A note about function parameters
All parameters where only the type is listed must be supplied. All parameters where a default value is listed are optional, and will use the default value if they are omitted.

```
VectorLine (name : String,
            points : List<Vector2> or List<Vector3>,
            texture : Texture = null,
            lineWidth : float,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.None) : VectorLine
```

Constructs a VectorLine object with the supplied parameters. The texture and parameters after **lineWidth** are optional and have the default values indicated, though if you supply joins, you must also supply lineType.

name is a string that's used to name the GameObject and mesh created for the vector line. The name is also used to identify bounds meshes made with VectorManager.ObjectSetup.

points is a Vector2 or Vector3 generic List, where each entry is a point in the line using screen-space coordinates (in the case of a Vector2 list), or world units (in the case of a Vector3 list). The list may be empty initially, with points added later. If the list is declared with an initial capacity (i.e., "new List<Vector2>(50)"), the list will have the capacity filled with Vector2.zero or Vector3.zero as appropriate, so the list.Count will equal the list.Capacity.

The line is drawn using the supplied **texture**. If the texture is omitted, the line is drawn as a solid color.

The **lineWidth** is the width in pixels. This is a float, so values like 1.5 are acceptable.

lineType is LineType.Discrete, LineType.Continuous, or LineType.Points. For discrete lines, each line segment is made from two consecutive entries in the Vector2 or Vector3 list. For continuous lines, the line starts at entry 0, and each segment is continuously connected to the next point until the end is reached. For points, each entry in the points list is a separate point.

joins is one of Joins.None, Joins.Fill, or Joins.Weld. Joins.None is the default and draws line segments as standard rectangles, primarily used with thin lines. Joins.Fill will fill in the gaps seen where lines join at an angle, primarily used with thick lines with no texture. This only works if using LineType.Continuous. Joins.Weld is similar, except vertices of sequential line segments are welded, which makes it more appropriate for textured lines. Joins.Weld works with LineType.Discrete, but only for sequential line segments. When using LineType.Points, only Joins.None can be used.

active

```
var active : boolean = true;
```

Sets a VectorLine active or inactive. Inactive VectorLines won't be rendered, and if an inactive VectorLine is used with functions like Draw, the function will return immediately without doing anything.

alignOddWidthToPixels

```
var alignOddWidthToPixels = false;
```

If true, offsets a line's RectTransform by (0.5, 0.5). Line segments drawn with odd line widths (1, 3, 5, etc.) are positioned "between" pixels when they are straight horizontal or vertical, which can result in a blurred appearance if FSAA is used, most noticeable with a 1-pixel width. Setting alignOddWidthToPixels to true will align odd line widths to the pixel grid, in order to prevent FSAA blurring.

capLength

```
var capLength : float = 0;
```

Sets the line's **capLength** to the supplied float. This is the number of pixels added to either end of the line (0 by default). Typically used for filling in gaps seen in thick lines when they are joined at right angles, such as when drawing selection boxes or other rectangles.

collider

```
var collider : boolean = false;
```

If true, the line will generate a matching 2D collider when it's drawn. The collider is updated if the line is changed and re-drawn. Setting this to false will disable the collider, if one existed previously. The collider works with lines made with Vector3 points as well as lines made with Vector2 points. Since 2D colliders are on the X/Y plane only, the camera should be facing along the Z axis, with no rotation, in order for the collider to correctly match the line.

color

```
var color : Color = Color.white;
```

The color of the VectorLine. When set, all line segments are colored with the supplied value. See also VectorLine.SetColor and VectorLine.SetColors. Note that VectorLine.color will retain its original value even if all line segments have been set to different colors when using SetColor or SetColors. The default value is Color.white when a VectorLine is first constructed.

continuousTexture

```
var continuousTexture : boolean = false;
```

If this is set to true, then any texture used for the line will be stretched evenly along the entire length of the line, regardless of how many points make up the line or how far apart the points are spaced. If set to false (which is the default), then the texture is repeated once for every line segment.

drawDepth

```
var drawDepth : int;
```

The sibling index of the VectorLine as drawn in the vector canvas. Lines with higher drawDepth values are drawn on top of lines with lower drawDepth values. Each line has a unique drawDepth value, which is assigned in order as more VectorLines are created, so changing the drawDepth value for one line may change the values of others. The minimum is 0, and the maximum depends on the number of VectorLines parented to an object (which is the vector canvas by default, but can be changed using VectorLine.rectTransform). VectorLine.drawDepth can't be used on lines drawn with Draw3D.

drawEnd

```
var drawEnd : int;
```

Specifies the index in the VectorLine's points list at which line drawing should end. Any later points will be erased when VectorLine.Draw is called. (Erased on-screen, not from the points list.) When used with a discrete line, drawEnd should be an odd number, although it can be 0. If it's even (aside from 0), it will be incremented to the next highest integer. It's clamped between 0 and points.Count-1. This can be used with drawStart to specify a range of points to draw in a line.

drawStart

```
var drawStart : int;
```

Specifies the index in the VectorLine's points list at which line drawing should start. Any earlier points will be erased when VectorLine.Draw is called. (Erased on-screen, not from the points list.) When used with a discrete line, drawStart should be an even number. If it's odd, it will be incremented to the next highest integer. It's clamped between 0 and points2.Length-1 or points3.Length-1.

drawTransform

```
var drawTransform : Transform;
```

When drawn, the line is modified by the supplied transform. For example, if the transform is located at Vector3(5.0, 2.0, 0.0), then the line is offset by 5 units on the X axis and 2 units on the Y axis. A transform that's rotated will cause the line to be rotated by the same amount, and a transform that's scaled will cause the line to be scaled by the same amount, though it will always be drawn with the correct thickness. If the transform is modified, the line will need to be re-drawn in order to reflect the changes.

endCap

```
var endCap : String;
```

Specifies an end cap for the VectorLine, using a name as defined in SetEndCap. The name is case-sensitive, and SetEndCap must have been called before an end cap can be added. If null or "" is used for the string, then the end cap is removed from the VectorLine.

endPointsUpdate

```
var endPointsUpdate : int = 0;
```

The number of points from the end of the VectorLine that are actually updated when Draw or Draw3D is called. The other points will be untouched. This is intended to be used as an optimization for cases where most of the line stays the same and does not need to be recomputed when re-drawn.

