INDEX

VectorLine

VectorPoints

Vector functions

VectorManager variables and functions

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[],
            material : Material,
            lineWidth : float,
            capLength : float = 0.0,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : VectorLine
```

Constructs a VectorLine object for a Vector2 array with the supplied parameters. The parameters after **lineWidth** are optional and have the default values indicated. If one of the later optional parameters is supplied, then the earlier ones must be supplied too (i.e., if you supply lineType, then you must supply capLength and depth).

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.

points is a Vector2 array, where each entry is a point in the line using screen-space coordinates.

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 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.

The **capLength** is the number of pixels added to either end of the line. 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. This is also a float, like lineWidth.

Lines with a higher **depth** are drawn on top of lines with a lower depth. This is an integer clamped between 0 and 100. As noted above, depth requires a shader that uses ZWrite On.

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 either Joins.Fill or Joins.Open. Joins.None can be used instead of Joins.Open. Joins.Fill will fill in the gaps seen where thick lines join at an angle. This only works if using LineType.Continuous.

```
VectorLine (name : String,
            points : Vector2[],
            color : Color,
            material : Material,
            lineWidth : float,
            capLength : float = 0.0,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : 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**.

```
VectorLine (name : String,
            points : Vector2[],
            color : Color[],
            material : Material,
            lineWidth : float,
            capLength : float = 0.0,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : VectorLine
```

As above, but the supplied **color** 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.

```
VectorLine (name : String,
            points : Vector3[],
            material : Material,
            lineWidth : float,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : VectorLine
```

Constructs a VectorLine object for a Vector3 array with the supplied parameters. The parameters after **lineWidth** are optional and have the default values indicated. If one of the optional parameters is supplied, then the rest must be supplied too. Compared to using a Vector2 array to construct a VectorLine, there is no capLength parameter, since it's not used when drawing 3D vector lines.

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.

points is a Vector3 array, where each entry is a point in the line using world coordinates.

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 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.

Lines with a higher **depth** are drawn on top of lines with a lower depth. This is an integer clamped between 0 and 100. As noted above, depth requires a shader that uses ZWrite On.

lineType is either LineType.Discrete or LineType.Continuous. For discrete lines, each line segment is made from two entries in the Vector3 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 either Joins.Fill or Joins.Open. Joins.None can be used instead of Joins.Open. Joins.Fill will fill in the gaps seen where thick lines join at an angle. This only works if using LineType.Continuous.

```
VectorLine (name : String,
            points : Vector3[],
            color : Color,
            material : Material,
            lineWidth : float,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : 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**.

```
VectorLine (name : String,
            points : Vector3[],
            color : Color[],
            material : Material,
            lineWidth : float,
            depth : int = 0,
            lineType : LineType = LineType.Discrete,
            joins : Joins = Joins.Open) : VectorLine
```

As above, but the supplied **color** 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.

VectorLine.smoothWidth

```
var smoothWidth : boolean;
```

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 Vector.SetWidths.

VectorLine.Resize

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

Resizes the Vector2 points array in this VectorLine to that supplied by **linePoints**. Vector.DrawLine must be called afterwards for the new line to show up. If line segment widths have been supplied with Vector.SetWidths, they must be reset, and the same applies to Vector.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).

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

As above, but for lines using a Vector3 array.

```
static 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);
Vector.DrawLine(line);
```

```
VectorPoints (name : String,
              points : Vector2[],
              material : Material,
              width : float,
              depth : int = 0) : VectorPoints
```

Constructs a VectorPoints object for a Vector2 array with the supplied parameters. This is used when calling Vector.DrawPoints, rather than a VectorLine object.

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 array, where each entry is a point using screen-space coordinates.

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 the depth parameter, but has no texture.

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

Points with a higher **depth** are drawn on top of points (and lines) with a lower depth. This is an integer clamped between 0 and 100. As noted above, depth requires a shader that uses ZWrite On.

Most functions that accept a VectorLine object will also work with VectorPoints.

```
VectorPoints (name : String,
              points : Vector2[],
              color : Color,
              material : Material,
              width : float,
              depth : int = 0) : VectorPoints
```

As above, but the supplied **color** creates a Color array of the appropriate length, filled with **color**.

