

# 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. 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 3.2 or later, and may not work with earlier versions.

This documentation is divided into these sections:

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

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

[Setting the Camera \(page 4\)](#): Information about the vector camera.

[Setting up Lines \(page 5\)](#): Information about lines, including textures, materials, and various line options.

[Drawing Lines \(page 9\)](#): Putting lines on the screen after they've been set up.

[Line Extras \(page 10\)](#): More things you can do with lines, including colors, widths, removing, and resizing.

[Uniform-Scaled Textures \(page 14\)](#): How to make things like dotted and dashed lines.

[Drawing Points \(page 16\)](#): Making dots instead of lines.

[Vector Utilities \(page 17\)](#): Various things to make line creation easier, such as boxes, curves, etc.

[3D Lines \(page 24\)](#): Lines that exist in the scene, rather than being drawn on top of everything.

[Vector Manager \(page 25\)](#): Utilities for working with 3D vector objects.

[Editor Scripts \(page 28\)](#): Editor utilities to help create 3D vector objects.

[Troubleshooting \(page 31\)](#): Q & A for common problems.

[Appendix \(Project Settings for Tank Zone\) \(page 32\)](#): 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.

Included in this package are the standard Vectrosity scripts, a number of demo scenes, the Tank Zone project, and complete documentation (which you've already found if you're reading this).

The documentation is in two files, the user guide (this file) and the reference guide. The user guide explains the concepts of Vectrosity and includes various programming examples. The reference guide is useful for quickly looking up information about the VectorLine, Vector, and VectorManager classes. It's probably most useful when you have some familiarity with how Vectrosity works.

The standard Vectrosity scripts are in Standard Assets/VectorScripts. If you want to use Vectrosity in your own projects, you can copy just this folder. You should also copy the Editor folder from the Vectrosity folder if you want the editor scripts.

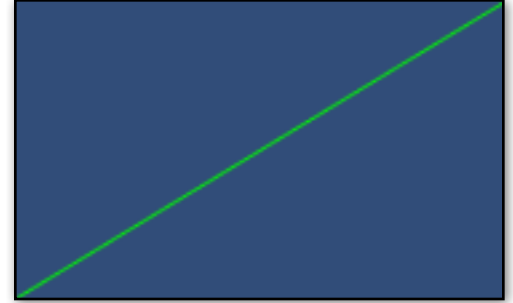
The demo scenes are located in the “\_Scenes” folder. Some scenes have several related scripts — check the Main Camera object and enable/disable the ones you want. All the scripts are, of course, in the Scripts folder. The Standard Assets folder needs to be in the root of your project (outside the Vectrosity folder, one level up) in order for the demo scripts to compile.

The Tank Zone project is located in the “\_Tank Zone” folder. There are also a few scripts in Standard Assets (outside the “VectorScripts” folder) that Tank Zone needs. You can load the \_TankZone scene and play the game, read the scripts, and generally mess around. (Please note that the assets are ©2011 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 version of Unity and won't run on the iPhone or Android versions without some work.) Tank Zone needs some project-specific settings to work properly; if these are not imported with the package, see the [Appendix](#) for setting them up.

The simplest way to draw lines is with the **SetLine** command. This 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:

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

(You can use C# as well; Javascript is 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**.) 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. Every point you add in **SetLine** will create an additional line segment. Note that the script doesn't have to be attached to a camera; it can be attached to any object in the scene.



This is all fine if you need to draw a line once, but what if you want to change it later? In fact, **SetLine** returns a **VectorLine** object. This is a type of object in Vectrosity that contains information about lines. So you can assign **SetLine** to a variable, and change this variable wherever you need to. For example, the following script will draw the line as above, then if you press the space key, it will flip the line:

```
private var myLine : VectorLine;  
  
function Start () {  
    myLine = Vector.SetLine (Color.green, Vector2(0, 0), Vector2(Screen.width-1,  
Screen.height-1));  
}  
  
function Update () {  
    if (Input.GetKeyDown (KeyCode.Space)) {  
        myLine.points2[0] = Vector2(0, Screen.height-1);  
        myLine.points2[1] = Vector2(Screen.width-1, 0);  
        Vector.DrawLine (myLine);  
    }  
}
```

The **VectorLine** object contains a **points2** array, with one entry for each point you defined when using **SetLine**. If you used **Vector3** points instead of **Vector2**, then you should use **points3** instead of **points2**. In order to re-draw a line that already exists, you use **Vector.DrawLine**. You can also use **SetLine3D** for creating lines that actually exist in 3D space, instead of being drawn by a separate camera on top of other objects. Otherwise, it works the same way as **SetLine**.

There is much more you can do with line drawing; **SetLine** is for very simple lines only, so you may prefer to create **VectorLine** objects directly, which gives you more control. This is covered in detail in the next sections.

Lines are normally shown by a separate camera, unless you use `DrawLine3D` or `SetLine3D`. This camera will appear in the hierarchy as an object called “VectorCam”. This is set up for you automatically as long as the camera you’re using is tagged “**Main Camera**”. However, you can also set up the vector camera manually, which you’ll need to if you want to use anything other than the default parameters. Otherwise, you may want to skip this section for now, though it’s worth knowing how the camera works in Vectrosity if you need to do anything complex. The basic format for setting the camera manually is:

```
Vector.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 “Main Camera”. 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 “Main Camera” tag:

```
Vector.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:

```
Vector.SetCamera (CameraClearFlags.SolidColor);
```

You can pass both at once:

```
Vector.SetCamera (camera, CameraClearFlags.SolidColor);
```

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, `Vector.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, 3D lines can show glitches in certain circumstances, which is why it’s false by default. If you don’t use 3D lines, you might as well set orthographic mode to true.

**A NOTE ABOUT LAYERS:** In order for Vectrosity to work, it creates meshes which are drawn by the vector camera. 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 “`Vector.vectorLayer = x`”, where `x` is the layer number you want. Do this before calling `SetCamera`. So if you notice weird things happening like lines being seen in 3D, **make sure your normal camera always ignores the vector layer**.

As described earlier, you can use `Vector.SetLine` to make simple lines. The next few pages describe how to set up `VectorLine` objects directly, which allows control over width, material, and so on.

In order to set up a line, you need some points. Typically this is done with a `Vector2` array, though you can use a `Vector3` array for 3D lines (see also the `VectorManager` section). 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. 3D lines are often used for drawing 3D objects, naturally, so they use the same coordinates as normal 3D objects.

## Line points

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

```
var linePoints = new Vector2[2];
```

That creates a `Vector2` array with 2 points. You can have a maximum of about 16,000 points per line, depending on what sort of line you're drawing. You'll get an error at runtime if you try to exceed the maximum. You can set the points later, or set them when you're declaring the array:

```
var linePoints = [ Vector2(20, 30), Vector2(100, 50) ];
```

It's fine to 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 `Vector.DrawLine` to draw the line with the new points.

## Materials

Another thing you need is a material for the lines. The material should use an unlit shader that does not use mesh normals, and it should use vertex colors if you want to have different colors for different line segments in the same line. 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.

Note, however, that if you want to use the depth property of lines, so that lines draw on top or underneath each other correctly, the shader needs to write to the zbuffer. See the `ParticleZwrite` shader in the `Shaders` folder in the `VectrosityDemos` project.

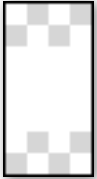
Several other shaders in the `VectrosityDemos` project are appropriate for different types of lines. `SolidColor` is an extremely simple shader that's appropriate for single-color lines that don't use the depth property. `Unlit` is for single-color lines that do use the depth property, and can use a texture. `UnlitAlpha` is the same, except it supports textures with alpha, and doesn't have zbuffer writing (this can look ugly with alpha when lines overlap at different depths).

