INDEX

VectorLine constructor and variables

VectorPoints

VectorLine functions

VectorLine static functions and variables

INDEX

VectorManager functions and variables

A note about parameter conventions

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 : Vector2[] or Vector3[],
            material : Material,
            lineWidth : float,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.None) : VectorLine
```

Constructs a VectorLine object with the supplied parameters. The 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 mesh created for the vector line. It's also used in the name of the GameObject that this constructor generates, where the complete name is "Vector " plus the supplied name. The name is also used to identify bounds meshes made with VectorManager.ObjectSetup.

points is a Vector2 or Vector3 array, where each entry is a point in the line using screen-space coordinates (in the case of a Vector2 array), or world units (in the case of a Vector3 array).

The line is drawn using the material supplied by **material**. If different line depths are used, the material should use a shader that writes to the depth buffer for this to work reliably. The shader should use vertex colors in order for line segment colors to work. **Note:** if null is passed for the material, a default material is used. This material uses a basic shader which works with vertex colors and line depths, but has no texture.

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

lineType is either LineType.Discrete or LineType.Continuous. For discrete lines, each line segment is made from two entries in the Vector2 array. For continuous lines, the line starts at entry 0, and each segment is continuously connected to the next point until the end is reached.

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. This works with LineType.Discrete, but only for sequential line segments.

The default color for the line is Color.white. This can be overridden; see below.

```
VectorLine (name : String,
            points : Vector2[] or Vector3[],
            color : Color,
            material : Material,
            lineWidth : float,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.None) : VectorLine
```

As above, but the supplied **color** creates a Color array of the appropriate length for the **lineType** (half the length of the **points** array for Discrete, the length minus one for Continuous), filled with **color**. The line will be drawn with this color as long as the shader used in the material for the line works with vertex colors.

```
VectorLine (name : String,
            points : Vector2[] or Vector3[],
            colors : Color[],
            material : Material,
            lineWidth : float,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.None) : VectorLine
```

As above, but the supplied **colors** is a Color array where each entry describes a color for a line segment in the **points** array. This Color array must be half the length of the **points** array for LineType.Discrete (since each line segment is created from two points), and the length minus one for LineType.Continuous. In other words, one entry in the Color array for each line segment.

active

```
var active : boolean;
```

Sets a VectorLine active or inactive. Inactive VectorLines have their renderer disabled, and if an inactive VectorLine is used with functions like Draw, the function will return immediately and do nothing.

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. If the line has been drawn with a collider, setting this to false will disable the collider. Only works with 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; (read only)
```

Returns the color that was used to make the VectorLine. If a color array was used, it returns the first entry in that array. Use the VectorLine.SetColor function to change the color, or the VectorLine.SetColors function to change the color array.

continuous

```
var continuous : boolean; (read only)
```

True if the VectorLine was made with LineType.Continuous, false if it was made with LineType.Discrete.

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, then the texture is repeated once for every line segment (which is the default).

depth

```
var depth : int = 0;
```

Sets the line's depth to the supplied integer value. It will be clamped between 0 and 100. This is used to specify the order in which lines are drawn, primarily used to determine which lines are drawn on top in the case when they overlap. Lines with higher values are drawn on top of lines with lower values, as long as the lines are drawn with materials that use shaders with ZWrite On, so they write to the zbuffer. VectorLine.Draw must be called after setting the lineDepth value for the effect to be visible on-screen.

drawEnd

```
var drawEnd : int;
```

Specifies the index in the VectorLine's points array at which line drawing should end. Any later points will be erased when VectorLine.Draw is called. (Erased on-screen, not from the points array.) When used with a discrete line, drawEnd should be an odd number. If it's even, it will be incremented to the next highest integer. It's clamped between 0 and pointsArray.Length-1. This can be used with drawStart to specify a range of points to draw in a line without having to manually erase points in the array.

drawStart

```
var drawStart : int;
```

Specifies the index in the VectorLine's points array at which line drawing should start. Any earlier points will be erased when VectorLine.Draw is called. (Erased on-screen, not from the points array.) 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 pointsArray.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.

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. Also does nothing if the line is VectorPoints and either Joins.Fill or Joins.Weld are used.

layer

```
var layer : int = 31;
```

Sets the line object's layer to the supplied integer value. Layers are used with camera culling masks. Lines by default use layer 31, unless drawn with DrawLine3D, in which case they use layer 0 by default.

