



Version 5.1

Thank you for purchasing Vectrosity! The goal is to make line-drawing easy, flexible, and fast. This document will help you get started with Vectrosity, and show you how to create lines in the editor. For information about line-drawing with scripts, as well as more detailed explanations of the various line parameters, see the **Vectrosity 5 Coding** document.

Note that Vectrosity 5 requires Unity 5.2.2 or later. If you're using Unity 4.6 - Unity 5.1, you'll need to use the files in the Vectrosity 4.4 folder. If you're using Unity 4.0 - Unity 4.5, you'll need to use the files in the Vectrosity 3.1.2 folder. (Unity 5.2.0 and 5.2.1 are not supported, due to Unity API changes.) This documentation only applies to Vectrosity 5, so if you use Vectrosity 3 or Vectrosity 4, you should stop reading now and open the appropriate documentation instead.

If you have any questions you can contact sales@starscenesoftware.com.

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

[Creating a Line \(page 3\)](#): Making a line on a canvas.

[Editing a Line in the Scene \(page 4\)](#): Adding, moving, and deleting points.

[The Line Inspector \(page 8\)](#): Customizing your line in various ways, with [Style](#), [Texture](#), [Partial Line](#), [Points](#), [Colors](#), and [Widths](#).

[Masking \(page 14\)](#): Mask your line with other shapes.

[Canvas Controls \(page 15\)](#): Anchors aweigh! And other stuff like canvas scaling.

[Using Lines in Code \(page 16\)](#): Control your lines with scripting.

[LineMaker \(page 17\)](#): A utility to help create 3D vector objects.

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

- **Vectrosity5:** The core Vectrosity package for Vectrosity 5, which contains all Vectrosity scripts in a .dll. Since it's a .NET .dll, rather than native code, it will work on any platform.
- **Vectrosity5Source:** This contains the core Vectrosity 5 scripts as source code. You'd generally want to use the .dll package instead, since .dlls are more convenient and can make script compiling faster, but the source is provided if you want to make modifications.

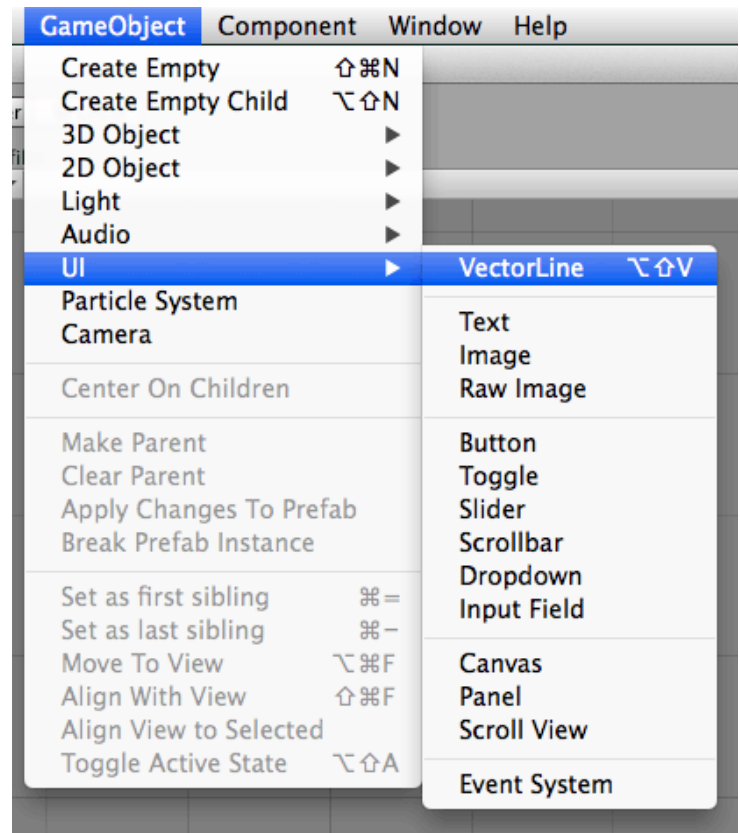
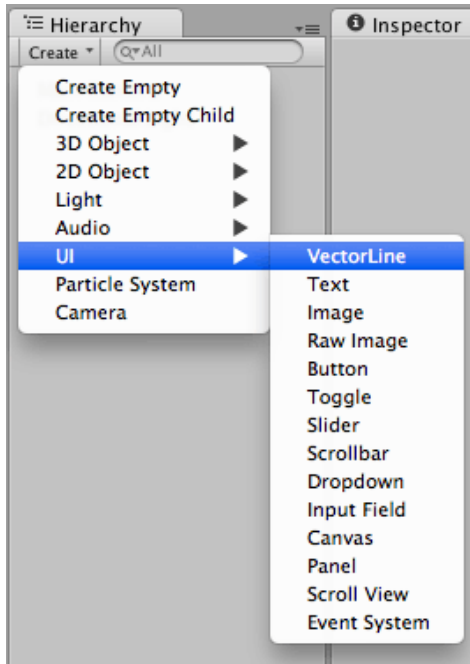
The following package can be imported after you've imported one of the above packages. It's not required, but it's useful for showing a range of Vectrosity functions.

- **Vectrosity5Demos:** Many demonstrations of various Vectrosity functions in a number of scenes. The demo scenes are located in the "Vectrosity/Demos/_Scenes" folder after importing the Vectrosity5Demos package. Some scenes have several related scripts — check the Main Camera object and enable/disable the ones you want.

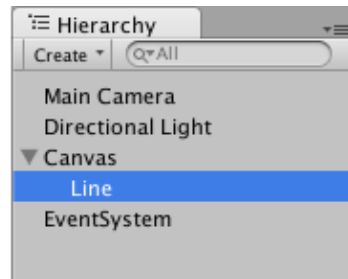
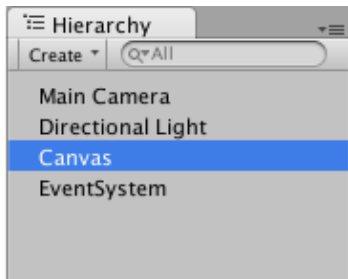
There are several documentation files:

- **Vectrosity5 Documentation:** You're reading it! This explains the basics of Vectrosity, and how to create lines in the editor.
- **Vectrosity5 Coding:** How to use the power of Vectrosity in your scripts.
- **Vectrosity5 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.
- **Vectrosity5 Upgrade Guide:** Have a look at this if you're upgrading from an older version. 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.

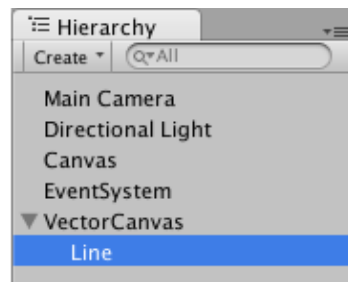
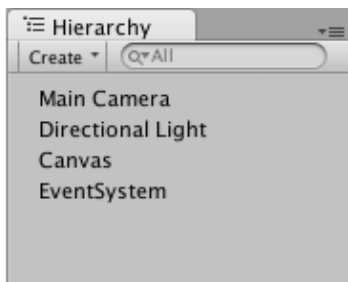
To create a line, use the **GameObject** menu, and select **VectorLine** from the **UI** submenu. You can also use the alt-shift-V keyboard shortcut, or use the **Create** menu.



If you have a canvas selected before creating the line, then the line will be created as a child of that canvas.



If you don't have a canvas selected, then Vectrosity will create a new canvas in the scene, called VectorCanvas, and the line will be parented to that.