Note that you can pass in null instead of a material. In this case, Vectrosity will use a basic shader that uses vertex colors and zbuffer writing, but no texture. So if you don't have any particular need for a texture, just specify null for the material and Vectrosity will take care of it for you.

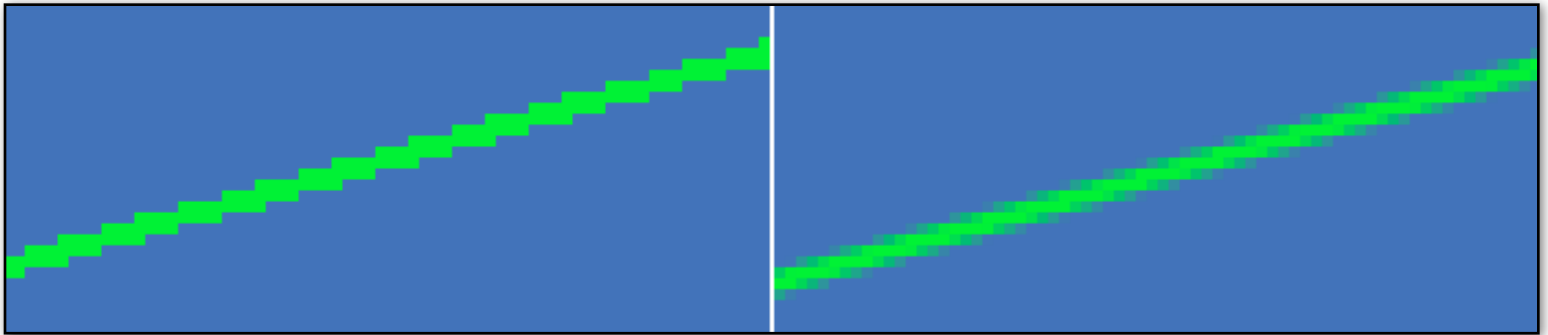
Also note that some shaders don't appear to work with deferred rendering with Vectrosity's camera setup, such as the built-in particle shaders. If your lines don't show up, try a different shader.

## “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 Unity iPhone, since there is currently no built-in way to get FSAA there. To do this, you need to use a material that uses a shader that has alpha texture support, such as Particles/Additive or UnlitAlpha. Then you need a texture for this material that has transparent pixels at the top and bottom. The VectrosityDemos project 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:



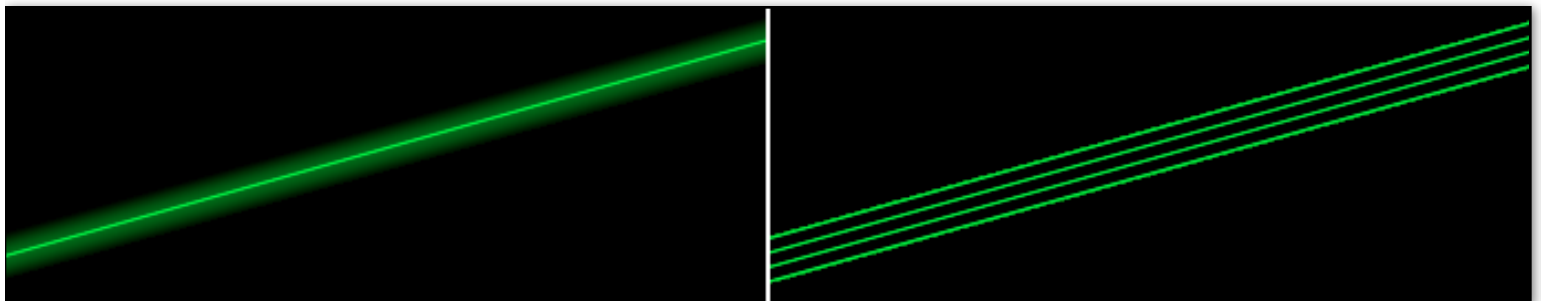
Make sure the textures are set to bilinear filtering and not point. The line anti-aliasing works by taking advantage of the inherent anti-aliasing you get when using bilinear filtering.



**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 the transparent areas in the ThinLine texture will get stretched out and blurry if used for thicker lines, 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 below, which describes making dotted, dashed, and similar lines.



## VectorLine

When creating a `VectorLine` object, in the simplest form you need to supply a name, the array of points that make 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 above), and the width of the line in pixels:

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

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. Note that the pixel width is a float; it's fine to have a line width of 2.5, for example. 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.

This won't actually draw the line yet (see below for that), it just creates a `VectorLine` object that will be used to draw the line later. It's of type `VectorLine`, so to declare a null variable like this, you can do this:

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

With Javascript (and C# in Unity 3.0 and later) you can use type inferencing when declaring `VectorLine` objects, such as the example code at the top — in that case `myLine` is inferred as type `VectorLine`. C# without type inferencing would be written like this:

```
VectorLine myLine = new VectorLine("MyLine", linePoints, lineMaterial, 2.0f);
```

## Segment Cap

There are additional parameters for `VectorLine`. For one, you can also specify a segment cap length:

```
var myLine = new VectorLine("MyLine", linePoints, lineMaterial, 8.0, 4.0);
```

The segment cap length is used primarily when you have thick lines. With thin lines, you can usually leave it at 0. This adds a given number of pixels to either end of a given line segment. Primarily this is used for things like squaring off rectangular shapes, because otherwise the line thickness leaves gaps at the ends. Usually in this case you'd want to use exactly half the line width, but you can use different numbers for different effects. (And remember this uses floats, so if your line width is 3, you can use 1.5 for the end cap.)



Left: a 14-pixel-thick line with a segment cap length of 0. Middle: the same line with a segment cap length of 7. Right: a segment cap length of 14.

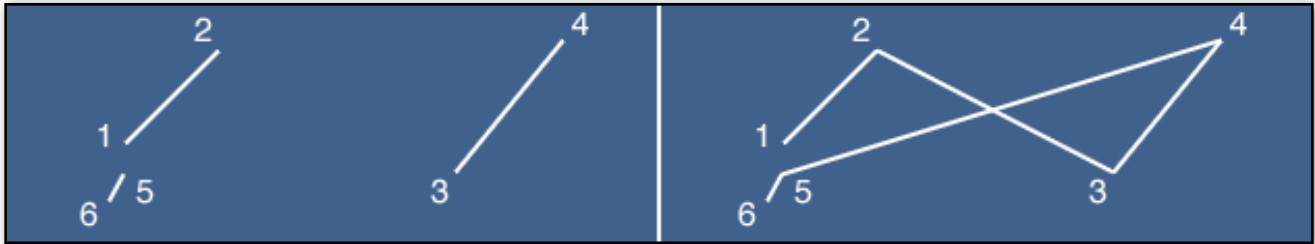


## Depth, LineType, and Joins

The remaining parameters for VectorLine are **depth**, **LineType**, and **Joins**.