Note that when used with discrete lines, endPointsUpdate always uses even numbers. If odd numbers are supplied, they will be rounded up to the next even number, so 1 becomes 2 and so on.

is2D

```
var is2D : boolean;
```

Returns true if the VectorLine was made with Vector2 points, false otherwise.

joins

```
var joins : Joins;
```

Returns the Joins type that was used when the line was created, and can be used to change the Joins type at any time. Does nothing if Joins.Fill is used with a discrete line. Lines made with LineType.Points can only use Joins.None.

layer

```
var layer : Layer;
```

The GameObject layer of the line, for use with culling masks. By default this is the UI layer.

lineType

```
var lineType : LineType;
```

Whether a line is LineType.Continuous, LineType.Discrete, or LineType.Points. Changing the line type removes any individual line segment colors or widths (such as those set by SetColors or SetWidths), since different line types have different numbers of segments.

lineWidth

```
var lineWidth : float;
```

The width of the line, in pixels. This can be used to change the width of a VectorLine object after it’s been declared. If a VectorLine has multiple widths, then setting lineWidth will set all widths in the line to the supplied value, and reading lineWidth will return the value of the first entry in the list. VectorLine.Draw must be called after setting the lineWidth value for the effect to be visible on-screen.

material

```
var material : Material;
```

Assigns a material to the VectorLine, overriding the default. The default for lines drawn with Draw is the default UI shader, and the default for lines drawn with Draw3D is the DefaultLine3D material located in Assets/Vectrosity/Resources.

matrix

```
var matrix : Matrix4x4;
```

The line will be modified by the supplied Matrix4x4 when it’s drawn. This is similar to using drawTransform, except no other object is required since the matrix is supplied directly.

maxWeldDistance

```
var maxWeldDistance : float;
```

The number of pixels that a weld can extend, from where a given line segment joins with the next line segment. If this distance is exceeded, the weld is cancelled. By default, maxWeldDistance is twice the pixel width of the line when it was created.

name

```
var name : string;
```

Returns the name of the VectorLine, which is initially the string that was supplied when the VectorLine was constructed. The name can be changed at any time, which applies to both the GameObject and Mesh used for the line.

physicsMaterial

```
var physicsMaterial : PhysicsMaterial2D;
```

If a collider is used for the line, it will have the supplied PhysicsMaterial2D.

points2

```
var points2 : List<Vector2>;
```

This is a reference to the list of Vector2 points that was used when the line was set up. The entries in the list can be changed at any time, and then VectorLine.Draw used to draw the line with the updated points. Standard generic List functions such as Add and RemoveAt can be used to change the count of the list. (See also Resize.) If the line was made with Vector3 points, then accessing points2 will result in an error. Use is2D to determine if a line was made with Vector2 points.

points3

```
var points3 : List<Vector3>;
```

Similar to points2, except it's a reference to the list of Vector3 points that was used when the line was set up. If the line was made with Vector2 points, then accessing points3 will result in an error. Use is2D to determine if a line was made with Vector2 points.

rectTransform

```
var rectTransform : RectTransform;
```

A reference to the RectTransform component of the VectorLine. This can be used to manipulate the position of the line, but in most cases this is not recommended; use drawTransform instead.

smoothColor

```
var smoothColor : boolean = false;
```

Should line segment colors in this VectorLine be smoothly interpolated between segments? By default this is false, so each segment has its own discrete color. Line segment colors are set with VectorLine.SetColor and VectorLine.SetColors.

smoothWidth

```
var smoothWidth : boolean = false;
```

Should line segment widths in this VectorLine be smoothly interpolated between segments? By default this is false, so each segment has its own discrete width. Line segment widths are set with VectorLine.SetWidth and VectorLine.SetWidths.

texture

```
var texture : Texture;
```

If the line was created with a texture, this is a reference to that texture, otherwise it will be null initially. The texture can be set at any time. If VectorLine.endCap has been set, the line’s texture is derived from the end cap instead.

textureOffset

```
var textureOffset : float = 0.0;
```

Used in combination with textureScale. If the value of textureScale is not 0.0, then the texture will be horizontally offset (relative to the line) proportionally by the amount specified in textureOffset.

textureScale

```
var textureScale : float = 0.0;
```

Used for making uniform-scaled textures, such as dotted or dashed lines. If not 0.0, then the width of the texture supplied by the line’s material is scaled by the specified amount compared to its height, and the texture is repeated this way for the length of the line. For example, a value of 1.0 means a square texture would be as wide as it is high.

trigger

```
var trigger : boolean = false;
```

If VectorLine.collider has been set, then VectorLine.trigger controls whether the collider is set as a trigger or not.

useViewpointCoords

```
var useViewportCoords : boolean = false;
```

If true, the line (or points) is drawn with viewport coordinates, where Vector2(0.0, 0.0) is the lower-left corner of the screen, and Vector2(1.0, 1.0) is the upper-right corner, regardless of resolution. Works with Vector2 coordinates only.

VectorLine functions are used with a VectorLine object. For example, if “myLine” is a VectorLine, then “myLine.Draw()” or “myLine.MakeRect (someRect)”.

AddNormals

```
function AddNormals () : void
```

Creates normals for a VectorLine. This should be called when a material that has a shader which requires normals is used for the VectorLine. If the line has already been drawn, it should be re-drawn after using AddNormals in order for the normals to actually be applied.

AddTangents

```
function AddTangents () : void
```

Creates mesh tangents for a VectorLine. This should be called when a material that has a shader which requires tangents is used for the VectorLine. AddNormals is called automatically whenever AddTangents is called, since tangents require normals. If the line has already been drawn, it should be re-drawn after using AddTangents in order for the tangents to actually be applied.

