLines instead of one discrete VectorLine, so in situations where you're not constrained by draw calls, don't kill yourself trying to cram everything into a single discrete VectorLine.



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

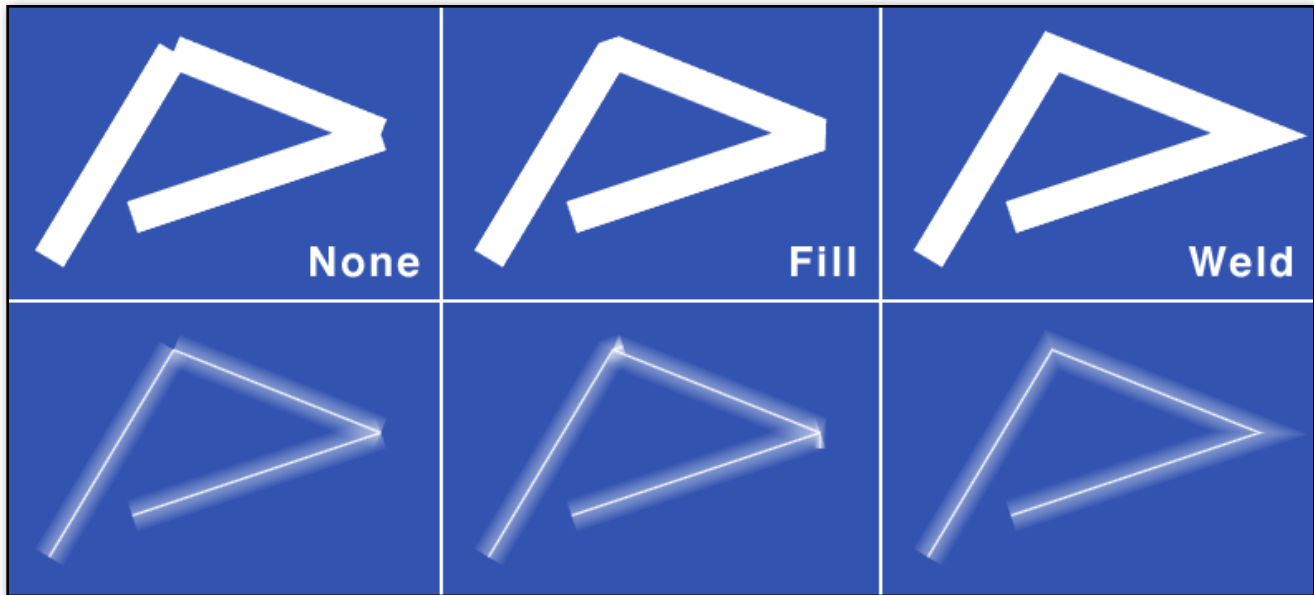
Since discrete lines take two Vector2 or Vector3 points to describe each line segment, they must use a points array with an even length. Continuous lines can have a points array of any size. They can also use Joins.Fill (see below), whereas discrete lines can't.

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, 2.0, LineType.Continuous, Joins.Fill);
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0, 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, and is efficient, since it only adds some triangles to line meshes, and isn't any slower than `Joins.None` when drawn with `VectorLine.Draw`. It has a couple of drawbacks though: 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.

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.

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 some more 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 they aren't included in the declaration (otherwise it would start getting complicated and messy). The additional options include: active, capLength, depth, layer, drawStart, drawEnd, minDrawIndex, maxDrawIndex, smoothWidth, 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();
```

Since a line is actually a mesh, once it's drawn, `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.

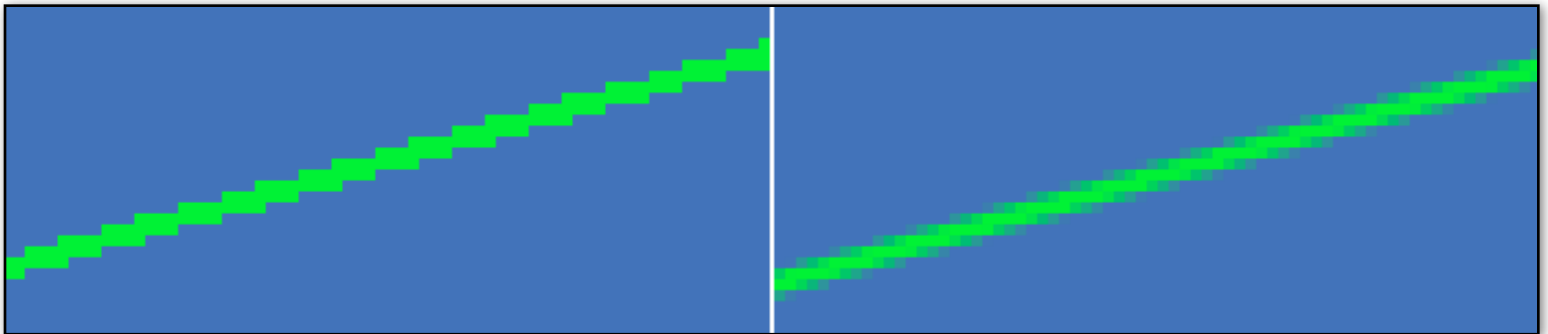
If you need to draw lines that exist in 3D space, then you can use **Draw3D**. See [3D Lines](#) for more information on 3D line drawing. It's also possible to draw points instead of line segments. See [Drawing Points](#) for more information about that.

“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 `UnlitAlpha` from the `VectrosityDemos` package. 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:



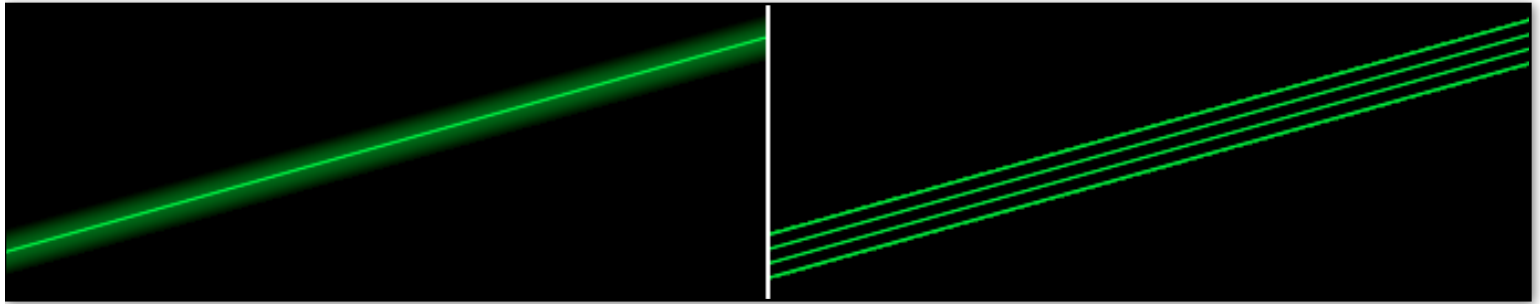
Make sure the texture is set to bilinear filtering and not point. The line anti-aliasing works by taking advantage of the inherent anti-aliasing you get when using bilinear filtering, as illustrated below:



Left: a close-up of a line using a material with the `SolidColor` shader and no FSAA. Right: a line using the `UnlitAlpha` shader and the `ThinLine` texture, and still no FSAA. The AA comes from the texture itself.

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 uses its own camera for drawing lines, but as long as the camera you're using is tagged **"MainCamera"**, you don't need to do anything yourself most of the time. So you may want to skip this section for now. However, you can also set up the vector camera manually, which you'll need to do if you want to use anything other than the default parameters, or if you're using a complex camera setup.

The basic format for setting the camera manually is:

```
VectorLine.SetCamera();
```

This normally only has to be done once, such as in an Awake function. If you allow resolution switching in your project, it has to be called once after every resolution change. SetCamera uses info from your normal camera. As mentioned, by default it uses the first camera in the scene tagged "MainCamera". The vector camera will survive level changes; however, you may need to call SetCamera again after changing levels, unless the normal camera also survives level changes (see the note below about layers).

You can optionally pass a specific camera instead of relying on the "MainCamera" tag:

```
VectorLine.SetCamera (camera);
```

That, for example, would use the camera component of whatever object the script is attached to.

By default SetCamera uses **CameraClearFlags.DepthOnly** for the vector camera's clear flags. This is so you can see your normal camera view under the lines. In certain cases you may want to change that—for instance, the Xray demo scene in the VectrosityDemos package uses a black background for the vector camera. In this case you can pass different clear flags:

```
VectorLine.SetCamera (CameraClearFlags.SolidColor); // Or pass both at once:  
VectorLine.SetCamera (camera, CameraClearFlags.SolidColor);
```

Note that VectorLine.SetCamera returns the vector camera, so you can assign the result to your own variable if you want to do additional things with the vector camera after it's been created.

The vector camera's layer can be set with **SetVectorCamDepth**. Normally the vector camera is drawn on top of the normal camera; you'd only change this for special effects, such as the Xray demo does.

You can also add "true" as the last or only parameter — for example, VectorLine.SetCamera(camera, true) — to make the vector camera use orthographic mode. The visual difference is normally small at best, but orthographic mode may render lines slightly more accurately. On the other hand, lines with Vector3 points can show glitches in certain circumstances, such as when they're behind the camera, which is why it's false by default. If you don't use Vector3 lines, you might as well set orthographic mode to true.

A NOTE ABOUT LAYERS: Normally you don't have to worry about this, and things will just work, but if you have a more complex camera setup, it's best to be aware of how things work behind the scenes. So: in order for Vectrosity to work, it creates meshes which are drawn by the vector camera (for 2D lines, anyway). You don't normally want these to be seen by any other camera. In order to accomplish this, the vector camera only sees user layer 31, which is where the line meshes are drawn. When you call SetCamera, your normal camera is set to ignore layer 31 (other culling mask settings are untouched). You may find it helpful to use the layer manager in Unity to name layer 31 "Vector" or something similar. Otherwise it's just called "Unnamed 31". If you need to change the layer used, you can do this by setting "VectorLine.vectorLayer = x", where x is the layer number you want. Do this before calling SetCamera. So if you notice weird things happening like unwanted lines being seen, **make sure your normal camera always ignores the vector layer**.

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 pretty 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 vectorObject

Warning! In general you shouldn't use the vectorObject property unless you know what you're doing and are using it for optimization purposes. It's not necessary to use this and doing so can easily cause Vectrosity to break. The information is included here for advanced users. If you have any doubts, don't use vectorObject; everything can be accomplished without it.

That said, it's possible to move the line itself directly with vectorObject, since technically it's an object in 3D space. In this case you can use **VectorLine.vectorObject** to refer to the actual line GameObject. For example,

```
myLine.vectorObject.transform.Rotate (Vector3.forward * 45.0);
```

That would rotate the line 45° around the Z axis. Note that since lines are drawn as flat 2D objects, rotating around the X or Y axes this way won't work very well, and neither will moving it forward or back on the Z axis. So, as mentioned, generally you probably don't want to mess with the VectorLine.vectorObject variable unless you know what you're doing. Most of the time, you would want to pass in the transform of an object using the drawTransform property to move or rotate VectorLine objects on any axis, as well as scaling them, and they will still be drawn correctly, since in this case the line mesh itself isn't actually rotated or translated —instead the line points are just calculated that way.

However, it can occasionally be useful to use VectorLine.vectorObject in certain specific cases, since moving objects in this manner doesn't require the line to be re-drawn, so it's somewhat more efficient. Think of the little triangular spaceship in old vector games like Asteroids...you could draw the ship once using VectorLine.Draw, and from then on you could move it around and rotate it using VectorLine.vectorObject.

Since some Unity objects are made when creating a VectorLine object, simply setting it to null won't remove those objects. (And for the technically inclined, it doesn't seem to be possible to remove Unity objects from inside a class's destructor.) Instead, you should use **VectorLine.Destroy**:

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

This will properly dispose of the VectorLine. 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 is usually used when you're using a GameObject's transform to control a VectorLine object:

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

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.

Note that these functions use reference parameters for the VectorLine, so C# users need to specify "ref":

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

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). In that case, destroying the GameObject passed into ObjectSetup will destroy the respective VectorLine automatically.

Though it's usually better to use a fixed array size and keep the unused points as `Vector2.zero` or `Vector3.zero`, at times you may need to actually change the number of points in a line. One possibility is to destroy the line using `VectorLine.Destroy`, and then recreate the line with the new points. However, a more convenient and efficient way is to use `VectorLine.Resize`. There are a couple different ways to do this, and the first one works like this:

```
myLine.Resize (linePoints);
```

This assumes "myLine" is the `VectorLine` you want to resize, and "linePoints" is a `Vector2` (or `Vector3`) array containing the new points. In this case, the existing line is kept as-is, except it's rebuilt with the new `Vector2` or `Vector3` array. You then call `VectorLine.Draw` in order to update the line on-screen with the new points. For an example of line resizing in action, see the `DrawGrid` script in the Vectrosity demos — the line is resized when the number of grid lines increase or decrease.

The second way works by passing in an integer:

```
myLine.Resize (50);
```

In this case, the line is resized to 50 points, which are either `Vector2` or `Vector3` depending on how the line was originally made. Note that the points will all be empty (or rather, `Vector2.zero` or `Vector3.zero`), and the points array that was originally used for the line won't be used any more. This means you need a new array, which you can get from the `points2` or `points3` array of the `VectorLine`:

```
myLine.Resize (50);  
var newPoints = myLine.points2; // or points3 if you used a Vector3 array originally
```

Here's an example of a simple line using 4 points:

```
var points = [Vector2(0, 0), Vector2(200, 200), Vector2(400, 0), Vector2(200, 200)];  
var myLine = new VectorLine("Line", points, null, 2.0);  
myLine.Draw();
```

And here's how you could resize it to 2 points:

```
myLine.Resize (2);  
var newPoints = myLine.points2;  
newPoints[0] = Vector2(100, 100);  
newPoints[1] = Vector2(150, 150);  
myLine.Draw();
```

Or, you could use the first technique:

```
var newPoints = [Vector2(100, 100), Vector2(150, 150)];  
myLine.Resize (newPoints);  
myLine.Draw();
```

Use whatever technique is more appropriate for your code.

Note that if you were using segment colors, all colors after resizing will be set to the first color in the color array. So if you were using multiple colors, you'll have to set them again with `VectorLine.SetColors`. This also applies if you were using multiple line segment widths.

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.

Segment Cap

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 setting VectorLine.capLength after the VectorLine is created:

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

Depth

Depth is the order in which lines are drawn. This ranges from 0 to 100 (the default is 0), and is an integer. If you have overlapping lines, you can use this to control which lines are drawn on top. Lines with a higher depth are drawn on top of lines with a lower depth, **as long as the line is using a material with a shader that uses ZWrite On** (see the [Material](#) section earlier). Without zbuffer writing, lines will generally not handle depth well. The default Vectrosity-generated material does use ZWrite On, so it works fine with line depths. You set this by using VectorLine.depth:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0);  
myLine.depth = 10;
```

If you change the depth after the line has been drawn, you need to redraw the line with VectorLine.Draw in order for the depth to actually change on-screen. Note that if you're using Draw3D, depth has no effect.