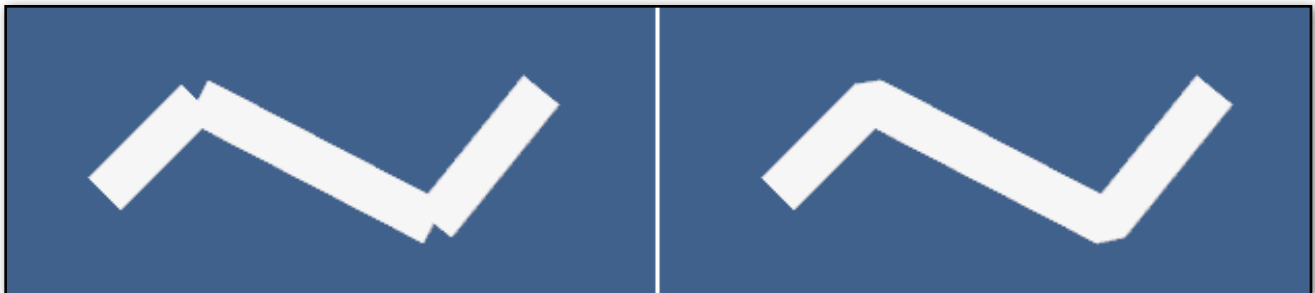
Depth is the order in which lines are drawn. This has a default of 0, ranges from 0 to 100, and is an integer. Lines with a higher depth are drawn on top of lines with a lower depth, **as long as the line is using a material with a shader that uses ZWrite On** (see the Materials section earlier). Without zbuffer writing, lines will generally not handle depth well.

LineType is either **LineType.Discrete** (the default) or **LineType.Continuous**. Discrete lines take two Vector2 or Vector3 points to describe each line segment. Therefore, a discrete line must use a points array with an even length. Since each line segment is described individually, a single VectorLine object can have the appearance of many separate lines (if you want), but will only take one draw call. Continuous lines, on the other hand, draw a single unbroken line starting from the first point, continuing to the last. This makes them less flexible to use, but they are a little more efficient to draw. They can also use Joins.Fill (see below), whereas discrete lines can't.



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

The last parameter is either **Joins.Fill** or **Joins.Open** (you can also use Joins.None). This is primarily intended for thick lines, and as mentioned above can only be used with continuous lines. Like the end cap length, it's intended to fill the ugly gaps you get at the joins of thick line segments. But while the end cap length is good for lines that meet at 90° angles, it doesn't necessarily work well in a general case. Joins.Fill, on the other hand, will fill in the gaps at any angle.



Left: a 22-pixel-thick line with Joins.Open. Right: the same line with Joins.Fill.

Here's a VectorLine that uses linePoints for the line's screen points, is colored red, uses lineMaterial, is 10 pixels thick, 0 segment cap length, depth 1, is continuous and fills in joins:

```
myLine = VectorLine("MyLine", linePoints, Color.red, lineMaterial, 10.0, 0.0,
    1, LineType.Continuous, Joins.Fill);
```



At last, we'll actually draw a line! This is pretty simple: **Vector.DrawLine**. Just supply the VectorLine object that you've set up and now want to be drawn:

```
Vector.DrawLine (myLine);
```

Since a line is actually a mesh, once it's drawn, Vector.DrawLine doesn't have to be called again unless the line changes in some way. So DrawLine can generally be called from pretty much anywhere.

You can optionally specify the transform of an object. This is particularly useful for 3D lines, but can also be used to move or rotate 2D lines. See the DrawLines scene in the VectrosityDemos package for an example of rotating a line left and right. You do this by specifying the transform:

```
Vector.DrawLine (myLine, transform);
```

That would use the transform component of whatever object the script it attached to. You can pass in the transform of any object, of course. If you're continuously updating a line with a transform every frame, it's generally best to call Vector.DrawLine in LateUpdate to make sure it's updated correctly.

## Advanced movement with vectorObject

It's also possible to move the line itself, 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 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 to move or rotate VectorLine objects on any axis, 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).

## VectorLine colors

When making a line, you can also specify a color:

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

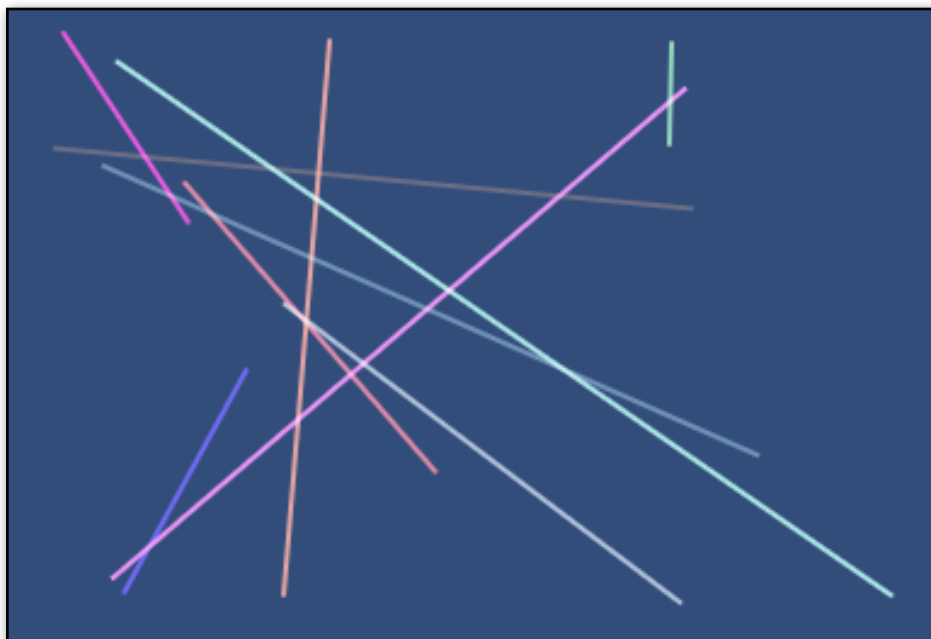
Normally, the line color comes from whatever material you specify. In this example, the line will use `Color.blue` for the color, **as long as the material's shader supports vertex colors**. For example, the `Particles/Additive (Soft)` shader or similar. In fact, with vertex-color-only shaders like that, this is the only way to specify line colors, since you can't specify a color in the material. If you use `null` for the `lineMaterial`, the default material uses vertex colors just fine.

Instead of a single color, you can use an array of colors. For this, you need a **Color array**. Each entry in the color array represents one segment of the line. Therefore, the length of the color array depends on whether the line is continuous or discrete. For a continuous line, the length should be the length of the `Vector2` (or `Vector3`) points array minus one. For a discrete line, the length should be exactly half that of the points array, since you need two points to describe a single line segment. By default, a `VectorLine` uses a discrete line, so by default the `Color` array should be half the length of the points array.

Here's another example with a `Color` array:

```
var linePoints = new Vector2[20];  
var lineColors = new Color[10];  
var myLine = VectorLine("MyLine", linePoints, lineColors, null, 2.0);
```

This makes a `VectorLine` object with 20 points (10 line segments) and 10 colors. Depending on what you do with the points and colors, you could make a `VectorLine` object that looks like this when drawn:



Note that using the single-color option actually creates a `Color` array of the appropriate length for you. Therefore, you can specify one color when the line is created, and later make use of **Vector.SetColors** (see the `Vector Utilities` section below) to change the individual line segment colors later.