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 array to the supplied value, and reading lineWidth will return the value of the first entry in the array. VectorLine.Draw must be called after setting the lineWidth value for the effect to be visible on-screen.

material

```
var material : Material;
```

Returns the material that was used when the VectorLine was set up, and can be used to assign a different material at any time after the line has been created.

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.

maxDrawIndex

```
var maxDrawIndex : int;
```

When drawn, the line drawing routine will stop with this index in the Vector2 or Vector3 array. By default, this is the array's length-1. Used for updating part of the line instead of the whole thing, which can be used for optimization. If the line has been drawn previously, the segments after minDrawIndex are untouched, rather than being erased. Also affects updating of line segment colors with VectorLine.SetColors.

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.

mesh

```
var mesh : Mesh;
```

The mesh used for the actual GameObject that contains the VectorLine. Use this rather than VectorLine.VectorObject.GetComponent(MeshFilter).mesh.

minDrawIndex

```
var minDrawIndex : int;
```

When drawn, the line drawing routine will start with this index in the Vector2 or Vector3 array. This is 0 by default. Used for updating part of the line instead of the whole thing, which can be used for optimization. If the line has been drawn previously, the segments before minDrawIndex are untouched, rather than being erased. Also affects updating of line segment colors with VectorLine.SetColors.

name

```
var name : string;
```

Returns the name of the line that was supplied when the VectorLine was constructed. If the name is changed using VectorLine.name after the line has been constructed, then the VectorLine.vectorObject and mesh names will also change appropriately.

physicsMaterial

```
var physicsMaterial : PhysicsMaterial2D;
```

If a collider is used for the line, it will have the supplied PhysicsMaterial2D. Requires Unity 4.3 or later.

points2

```
var points2 : Vector2[];
```

This is a reference to the array of Vector2 points that was used when the line was set up. The entries in the array can be changed at any time, and then VectorLine.Draw used to draw the line with the updated points. If points2 is resized, an error will be generated when the line is drawn. If it's necessary to resize this array, use the VectorLine.Resize function. If the line was made with Vector3 points, then points2 is null.

points3

```
var points3 : Vector3[];
```

Similar to points2, except it's a reference to the array of Vector3 points that was used when the line was set up. If the line was made with Vector2 points, then points3 is null.

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.SetWidths.

sortingLayerID

```
var sortingLayerID : int = 0;
```

The line will be drawn with the specified sorting layer, if it exists. This can be used to control the drawing order between the line and other objects with a different sorting layer. Requires Unity 4.3 or later.

sortingOrder

```
var sortingOrder : int = 0;
```

The line will be drawn with the specified sorting order. This can be used to control the drawing order between the line and other objects within the same sorting order. Requires Unity 4.3 or later.

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.

useViewpointCoords

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

If true, the line 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 points only.

vectorLayer

```
static var vectorLayer : int = 31;
```

Sets the default layer that VectorLines will be drawn in, and which the vector camera will see. Should be set before using VectorLine.SetCamera for it to have any effect. Note that this is a static variable and applies to all lines. Therefore it should be used like "VectorLine.vectorLayer = 20;".

vectorLayer3D

```
static var vectorLayer3D : int = 0
```

Sets the default layer that lines drawn with DrawLine3D will be drawn in. Note that this is a static variable and applies to all lines. Therefore it should be used like "VectorLine.vectorLayer3D = 1;".

vectorObject

```
var vectorObject : GameObject;
```

This is a reference to the actual GameObject that's used to hold the line mesh. It's possible to move lines around by using vectorObject.transform, but care should be used when doing this, since it's possible to break the way Vectrosity works if the transform is rotated or scaled inappropriately. In most cases it it's more appropriate to pass in a different transform using drawTransform instead of directly manipulating the vectorObject.

```
VectorPoints (name : String,
              points : Vector2[] or Vector3[],
              material : Material,
              width : float) : VectorPoints
```

Constructs a VectorPoints object for a Vector2 array with the supplied parameters. **name** is a string that's used to name the mesh created for the vector points. It's also used in the name of the GameObject that this constructor generates, where the complete name is "Vector " plus the supplied name.

points is a Vector2 or Vector3 array, where each entry is a point using screen-space coordinates (for Vector2 arrays) or world-space coordinates (for Vector3 arrays).

The points are drawn using the material supplied by **material**. If different depths are used, the material should use a shader that writes to the depth buffer for this to work reliably. The shader should use vertex colors for point colors to work. If null is passed for the material, a default material is used. This material uses a basic shader which works with vertex colors and line depths, but has no texture.