Layer

Normally this is not something you'd set manually, but it can be useful for certain effects in some cases. See the note about layers in the [Setting the Camera](#) section. You set this by using VectorLine.layer:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0);  
myLine.layer = 8;
```

SortingLayerID and SortingOrder

If you're using Unity 4.3 or later, you can use the same properties that sprites use, to control the drawing order of Vectrosity lines. NOTE: this requires Draw3D and has no effect if using Draw. The sorting order can be used to make lines draw over or under sprites, as well as being a way to make lines draw over or under other lines. These properties don't require any special shader. To set the sorting layer ID to 1:

```
myLine.sortingLayerID = 1;
```

To set the sorting order within a layer:

```
myLine.sortingOrder = 5;
```

See the Unity documentation for sprites for more details about sorting layers and sorting order.

DrawStart and DrawEnd

If you want to draw only part of a line, you have two choices: you can zero out the unwanted parts in the points array, or you can set **drawStart** and **drawEnd**. The first option is less efficient, and of course it changes the actual line points data, which is frequently not what you want to do. So instead you can use drawStart and drawEnd to specify what part of the line should be drawn, but leave the points array untouched. This example makes an array of 10 points across the screen, draws the line, waits a couple of seconds, then draws the line again from point 3 through point 6, so only the middle few line segments are drawn:

```
function Start () {
    var linePoints = new Vector2[10];
    for (var i = 0; i < 10; i++) {
        linePoints[i] = Vector2(Screen.width/9 * i, Screen.height/2);
    }
    var line = new VectorLine("Line", linePoints, null, 2.0, LineType.Continuous);
    line.Draw();
    yield WaitForSeconds(2.0);
    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.

MinDrawIndex and MaxDrawIndex

There are times when you don't necessarily want to update the entire line. For example, you're making a line-drawing game, where the player can draw a free-form line on the screen. It would be *much* more efficient to set up a `VectorLine` with the maximum number of points that could be drawn, rather than constantly resizing the `VectorLine`. So you'd use a `Vector2` array that has, say, 1000 points, and starts off filled with `Vector2.zero`. When the line is drawn, you'd set entry 0 in the array first, setting additional points as the line gets longer and longer, calling `VectorLine.Draw` each time the line needs to be updated.

But when you update the line with new points, the earlier segments are always the same...it seems like kind of a waste to constantly re-draw them, especially if the line gets really long. So wouldn't it be nice if you could just update the part of the line that actually needs it? That's where **minDrawIndex** and **maxDrawIndex** come in. This can also prevent the "connecting to `Vector2.zero`" effect when using a continuous line. That is, if you have a continuous line with 1000 points and are using the first 250, then point 250 will be connected on-screen to (0, 0). Using `maxDrawIndex` correctly will prevent that part from being drawn.

`VectorLine.minDrawIndex` indicates that line drawing should start at the specified index. For example, if you have an array with 1000 points, setting `minDrawIndex` to 500 will cause `Draw` or `Draw3D` to skip over elements 0-499. Note that they're not erased; any line segments that have already been drawn will stay on-screen. The default for `minDrawIndex` is 0, naturally. When you assign a value to `minDrawIndex`, it's always clamped between 0 and the maximum array index. So, trying to use a `minDrawIndex` of -5 will actually result in it being set to 0. You set `minDrawIndex` after the line is declared:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0);  
myLine.minDrawIndex = 24;
```

You can look at the `DrawLinesTouch` and `DrawLinesMouse` scripts in the `VectrosityDemos` package for examples of `minDrawIndex` and `maxDrawIndex` in action.

Complementing `minDrawIndex`, of course, is `maxDrawIndex`, which sets the maximum index in the points array that will be drawn. For example, in an array with 1000 points, setting `maxDrawIndex` to 600 will cause any drawing routines to skip over elements 601-999. Again, segments that are skipped over aren't erased, they're just not touched when a line is re-drawn. So a `minDrawIndex` of 500 and a `maxDrawIndex` of 600 means only elements 500-600 will be updated if you call `VectorLine.Draw`.

The default for `maxDrawIndex` is the length of the points array minus one, so for an array of 50 points, the default `maxDrawIndex` will be 49. Like `minDrawIndex`, it's clamped, so with that array of 50 points, trying to use a `maxDrawIndex` of 75 will actually result in it being set to 49 again.

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0);  
myLine.maxDrawIndex = 150;
```

Both `minDrawIndex` and `maxDrawIndex` also work with `VectorLine.SetColors`. In our example array of 1000 points, with `minDrawIndex` at 500 and `maxDrawIndex` at 600, calling `VectorLine.SetColors` means only the line segments that correspond to the points of 500-600 will be updated.

Remember that `minDrawIndex` and `maxDrawIndex` are primarily optimizations, used when most of a line is unchanged, and you want to save some CPU cycles by only updating part of it. If you want to actually show only part of a line, use `drawStart` and `drawEnd` instead.

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;
```

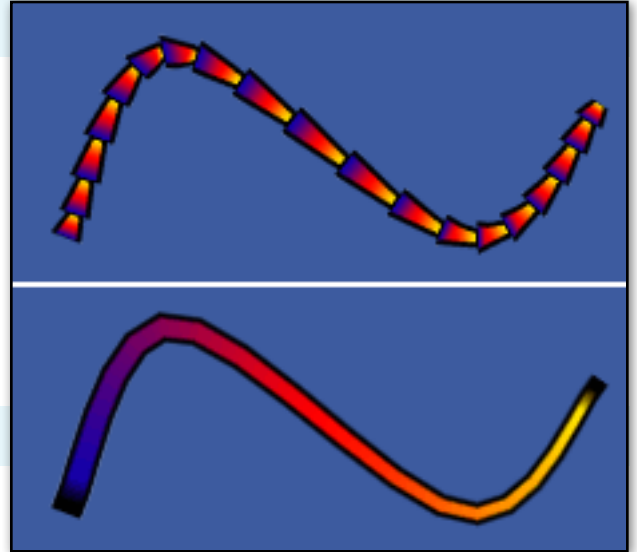
You can also read `VectorLine.material` to get the material currently being used by the `VectorLine`.

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



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`**:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0);  
myLine.lineWidth = 4.0;
```

Note that if you had been using multiple segment widths with `SetWidth` or `SetWidths`, 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.

Active

This is 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` (and similar functions) with an inactive line won't do anything.

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

Shortcut

As you've seen, there are a number of options when making lines. There are times when you want a bunch of lines that have most of the same options, but you're not using the defaults, and it would be nice if you didn't have to specify all the options again for every line. In this case you can use **VectorLine.MakeLine** as a shortcut. First you must initialize it by using **VectorLine.SetLineParameters**. For example:

```
VectorLine.SetLineParameters (lineColor, lineMaterial, lineWidth, capLength,  
                             lineDepth, LineType.Continuous, Joins.Fill);
```


This assumes "lineColor" is a Color variable, "lineMaterial" is a Material variable, "lineWidth" is a float, "capLength" is another float, and "lineDepth" is an integer. This will set defaults for any lines made with MakeLine afterward. You can use SetLineParameters to set different defaults whenever you like. Using the above example, you can then do this:

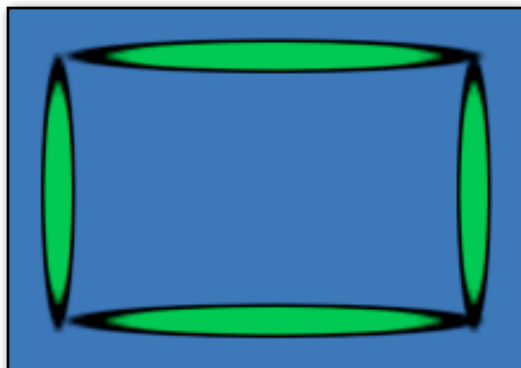
```
var myLine = VectorLine.MakeLine ("LineName", linePoints);
```

and the rest of the parameters will be as specified in SetLineParameters. You can optionally supply a different color or color array, which overrides what you supplied before:

```
var myLine = VectorLine.MakeLine ("LineName1", linePoints, Color.white);  
var anotherLine = VectorLine.MakeLine ("LineName2", linePoints, lineColors);
```

The rest of the parameters will be the same.