Here's a complete script that will create a line made of random line segments with random colors, similar to the picture above. You can use a new empty scene that has only a camera in it (which should be tagged "Main Camera") 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);
    Vector.DrawLine(line);
}
```

## VectorLine widths

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

```
var myWidths = [1.0, 2.0, 3.0, 10.0, 20.0];
Vector.SetWidths (myLine, 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 two line segments with a discrete line, and sets the first segment to 2 pixels, and the second segment to 6 pixels:

```
var points = [Vector2(100, 100), Vector2(200, 100), Vector2(200, 100), Vector2(300, 100)];
var line = new VectorLine("Line", points, null, 2);
var widths = [2.0, 6.0];
Vector.SetWidths (line, widths);
Vector.DrawLine (line);
```

This results in a line which looks like this:

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

```
line.smoothWidth = true;
```

If that line is used in the above script, you get this result instead:



## Shortcut

Even though you there are some defaults and you don't have to specify all the parameters, there are times when you want a number of lines that have the same basic look, and you just want to use most of the same parameters. In this case you can use **Vector.MakeLine** as a shortcut. First you must initialize it by using **Vector.SetLineParameters** once. For example:

```
Vector.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 = Vector.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 = Vector.MakeLine ("LineName", linePoints, Color.white);  
var anotherLine = Vector.MakeLine ("LineName", linePoints, lineColors);
```

The rest of the parameters will be the same.

## Removing VectorLines

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

```
Vector.DestroyLine (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 using **Vector.DestroyObject**, which is usually used when you have a GameObject that's controlling a 3D VectorLine object:

```
Vector.DestroyObject (myLine, gameObject);
```

This is almost the same as writing:

```
Vector.DestroyLine (myLine);  
Destroy (gameObject);
```

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

## Resizing VectorLines

At times you may want to add more points to a line, or maybe remove some. One possibility is to destroy the line using `Vector.DestroyLine`, and then recreate the line with the new points. However, a more convenient 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 `Vector.DrawLine` in order to update the line on the 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);  
Vector.DrawLine (myLine);
```

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);  
Vector.DrawLine (myLine);
```

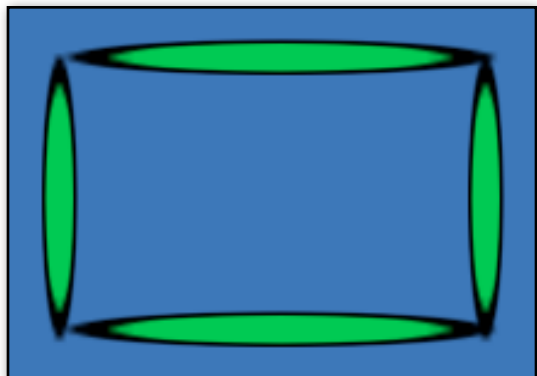
Or, you could use the first technique:

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

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 `Vector.SetColors`. This also applies if you were using multiple line segment widths.

So far you've seen various sorts of lines, where the textures used are stretched the length of each line segment. 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 **Vector.SetTextureScale** comes in. You call this function after setting up a **VectorLine** — whether you use it before or after calling **Vector.DrawLine** doesn't matter. The basic format is this:

```
Vector.SetTextureScale (vectorLine, textureScale);
```

“vectorLine” is whatever **VectorLine** object you pass in, and “textureScale” is a float. This is probably most commonly 1.0, but it can be anything. Let's set up a rectangle:

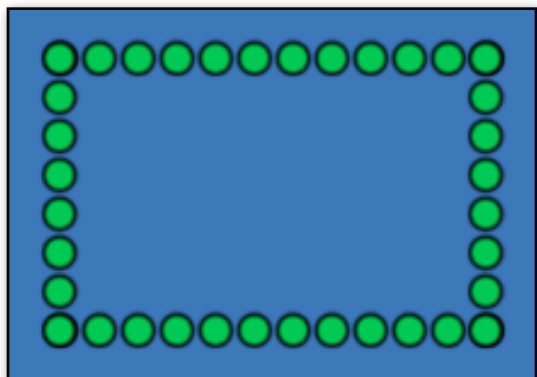
```
var lineMaterial : Material;

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

That results in the above image, assuming a **Material** is used that contains the green dot texture. Now add another line of code, after “**Vector.MakeRectInLine**” is called:

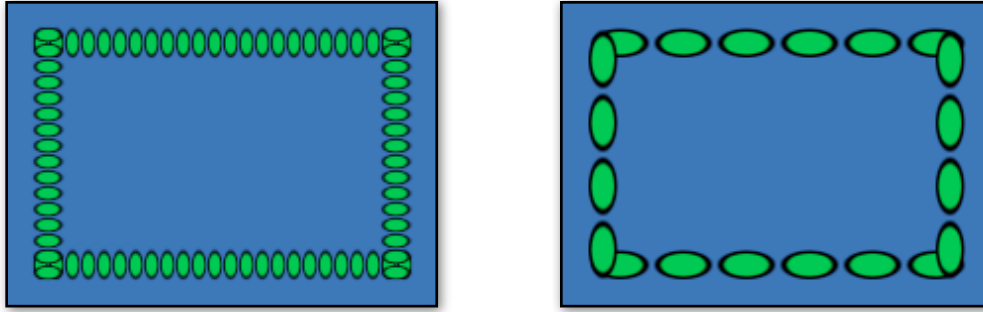
```
Vector.SetTextureScale (rectLine, 1.0);
```


And we get this result instead:



Note that if you're using uniform-scaled textures, you should call **Vector.SetTextureScale** whenever a line is redrawn, in order to keep the texture scale consistent. In other words, whenever you call **Vector.DrawLine**, you should also call **Vector.SetTextureScale**, but only if you're using uniform-scaled textures. With standard textures, all you need is **Vector.DrawLine**, of course.

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:



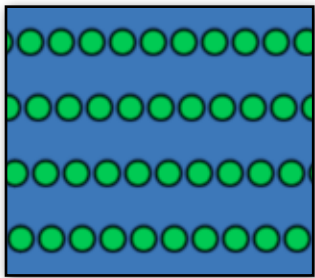
This is particularly useful for dashed lines, where you might want to make longer or shorter dashes, and it's also useful for non-square textures. This 32x16 texture, for example:  A textureScale of 1.0 results in the left image, and using 2.0 results in the right image:



You can also optionally supply an offset in the third parameter:

```
Vector.SetTextureScale (rectLine, 1.0, .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 calling `Vector.SetTextureScale` in the same way that you would using `renderer.material.mainTextureScale`, although it's faster to set `mainTextureScale`, so normally you'd prefer to use that for animation. Using offsets of .25, .5, .75, and 1.0 (same as 0.0) would result in this:



For some more code examples of `Vector.SetTextureScale`, see the "SelectionBox2" script 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 its default look before you called `Vector.SetTextureScale`, you can use **Vector.ResetTextureScale**:

```
Vector.ResetTextureScale (vectorLine);
```

This removes all custom texture scaling information from whatever `VectorLine` object you pass in.

**iPhone note:** the iPhone GPU tends not to render textures well if texture UVs are too far away from the 0 to 1 range. `Vector.SetTextureScale` 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.



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 a little bit like a particle system.

To set up your points, use **VectorPoints**. This is similar to **VectorLine**, except there are fewer options, and it returns type **VectorPoints** rather than **VectorLine**. You pass in the name of the points, a **Vector2** array containing the screen-space coordinates of the points (each entry in the array corresponds to one point), a material (can be null, which uses the same basic line material that's created if you use null with vector lines), and the size in pixels (which is a float, same as with lines). The depth is 0 by default, but you can optionally add that if needed; as with lines, the depth is an integer from 0 to 100.

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

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 **Vector2** array.

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

As with lines, you can pass in a single color:

```
var points = new VectorPoints("Points", linePoints, Color.red, lineMaterial, 2.0);
```

Most functions that accept a **VectorLine** object will also work with **VectorPoints**, such as **SetColor**, **SetColors**, etc.

To draw points, use **Vector.DrawPoints**:

```
Vector.DrawPoints (myPoints);
```

You can pass in a transform with **DrawPoints** too, which works like it does with **DrawLines**:

```
Vector.DrawPoints (myPoints, 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);
    Vector.DrawPoints(dots);
}
```

There are a number of utilities in the Vector class that help with constructing lines.

## SetColor and SetColors

If you made a VectorLine object using a color or array of colors, you can change the colors at any time by using **Vector.SetColor** or **Vector.SetColors**. SetColor takes a VectorLine object and a color:

```
Vector.SetColor (myLine, Color.yellow);
```

SetColors takes a Color array. As always, 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).