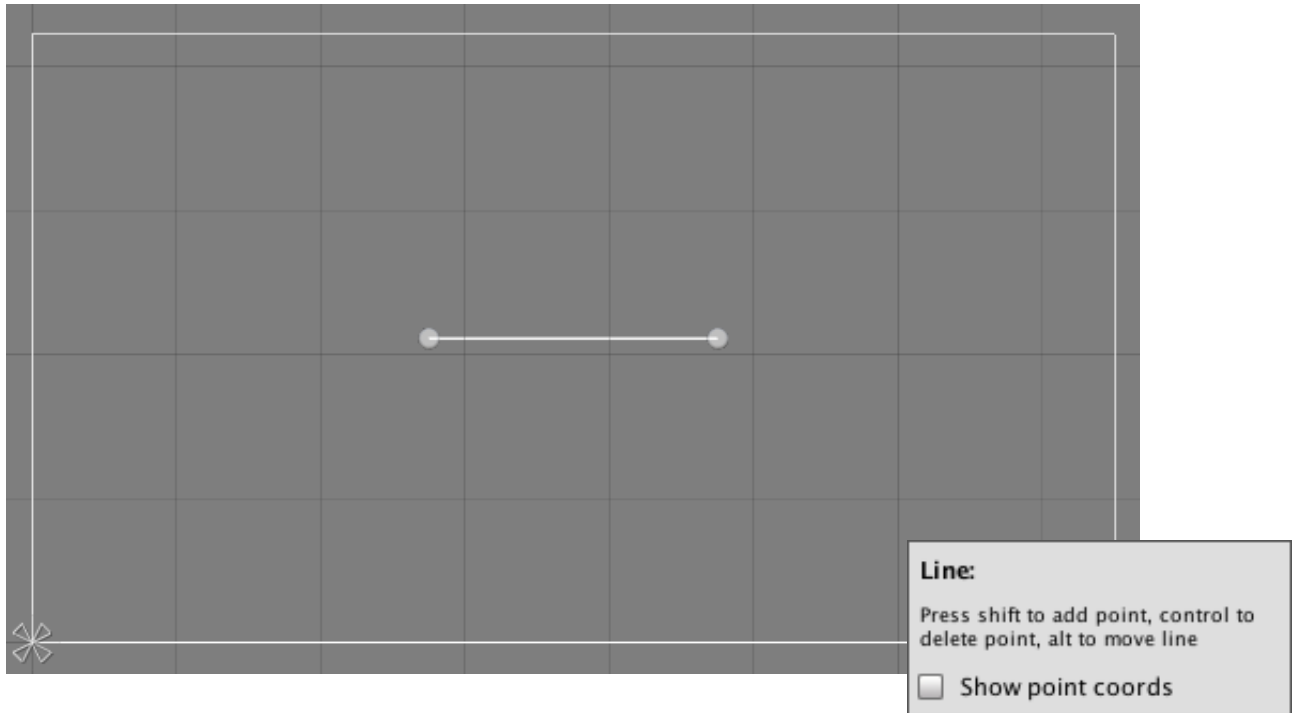


You can move the line to a different canvas after it's created.

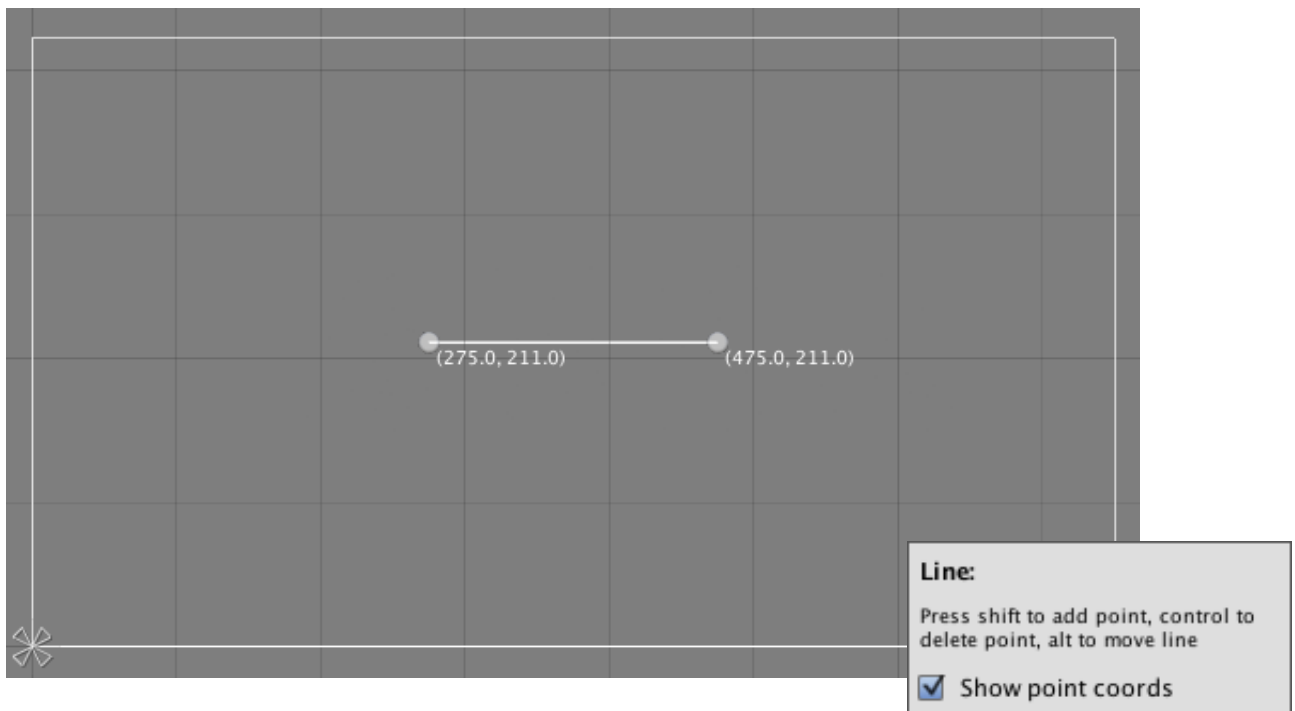
Line editing is done in the scene view in Unity. Make sure 2D mode is active; otherwise the line editing won't work. Click on the 2D button in the scene view toolbar if necessary.



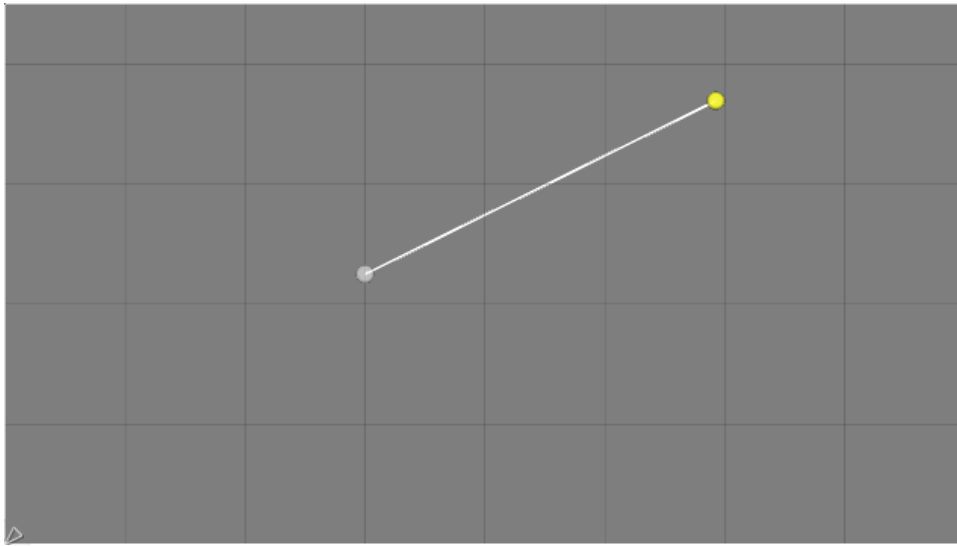
When you create a line on a canvas, it starts with two points. (If the scene view is zoomed in so you can't see the line, the fastest way to get a usable view is to double-click on the canvas in the hierarchy.)



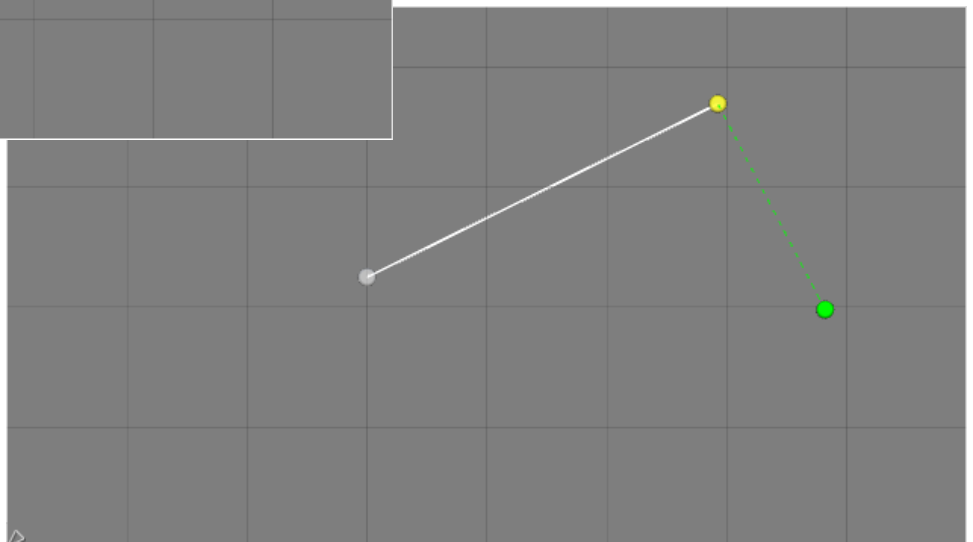
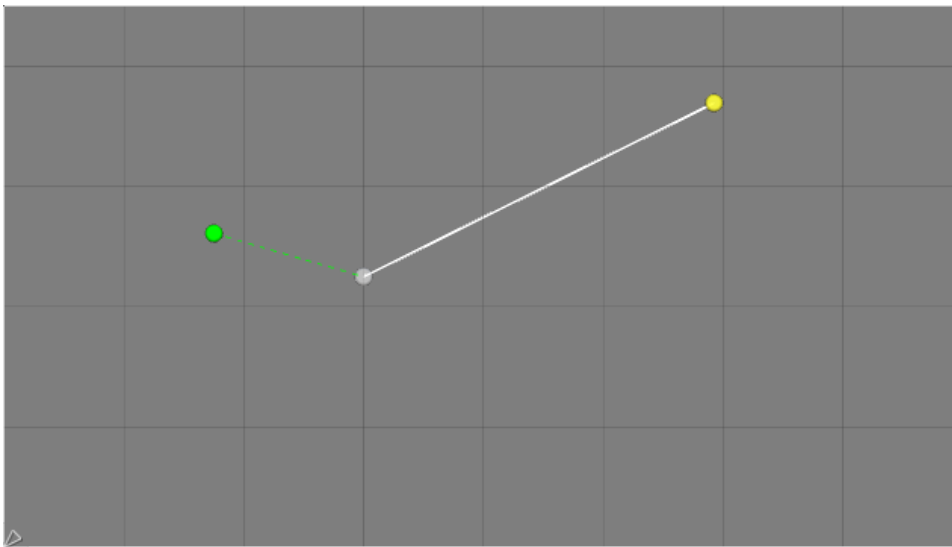
You will also see a small window in the corner of the scene view, which has reminders of the keyboard controls, and an option to show point coords. When active, the coordinates are shown next to each point:



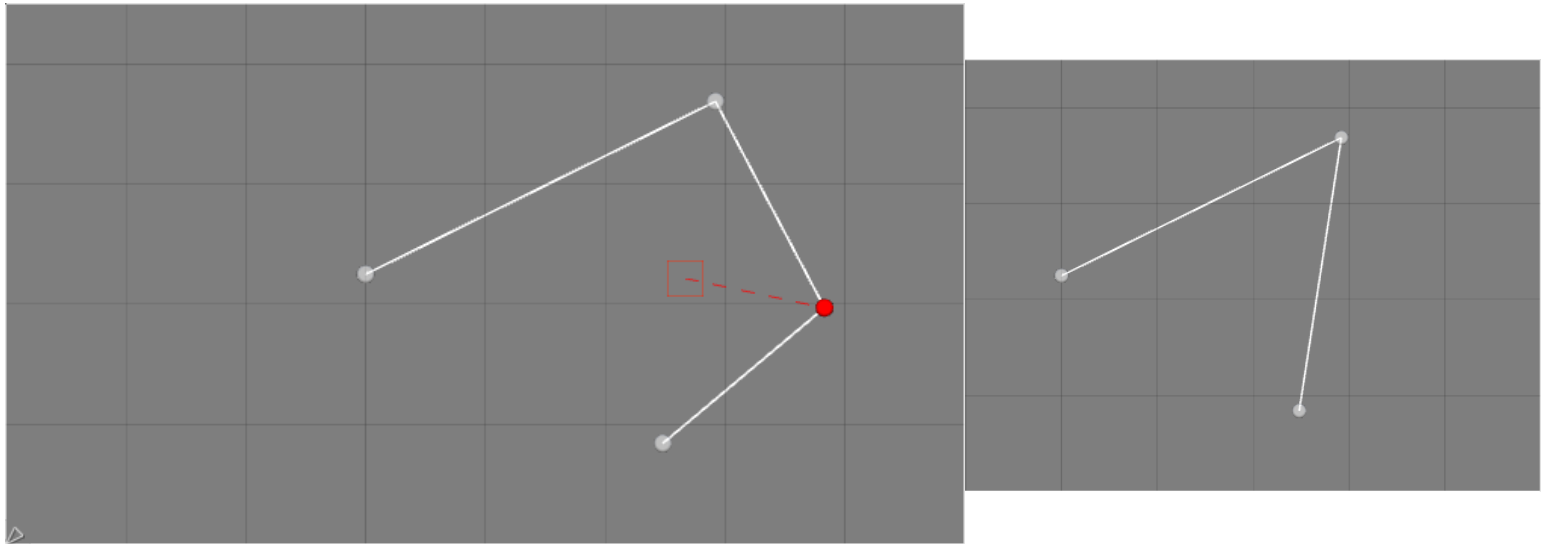
Each point on the line has a control point, which can be dragged. If you hold down **Command** (Mac) or **Control** (Windows) while dragging, the point will be snapped to the nearest pixel value.



If you hold down the **Shift** key, you can add a point to the line by clicking. The point is added to either the front of the line, or the end, depending on whether the cursor is currently closer to the front or end points. A green dot shows where the next point will be, with a dashed green line showing a preview of the line segment that will be added.

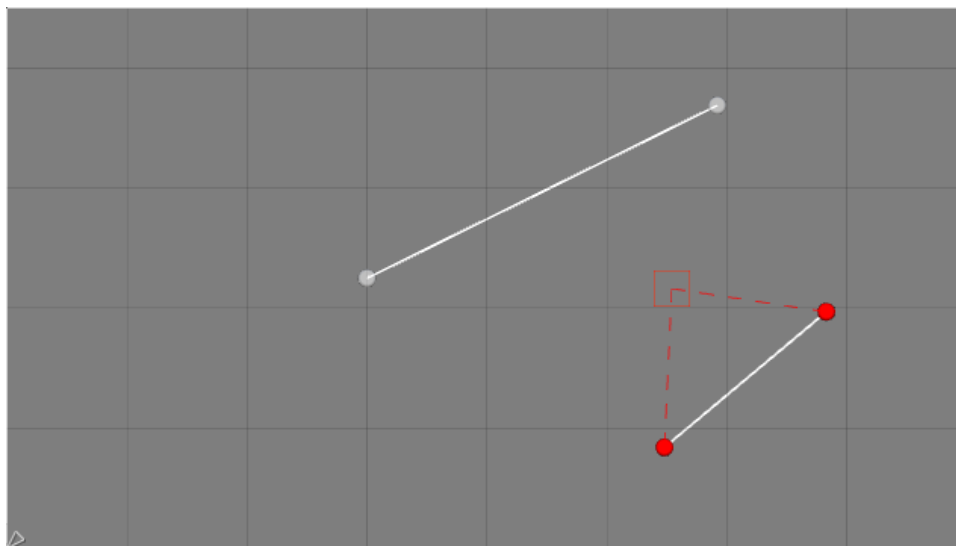


If you hold down the **Control** key, clicking will delete the point that the cursor (highlighted by a red square outline) is currently closest to. A red dashed line is connected to the point that will be deleted (which can be useful if you're zoomed in and the point isn't visible in the scene view). Once clicked, the point is deleted:

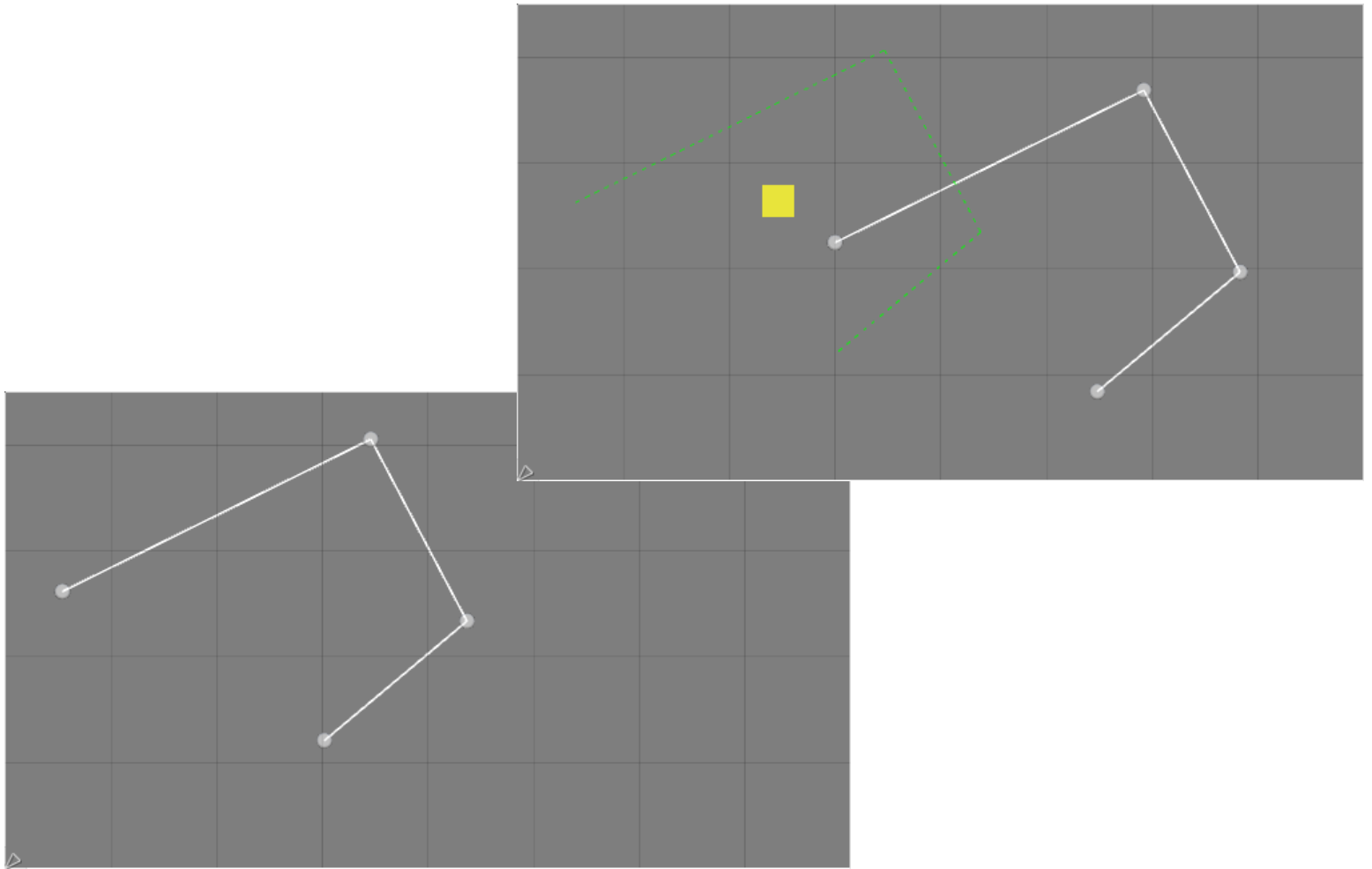


Note that there is always a minimum of two points in the line, so you won't be able to delete points if there are only two left.

If you're using a discrete line ([described in the next section](#)), two points—one line segment—will be deleted, rather than one point:



Finally, you can hold down the Alt key to move the entire line to wherever the mouse cursor is, highlighted by a yellow square. A preview made of dashed green line segments shows where the line will be once you click:



When you have a line selected in the hierarchy, you can see various parameters in the inspector. This is divided into a number of sections. Each section can be collapsed or expanded by clicking on the appropriate arrow. Let's start by taking a look at the first three: Style, Texture, and Partial Line.

Style

This section has some options that control how the line looks. The first is **Width**, which is the number of pixels wide the line is. This can be a floating-point number, such as 2.5. It has no particular upper limit, but must be at least 1. Below, the width has been changed to 40:

▼ Style

Width

2

Cap Length

0

Color

LineType

Continuous

Joins

None

Collider

☐

▼ Texture

Texture

None (Texture)

Select

Use Texture Scale

☐

Texture Scale

0

Texture Offset

0

▼ Partial Line

Use Partial Line

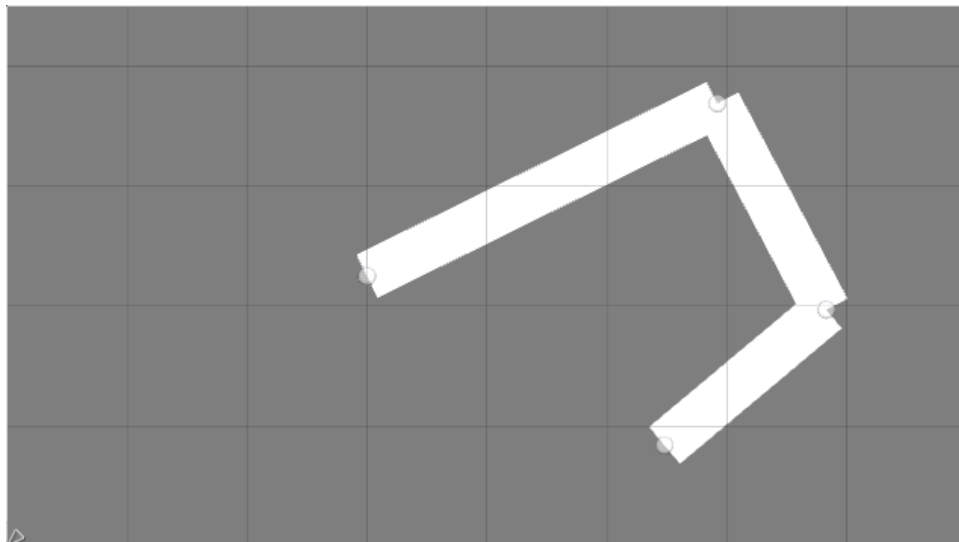
☐

Draw Start

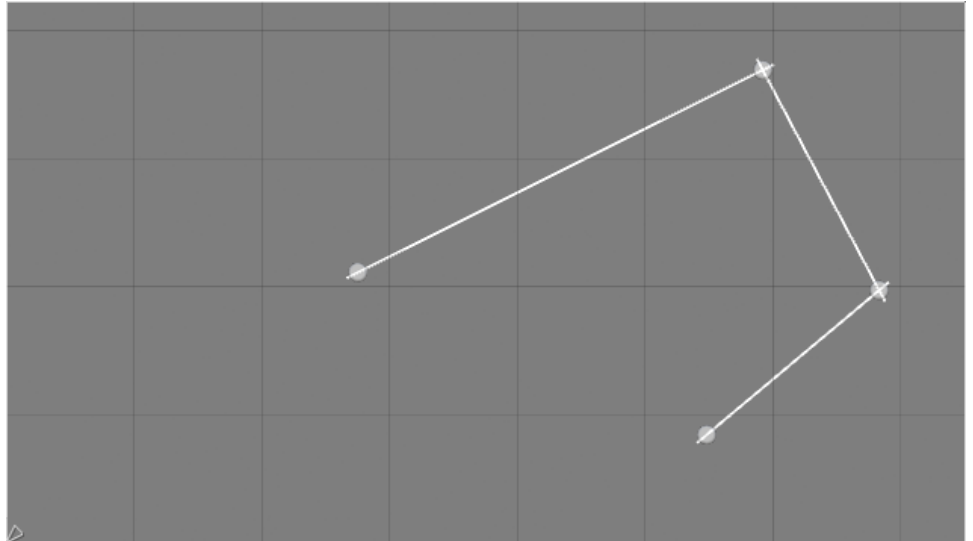
0

Draw End

3

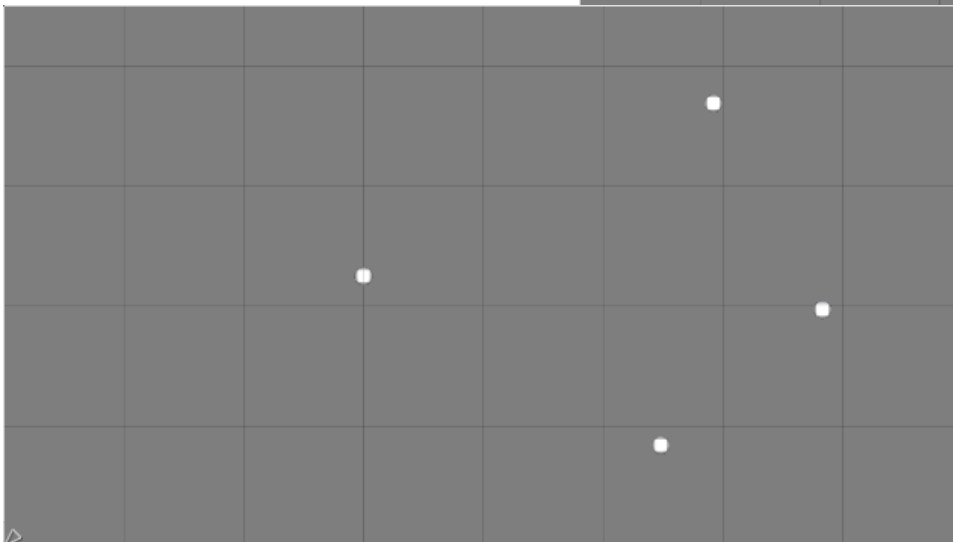
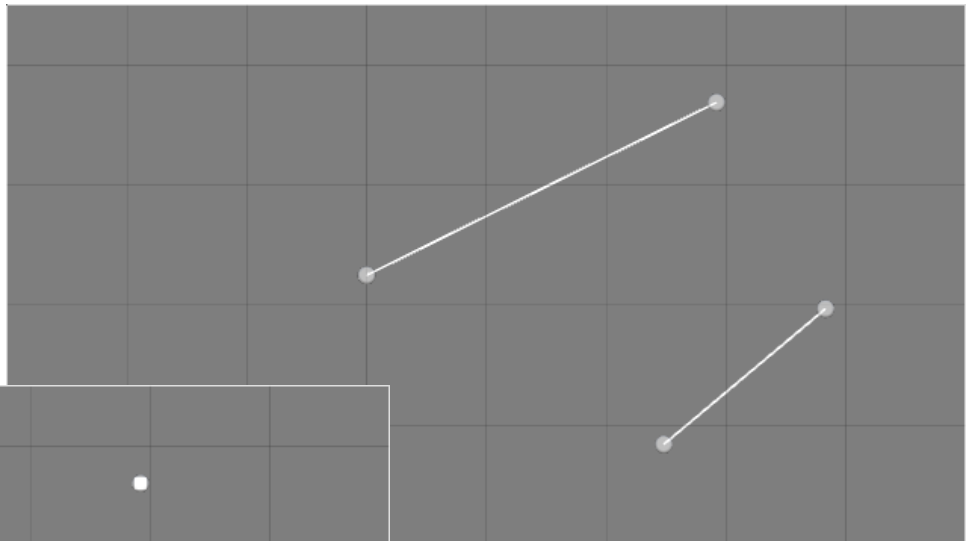