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:

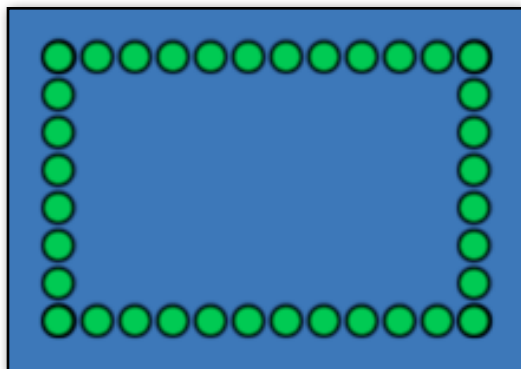
```
var lineMaterial : Material;
```

```
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();
}
```

That 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:

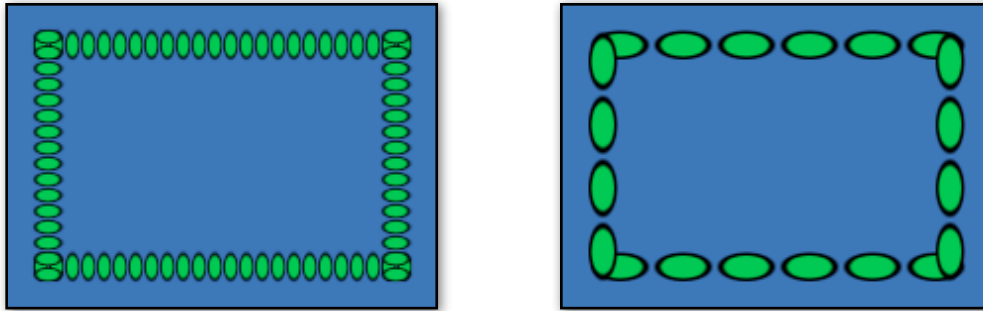
```
rectLine.textureScale = 1.0;
```

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:



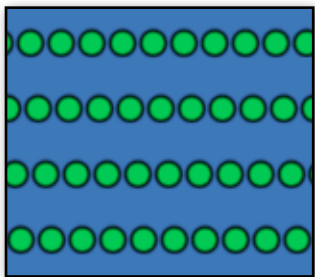
This is particularly useful for dashed lines, where you might want to make longer or shorter dashes, and it's also useful for non-square textures. This 32x16 texture, for example: ∞. A textureScale of 1.0 results in the left image, and using 2.0 results in the right image:



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 to the way it looked before you used `VectorLine.textureScale`, you can use **`VectorLine.ResetTextureScale`**:

```
myLine.ResetTextureScale();
```

This removes all custom texture scaling information from the `VectorLine` object.

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 smaller 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);
```



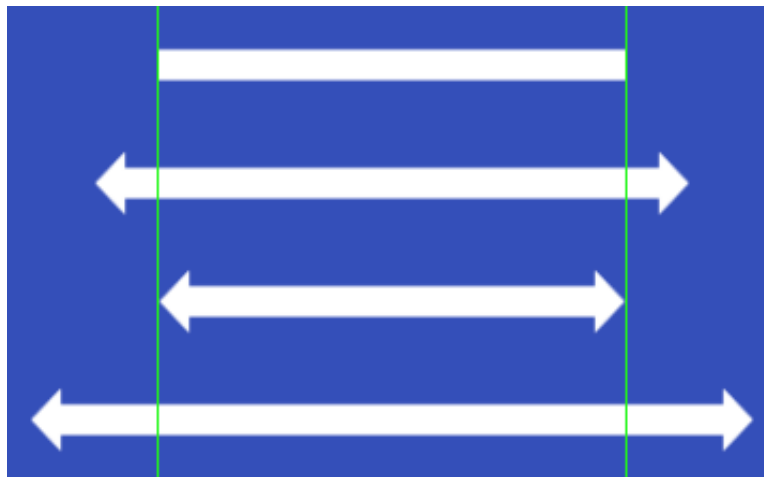
For another example of this, check out the DrawLinesTouch demo scene. Click on the Main Camera, then check “Use End Cap” on one of the scripts, and change the line width to something larger like 15.0.

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, -1.0, frontTex);
```

If left out, the default is 0.0. This illustration shows a line without end caps, followed by a line drawn with the default offset, followed by an offset of -1.0 (so the line is as long as it would have been without the end cap), and finally an offset of 1.0 (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.

Removing end caps

To remove an end cap from a line, set `VectorLine.endCap` to null or `""`. 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`.

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 be accomplished is with an array of floats or ints, plus **VectorLine.SetWidths**:

```
var myWidths = [1, 2, 3, 10, 20];
myLine.SetWidths (myWidths);
```

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 in 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 a particular segment only, which you specify by supplying a width and an index value that corresponds to that segment:

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

This example sets line segment 5 to a width of 10.0. You can use **VectorLine.GetWidth** to get the width of a particular segment:

```
Debug.Log (myLine.GetWidth (5));
```

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



After the line has been created, you can change the color or colors of a line at any time by using **VectorLine.SetColor** or **VectorLine.SetColors**. SetColor is for when you want to set a single color, and SetColors sets all the colors in the line using an array. 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
```

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 blue (remember that you start at 0 when counting segments):

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

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

```
myLine.SetColor (Color.red, 4, 12);
```

In this case, the segments starting with 4 up to and including 12 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 10, then segments 4-10 would be changed.

By comparison, SetColors takes a Color array. As always, since each color corresponds to a line segment, the length of the Color array must be the length of the Vector2 or Vector3 array used to make the VectorLine object minus one (for LineType.Continuous) or divided by two (for LineType.Discrete). If you're using points, the length of the Color array must be the same as the points array.