Draw

```
function Draw () : void
```

Draws the VectorLine object using a canvas. Lines with either Vector2 or Vector3 points can be used. If the line uses Vector3 points, and VectorLine.SetCamera3D has not been called, then SetCamera3D is called automatically (using Camera.main) the first time Draw is used.

Draw3D

```
function Draw3D () : void
```

Draws the VectorLine object in 3D space. The points used to create the VectorLine object must be Vector3. If VectorLine.SetCamera3D has not been called, then it’s called automatically (using Camera.main) the first time Draw is used. Since the lines exist in the scene and are not drawn in a separate overlay as they are with Draw, they must be updated when the camera used in SetCamera3D changes position or else they will no longer have the correct appearance. See also Draw3DAuto.

Draw3DAuto

```
function Draw3DAuto (time : float = Mathf.Infinity) : void
```

Draws the VectorLine object in the same way as Draw3D, but automatically updates the line every frame. Therefore, lines drawn with Draw3DAuto don't need to be updated manually and will always appear correct regardless of camera movement and so on.

The optional **time** is the number of seconds for which the line will be drawn, after which it's destroyed. The default is that the line is never destroyed.

GetColor

```
function GetColor (index : int) : Color32
```

Returns the color of the line segment specified by the **index**.

GetLength

```
function GetLength () : float
```

Returns the total length of all line segments that make up the VectorLine. The length refers to pixels when used with Vector2 lines, and world units when used with Vector3 lines. If the points that make up the line are changed after GetLength is called, then SetDistances should be called before using GetLength again in order to maintain accurate segment distances.

GetPoint

```
function GetPoint (distance : float) : Vector2
```

Returns a Vector2 in screen-space coordinates that corresponds to the given **distance** in the VectorLine. If **distance** is equal to or less than 0, then the first point in the VectorLine is returned. If the distance is equal to or greater than the total length of the line as defined by GetLength, then the last point in the VectorLine is returned. The first and last points are clamped by the VectorLine's drawStart and drawEnd values. If the points that make up the line are changed after GetPoint is called, then SetDistances should be called before using GetPoint again in order to maintain accurate segment distances.

```
function GetPoint (distance : float
                   out index : int) : Vector2
```

As above, but the out parameter **index** will contain the line segment index that corresponds to the distance.

GetPoint01

```
function GetPoint01 (distance : float) : Vector2
```

Same as GetPoint, except that **distance** is normalized between 0.0 and 1.0, where 0.0 is the first point in the VectorLine and 1.0 is the last point in the VectorLine.

```
function GetPoint01 (distance : float
                     out index : int) : Vector2
```

As above, but the out parameter **index** will contain the line segment index that corresponds to the distance.

GetPoint3D

```
function GetPoint3D (distance : float) : Vector3
```

Returns a Vector3 in world-unit coordinates that corresponds to the given **distance** in the VectorLine. If distance is equal to or less than 0, then the first point in the VectorLine is returned. If the distance is equal to or greater than the total length of the line as defined by GetLength, then the last point in the VectorLine is returned. The first and last points are clamped by the VectorLine's drawStart and drawEnd values. If the points that make up the line are changed after GetPoint3D is called, then SetDistances should be called before using GetPoint3D again in order to maintain accurate segment distances.

```
function GetPoint3D (distance : float
                     out index : int) : Vector2
```

As above, but the out parameter **index** will contain the line segment index that corresponds to the distance.

GetPoint3D01

```
function GetPoint3D01 (distance : float) : Vector3
```

Same as GetPoint3D, except that **distance** is normalized between 0.0 and 1.0, where 0.0 is the first point in the VectorLine and 1.0 is the last point in the VectorLine.

```
function GetPoint3D01 (distance : float
                       out index : int) : Vector2
```

As above, but the out parameter **index** will contain the line segment index that corresponds to the distance.

GetSegmentNumber

```
function GetSegmentNumber () : int
```

Returns the number of line segments that it would be possible to make with the given VectorLine. This is the number of points in VectorLine.points2 or VectorLine.points3 minus one for LineType.Continuous, and half the number of points for LineType.Discrete. For LineType.Points, it returns the total number of points.

GetWidth

```
function GetWidth (index : int) : float
```

Returns the width of the line segment specified by the **index**.

MakeArc

```
function MakeArc (origin : Vector2 or Vector3,
                  up : Vector3 = Vector3.forward,
                  xRadius : float,
                  yRadius : float,
                  startDegrees : float,
                  endDegrees : float,
                  segments : int = all,
                  index : int = 0) : void
```

Creates a section of an ellipse in VectorLine.points2 or VectorLine.points3. For Vector2, the supplied **origin** is in screen space pixels, as are the supplied radii. Vector3 uses world space coordinates. **xRadius** is the horizontal radius of the ellipse, and **yRadius** is the vertical radius. If both are the same, the arc will be made from a circle.

The optional **up** vector indicates the orientation of arcs when drawn using Vector3 points. It has no effect when using Vector2 points. The default up vector of Vector3.forward means that the arc is drawn in the X/Y plane. Using Vector3.up would mean that the ellipse would be drawn in the X/Z plane. The up vector can be any arbitrary vector and does not need to be normalized.

The **startDegrees** is the number of degrees around the ellipse that the arc will be started. The degrees go in a clockwise direction. The **endDegrees** is the number of degrees around the ellipse that the arc will finish. If the degrees are over 360 or below 0, they wrap around.

The optional **segments** indicates how many line segments will be used to create the arc, with a minimum of 3. This is normally used with the index parameter to create multiple arcs in the same VectorLine. If segments is omitted, then all of the points in VectorLine.points2 or VectorLine.points3 will be used.

The optional **index** is 0 by default, though it can be anything, as long as the number of segments would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple arcs in the same line, since the points used to create the arc start at the value defined by **index**. The count of VectorLine.points2 or VectorLine.points3 must be at least the number of **segments** specified plus one if the line was created as continuous line, or twice the number of **segments** if it was created as a discrete line. Note that specifying the index also requires that segments be specified.