```
Vector.SetColors (myLine, myColors);
```

When SetColor or SetColors are called, the line colors are changed immediately, and you don't have to redraw the line using Vector.DrawLine. Note that these functions require a shader that uses vertex colors to have any visible effect.

## SetColorsSmooth

Normally lines using an array of colors use a specific color for each line segment, as dictated by the color array. In some cases you may prefer that colors blend smoothly together instead. This is useful for things like lines that gradually fade out, rather than having visible "steps" for each line segment.

```
Vector.SetColorsSmooth (myLine, myColors);
```

Aside from blending colors, this works the same as Vector.SetColors.

## SetDepth

You can set the depth for a VectorLine object when creating it (see the **Setting Up Lines** section above), but with an already-existing line, you can change the depth using Vector.SetDepth:

```
Vector.SetDepth (myLine, 3);
```

The depth is an integer from 0 to 100. You need to redraw the line in order for the depth to actually change on-screen. You can also use it with a transform, for an object that is not a VectorLine:

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

This would generally be used for special circumstances. The Tank Zone demo package uses it 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.

## MakeRectInLine

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 `Vector2`s 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.

`MakeRectInLine` works with `Vector3` arrays as well as `Vector2` arrays. In the case of `Vector3` arrays, you can pass in two `Vector3`s instead of two `Vector2`s, and the `.z` element of the `Vector3`s will be the depth in world space. `Rect`s 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.)

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

```
Vector.MakeRectInLine (vectorLine, Rect, index);
```

or:

```
Vector.MakeRectInLine (vectorLine, 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);  
Vector.MakeRectInLine (squareLine, Rect(300, 200, 100, 100));  
Vector.DrawLine (squareLine);
```

Making a selection box might look like this, assuming “`selectionLine`” is a `VectorLine` made with `LineType.Continuous`, and “`originalPos`” is the position where the mouse was originally clicked:

```
Vector.MakeRectInLine (selectionLine, 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.

## MakeCircleInLine

This is for easily creating circles. As with `MakeRectInLine`, it calculates the appropriate values in a `VectorLine`'s `Vector2` or `Vector3` array; you still use `Vector.DrawLine` to actually draw the circle after using `MakeCircleInLine`.

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 `MakeRectInLine`, 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:

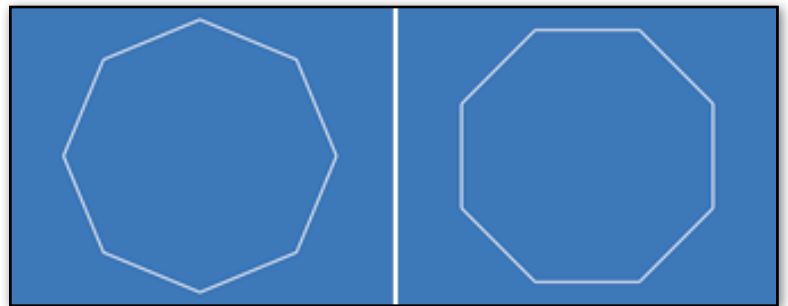
```
Vector.MakeCircleInLine (vectorLine, origin, radius, segments, pointRotation, index);
```

“`vectorLine`” is a `VectorLine` object that you’ve already declared. The size of the `Vector2`/`Vector3` array that’s used for this `VectorLine` must be 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.

“`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. “`radius`” is a float, which describes half the total width of the circle in screen pixels. 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). “`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 `MakeRectInLine`. For example, a discrete line with a `Vector2` array with 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.)

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

## MakeEllipseInLine

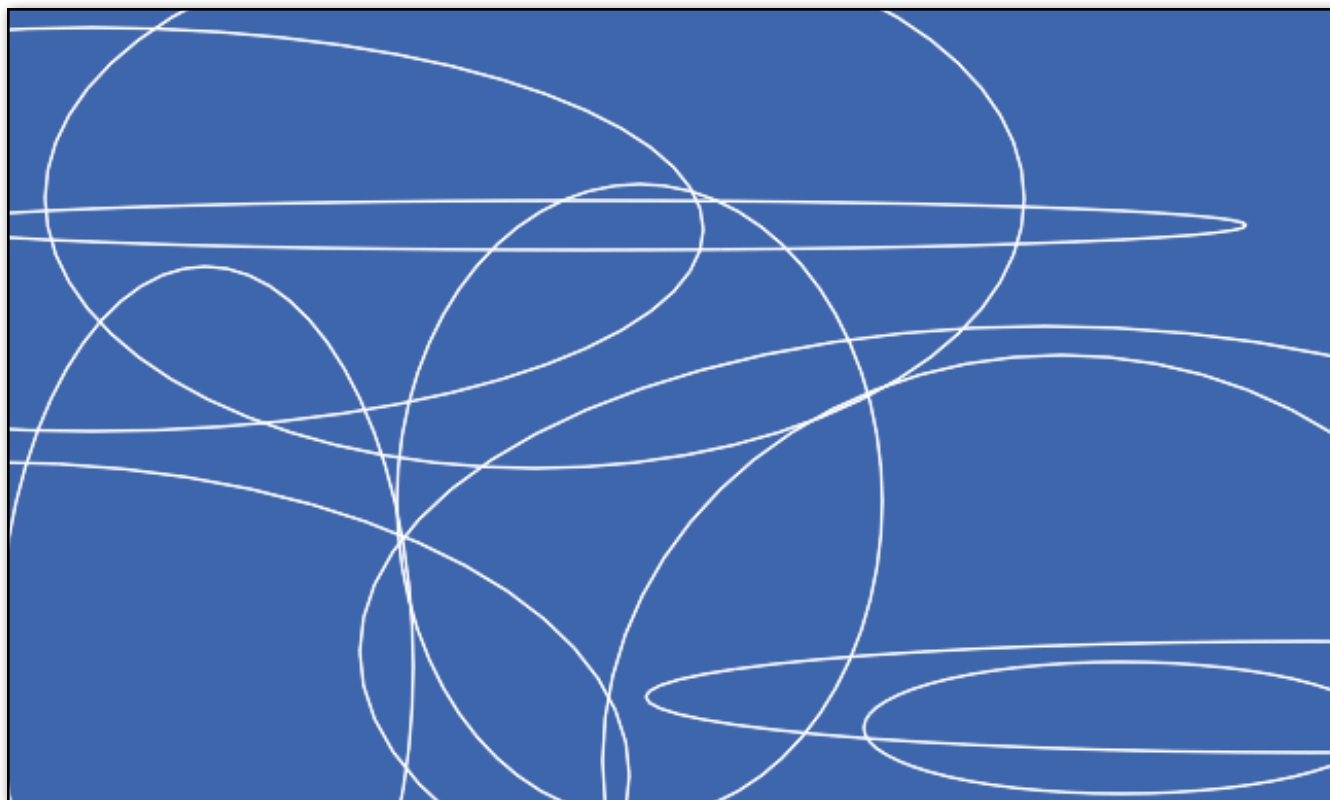
This is nearly identical to `MakeCircleInLine`, with the exception that two radius values are used instead of just one. You can specify the x and y radius values to make ellipses of different widths/heights. `MakeCircleInLine` actually uses this routine, but it passes one radius value for both x and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

```
Vector.MakeEllipseInLine (vectorLine, origin, xRadius, yRadius, segments, pointRotation, index);
```

Both `xRadius` and `yRadius` are floats that specify the number of screen pixels for the respective radii. The usage otherwise is the same as `MakeCircleInLine`. 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.