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

Most functions that accept a VectorLine object will also work with VectorPoints, such as Draw, Draw3D, MakeCurve, etc. Most of the variables for VectorLine (depth, minDrawIndex, etc.) work with VectorPoints, except for capLength, joins, maxWeldDistance, and smoothWidth.

```
VectorPoints (name : String,
              points : Vector2[],
              color : Color,
              material : Material,
              width : float) : VectorPoints
```

As above, but the supplied **color** creates a Color array of the appropriate length, filled with **color**. The points will be drawn with this color as long as the shader used in the material for the line works with vertex colors.

```
VectorPoints (name : String,
              points : Vector2[],
              color : Color[],
              material : Material,
              width : float) : VectorPoints
```

As above, but the supplied **color** is a Color array where each entry describes a color for the corresponding point in the **points** array. All points will therefore be drawn with the respective color, as long as the shader used in the material for the line works with vertex colors.

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

AddNormals

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

Creates mesh normals for a VectorLine. This should be called when a material that has a shader which requires normals is used for the VectorLine.

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.

Draw

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

Draws the VectorLine or VectorPoints object on the screen. If VectorLine.SetCamera has not been called, it’s called the first time Draw is used, using the default parameters. Lines with either Vector2 or Vector3 arrays can be used.

Draw3D

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

Draws the VectorLine object on the screen in 3D space (in contrast to Draw, which draws all lines in 2D with a separate camera overlaid on top of the standard camera). The array of points used to create the VectorLine object must be of type Vector3[]. If VectorLine.SetCamera3D has not been called, it’s called the first time DrawLine is used, using the default parameters. 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 SetCamera changes position or else they will no longer have the correct appearance.

Draw3D doesn’t use the vector camera object as created by SetCamera. Lines drawn with Draw3D are put on layer 0 by default. This default layer can be changed by setting VectorLine.vectorLayer3D.

Draw3DAuto

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

Draws the VectorLine object as DrawLine3D does, 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) : Color
```

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 array 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 array are changed after GetPoint is called, then SetDistances should be called before using GetPoint again in order to maintain accurate segment distances.

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.

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 array are changed after GetPoint3D is called, then SetDistances should be called before using GetPoint3D again in order to maintain accurate segment distances.

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.

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 the Vector2 or Vector3 array minus one for continuous lines, and half the number of points for discrete lines.

GetWidth

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

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

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 the Vector2 or Vector3 array for the VectorLine. 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 arrays. It has no effect when using a Vector2 array. The default 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. If segments is omitted, then all of the segments in the VectorLine will be used. This is normally used with the index parameter to create multiple circles in the same VectorLine.

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 the Vector2 or Vector3 array. 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 the Vector2 or Vector3 array 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 the Vector3 array (which must have a length of at least 24) for the VectorLine, which must use a discrete line. 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 length of the array 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 the Vector2 or Vector3 array for the VectorLine. 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 the **line** array. 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 length of the array 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.

Example: a curve with 10 segments would require an array in **line** that contains 11 points, as long as **line** is a continuous line. If it was a discrete line, the array 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 entire Vector2 or Vector3 array 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 the Vector2 or Vector3 array for the VectorLine. 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 arrays. It has no effect when using a Vector2 array. The default 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. If segments is omitted, then all of the segments in the VectorLine will be used. This is normally used with the index parameter to create multiple ellipses in the same VectorLine.

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 the Vector2 or Vector3 array. 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 length of the Vector2 or Vector3 array 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.

MakeRect

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

Creates a rectangle in the Vector2 or Vector3 array for the VectorLine. 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 the Vector2 or Vector3 array. 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 the Vector2 or Vector3 array used for **line** 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 (topLeft : Vector2 or Vector3,
                   bottomRight : Vector2 or Vector3,
                   index : int = 0) : void
```

Creates a rectangle in the Vector2 or Vector3 array for the VectorLine. The supplied **topLeft** and **bottomRight** describe the respective corners of the rectangle in screen space pixels if using Vector2, or world space coordinates if using Vector3. Otherwise this is the same as MakeRectInLine using a Rect.

MakeSpline

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

Creates a Catmull-Rom spline in the Vector2 or Vector3 array for the VectorLine. 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. **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 the Vector2 or Vector3 array. 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 the Vector2 or Vector3 array 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.