MakeCircle

```
function MakeCircle (origin : Vector2 or Vector3,
                     up : Vector3 = Vector3.forward,
                     radius : float,
                     segments : int = all,
                     pointRotation : float = 0.0,
                     index : int = 0) : void
```

Creates a circle in VectorLine.points2 or VectorLine.points3. If using Vector2, the supplied **origin** is in screen space pixels, as is the supplied **radius**. If using Vector3, the coordinates are world space.

The optional **up** vector indicates the orientation of circles when drawn using Vector3 points. It has no effect when using Vector2 points. The default up vector of Vector3.forward means that the circle is drawn in the X/Y plane. Using Vector3.up would mean that the circle would be drawn in the X/Z plane. The up vector can be any arbitrary vector and does not need to be normalized.

The optional **segments** indicates how many line segments will be used to create the circle, with a minimum of 3. This is normally used with the index parameter to create multiple circles in the same VectorLine. If segments is omitted, then all of the segments in the VectorLine will be used.

The optional **pointRotation** describes how many degrees clockwise the points will be rotated around the origin. Negative values rotate the points counter-clockwise.

The optional **index** is 0 by default, though it can be anything, as long as the number of segments would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple circles in the same line, since the points used to create the circle start at the value defined by **index**. The length of VectorLine.points2 or VectorLine.points3 used for **line** must be at least the number of **segments** specified plus one if the line was created as continuous line, or twice the number of **segments** if it was created as a discrete line. Note that specifying the index also requires that segments be specified.

MakeCube

```
function MakeCube (origin : Vector3,
                   xSize : float,
                   ySize : float,
                   zSize : float,
                   index : int = 0) : void
```

Creates a cube in VectorLine.points3 (which must have a count of at least 24). The VectorLine must be LineType.Discrete. The **origin** is the world-space coordinate for the center of the cube. The **xSize**, **ySize**, and **zSize** are floats indicating how many units in size the cube is for those dimensions. The optional **index** is 0 by default, though it can be anything, as long as the count of VectorLine.points3 is at least 24 plus the index.

MakeCurve

```
function MakeCurve (curvePoints : Vector2[] or Vector3[],
                    segments : int,
                    index : int = 0) : void
```

Creates a bezier curve in VectorLine.points2 or VectorLine.points3. The supplied **curvePoints** is a Vector2 or Vector3 array that must contain four elements, where the elements are Vector2s using screen space pixel coordinates or Vector3s using world space coordinates, and are defined as follows:

curvePoints[0] = the first anchor point of the curve
curvePoints[1] = the first control point of the curve
curvePoints[2] = the second anchor point of the curve
curvePoints[3] = the second control point of the curve

The supplied **segments** indicates how many line segments will be used to create the curve, with a minimum of 2. The optional **index** is 0 by default, though it can be anything, as long as the number of segments would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple curves in the same line, since the points used to create the curve start at the value defined by **index**. The count of VectorLine.points2 or VectorLine.points3 must be at least the number of **segments** specified plus one if the line was created as continuous line, or twice the number of **segments** if it was created as a discrete line.

Example: a 2D curve with 10 segments would require that VectorLine.points2 contains 11 points, as long as the line is continuous. If it's a discrete line, VectorLine.points2 would need to contain 20 points. Two curves of 10 segments each in the same line would require double the number of points, or 22 and 40 respectively. The index for the second curve would be 11 for a continuous line or 20 for a discrete line.

```
function MakeCurve (anchor1 : Vector2 or Vector3,
                    control1 : Vector2 or Vector3,
                    anchor2 : Vector2 or Vector3,
                    control2 : Vector2 or Vector3,
                    segments : int,
                    index : int = 0) : void
```

The anchor and control points for the curve are defined as individual Vector2s or Vector3s rather than a Vector2[] or Vector3[] array, but otherwise this is the same as above. The Vector2s use screen space pixel coordinates, and Vector3s use world space coordinates.

```
function MakeCurve (curvePoints : Vector2[] or Vector3[]) : void
```

As above, but since the number of segments isn't specified, the maximum allowed by the total count in VectorLine.points2 or VectorLine.points3 will be used.

MakeEllipse

```
function MakeEllipse (origin : Vector2 or Vector3,
                      up : Vector3 = Vector3.forward,
                      xRadius : float,
                      yRadius : float,
                      segments : int = all,
                      pointRotation : float = 0.0,
                      index : int = 0) : void
```

Creates an ellipse in VectorLine.points2 or VectorLine.points3. For Vector2, the supplied **origin** is in screen space pixels, as are the supplied radii. Vector3 uses world space coordinates. **xRadius** is the horizontal radius of the ellipse, and **yRadius** is the vertical radius.

The optional **up** vector indicates the orientation of ellipses when drawn using Vector3 points. It has no effect when using Vector2 points. The default up vector of Vector3.forward means that the ellipse is drawn in the X/Y plane. Using Vector3.up would mean that the ellipse would be drawn in the X/Z plane. The up vector can be any arbitrary vector and does not need to be normalized.

The optional **segments** indicates how many line segments will be used to create the ellipse, with a minimum of 3. This is normally used with the index parameter to create multiple ellipses in the same VectorLine. If segments is omitted, then all of the points in VectorLine.points2 or VectorLine.points3 will be used.

The optional **pointRotation** describes how many degrees clockwise the points will be rotated around the origin. Negative values rotate the points counter-clockwise.

The optional **index** is 0 by default, though it can be anything, as long as the number of segments would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple ellipses in the same line, since the points used to create the ellipse start at the value defined by **index**. The count of VectorLine.points2 or VectorLine.points3 must be at least the number of **segments** specified plus one if the line was created as continuous line, or twice the number of **segments** if it was created as a discrete line. Note that specifying the index also requires that segments be specified.

MakeRect

```
function MakeRect (rect : Rect,
                   index : int = 0) : void
```

Creates a rectangle in VectorLine.points2 or VectorLine.points3. The supplied **rect** is in screen space pixels if using Vector2, or world space coordinates if using Vector3.