The **Cap Length** adds the given number of pixels to the ends of each line segment. (See the Vectrosity 5 Coding document for more info on cap length.) Here, a cap length of 10 has been used:

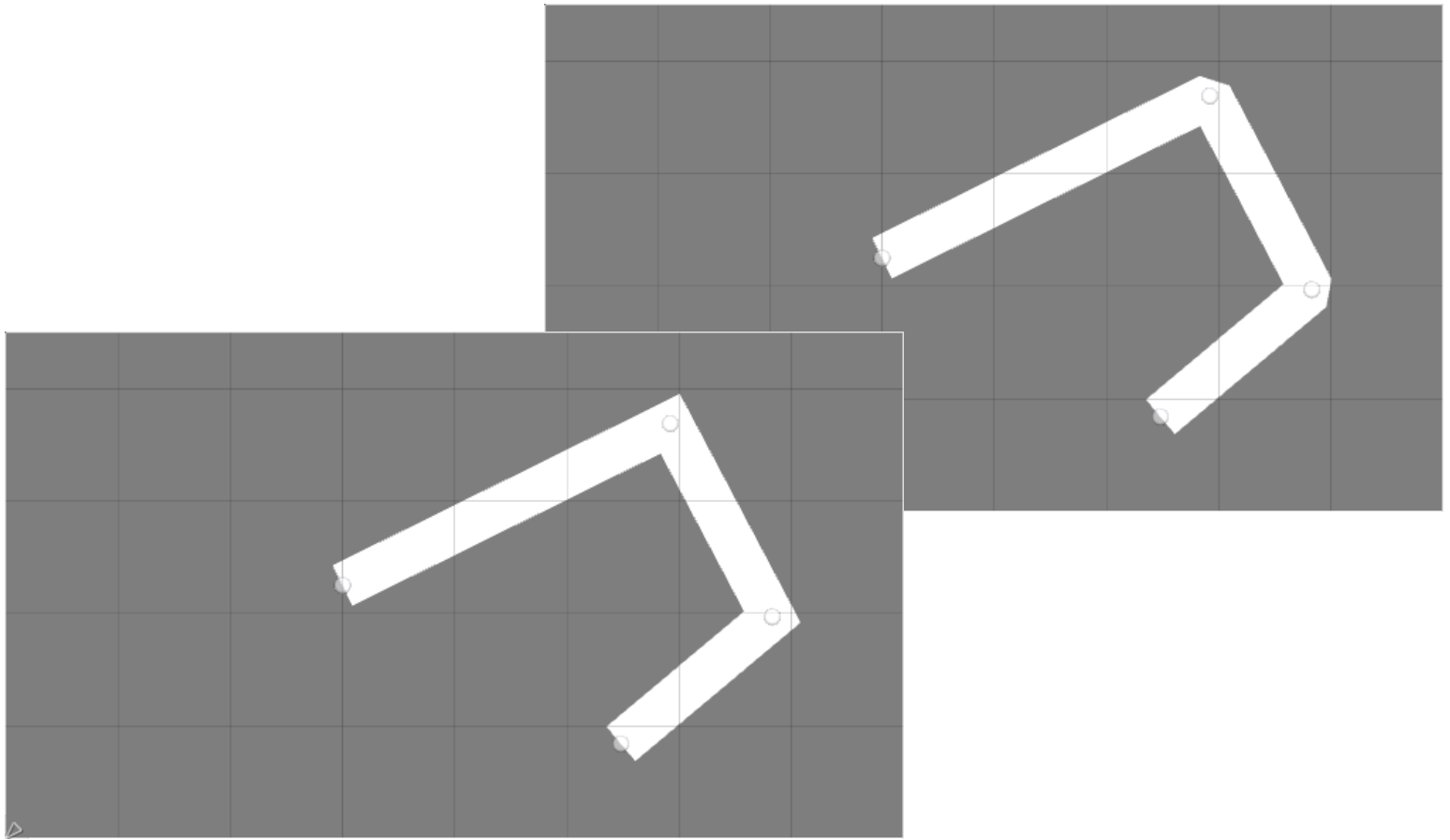


The **Color**, naturally, is the color of the line. This can be overridden on a segment-by-segment basis by using the entries in the [Colors](#) section.

The **LineType** is Continuous, Discrete, or Points. Continuous lines are made by connecting all points, Discrete lines are made with a separate segment for every two points, and Points just draws the actual points, rather than line segments.

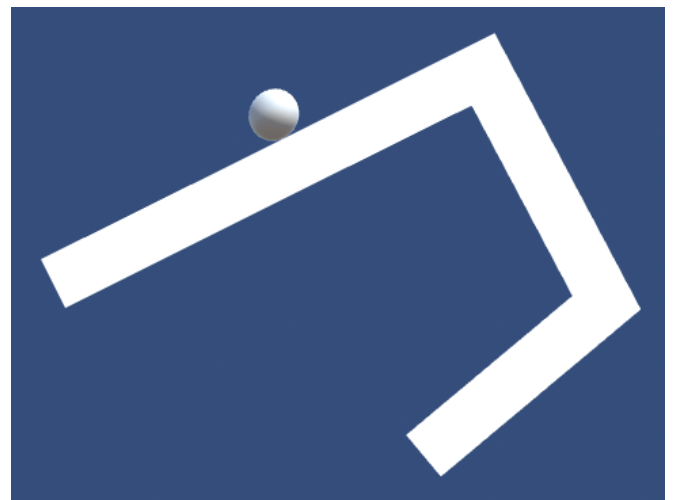
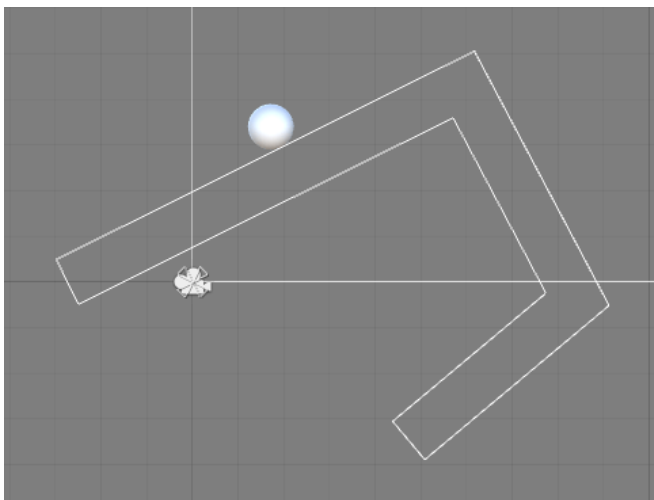


Joins can be Fill, Weld, or None. So far we've seen Joins.None, the default. Joins.Fill fills in the gaps that are visible with thick lines, and Joins.Weld moves vertices in line segments to weld them together.



For more information on LineType and Joins, see the Vectrosity 5 Coding document.

Finally, **Collider** adds a 2D physics collider component to the line. Note that the collider is in world space, so objects in the scene can collide with it, as long as they use 2D physics colliders (not 3D). This means it doesn't visually match the line that you edit in the canvas, so you'd have to zoom in to see it in the scene view. (Known issue: if you change the line type while it has a collider, the editor will print a "missing reference exception" error. This is generated by Unity code and seems to be a bug, but is harmless.)