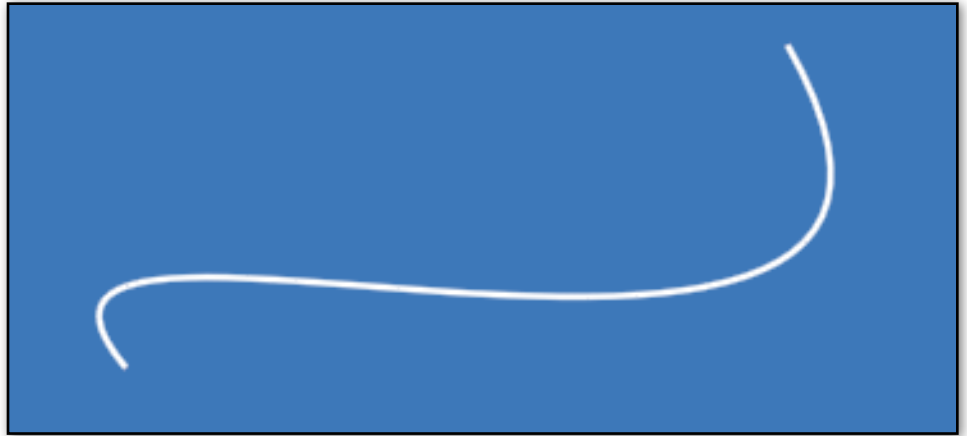
See the **Ellipse** scene in the **VectrosityDemos** package for a couple of example scripts. The **Main Camera** object in that scene has two scripts attached, **Ellipse1** and **Ellipse2**, which you can enable/disable to see the different effects. **Ellipse1** creates a single ellipse using a continuous 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. Running the **Ellipse2** script results in something like this:



## MakeCurveInLine

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/EllipseInLine sections. It results in curves that might look like this, depending on how the anchor and control points are positioned:



The format is either:

```
Vector.MakeCurveInLine (vectorLine, curvePoints, segments, index);
```

or

```
Vector.MakeCurveInLine (vectorLine, anchor1, control1, anchor2, control2, segments,  
                        index);
```

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

“segments” is an int, and works like it does in MakeCircle/EllipseInLine: 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.

“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.

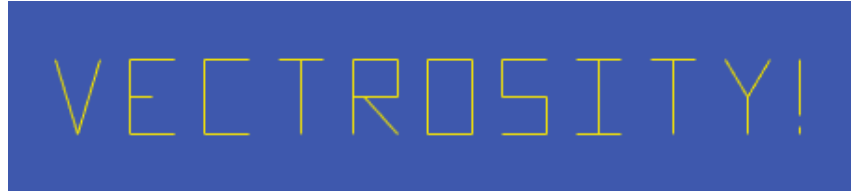
If you’re unfamiliar with the concept of how bezier curves work, open the **Curve** scene in the VectrosityDemos package, and enable the **DrawCurve** script on the **Main Camera** object. (By default, the **SimpleCurve** script is enabled, which draws a single curve using a Vector2 array of 4 points, which you can specify in the inspector.) 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. With the **DrawCurve** script active, you can hit Play, and interactively create curves by dragging anchor and control points around the screen.

**A note about Joins.Fill:** since MakeCurveInLine uses a pre-existing Vector2 array, you probably leave the array empty, which actually means it’s filled with Vector2(0, 0). Joins.Fill looks at the first and last points in the array to see if they are the same, and if so, they are connected, which is what will happen if they’re all (0, 0). That’s fine for rects and ellipses, which are closed shapes, but usually not what you want for curves. The solution is to *first* make either the first or last element of the Vector 2 array something other than Vector2(0, 0), and *then* create the VectorLine object (and then call MakeCurveInLine).

## MakeTextInLine

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 `Vector.MakeTextInLine` 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:



```
Vector.MakeTextInLine (myLine, "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, since 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 `MakeTextInLine`.

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:

```
Vector.MakeTextInLine (myLine, "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:

```
Vector.MakeTextInLine (myLine, "Hello world!", Vector2(100, 100), 30.0, false);
```

Any characters in the string which don't exist in the Vectrosity "font" are ignored. Currently the default font doesn't contain lower-case characters, so the above example will just print `"H !"`.

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.



## 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.

**NOTE: Unity 2.6 has a bug in TextAsset.bytes that sometimes results in corrupted data. So it's likely you should avoid this technique if you're using Unity 2.6. Unity iPhone and Unity 3 don't have this bug.**

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 = Vector.BytesToVector2Array (lineData.bytes);
    var line = new VectorLine("Line", 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...remember that this might not work in Unity 2.6.

## ZeroPointsInLine

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:

```
Vector.ZeroPointsInLine (myLine);
```

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, only the entries from that point on will be zeroed.

```
Vector.ZeroPointsInLine (myLine, 20);
```

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 `Vector.DrawLine3D` rather than `Vector.DrawLine`. This can only be used with `VectorLine` objects created with `Vector3` arrays, but otherwise works the same:

```
Vector.DrawLine3D (myLine);
```

As with `DrawLine`, you can optionally pass in a transform:

```
Vector.DrawLine3D (myLine, transform);
```

Since lines drawn this way are in 3D space, they will not be seen by the vector camera. And since `VectorLine` objects are created on a layer not normally seen by your regular camera, they won't be seen by that either. In order to make these 3D lines actually be visible, there are a couple of different options.

The first is useful if you're not planning on using anything but 3D lines. In this case the vector camera would be useless, so you can make sure that it's never created in the first place. Do this by calling `Vector.SetCamera3D` instead of `Vector.SetCamera`. Since `Vector.SetCamera` is called by default the first time you use `Vector.DrawLine`, this means you should explicitly call `Vector.SetCamera3D` yourself in a `Start` or `Awake` function before drawing any 3D lines.

`SetCamera3D` is used exactly the same way as `SetCamera`, aside from the name. The main differences being that the vector camera isn't created, and the layer culling mask of your regular camera isn't changed. `VectorLines` are still created on the layer used for lines, which is 31 by default. Normally this doesn't make any difference, but if you want to use a different layer, set `Vector.vectorLayer` to something other than 31 (such as 0 for the default layer).

The second option is to keep using `Vector.SetCamera`, but use `Vector.SetLayer` for your lines. This is used like so:

```
Vector.SetLayer (vectorObject, layerNumber);
```

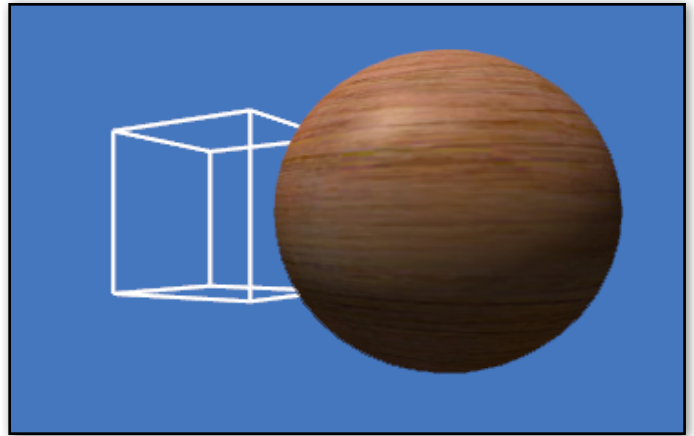
"layerNumber" is an int referring to a particular, to which the line object "vectorObject" is set. As mentioned, 0 is the default layer, so you could set your lines to be drawn by `Vector.DrawLine3D` to layer 0, assuming your regular camera can see that layer. For example:

```
var myLine = new VectorLine ("Line", linePoints, null, 1.0);  
Vector.SetLayer (myLine, 0);  
Vector.DrawLine3D (myLine);
```

If you want `VectorManager` (see below) to use 3D lines instead of standard lines, this can be done by setting **useDrawLine3D**, which is false by default:

```
VectorManager.useDrawLine3D = true;
```

The **Simple3D 3** script on the **Cube** object in the **\_Simple3DObject** scene in the **VectrosityDemos** package has an example of using a 3D line.