The optional **index** is 0 by default, though it can be anything, as long as the rect would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple rectangles in the same line, since the points used to create the rectangle start at the value defined by **index**. The length of VectorLine.points2 or VectorLine.points3 must be at least 5 if the line was created as continuous line, or 8 if it was created as a discrete line.

```
function MakeRect (bottomLeft : Vector2 or Vector3,
                   topRight : Vector2 or Vector3,
                   index : int = 0) : void
```

As above, but the supplied **bottomLeft** and **topRight** describe the respective corners of the rectangle in screen space pixels if using Vector2, or world space coordinates if using Vector3.

MakeRoundedRect

```
function MakeRoundedRect (rect : Rect,
                          cornerRadius : float,
                          cornerSegments : int,
                          index : int = 0) : void
```

Creates a rectangle in VectorLine.points2 or VectorLine.points3. The supplied **rect** is in screen space pixels if using Vector2, or world space coordinates if using Vector3.

The **cornerRadius** is the radius for each corner, measured in the number of pixels (for Vector2 lines) or world units (for Vector3 lines).

The **cornerSegments** is how many line segments will be used for each corner. This must be at least 1.

The optional **index** is 0 by default, though it can be anything, and is normally used to create multiple rectangles in the same line, since the points used to create the rectangle start at the value defined by **index**. The number of segments required for a rounded rect is 4 * cornerSegments + 4 (to get the number of points for a continuous line, add 1, and for a discrete line, multiply by 2).

If the VectorLine doesn't contain enough points to hold the rounded rect, more will automatically be added as needed.

```
function MakeRoundedRect (bottomLeft : Vector2 or Vector3,
                          topRight : Vector2 or Vector3,
                          cornerRadius : float,
                          cornerSegments : int,
                          index : int = 0) : void
```

As above, but the supplied **bottomLeft** and **topRight** describe the respective corners of the rectangle in screen space pixels if using Vector2, or world space coordinates if using Vector3.

MakeSpline

```
function MakeSpline (splinePoints : Vector2[] or Vector3[],
                     segments : int,
                     index : int = 0,
                     loop : boolean = false) : void
```

Creates a Catmull-Rom spline in VectorLine.points2 or VectorLine.points3. The supplied **splinePoints** are a Vector2 (screen space) or Vector3 (world space) array defining the points that should be used to create the spline. The resulting curve will pass directly through all points in the splinePoints array. The supplied **segments** indicates how many line segments will be used to create the circle, with a minimum of 3. The optional **loop** indicates whether the first and last points in the splinePoints array will be connected or not.

The optional **index** is 0 by default, though it can be anything, as long as the number of segments would fit in VectorLine.points2 or VectorLine.points3. This allows creation of multiple splines in the same line, since the points used to create the spline start at the value defined by **index**. The length of VectorLine.points2 or VectorLine.points3 must be at least the number of **segments** specified plus one if the line was created as continuous line, or twice the number of **segments** if it was created as a discrete line.

```
function MakeSpline (splinePoints : Vector2[] or Vector3[],
                     loop : boolean = false) : void
```

As above, but since the number of segments isn't specified, all the points in VectorLine.points2 or VectorLine.points3 will be used.

MakeText

```
function MakeText (text : String,
                   position : Vector2 or Vector3,
                   size : float,
                   characterSpacing : float = 1.0,
                   lineSpacing : float = 1.5,
                   uppercaseOnly : boolean = true) : void
```

Creates the string **text** in the VectorLine. The text is placed at **position** using screen-space pixel coordinates if used with Vector2 points, or world space coordinates if used with Vector3 points. **size** is the number of pixels in height if used with Vector2 points, or world units if used with Vector3 points. Text is always monospaced. Any characters not present in the default font are ignored. “\n” can be used as a newline character. VectorLine.points2 or VectorLine.points3 will be resized as necessary to contain the number of points needed for the text.

The optional **characterSpacing** is 1.0 by default, which is a relative value, where 1.0 equals **size**, 0.5 would be half of **size**, etc. The optional **lineSpacing** is 1.5 by default, and is also a relative value in the same way. If supplying either the character spacing or the line spacing, both values must be supplied.

The optional **uppercaseOnly** is true by default, which makes the text always display using uppercase characters, even if **text** contains lowercase characters.

MakeWireframe

```
function MakeWireframe (mesh : Mesh) : void
```

Creates a wireframe in the VectorLine using all the triangles in **mesh**. The points used to create the VectorLine must be Vector3, and the line must be created using LineType.Discrete. VectorLine.points3 will be resized as necessary to contain the number of points needed for the wireframe.

Resize

```
function Resize (newCount : int) : void
```

Resizes the VectorLine.points2 or VectorLine.points3 count to the number specified by **newCount**. The size must be at least 0 and less than the maximum available for the line type (16383 for LineType.Continuous and LineType.Points, and 32767 for LineType.Discrete). Existing content of VectorLine.points2 or VectorLine.points3 is untouched. If newCount is larger than the existing count, the new points will be filled with Vector2.zero or Vector3.zero.

Selected

```
function Selected (position : Vector2,
                   cam : Camera = Camera.main) : boolean
```

Returns true if this VectorLine contains **position**, which is a screen-space coordinate, such as that provided by Input.mousePosition. (Vector3 is implicitly cast to Vector2, so Input.mousePosition will work directly even though it's technically a Vector3.) If the camera is not supplied, the first camera tagged “MainCamera” is used, if one exists.

```
function Selected (position : Vector2,
                   out index : int,
                   cam : Camera = Camera.main) : boolean
```

Returns true if this VectorLine contains **position**. The **index** is an integer variable declared elsewhere, which will contain the line segment index when passed in to the Selected function. If Selected returns false, then **index** will contain -1. If using LineType.Points, then **index** is the point that's currently selected rather than the line segment. If the camera is not supplied, the first camera tagged “MainCamera” is used, if one exists.

```
function Selected (position : Vector2,
                   extraWidth : int,
                   out index : int
                   cam : Camera = Camera.main) : boolean
```

Returns true if this VectorLine contains **position**. The **extraWidth** parameter is the number of pixels that a line's width is expanded by for the purposes of determining whether it contains **position**, so that the position doesn't need to be exact in order to register as being inside the line. The extra distance is for computation only and does not affect the visual appearance of the line. As above, the **index** is an integer variable which will contain the line segment index. If the camera is not supplied, the first camera tagged “MainCamera” is used, if one exists.

```
function Selected (position : Vector2,
                   extraWidth : int,
                   extraLength : int,
                   out index : int
                   cam : Camera = Camera.main) : boolean
```

Returns true if this VectorLine contains **position**. The **extraWidth** parameter is the number of pixels that a line's width is expanded by for the purposes of determining whether it contains **position**. The **extraLength** parameter is the number of pixels that each line segment is extended by for the purposes of determining whether the line contains **position**; the actual visible line segments remain the same. The **index** is an integer variable which will contain the line segment index. If the camera is not supplied, the first camera tagged “MainCamera” is used, if one exists.