```
var myColors = new Color[10]; // If the line has 10 segments
for (color in myColors) {
    color = new Color(Random.value, Random.value, Random.value);
}
myLine.SetColors (myColors);
```

When SetColor or SetColors are called, the line colors are changed immediately, and you don't have to redraw the line using VectorLine.Draw. 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. If a line uses minDrawIndex or maxDrawIndex, only the relevant colors for the appropriate line segments will be updated.

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:

```
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 6 (keep in mind segments start at 0):

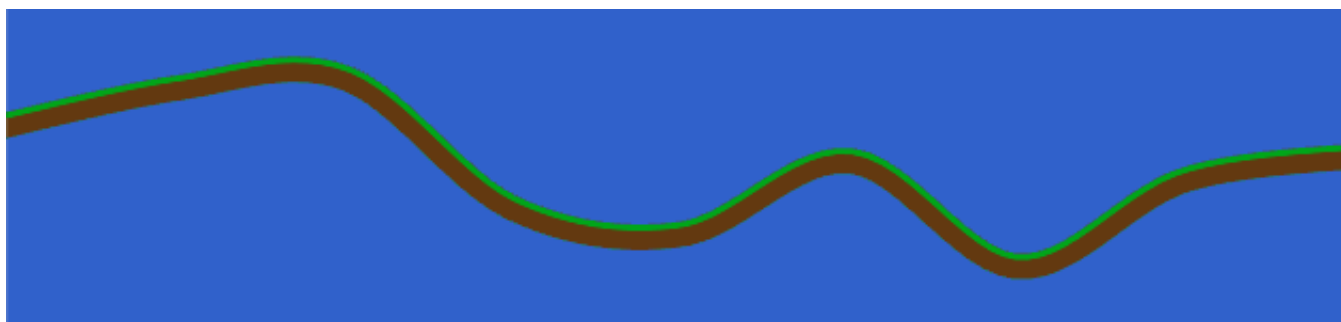
```
Debug.Log (myLine.GetColor (5));
```

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:

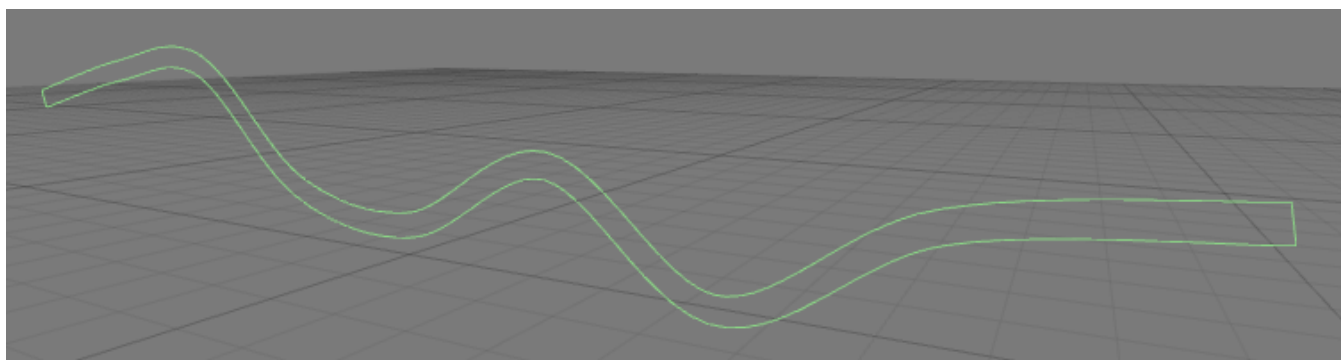
```
myLine.collider = true;
```

Note that this is for 2D physics only, and requires Unity 4.3 or later. Since it's 2D-only, this means it's on the X/Y plane, and the camera shouldn't be rotated (otherwise 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.) See the RandomHills scene in the VectrosityDemos package for an example of the collider property in action.

Lines made with Vector3 points can use colliders too. Note that 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.

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;
var lineMaterial : Material;

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

Making colliders be a trigger

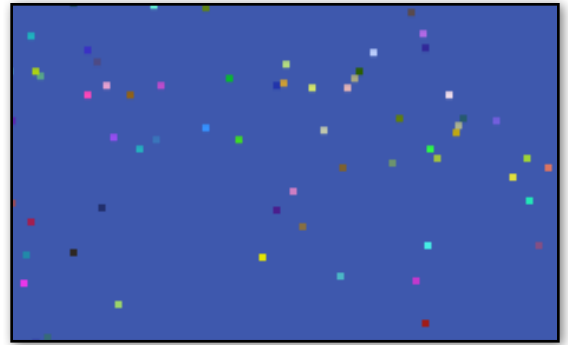
In some cases you may want to use a trigger instead of a standard collider. In this case, use `VectorLine.trigger`:

```
var myLine = new VectorLine("ColliderLine", new Vector2[100], lineMaterial, 20.0);
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 if you use

[VectorLine.useViewportCoords](#).



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, 2.0);
```

As with **VectorLines**, you can also optionally pass in an array of colors. With the color array, every entry corresponds to a point, so it should be the same length as the points array.

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

You can pass in a single color too, the same as with a **VectorLine**.

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, such as **depth**, **maxDrawIndex**, etc.

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:

```
var dotSize = 1.0;
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 Color[numberOfDots];
    for (c in dotColors)
        c = Color(Random.value, Random.value, Random.value);
    var dots = new VectorPoints("Dots", dotPoints, dotColors, null, dotSize);
    dot.Draw();
}
```


Unity has the ability to specify lines and points for meshes, in addition to triangles. Lines and points are limited to one pixel in width, but are several times faster than using triangles.

The downside is that not all platforms necessarily support these additional drawing methods well. So, by default, Vectrosity always uses triangles for all lines and points. In those cases where you're sure that your target platform has proper support for lines and points, you can specify that Vectrosity should use them where applicable, by using **useMeshLines** and **useMeshPoints**.

```
VectorLine.useMeshLines = true;
```

This causes any 1-pixel-thick lines to be drawn using MeshTopology.Lines. They can't support textures or end caps, but are faster and more efficient than triangles. Lines that are thicker than 1 pixel are drawn with triangles, regardless of whether useMeshLines is true or not.

```
VectorLine.useMeshPoints = true;
```

As above, but it applies to 1-pixel-thick points, using MeshTopology.Points.

Note that once any VectorLines have been created, the method used for drawing lines can't be changed. So you should set useMeshLines or useMeshPoints as desired before creating any VectorLines. You'll get a warning message if you attempt to change the drawing method when it's no longer possible to do so.

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

SetDepth

Aside from setting the depth of lines, you can set the depth of an arbitrary transform. This might seem like an odd thing to do, but it can occasionally be useful for special effects. The Tank Zone demo package uses this, for example, to set a plane between the green game view graphics (drawn at depth 0) and the red info graphics at the top of the screen (drawn at a higher depth). The plane is set to the vector layer (normally layer 31) so it's seen by the vector camera, and positioned so that it covers the top part of the screen. This way the green lines are blocked off from showing up behind the info graphics, since the plane is the same color as the background. You use SetDepth by passing in a transform and the desired depth:

```
VectorLine.SetDepth (transform, 10);
```

Since it's a static class that doesn't actually involve a VectorLine object, in this case you literally write "VectorLine.SetDepth" rather than using a VectorLine object.

AddNormals and AddTangents

By default, VectorLines don't use normals, and attempting to use a shader that requires normals will make Unity complain. If you want to use a shader that requires normals, however, you can call AddNormals to make Unity stop complaining:

```
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.

GetCamera

This is pretty simple: it returns the vector camera made with SetCamera. SetCamera also returns this camera, but you can call GetCamera at any time.

```
var vectorCam = VectorLine.GetCamera();
```

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], Color.yellow, 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; // Unityscript
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 Vector2 array 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 array 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 an array 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 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, 1.0);  
squareLine.MakeRect (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.



MakeCircle

This is for easily creating circles or other shapes like octagons. As with `MakeRect`, it calculates the appropriate values in a `VectorLine`'s `Vector2` or `Vector3` array; you still use `VectorLine.Draw` 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, and few segments 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);
```

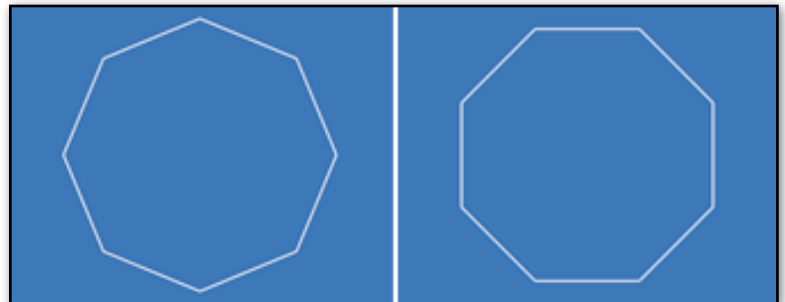
In this case, the circle will use all available segments in the `VectorLine`, so you don't have to specify the number of segments to use.