Texture

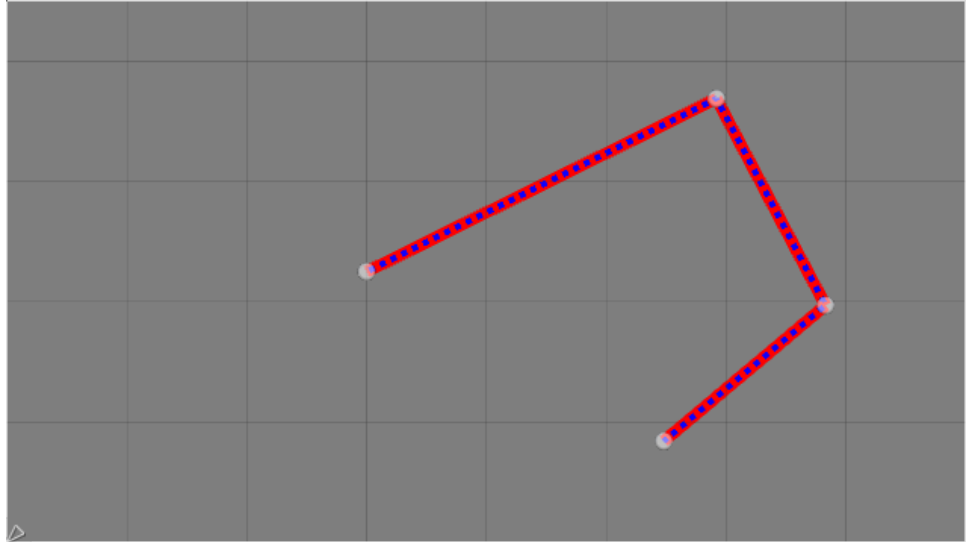
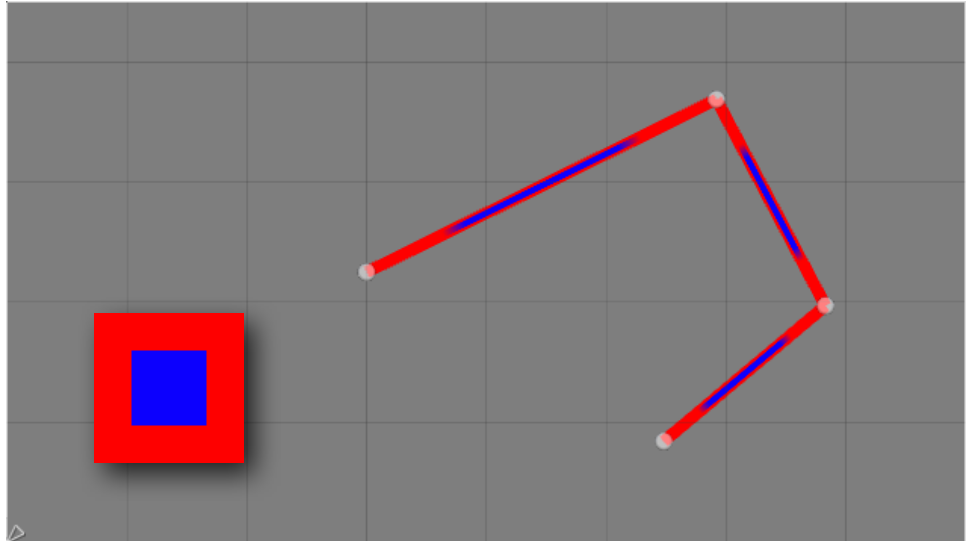
You can add a texture to the line using the **Texture** slot. By default, the texture is stretched along the length of each line segment.

If you check **Use Texture Scale**, the texture will be scaled uniformly instead. This requires that the texture uses a wrap mode of Repeat rather than Clamp. You'll see a message in the inspector if the texture isn't set to the correct wrap mode.

The **Texture Scale** number can be used to stretch a texture by a relative amount while it repeats.

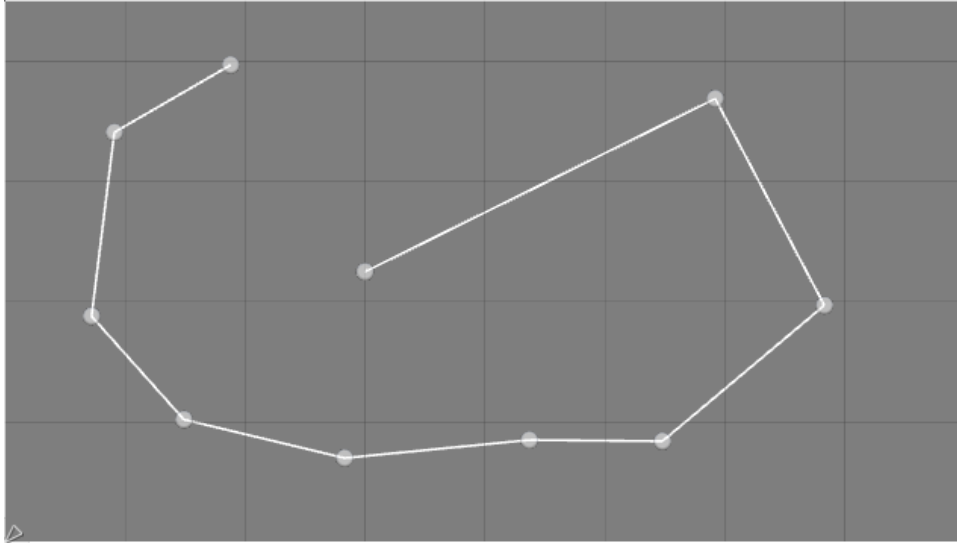
The **Texture Offset** number will offset the texture by a relative amount.

For more details about uniformed-scaled textures, see the Vectrosity5 Coding document.

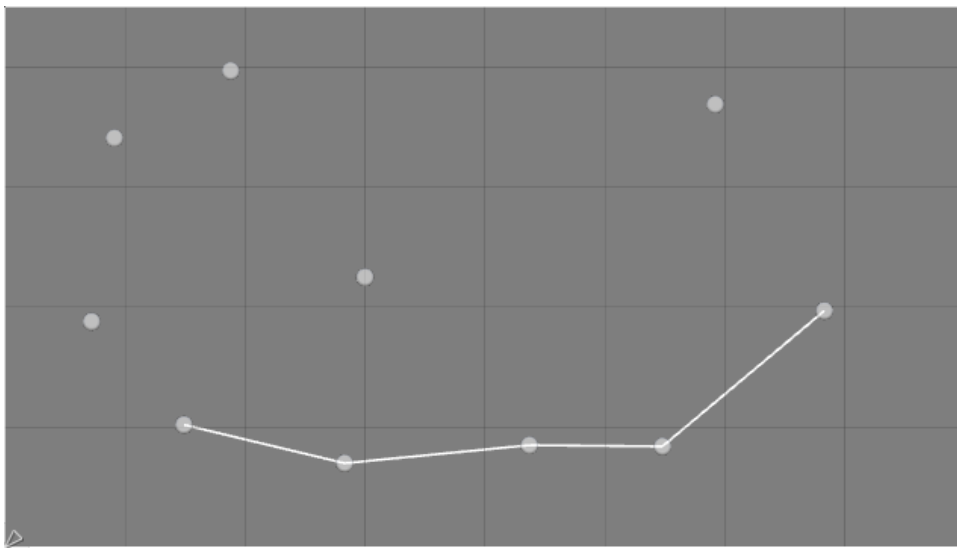


Partial Line

If you check **Use Partial Line**, then you can control how much of the line is drawn, where you set the **Draw Start** and **Draw End** values. The line will visibly start and end at the point numbers you specify, and the rest will be invisible, but you can still edit all the points. You can also use the slider to control this instead of manually typing numbers. If we have a line with 10 points:



And we set Draw Start to 2, and Draw End to 6, then only points 2-6 are drawn (remember that point numbers start at 0):



This is can be particularly useful for animation effects when controlled through scripting.

Line Points

In addition to editing points in the scene view, you can edit them in the inspector. The + button at the top adds a point to the end of the line, and the - button removes the last point.

Each point also has its own + and - buttons. The + button inserts a new point immediately after that point, using the same coordinates. The - button deletes that point. There must always be at least two points in a line, so the - buttons are grayed out if it's not possible to delete any more.

If you use a discrete line, as opposed to a continuous line, then points are added and removed two at a time, in order to keep the segments consistent.

Colors

Each line segment can have a different color. Click on the corresponding color control next to each segment to change the color.

Changing line segment colors overrides the overall line color from the Styles section. You can revert all segment colors to the base color by clicking the **Reset Colors** button.

If you check the **Smooth Color** box, the colors are smoothly interpolated from one segment to the next.

Widths

Each line segment can also have a different width. This works basically the same as the Colors section, except with line widths. Clicking **Reset Widths** resets each segment to the base width, and checking **Smooth Width** interpolates line segment widths.