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

As above, but since the number of segments isn't specified, the maximum allowed by the entire Vector2 or Vector3 array 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 VectorLine **line**. 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. If the points array for **line** is not large enough to contain the line segments that make up the text, it will be resized appropriately. If this happens, any variables referencing the original points array will no longer reference the new points array. If a new reference is needed, it should be assigned to line.points2 if the line was made with Vector2 points, or line.points3 if the line was made with Vector3 points.

The optional **characterSpacing** is 1.0 by default, which is a relative value, where 1.0 equals **size**, .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 array used to create **line** must be of type Vector3, and the line must be created using LineType.Discrete. If the points array for **line** is not large enough to contain the line segments that make up the wireframe, it will be resized appropriately.

ResetTextureScale

```
function ResetTextureScale () : void
```

Sets all UVs in the mesh object for the VectorLine to their initial values. Used after VectorLine.SetTextureScale has been called, in case a return to the default state is desired.

Resize

```
function Resize (linePoints : Vector2[]) : void
```

Resizes the Vector2 points array in this VectorLine to that supplied by **linePoints**. VectorLine.Draw must be called afterwards for the new line to show up. If line segment widths have been supplied with VectorLine.SetWidths, they must be reset, and the same applies to VectorLine.SetColors (if a color array has been used for the line, all entries will be initially set to whatever the first color in the color array was).

```
function Resize (linePoints : Vector3[]) : void
```

As above, but for lines using a Vector3 array.

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

Resizes the number of points for the Vector2 or Vector3 array in this VectorLine to the number supplied by **newsize**. The points in the newly sized array are all set to zero, and the reference to the original Vector2 or Vector3 array used for making this VectorLine will no longer be valid, so VectorLine.points2 or VectorLine.points3 should be used instead. In other words, here's a line set to 4 points first, and resized to 2 points:

```
var linePoints = [Vector2(10, 10), Vector2(50, 50), Vector2(100, 100), Vector2(200, 200)];
var line = new VectorLine("Example", linePoints, null, 2.0);
line.Resize(2);
linePoints = line.points2;
linePoints[0] = Vector2(20, 20);
linePoints[1] = Vector2(150, 150);
line.Draw();
```

Selected

```
function Selected (position : Vector2) : 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.)

```
function Selected (position : Vector2,
                   out index : int) : 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 VectorPoints instead of VectorLine, then **index** is the point that's currently selected rather than the line segment.

```
function Selected (position : Vector2,
                   extraDistance : int,
                   out index : int) : boolean
```

Returns true if this VectorLine contains **position**. The **extraDistance** parameter is the number of pixels that a line 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.

SetColor

```
function SetColor (color : Color) : 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.

```
function SetColor (color : Color, 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.

```
function SetColor (color : Color, 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.

SetColors

```
function SetColors (colors : Color[]) : void
```

Sets all the line segment colors in the VectorLine to the supplied **colors** array. The line has its colors changed immediately without having to call VectorLine.Draw. Each entry in the color array corresponds to a line segment, so the length of the color array must be half the length of the Vector2 or Vector3 array in the VectorLine if using a discrete line, or the length of the Vector2 or Vector3 array minus one if using a continuous line.

SetColorsSmooth

```
function SetColorsSmooth (colors : Color[]) : void
```

Sets all the line segment colors in the VectorLine to the supplied color array, as with SetColors. Additionally, vertex colors are smoothly blended. The line has its colors changed immediately without having to call VectorLine.Draw.

SetDistances

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

Used to recompute line segment distances in the VectorLine after changing any of the Vector2 or Vector3 points that make up the line. 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. It's not necessary to call SetDistances before using GetLength or any of the GetPoint functions for the first time.

SetWidth

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

Sets the pixel width of the line segment specified by **index** to the specified **width.** VectorLine.Draw must be called afterwards in order for altered line segment to show up correctly.

SetWidths

```
function SetWidths (lineWidths : float[] or int[]) : void
```

Sets the pixel widths of the line segments in the VectorLine to the values supplied by the **lineWidths** array, which can be either a float array or an int array. Each entry in the line widths array corresponds to a line segment, so the length of the line widths array must be half the length of the Vector2 or Vector3 array in **line** if using a discrete line, or the length of the Vector2 or Vector3 array minus one if using a continuous line. VectorLine.Draw 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.

ZeroPoints

```
function ZeroPoints (index : int = 0) : void
```

Sets points in the Vector2 or Vector3 array in the VectorLine to Vector2.zero or Vector3.zero respectively. By default it starts from index 0 and sets all points to zero, but an **index** value greater than 0 can be supplied, which zeroes out points starting from that index (must be less than the Vector2 or Vector3 array length).