SetCanvas

```
function SetCanvas (canvas : Canvas,
                    worldPositionStays : boolean = true) : void
```

Sets the line to the specified **canvas**. The canvas must use RenderMode.ScreenSpaceOverlay or RenderMode. ScreenSpaceCamera. Attempting to use a canvas with RenderMode.WorldSpace will generate an error. Attempting to use SetCanvas with a line drawn with Draw3D will also generate an error.

The worldPositionStays boolean, if omitted, is true by default. This controls whether lines keep their world position when added to a canvas, which controls whether or not a canvas scaler component will affect the line's position or not.

```
function SetCanvas (canvasObject : GameObject,
                    worldPositionStays : boolean = true) : void
```

As above, but uses the Canvas component attached to the supplied **canvasObject**. Attempting to use a GameObject without a Canvas component will generate an error.

The worldPositionStays boolean, if omitted, is true by default. This controls whether lines keep their world position when added to a canvas, which controls whether or not a canvas scaler component will affect the line's position or not.

SetColor

```
function SetColor (color : Color32) : void
```

Sets all the line segment colors in the VectorLine to the supplied **color**. The line has its color changed immediately without having to call VectorLine.Draw or Draw3D.

```
function SetColor (color : Color32, index : int) : void
```

Sets the line segment specified by **index** to the supplied **color**. The other line segments are unaffected. The line has its color changed immediately without having to call VectorLine.Draw or Draw3D.

```
function SetColor (color : Color32, startIndex : int, endIndex : int) : void
```

Sets the line segments specified by **startIndex** up to and including **endIndex** to the supplied **color**. The other line segments outside of this range are unaffected. The line has its color changed immediately without having to call VectorLine.Draw or Draw3D.

SetColors

```
function SetColors (colors : List<Color32>) : void
```

Sets all the line segment colors in the VectorLine to the supplied **colors** list. The line has its colors changed immediately without having to call VectorLine.Draw or Draw3D. Each entry in the color list corresponds to a line segment, so the length of the color list must be half the count of VectorLine.points2 or VectorLine.points3 if using a discrete line, or the same as the points count minus one if using a continuous line. When using LineType.Points, the length of the list should equal the points count. Whether each line segment is a distinct color, or the colors are smoothly blended, is determined by VectorLine.smoothColor.

SetDistances

```
function SetDistances () : void
```

Used to recompute line segment distances in the VectorLine after changing any points in VectorLine.points2 or VectorLine.points3. If GetLength or any of the GetPoint functions have been used, and the line points are changed, those functions will no longer return accurate information unless SetDistances is called before using those functions again. SetDistances is automatically called when using GetLength or any of the GetPoint functions for the first time.

SetEndCapColor

```
function SetEndCapColor (color : Color) : void
```

Sets the end caps to the supplied **color**. The line must be using end caps, or an error is generated.

```
function SetEndCapColor (frontColor : Color, backColor : Color) : void
```

As above, but sets the front end cap to **frontColor** and the back end cap to **backColor**.

SetMask

```
function SetMask (mask : Mask) : void
```

Sets the line as a child of the specified **mask**. This will result in the VectorLine being masked by the texture of the mask component. If the line uses a custom material, the material must have a shader that uses the _Stencil property in order for masking to work. Attempting to use SetMask with a line drawn with Draw3D will generate an error.

```
function SetMask (maskObject : GameObject) : void
```

As above, but uses the Mask component attached to the supplied **maskObject**. Attempting to use a GameObject without a Mask component will generate an error.

SetWidth

```
function SetWidth (width : float) : void
```

Sets all the line segment widths in the VectorLine to the supplied **width**. VectorLine.Draw or Draw3D must be called afterwards in order for altered line segment to show up correctly.

```
function SetWidth (width : float, index : int) : void
```

Sets the line segment specified by **index** to the supplied **width**. The other line segments are unaffected. VectorLine.Draw or Draw3D must be called afterwards in order for altered line segment to show up correctly.

```
function SetWidth (width : float, startIndex : int, endIndex : int) : void
```

Sets the line segments specified by **startIndex** up to and including **endIndex** to the supplied **width**. The other line segments outside of this range are unaffected. VectorLine.Draw or Draw3D must be called afterwards in order for altered line segment to show up correctly.

SetWidths

```
function SetWidths (lineWidths : List<float> or List<int>) : void
```

Sets the pixel widths of the line segments in the VectorLine to the values supplied by the **lineWidths** list, which can be either a float list or an int list. Each entry in the line widths list corresponds to a line segment, so the length of the line widths list must be half the count of VectorLine.points2 or VectorLine.points3 if using a discrete line, or the same as the points count minus one if using a continuous line. When using LineType.Points, the length of the list should equal the points count. VectorLine.Draw or Draw3D must be called afterwards in order for the new widths to show up. Whether each line segment is a distinct width, or the widths are smoothly blended, is determined by VectorLine.smoothWidth.

StopDrawing3DAuto

```
function StopDrawing3DAuto () : void
```

Stops the automatic update of any line drawn with Draw3DAuto.

VectorLine static functions are not used with a VectorLine object, but are called explicitly with VectorLine. For example, “VectorLine.SetCamera3D()”.