The size of the `Vector2`/`Vector3` array 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 `Vector` array must have at least 31 elements. Using 30 segments for a discrete line would require 60 elements in the `Vector` array.

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 `Vector2` array 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);
```

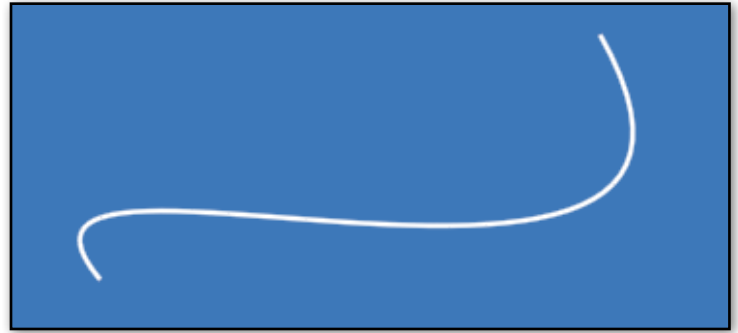
Both “xRadius” and “yRadius” are floats that specify the number of screen pixels for the respective radii (or world units if you’re using a `Vector3` line). The usage otherwise is the same as `MakeCircle`. Note that “pointRotation” only rotates the points clockwise or counterclockwise within the ellipse 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.

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, controll1, 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.

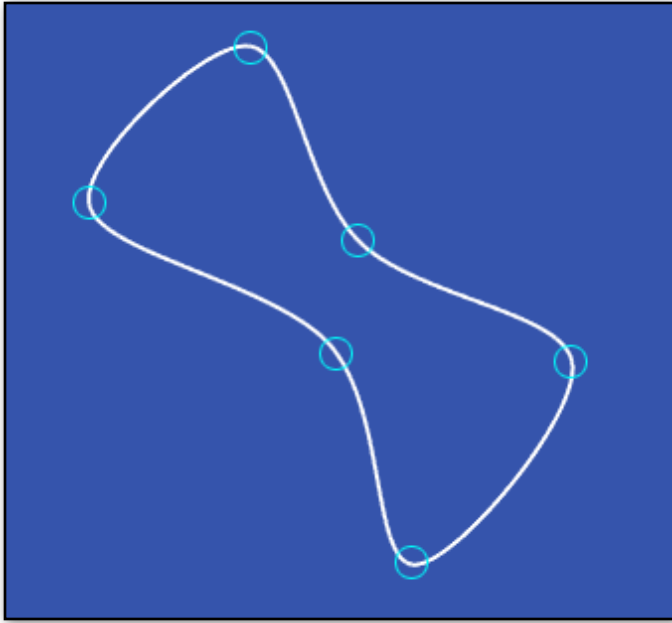
“segments” is an int, and works like it does in MakeCircle/MakeEllipse: the more segments, the smoother-looking the curve. Again, the number of elements in the Vector2 array should be the number of segments plus one for continuous lines, or twice the number of segments for discrete lines. Alternatively, 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 curve 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!", Vector2(100, 100), 30.0);
```

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 points array isn't large enough to hold them all, it's resized. If this happens, the original points array is no longer used, so you'd have to do something like “var newPoints = myLine.points2;” if you need a reference to the points after using `MakeText`.

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!", Vector2(100, 100), 30.0, .8, 1.2);
```

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!", Vector2(100, 100), 30.0, 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. 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).

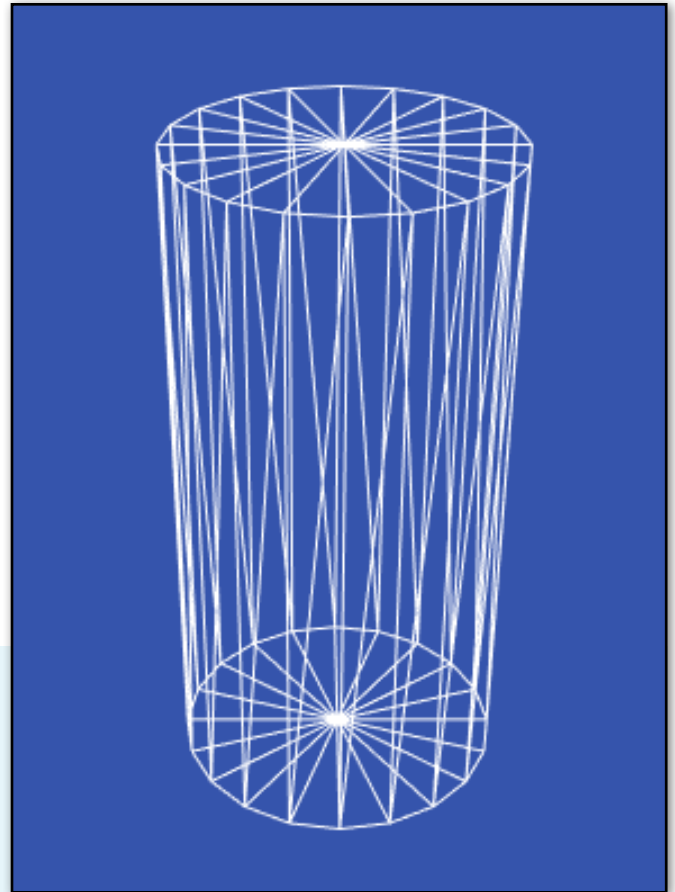
A convenient way to create characters is to use the **LineMaker** utility. For full details of how to use this, see the Editor Scripts section below. Briefly, you can drag the LetterGrid mesh from the Meshes folder into the scene, then select the Assets -> LineMaker menu item. This grid object is pretty 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 the array of `Vector3` points is — it 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 “lineMesh” 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 aMesh : Mesh;

function Start () {
    var line = new VectorLine("Wireframe", new
    Vector3[2], null, 1.0, LineType.Discrete);
    line.MakeWireframe (aMesh);
    line.Draw();
}
```



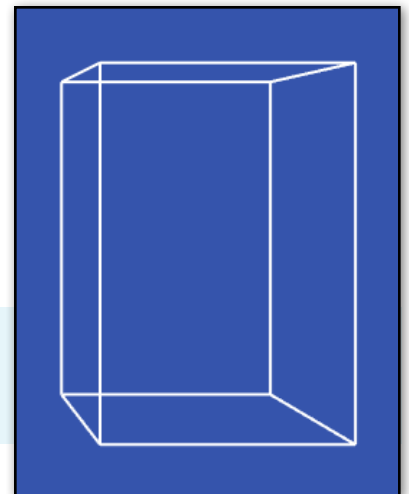
MakeCube

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 array 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);
```



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

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 for documentation on using this). You can create specific shapes for lines and store them as assets in your Unity project, and use drag'n'drop like 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;

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

`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.

SetCameraRenderTexture

If you have Unity Pro, you might sometimes want to render VectorLines to a texture. To do this, you can use SetCameraRenderTexture, which accepts a RenderTexture, and optionally a background color (which is black by default). You can also optionally pass in “true” to indicate that an orthographic camera should be used (see [SetCamera](#)). Once SetCameraRenderTexture has been used, all VectorLines will be drawn onto the specified RenderTexture and will not appear in the main camera’s view. You can pass null into SetCameraRenderTexture in order to stop using the RenderTexture and resume normal line-drawing.

SetCameraRenderTexture calls SetCamera, and likewise returns the vector camera, so you can use that to call Camera.Render() if desired.

The following script, assuming that the RenderTexture is 256x256, renders a line diagonally from one corner of the RenderTexture to the other (using a blue background), then it makes the vector cam actually render the texture, and then it draws the same line again normally using the regular camera:

```
var renderTex : RenderTexture;

function Start () {
    var cam = VectorLine.SetCameraRenderTexture (renderTex, Color.blue);
    var line = new VectorLine("Line", [Vector2(0, 0), Vector2(255, 255)], null, 1.0);
    line.Draw();
    cam.Render();
    VectorLine.SetCameraRenderTexture (null);
    line.Draw();
}
```

By default the camera background is a solid color (CameraClearFlags.SolidColor), but you can also optionally pass in a different CameraClearFlags value such as Depth or Nothing:

```
var cam = VectorLine.SetCameraRenderTexture (renderTex, CameraClearFlags.Depth);
```

ZeroPoints

At times you may want to erase some or all of the points in lines that already exist, without deleting the line itself. This is actually pretty simple to do yourself, but this function makes it even simpler:

```
myLine.ZeroPoints();
```

This sets all the points in the array for the line to Vector2.zero (or Vector3.zero). You need to redraw the line to see any effect. You can choose to erase only some of the points by supplying an index, which is 0 by default. When supplying an index, the entries from that point to the end will be zeroed.

```
myLine.ZeroPoints (20);
```

You can also erase a range:

```
myLine.ZeroPoints (20, 40);
```

In this case, all points from 20 up to (but not including) 40 will be zeroed.

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. For example, this will extend the selection area of a line by 10 pixels:

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

See also the `SelectLine` demo scene in the `VectrosityDemos` package.

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());
```


Draw3D

Normally, all lines (even lines made from `Vector3` arrays) are rendered in a flat plane by a separate camera, which is overlaid on top of your regular camera. 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`** rather than `VectorLine.Draw`. This can only be used with `VectorLine` objects created with `Vector3` arrays (`Vector2` arrays aren't allowed), but otherwise works the same:

```
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 be drawn in 3D space, by using `Draw3D` in the same way. 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.

SetCamera3D

When you use `VectorLine.Draw3D` for the first time, **`SetCamera3D`** is called for you, so normally you don't need to use it manually. The only reason for doing so is if your camera isn't tagged `MainCamera`, in which case you need `SetCamera3D` to specify which camera to use. In contrast to `SetCamera`, when you use `SetCamera3D`, the vector camera isn't created, and the layer culling mask of your regular camera isn't changed. If you're going to call it yourself, you should do so before using `Draw3D`.

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

VectorLine.vectorLayer3D

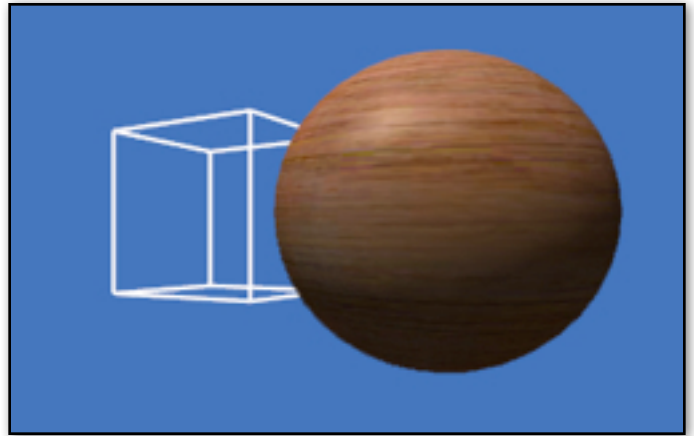
When you use `Draw3D`, the `VectorLine` object's layer is set to **`VectorLine.vectorLayer3D`**, which is 0 by default. (Normally, `VectorLine` objects are set to layer 31 by default; see "A Note About Layers" on the [Setting the Camera](#) page.) This is so `Draw3D` can be mixed with `Draw` without having to change `VectorLine` layers yourself. If you need 3D lines to use something other than layer 0, you can set `VectorLine.vectorLayer3D` to some other value.

```
VectorLine.vectorLayer3D = 16;
```

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

Since lines drawn with Vectrosity are actually meshes, that means once they are drawn, they 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 constantly call `VectorLine.Draw3D` (usually in `Update` or `LateUpdate`).

To alleviate this, you can use **Draw3DAuto** instead. It's used just like `Draw3D`, except that a `VectorLine` drawn with `Draw3DAuto` will automatically be updated 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 update the line in any way. For example:

```
function Start () {
    GameObject.CreatePrimitive (PrimitiveType.Cube);
    Camera.main.transform.position = Vector3(0,0,-10);
    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);
}
```

Note that `Draw3DAuto` is not necessary if using 1-pixel-thick lines and `VectorLine.useMeshLines` is set to `true`. In this case, `Draw3D` and `Draw3DAuto` effectively do the same thing, so use `Draw3D` instead, since it's more efficient.

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 points

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 for some things.

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 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. 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. Currently this uses the first entry in the Color array for an object, so any objects which have multiple colors for the line segments will only be drawn using the first color. Control over the fog effect is done with VectorManager.SetBrightnessParameters (see below).

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 it does by default, or if you explicitly supply "true" instead of "false", is create an invisible bounds mesh for the GameObject that you're using to control the vector object. The reason this is normally done is because of the ability of Visibility.Dynamic and Visibility.Static to stop updating lines when they're not visible. This is accomplished by using OnBecameVisible and OnBecameInvisible, and the only way those functions work is by using a mesh renderer. This may appear to be a bit problematic, since you normally don't actually want the GameObject to be visible, but only the vector object. The solution is to use an invisible bounds mesh — this is a mesh which consists only of eight vertices at the corners of a bounding box. Since this bounds mesh contains no triangles, it's not actually visible, but it's still enough to let OnBecameVisible and OnBecameInvisible work properly. Presto, problem solved.

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 "Ship", 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 all of them "X" or something.

So, why might you use false for makeBounds, anyway? You might if you're using Visibility.Dynamic or Visibility.Always, and you don't want the GameObject's mesh to be replaced by the invisible bounds mesh for whatever reason. For example, the Simple3D3 script in the VectrosityDemos package uses false for makeBounds, so that the cube mesh is still visible, which makes a solid-shaded cube with vector line highlights, as a special 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.

Minimum Brightness Distance: The distance from the camera at which brightness will be at the minimum (i.e., 0%). The default is 500. Anything beyond this distance will be drawn with only the fog color.

Maximum Brightness Distance: The distance from the camera at which brightness will be at the maximum (i.e., 100%). The default is 250. Anything closer than this distance will be drawn with only the first color entry in the `Color` array for this object. Anything between the min and max distances will be proportionally faded between that 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 max brightness distance is 600, the minimum is 200, there are 64 brightness levels, the routine runs 10 times per second, and fades to a dark blue:

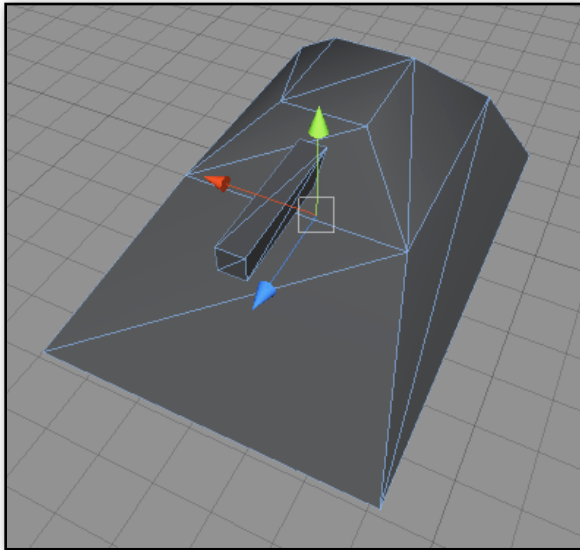
```
VectorManager.SetBrightnessParameters (600.0, 200.0, 64, .1, Color(0, 0, .25));
```

GetBrightnessValue

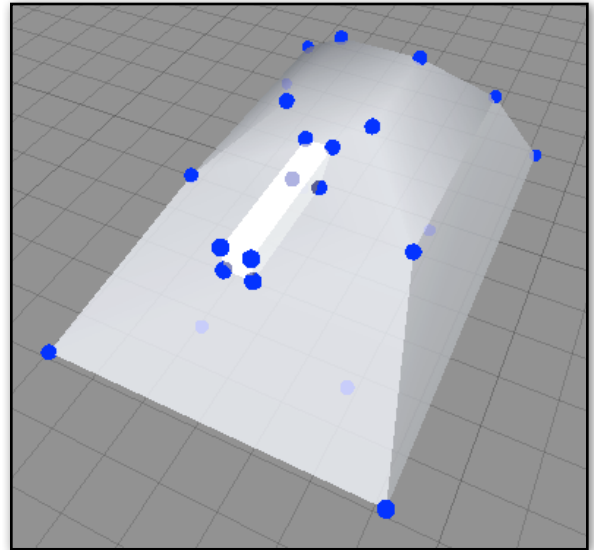
If you are doing some effects where it would be useful to know what brightness a 3D vector object should be at a certain distance, then you can use `VectorManager.GetBrightnessValue` (Tank Zone uses it in a couple of places). If you pass in the `Vector3` from a `GameObject`’s `transform.position`, it returns a value between 0 and 1, where 1 would be 100% brightness and 0 would be 0% brightness.

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 have the LineMaker editor script in a folder called Editor in your project.

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.

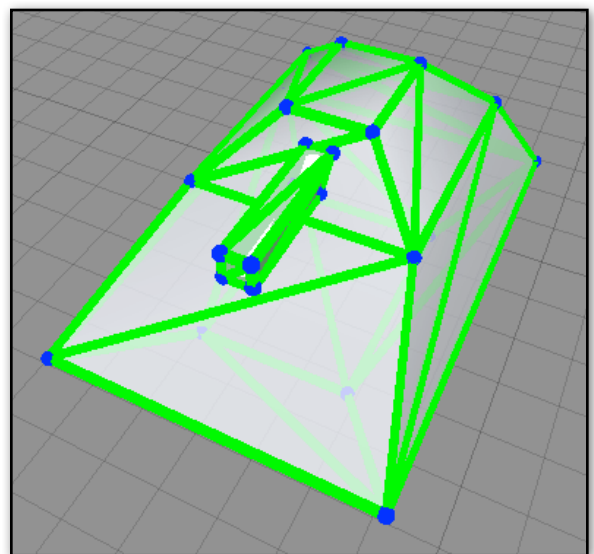
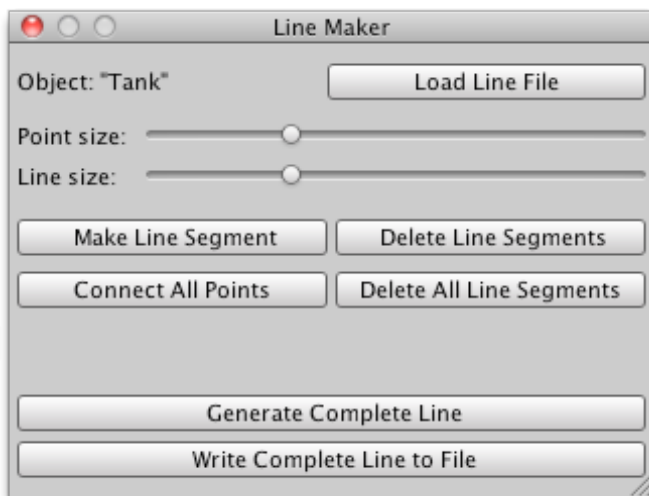


Then, with the object selected, choose **LineMaker...** from the Assets menu. Your mesh will become transparent, with blue dots placed at each vertex, and the LineMaker window will appear.



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.

Under certain circumstances, using deferred rendering can result in lines not showing up. This seems to happen with the built-in particle shaders, and has something to do with Vectrosity's camera setup and the fragment program. The solution is to use a simpler shader, such as the ones included with the Vectrosity demos (such as Unlit, UnlitAlpha, etc.), or the default shader that's used if you pass null as the line material.

Also, it seems that in some cases, dynamic batching can cause problems with lines not being visible, which apparently only happens on Windows. If this happens, try disabling dynamic batching in the player settings.

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.

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

A: Make sure you're using Unity 4.0 or later, since it uses some functions not available in earlier versions. Also, make sure you import either the source code, or one of the .dlls, but not both.

Q: My lines are messed up if I change the screen resolution at runtime.

A: After any resolution change, make sure you call VectorLine.SetCamera, and then re-draw all visible lines. You may also need to do this after changing levels; see [Setting the Camera](#) for more details. Note that this only applies to lines drawn with VectorLine.Draw; for Draw3D, calling SetCamera3D again isn't necessary.

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, and generally try not to use Resize either, unless it's only done occasionally. 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, allocate the maximum number of points that the line can have, and make use of VectorLine.minDrawIndex and .maxDrawIndex to update only part of the line as needed, rather than constantly resizing the 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 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.vectorObject, and it’s messing up the lines.

A: In most cases you shouldn’t use vectorObject 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 not necessary, and anything 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 vectorObject.

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, be sure to use VectorLine.SetCamera with the correct camera.

Q: I’ve updated some points in my points array, and called myLine.Draw(), but the line won’t update.

A: You probably recreated the points array, so it’s no longer associated with the VectorLine. You can re-associate the points array with the VectorLine by doing “myLine.Resize (newPointsArray)”. Another possibility is to use the myLine.points2 array, which is always associated with that line (or .points3 for a Vector3 array).

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.

The Tank Zone demo game needs some specific project settings in order to run correctly, and these settings may not import with the Vectrosity package. Therefore, if necessary, you should add 4 entries to the Input Manager, and use the settings below. If you only care about the arrow key controls, you can just add the entries (KeyLeft etc.) and not bother to actually set them up. Also, change “Fire1” to “Fire” and change the alt positive button from “mouse 0” to “space”.

▼ 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	<input checked="" type="checkbox"/>
Invert	<input type="checkbox"/>
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	<input checked="" type="checkbox"/>
Invert	<input checked="" type="checkbox"/>
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	<input checked="" type="checkbox"/>
Invert	<input type="checkbox"/>
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	<input checked="" type="checkbox"/>
Invert	<input checked="" type="checkbox"/>
Type	Joystick Axis
Axis	4th axis (Joysticks)
Joy Num	Get Motion from all Joysticks

The Audio Manager should have the Doppler Factor changed to 0:

▼ Audio Manager (Audio Manager)	
Volume	1
Speed Of Sound	347
Doppler Factor	0

The Time Manager should have the Fixed Timestep changed to .0166 (60 fps):

▼ Time Manager (Time Manager)	
Fixed Timestep	0.0166
Maximum Allowed Timestep	0.3333333
Time Scale	1