There is an additional class that makes 3D vector shapes behave almost exactly like regular GameObjects. This is the **VectorManager** script, which needs to be attached to a GameObject in the scene to function (this is necessary because it runs a LateUpdate function). See the **\_Simple3DObject** scene in the VectrosityDemos package for an example of making a 3D vector cube.

**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, always use Maximize On Play so that the scene view camera isn't used, and therefore doesn't interfere with things when you're testing your project.

## 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);
```

Depending on the parameters, this adds a couple of components to the object at runtime. You then have a 3D vector object that behaves just like the GameObject. Any movement is copied to the vector object, so you can use Transform.Translate, Transform.Rotate, a rigidbody and physics, or anything else you normally do with a GameObject. You don't even have to use Vector.DrawLine, since VectorManager will do this for you if you add a visibility component (although you're free to do so manually if you like).

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. See the section about **BoundsMaker** in the editor scripts section below for details on how to make an invisible bounds mesh that adds no extra draw calls and doesn't appear in the scene, but still allows OnBecameVisible and OnBecameInvisible to work. This is useful if you don't want to see the mesh of the normal object, only the 3D vector object.

**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. You also need a renderer for this to work, such as an invisible bounds mesh as mentioned above.

**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, in which case there isn't much point setting up an invisible bounds mesh.

**Visibility.None:** None of the VisibilityControl scripts will be added. Use this if you're manually managing the line drawing using Vector.DrawLine, but still want to use the brightness control (see below).

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. This also means that in order to use the fog effect, lines must be created using a color. Control over the fog effect is done with `VectorManager.SetBrightnessParameters` (see below).

**Brightness.Normal** (or `Brightness.None`): The line segment colors are left alone.

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);
```

## 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.

## DestroyObject

Normally you don't actually need to use this function yourself. When you destroy a GameObject that has a 3D vector object made using the ObjectSetup function, cleanup code in the Visibility scripts calls DestroyObject to take care of things automatically. In case you need to do this manually for some reason, you should know that if you're using the ObjectSetup function, destroying a GameObject alone isn't enough, since the vector objects would be left behind. So the DestroyObject function takes care of this. While VectorManager.DestroyObject isn't the same as Vector.DestroyObject, it does work the same, where you pass in the VectorLine that was used to create the vector object, and the GameObject that you want to destroy:

```
VectorManager.DestroyObject (line, gameObject);
```

Here's some example code, which creates a vector object from a text asset (see the **LineMaker** section below about creating these), waits 5 seconds, then destroys the object:

```
var vectorObject : TextAsset;

function Start () {
    var linePoints = Vector.BytesToVector3Array (vectorObject.bytes);
    var line = new VectorLine ("VectorObject", linePoints, null, 2.0);
    VectorManager.ObjectSetup (gameObject, line, Visibility.Dynamic, Brightness.Normal);

    yield WaitForSeconds (5.0);

    Destroy (gameObject);
}
```

The DestroyObject function call itself is in the Visibility.Dynamic script, in the OnDestroy function, so it doesn't need to be called manually.

There are two editor scripts which help with creating 3D vector objects: **BoundsMaker** and **LineMaker**. (Actually, LineMaker is good for making 2D vector objects as well.)

## BoundsMaker

As described in the **ObjectSetup** section, above, it's possible to use `OnBecameVisible` and `OnBecameInvisible` to save on having to compute 3D vector lines for `GameObjects` that aren't on-screen. But wait—usually you don't want to actually see the normal `GameObject` (except maybe for special effects), but only the 3D vector object, and you can't disable the mesh renderer or else `OnBecameVisible` and `OnBecameInvisible` won't work. So how is this done? The answer is **BoundsMaker**.

This makes a special mesh consisting only of vertices, and no triangles. Therefore nothing is actually drawn and no draw calls are added, but the vertices (8 of them to be precise, 1 for each corner of the bounding cube) are enough to make `OnBecameVisible` and `OnBecameInvisible` work.

If this script is in the Editor folder of your project, it adds an entry in the Assets menu in Unity called “Make Invisible Bounds Mesh...”. To use it:

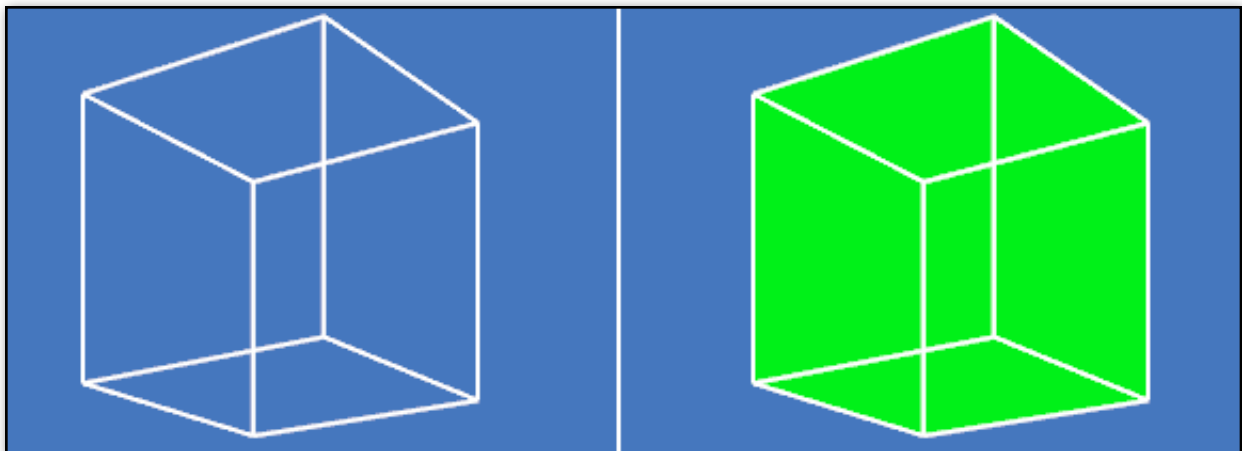
Select the mesh that you're using for your `GameObject`, which must be in the scene. If you were making a bounds mesh for the enemy tank in Tank Zone, for example, you'd drag the `Tank_Cube` mesh from `Meshes/tank` into the scene. Note that since the bounds mesh is only 8 points of a cube, the mesh doesn't have to be exact, but only the general size. Usually it's convenient to use the mesh that you made the 3D vector points from (see **LineMaker**, below).

With the mesh selected, go to the Assets menu and choose “Make Invisible Bounds Mesh...”.

Choose somewhere in your project to save the bounds mesh. By default it's called the same name as the original mesh, plus “\_bounds”. You can delete the original mesh from the scene now if you want.

Drag the bounds mesh to the `GameObject` that you'll be using for the 3D vector object.