BytesToVector2List

```
static function BytesToVector2List (lineBytes : byte[]) : List<Vector2>
```

Converts **lineBytes** to a Vector2 list, used for making complex line objects without having to use long strings of hard-coded Vector2 list data in scripts. Typically lineBytes would be supplied by a TextAsset file, specifically TextAsset.bytes. These TextAssets are made with the LineMaker editor script.

BytesToVector3List

```
static function BytesToVector3List (lineBytes : byte[]) : List<Vector3>
```

Converts **lineBytes** to a Vector3 list, used for making complex line objects without having to use long strings of hard-coded Vector3 list data in scripts. Typically lineBytes would be supplied by a TextAsset file, specifically TextAsset.bytes. These TextAssets are made with the LineMaker editor script.

canvas

```
static var canvas : Canvas;
```

A reference to the vector canvas used by VectorLine.Draw, so properties such as planeDistance, sortingOrder, etc. can be changed. By default it’s set to RenderMode.Overlay. See also SetCanvasCamera.

Lines created with VectorLine.Draw3D do not use a canvas.

Destroy

```
static function Destroy (ref line : VectorLine) : void
```

Removes a VectorLine and all associated Unity objects from the scene. If **line** is null, it's ignored and no null reference exception errors will occur.

```
static function Destroy (ref line : VectorLine,
                         gameObject : GameObject) : void
```

Removes a VectorLine and all associated Unity objects from the scene, and destroys **gameObject** at the same time. If **line** or **gameObject** are null, they are ignored and no null reference exception errors will occur.

```
static function Destroy (lines : VectorLine[]) : void
```

Removes all of the VectorLines in the supplied array.

```
static function Destroy (lines : List<VectorLine>) : void
```

Removes all of the VectorLines in the supplied generic List.

RemoveEndCap

```
static function RemoveEndCap (name : String) : void
```

Removes the end cap defined by **name** from the end cap library.

SetCamera3D

```
static function SetCamera3D (camera : Camera = Camera.main) : void
```

Sets the camera up for drawing lines made with Vector3 points. The optional **camera** is set to the first camera in the scene found tagged “MainCamera” by default. SetCamera3D is called automatically when using Draw or Draw3D for the first time, so it's normally not necessary to call it manually, unless using a camera other than “MainCamera” is desired. If the camera used with SetCamera3D is destroyed, such as by scene changes, SetCamera3D should be called again with the desired camera.

SetCanvasCamera

```
static function SetCanvasCamera (camera : Camera);
```

Sets the vector canvas to RenderMode.OverlayCamera, and the canvas worldCamera to **camera**. By contrast, the default RenderMode for the canvas is RenderMode.Overlay. SetCanvasCamera should be used for VectorLines that use a shader which requires lighting, and it can be used with cameras that make use of a RenderTexture, in order to render the vector canvas to a texture.

```
static function SetCanvasCamera (camera : Camera, canvasID : int);
```

As above, but the **canvasID** refers to an entry in the VectorLine.canvases list.

SetEndCap

```
static function SetEndCap (name : String,
                           capType : EndCap,
                           offsetFront : float = 0.0,
                           offsetBack : float = 0.0,
                           scaleFront : float = 1.0,
                           scaleBack : float = 1.0,
                           lineTexture : Texture2D,
                           capTexture1 : Texture2D,
                           capTexture2 : Texture2D) : void
```

Creates a named end cap that can be used with VectorLine.endCap. There can be any number of different end caps that exist in the end cap library, as long as each one uses a unique name. Once an end cap is set up, it can be used by any VectorLine in any script, so SetEndCap should only be called once for each end cap. (The end cap library exists only at runtime, not in the project.)

The **name** is case-sensitive.

The **capType** can be EndCap.Front, EndCap.Back, EndCap.Both, EndCap.Mirror, or EndCap.None.

EndCap.Front: only one cap texture needs to be supplied (**capTexture1**), and it will appear at the front of the line, as defined by the first point in the list of points that makes up the VectorLine.

EndCap.Back: only one cap texture needs to be supplied (**capTexture1**), and it will appear at the back of the line, as defined by the last point in the list of points that makes up the VectorLine.

EndCap.Both: two cap textures need to be supplied (**capTexture1** and **capTexture2**), the first of which will appear at the front of the line, and the second of which will appear at the back of the line.

EndCap.Mirror: only one cap texture needs to be supplied (**capTexture1**), which will appear at both the front and back of the line, and the texture at the back will be a mirror image of the front.

EndCap.None: the named end cap will be removed from the end cap library.

The **offsetFront** value is a percentage of the end cap’s length. By default (when the offset is 0.0), end caps are added to the ends of the line. If the offset is set to -1.0, for example, the end cap will be moved inward by its own length, so the line will be as long as it would have been without the end cap. If the offsetBack value is omitted, then offsetFront applies to both the front and the back caps.

The **offsetBack** value works the same as offsetFront, but it applies to the back end cap.

The **scaleFront** value is a percentage of the end cap’s size. By default (when the scale is 1.0), end caps are the same size as the line they are attached to. If the scale is set to 2.0, for example, the end cap would be double the default size. If the scale values are supplied, the offset values must also be supplied.

The **scaleBack** value works the same as scaleFront, but it applies to the back end cap. If scaleFront is supplied, scaleBack must also be supplied.

The **lineTexture** is the texture that will be applied to the line. This replaces any existing texture that might have been supplied with VectorLine.texture. It must be square (same width and height). The cap textures must be the same height as the line texture, but can be of arbitrary width.

SetLine

```
static function SetLine (color : Color,
                         time : float = Mathf.Infinity,
                         params points : Vector2[] or Vector3[]) : VectorLine
```

Creates a 1-pixel thick VectorLine using the supplied **points**, and draws it immediately using the supplied **color**. The points use screen-space coordinates if made with Vector2s, or world coordinates if made with Vector3s. “Params” means that each point is supplied individually, rather than as an array. At least two Vector2s are required; this will create a single line segment. Each additional Vector2 will extend the line to that point by adding another line segment from the last. SetLine returns a VectorLine object, so it can be assigned to a variable and used in any function that takes a VectorLine.