More info about line segment colors and widths can be found in the Vectrosity5 Coding document.

▼ Line Points

Number of points: 4 + -







0	X	275	Y	211	+	-
1	X	592	Y	369	+	-
2	X	683	Y	197	+	-
3	X	548	Y	84	+	-

▼ Colors

Smooth Color ☐

Reset colors

Number of segments: 3

0		
1		
2		

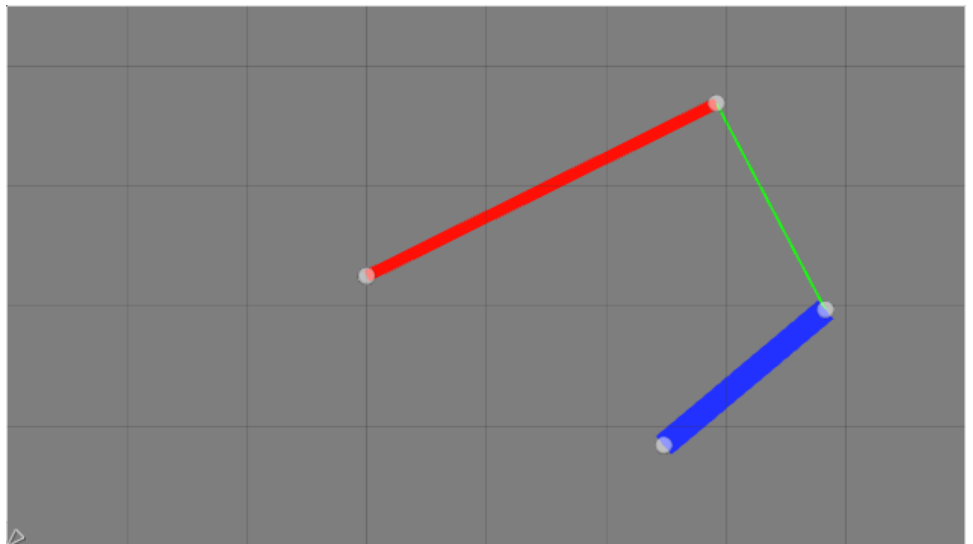
▼ Widths

Smooth Width ☐

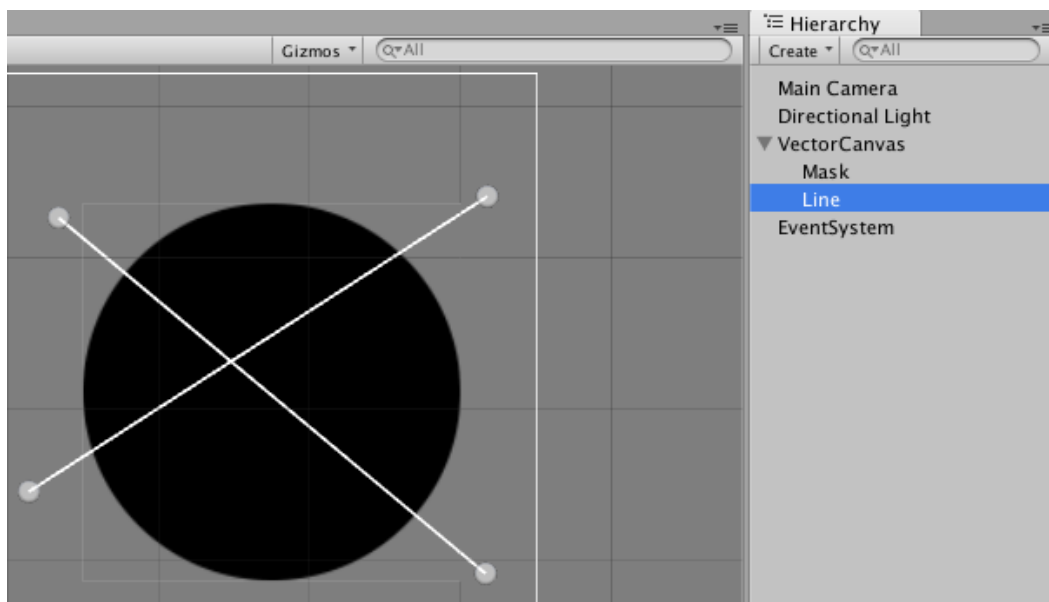
Reset widths

Number of segments: 3

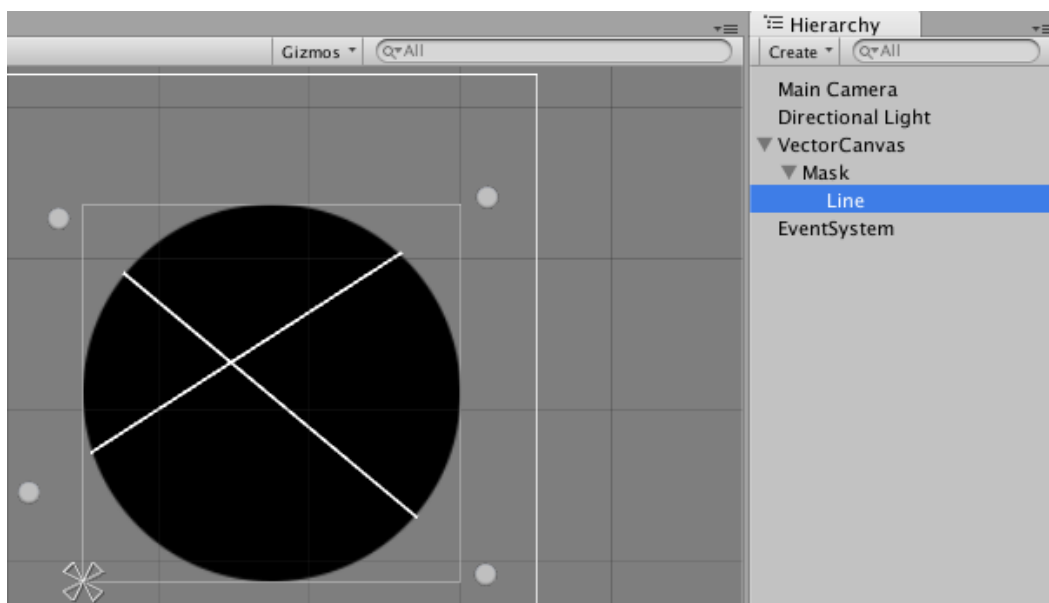
0	10
1	2
2	20



Lines can be masked by sprites. This is quite simple to do: first, create a UI **Image** object. Then add a UI **Mask** component to the image object, and add a sprite to the image component. Here we have a circle sprite mask under the VectorCanvas:



Now move the line object so it's a child of the mask object:



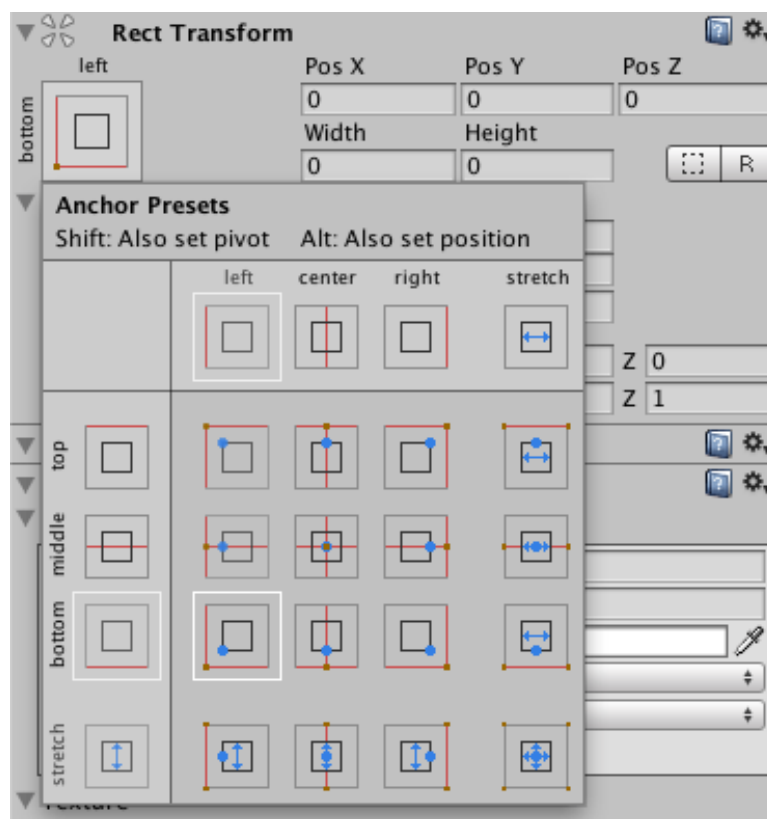
The mask can be invisible; uncheck “Show Mask Graphic” on the mask component. The line points will remain editable when masked.