Note that you can use an actual visible mesh for the `GameObject` if you like, instead of an invisible bounds mesh. This could be used for special effects...for example, in the **\_Simple3DObject** demo scene, the **Cube** object doesn't use an invisible bounds mesh (it's using `Visibility.Always`) and has a default cube mesh with its renderer disabled. If you enable it, you can see it using a solid color shader, and when run, the vector lines are drawn on top:

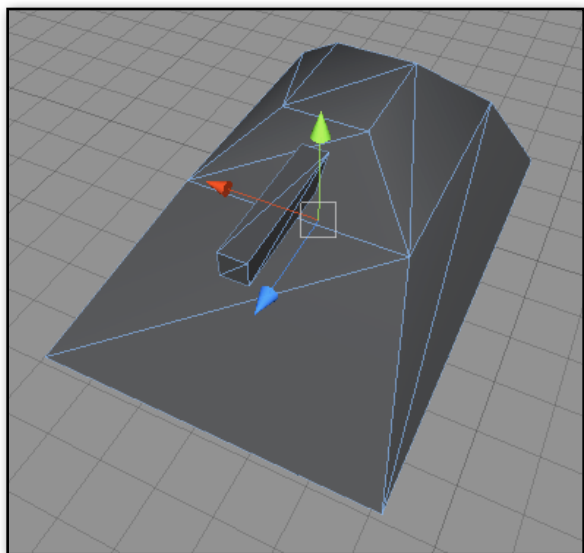




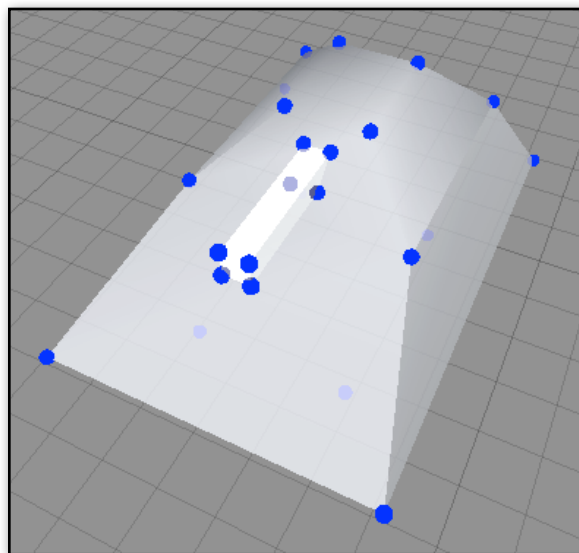
## LineMaker

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.

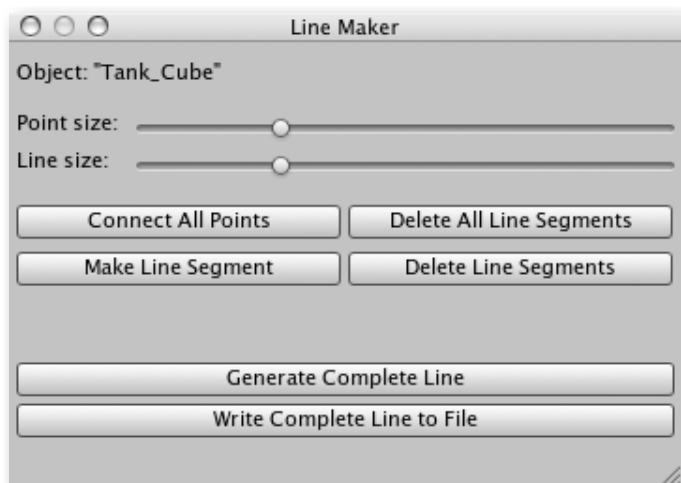
First, make a mesh in your 3D app of choice. Ideally this should be reasonably low-poly...LineMaker can get a little slow with high-poly objects. Drag the mesh into your scene.



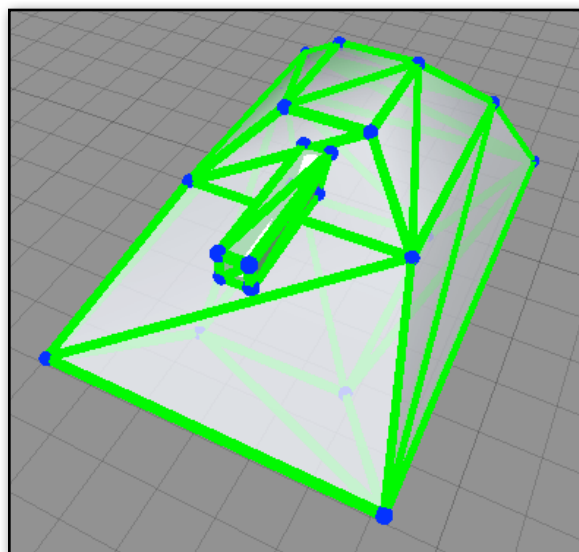
Then, with the object selected, choose **LineMaker...** from the Assets menu. Your mesh will become transparent, with blue dots placed at each vertex, and the LineMaker window will appear.



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.

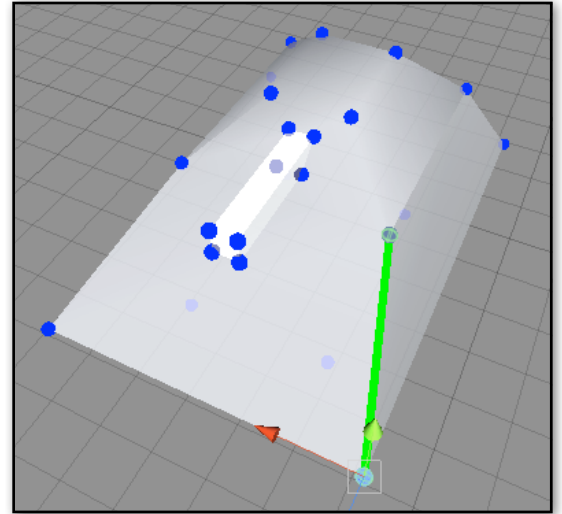
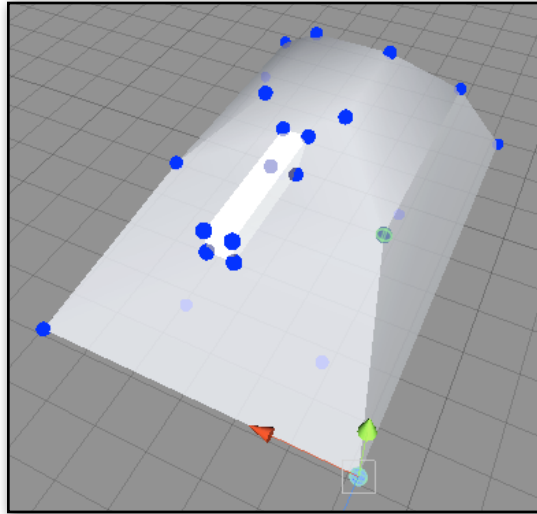


If you click on "Connect All Points", all points in the mesh will automatically be connected by green lines. These lines are what your 3D vector object will look like. You may find it easier to connect all points first, and then remove whatever line segments you don't want, rather than building it up 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, 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 (unless you’re using Unity 2.6, which has a bug that prevents TextAsset.bytes from working correctly). 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.

**Q: I can't see any lines!**

A: 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 3.2 or later; not all of the Vectrosity scripts are compatible with earlier versions.

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

The Physics Manager should have the Y axis of Gravity set to -19.81:

▼  Physics Manager (Physics Manager)  	
▼ Gravity	
X	0
Y	-19.81
Z	0