The optional **time** parameter specifies how long, in seconds, the line will be drawn before it’s destroyed. If this is left out, the line will never be removed.

If Vector3 points are used and VectorLine.SetCamera3D has not been called, it’s called automatically the first time SetLine is used, using the default parameters.

SetLine3D

```
static function SetLine3D (color : Color,
                           time : float = Mathf.Infinity,
                           params points : Vector3[]) : VectorLine
```

Creates a 1-pixel thick VectorLine using the supplied **points**, and draws it immediately using the supplied **color**. The points use world-space coordinates. “Params” means that each point is supplied individually, rather than as an array. At least two Vector3s are required; this will create a single line segment. Each additional Vector3 will extend the line to that point by adding another line segment from the last. SetLine3D returns a VectorLine object, so it can be assigned to a variable and used in any function that takes a VectorLine.

The lines created by SetLine3D are drawn in world space, rather than on top of other objects.

The optional **time** parameter specifies how long, in seconds, the line will be drawn before it’s destroyed. If this is left out, the line will never be removed.

If VectorLine.SetCamera3D has not been called, it’s called automatically the first time SetLine3D is used, using the default parameters.

SetRay

```
static function SetRay (color : Color,
                        time : float = Mathf.Infinity,
                        origin : Vector3,
                        direction : Vector3) : VectorLine
```

Creates a 1-pixel thick VectorLine from **origin** to **origin** + **direction**, and draws it immediately using the supplied **color**. SetRay returns a VectorLine object, so it can be assigned to a variable and used in any function that takes a VectorLine.

The optional **time** parameter specifies how long, in seconds, the line will be drawn before it's destroyed. By default, the line will never be removed.

If VectorLine.SetCamera3D has not been called, it's called automatically the first time SetRay is used, using the default parameters.

SetRay3D

```
static function SetRay3D (color : Color,
                          time : float = Mathf.Infinity,
                          origin : Vector3,
                          direction : Vector3) : VectorLine
```

Creates a 1-pixel thick VectorLine from **origin** to origin + **direction**, and draws it immediately using the supplied **color**. The line is drawn in world space, rather than on top of other objects. SetRay3D returns a VectorLine object, so it can be assigned to a variable and used in any function that takes a VectorLine.

The optional **time** parameter specifies how long, in seconds, the line will be drawn before it's destroyed. If this is left out, the line will never be removed.

If VectorLine.SetCamera3D has not been called, it's called automatically the first time SetRay is used, using the default parameters.

Version

```
static function Version () : String
```

Returns a string containing version information.

See the Vector Manager section in the Vectrosity5 Coding document for more details of how VectorManager functions are used.

GetBrightnessValue

```
static function GetBrightnessValue (position : Vector3) : float
```

Given the distance of **position** from the non-vector camera used in VectorLine.SetCamera, returns a float between 0.0 and 1.0, where 0.0 is 0% brightness and 1.0 is 100% brightness.

ObjectSetup

```
static function ObjectSetup (gameObject : GameObject,
                             vectorLine : VectorLine,
                             visibility : Visibility,
                             brightness : Brightness,
                             makeBounds : boolean = true) : void
```

Makes **gameObject** have a “shadow” 3D vector object as defined by **vectorLine**, which behaves according to the transform of **gameObject**. Depending on the values of **visibility** and **brightness**, one or more components may be attached to **gameObject** when this function is called.

visibility can be Visibility.Dynamic, Visibility.Static, Visibility.Always, or Visibility.None. Visibility.Dynamic causes the vector line to always be drawn when **gameObject** is visible to a camera, using the object’s transform. Visibility.Static only draws the vector line if the camera moves and the **gameObject** is visible to a camera, and is for objects that never move, since the object’s transform is only used once when the 3D vector line is initialized. Visibility.Always causes the vector line to be drawn every frame and has no optimizations. Visibility.None doesn’t add any visibility components, and removes any Visibility scripts if ObjectSetup has been called on this **gameObject** before.

brightness can be Brightness.Fog or Brightness.Normal. Brightness.Fog adds the BrightnessControl component, which makes the 3D vector object’s color behave according to the parameters given in VectorManager.SetBrightnessParameters. If SetBrightnessParameters hasn’t been called, the defaults for that function will be used. Currently only the first color in the array is used for the entire object. Brightness.None doesn’t add any component, so the vector line’s color won’t be altered, and removes BrightnessControl if ObjectSetup has been called on this **gameObject** before.

The optional **makeBounds** by default creates an invisible bounds mesh for the **gameObject**’s mesh filter, so that OnBecameInvisible and OnBecameInvisible will still work, which allows optimizations for Visibility.Dynamic and Visibility.Static. If set to false, then the **gameObject**’s mesh is not replaced by a bounds mesh. One bounds mesh is created for each VectorLine name, so VectorLines that share a name will also share a bounds mesh.

SetBrightnessParameters

```
static function SetBrightnessParameters (fadeOutDistance : float = 500,
                                         fullBrightDistance : float = 250,
                                         brightnessLevels : int = 32,
                                         frequency : float = 0.2,
                                         color : Color = Color.black) : void
```

Sets parameters for objects that use Brightness.Fog with VectorManager.ObjectSetup. **fadeOutDistance** is the distance that the object must be from the camera to be completely faded out, or in other words have 100% “fog” color. **fullBrightDistance** is the distance that the object must be from the camera in order to have the maximum amount (100%) of brightness. The VectorLine’s color will always be derived from VectorLine.color even if other colors have been set with SetColor or SetColors.

The vector line will have **brightnessLevels** “steps” between 0% and 100% brightness. The fewer steps, the more obvious the transitions become as the object moves closer and farther from the camera.

How often the distance of objects is checked depends on **frequency**, which by default is 5 times per second.

The **color** defines the “fog” color.

useDraw3D

```
var useDraw3D : bool = false
```

Tells the VectorManager routines to use VectorLine.Draw3D if VectorManager.useDraw3D is set to true, or VectorLine.Draw if set to false, which is the default. See VectorLine.Draw3D for more details about 3D lines.