The line can make use of RectTransform controls such as anchor presets. By default this is the bottom-left, but you can click on any of the 9 standard presets to anchor to different sides or corners of the screen. Note that the “stretch” options don’t do anything with VectorLines.

By default, the VectorCanvas uses a canvas scaler component set to “constant pixel size” with a scale factor of 1. This means line widths will always use a constant pixel value regardless of screen size. But if you set the scale factor to 2, for example, the line will now be double the size, and a line width of 5 would actually equal 10 pixels visually.

You can also set the canvas scaler to other values such as “scale with screen size”. In this case lines will be scaled up or down as the screen changes in size, so the line pixel width values will only be “correct” for the game window size that you used when creating the line. This can be a desirable effect, but if not, use the “constant pixel size” canvas scaler instead.

Note that the canvas render mode must be “Screen Space - Overlay” if you want to edit lines in the canvas visually. “Screen Space - Camera” and “World Space” won’t work, and will result in the line info window printing a warning. You can still use the inspector to edit lines, though.



There are many things you can do with lines in code, such as changing various parameters dynamically at runtime, including the points themselves. There are also some options that aren't yet available in the current version of the Vectrosity line inspector. The type of lines that you create in the editor is **VectorObject2D**. The actual lines themselves use the **VectorLine** type, since VectorLines can actually be 2D or 3D, which behave in different ways. So far we've only been discussing 2D lines; see the [LineMaker](#) section for a way to visually edit 3D lines. Typically the best way to get a reference to a VectorObject2D is to create a public variable in your script. You can then refer to the **vectorLine** property of this variable to control the line.

For lots of information about scripting VectorLines, see the **Vectrosity 5 Coding** document. Here's a simple script that moves the first point in a line around in a circle with a radius of 50 pixels. Create a line, then make an empty GameObject. Attach this script and drag your line onto the Line (Vector Object 2D) slot.

```
import Vectrosity;    // Unityscript

var line : VectorObject2D;
private var point0 : Vector2;

function Start () {
    point0 = line.vectorLine.points2[0];
}
function Update () {
    var point = line.vectorLine.points2[0];
    point.x = Mathf.Sin (Time.time * 2.0) * 50.0 + point0.x;
    point.y = Mathf.Cos (Time.time * 2.0) * 50.0 + point0.y;
    line.vectorLine.points2[0] = point;
    line.vectorLine.Draw();
}
```

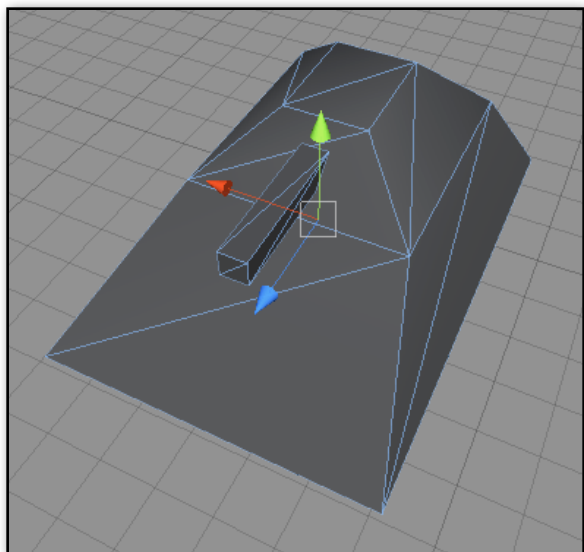
```
using UnityEngine;    // C#
using Vectrosity;

public class LineExample : MonoBehaviour {
    public VectorObject2D line;
    Vector2 point0;

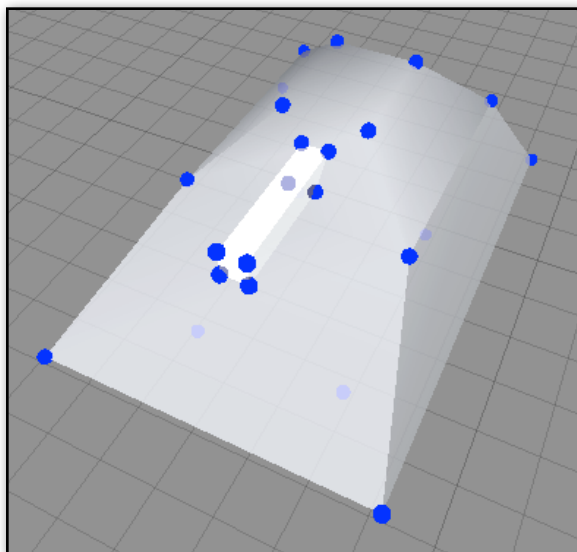
    void Start () {
        point0 = line.vectorLine.points2[0];
    }
    void Update () {
        var point = line.vectorLine.points2[0];
        point.x = Mathf.Sin (Time.time * 2.0f) * 50.0f + point0.x;
        point.y = Mathf.Cos (Time.time * 2.0f) * 50.0f + point0.y;
        line.vectorLine.points2[0] = point;
        line.vectorLine.Draw();
    }
}
```


VectorLines can use 3D points as well as 2D. The line inspector currently only works with 2D points, but you can use the LineMaker utility to do some 3D point editing, based off of existing meshes.

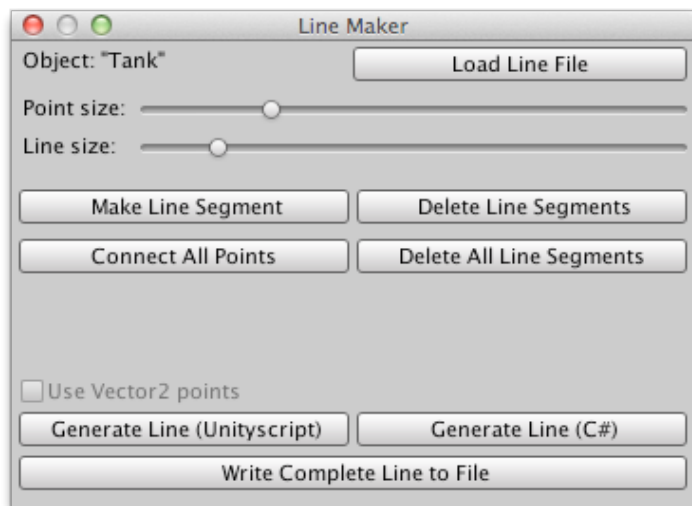
1) Make a mesh in your 3D app of choice. Ideally this should be reasonably low-poly...LineMaker may get a little slow with high-poly objects. Drag the mesh into your scene.



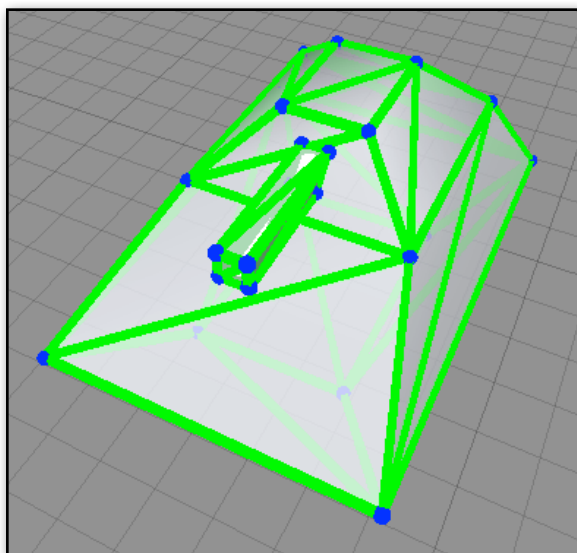
2) With the object selected, choose **LineMaker...** from the Assets menu. Your mesh will become transparent, with blue dots placed at each vertex, and the LineMaker window will appear.



3) 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.



4) 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 Line (Unityscript)” or “Generate Line (C#)”. Use whichever button corresponds to the language you’re using. 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 list. An empty list looks like this:

```
var tankLines = new List.<Vector3>(); // Unityscript
var tankLines = new List<Vector3>(); // C#
```

Paste the text between the parentheses (this example uses just one point for brevity...the real thing would be quite a bit longer!):

```
var tankLines = new List.<Vector3>([Vector3(1.748, -2, -2.513)]); // Unityscript
var tankLines = new List<Vector3>(){new Vector3(1.748f, -2f, -2.513f)}; // C#
```

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 **BytesToVector3List** in the **Vector Utilities** section in the **Vectrosity5 Coding** document for information on how to use these files.

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.

2D points

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 2D data. The “Use Vector2 points” toggle will be usable in this case; otherwise it’s grayed out. When using 2D points, the “generate line” buttons generate lists with Vector2, and the “write line to file” button will write data suitable for loading with **BytesToVector2List**.