```
function ZeroPoints (startIndex : int = 0, endIndex : int = 0) : void
```

As above, except it sets points to Vector2.zero or Vector3.zero in the range from **startIndex** up to (but not including) **endIndex**.

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

BytesToVector2Array

```
static function BytesToVector2Array (lineBytes : byte[]) : Vector2
```

Converts the bytes from TextAsset.bytes to a Vector2 array, used for making specific lines without having to use long strings of hard-coded Vector2 array data in scripts. These TextAssets are made with the LineMaker editor script.

BytesToVector3Array

```
static function BytesToVector3Array (lineBytes : byte[]) : Vector3
```

Converts the bytes from TextAsset.bytes to a Vector3 array, used for making specific lines without having to use long strings of hard-coded Vector3 array data in scripts. These TextAssets are made with the LineMaker editor script.

Destroy

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

Removes a VectorLine (or VectorPoints) and all associated Unity objects from the scene. If **line** is null, it’s ignored and no null reference exception errors are possible.

```
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 are possible.

```
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.

GetCamera

```
static function GetCamera () : Camera
```

Returns the camera used for line drawing, as made with SetCamera.

MakeLine

```
static function MakeLine (name : String,
                          points : Vector2[] or Vector3[]) : VectorLine
```

Creates a VectorLine from the parameters given in VectorLine.SetLineParameters, using the supplied **name** and Vector2 or Vector3 array in **points**. This can be used as a shortcut, instead of supplying all parameters when constructing a VectorLine. If SetLineParameters has not been called first, an error will be generated.

```
static function MakeLine (name : String,
                          points : Vector2[] or Vector3[],
                          color : Color) : VectorLine
```

Creates a VectorLine from the parameters given in VectorLine.SetLineParameters, using the supplied **name** and Vector2 or Vector3 array in **points**, and overrides the default color with the supplied **color**.

```
static function MakeLine (name : String,
                          points : Vector2[] or Vector3[],
                          colors : Color[]) : VectorLine
```

Creates a VectorLine from the parameters given in VectorLine.SetLineParameters, using the supplied **name** and Vector2 or Vector3 array in **points**, and overrides the default color with the supplied **colors** array.

RemoveEndCap

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

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

SetCamera

```
static function SetCamera (camera : Camera = Camera.main,
                           clearFlags : CameraClearFlags = CameraClearFlags.DepthOnly,
                           useOrtho : boolean = false) : Camera
```

Sets the camera up for line drawing. This generally has to be done once at startup, though it also needs to be called after any screen resolution changes, and may need to be called after level changes, if any non-vector cameras aren't manually set to ignore layer 31.

This function is used automatically with the default parameters the first time VectorLine.Draw, VectorLine.DrawViewport, VectorLine.SetLine, or VectorLine.SetRay is called. Therefore it only needs to be called manually if parameters other than the defaults are desired.

The optional non-vector **camera** is set to the first camera in the scene found tagged "Main Camera" by default. This camera is used for supplying the viewpoint for any 3D vector objects, and has its culling mask set to ignore layer 31, so lines aren't visible except to the vector camera. Setting VectorLine.vectorLayer to an integer, before calling SetCamera, will cause the line-drawing layer to be that supplied rather than 31.

The optional **clearFlags** for the vector camera is DepthOnly by default. Passing **clearFlags** will change this to the supplied value instead.

The optional **useOrtho** is false by default. Passing true will cause the vector camera to use orthographic mode, which may render lines slightly more accurately, but can potentially cause anomalies in 3D lines under certain circumstances.

SetCamera returns the vector camera that's created. This can be used if further modifications to the vector camera are desired.

SetCamera3D

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

Sets the camera up for drawing lines with VectorLine.Draw3D. The optional **camera** is set to the first camera in the scene found tagged "Main Camera" by default. In contrast to VectorLine.SetCamera, the culling mask for the supplied camera is not altered, and the vector camera normally used for drawing lines is not created. This function is called automatically when DrawLine3D is used for the first time, so it's normally not necessary to call it manually, unless using a camera other than "Main Camera" is desired.

SetCameraRenderTexture

```
static function SetCameraRenderTexture (renderTexture : RenderTexture,
                                        color : Color = Color.black,
                                        useOrtho : boolean = false) : Camera
```

Makes the vector camera render to the specified **renderTexture** instead of the main screen. The renderTexture will have the optionally supplied **color** as the background color (black by default). The camera used for the renderTexture will be orthographic if the optional **useOrtho** is true.

Passing in null as the renderTexture will set the vector camera back to normal. SetCameraRenderTexture calls SetCamera and returns the vector camera. Note: requires Unity Pro.

SetDepth

```
static function SetDepth (transform : Transform,
                          depth : int) : void
```

Sets the line depth of **transform** to the value supplied by **depth**. This can be used for special effects, where transforms (usually planes) are positioned in the space used by vector lines. The object must be drawn on the same layer as lines for it to be visible to the vector camera. The depth for VectorLines is set with VectorLine.depth.

SetEndCap

```
static function SetEndCap (name : String,
                           capType : EndCap,
                           material : Material = null,
                           offset : float = 0.0;
                           texture1 : Texture2D = null,
                           texture2 : Texture2D = null) : 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 texture needs to be supplied (**texture1**), and it will appear at the front of the line, as defined by the first point in the array of points that makes up the VectorLine.

EndCap.Back: only one texture needs to be supplied (**texture1**), and it will appear at the back of the line, as defined by the last point in the array of points that makes up the VectorLine.

EndCap.Both: two textures need to be supplied (**texture1** and **texture2**), 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 texture needs to be supplied (**texture1**), which will appear at both the front and back of the line, and the texture's appearance at the back will be a mirror image compared to the front.

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

The **material** should normally (but not necessarily) be the same material that will be used when creating a particular VectorLine, so that the end cap material will match the material used for the rest of the line. Note that lines with end caps will have two draw calls instead of one. Also, only one pair of end caps can be used for any given VectorLine, even if LineType.Discrete is used.

The **offset** 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 caps will be moved inward by their own length, so the line will be as long as it would have been without end caps.

SetLine

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

Creates a 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 VectorLine.SetCamera has not been called, it's called the first time SetLine is used, using the default parameters.

SetLine3D

```
static function SetLine3D (color : Color,
                           params points : Vector3[]) : VectorLine
```

Creates a 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, and are not seen by the vector camera, which is not required for 3D lines.

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

SetLineParameters

```
static function SetLineParameters (color : Color,
                                   material : Material,
                                   lineWidth : float,
                                   capLength : float,
                                   depth : int,
                                   lineType : LineType,
                                   joins : Joins) : void
```

Used to set up the default parameters for VectorLine.MakeLine. The color, material, line width, end cap length, depth, LineType, and Joins are then used whenever constructing a line with VectorLine.MakeLine. When this is called again, any further lines will then use the new defaults, but already-created lines will be unaffected.

SetRay

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

Creates a 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.SetCamera has not been called, it's called 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 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, and is not seen by the vector camera, which is not required for 3D lines. 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 the first time SetRay is used, using the default parameters.

SetVectorCamDepth

```
static function SetVectorCamDepth (depth : int) : void
```

Sets the depth of the vector cam used to show lines drawn with VectorLine.Draw. By default, when VectorLine.SetCamera is called, the depth is one greater than the depth of the standard camera, so that lines are drawn on top of the view. By setting **depth** to a different value, the depth of the vector camera can be changed, which is normally only done for special effects.

useMeshLines

```
static var useMeshLines : boolean = false;
```

Unity 4.0 or higher only. If true, VectorLines will be created using MeshTopology.Lines if they are 1 pixel thick. This must be called before any VectorLines are created; otherwise a warning is printed and MeshTopology.Lines will not be used.

useMeshPoints

```
static var useMeshPoints : boolean = false;
```

Unity 4.0 or higher only. If true, VectorPoints will be created using MeshTopology.Points if they are 1 pixel big. This must be called before any VectorLines or VectorPoints are created; otherwise a warning is printed and MeshTopology.Points will not be used.

Version

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

Returns a string containing version information.

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 and 1, where 0 is 0% brightness and 1 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 (minBrightnessDistance : float = 500,
                                         maxBrightnessDistance : float = 250,
                                         brightnessLevels : int = 32,
                                         frequency : float = .2,
                                         color : Color = Color.black) : void
```

Sets parameters for objects that use Brightness.Fog with VectorManager.ObjectSetup. **minBrightnessDistance** is the distance that the object must be from the non-vector camera in order to have the minimum amount of brightness, or in other words have 100% “fog” color. **maxBrightnessDistance** is the distance that the object must be from the non-vector camera in order to have the maximum amount (100%) of brightness. The color of this brightness is taken from entry 0 in the vector line’s segmentColors array. 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.