```
VectorPoints (name : String,
              points : Vector2[],
              color : Color[],
              material : Material,
              width : float,
              depth : int = 0) : 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.

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.

DestroyLine

```
static function DestroyLine (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 are possible.

DestroyObject

```
static function DestroyObject (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.

DrawLine

```
static function DrawLine (line : VectorLine) : void
```

Draws the VectorLine object **line** on the screen. If Vector.SetCamera has not been called, it's called the first time DrawLine is used, using the default parameters.

```
static function DrawLine (line : VectorLine,
                          transform : Transform) : void
```

Draws the VectorLine object **line** on the screen, with the matrix from **transform** applied. If Vector.SetCamera has not been called, it's called the first time DrawLine is used, using the default parameters.

DrawLine3D

```
static function DrawLine3D (line : VectorLine) : void
```

Draws the VectorLine object **line** on the screen in 3D space (in contrast to DrawLine, 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 Vector.SetCamera3D has not been called, it's called the first time DrawLine is used, using the default parameters.

DrawLine3D doesn't use the vector camera object as created by SetCamera. If SetCamera has been used instead of SetCamera3D, then Vector.SetLayer should be used to set the line to a layer visible by the standard camera.

```
static function DrawLine3D (line : VectorLine,
                            transform : Transform) : void
```

Draws the VectorLine object **line** on the screen in 3D space, with the matrix from **transform** applied. Otherwise the behavior is the same as DrawLine3D without the transform parameter.

DrawPoints

```
static function DrawPoints (points : VectorPoints) : void
```

Draws points rather than lines. The **points** must be created using VectorPoints rather than VectorLine.

```
static function DrawPoints (points : VectorPoints,
                            transform : Transform) : void
```

Draws points rather than lines, with the matrix from **transform** applied. Otherwise the behavior is the same as DrawPoints without the transform parameter.

MakeLine

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

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

Creates a VectorLine from the parameters given in Vector.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[],
                          color : Color) : VectorLine

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

Creates a VectorLine from the parameters given in Vector.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[],
                          color : Color[]) : VectorLine

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

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

MakeCircleInLine

```
static function MakeCircleInLine (line : VectorLine,
                                  origin : Vector2,
                                  radius : float,
                                  segments : int,
                                  pointRotation : float = 0.0,
                                  index : int = 0) : void

static function MakeCircleInLine (line : VectorLine,
                                  origin : Vector3,
                                  radius : float,
                                  segments : int,
                                  pointRotation : float = 0.0,
                                  index : int = 0) : void
```

Creates a circle in the Vector2 or Vector3 array for VectorLine **line**. 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 supplied **segments** indicates how many line segments will be used to create the circle, with a minimum of 3.

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.

MakeCurveInLine

```
static function MakeCurveInLine (line : VectorLine,
                                 curvePoints : Vector2[],
                                 segments : int,
                                 index : int = 0) : void

static function MakeCurveInLine (line : VectorLine,
                                 curvePoints : Vector3[],
                                 segments : int,
                                 index : int = 0) : void
```

Creates a bezier curve in the Vector2 or Vector3 array for VectorLine **line**. 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 Vector2 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 Vector2 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.

```
static function MakeCurveInLine (line : VectorLine,
                                 anchor1 : Vector2,
                                 control1 : Vector2,
                                 anchor2 : Vector2,
                                 control2 : Vector2,
                                 segments : int,
                                 index : int = 0) : void

static function MakeCurveInLine (line : VectorLine,
                                 anchor1 : Vector3,
                                 control1 : Vector3,
                                 anchor2 : Vector3,
                                 control2 : 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.

MakeEllipseInLine

```
static function MakeCircleInLine (line : VectorLine,
                                  origin : Vector2,
                                  xRadius : float,
                                  yRadius : float,
                                  segments : int,
                                  pointRotation : float = 0.0,
                                  index : int = 0) : void

static function MakeCircleInLine (line : VectorLine,
                                  origin : Vector3,
                                  xRadius : float,
                                  yRadius : float,
                                  segments : int,
                                  pointRotation : float = 0.0,
                                  index : int = 0) : void
```

Creates an ellipse in the Vector2 or Vector3 array for VectorLine **line**. 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 supplied **segments** indicates how many line segments will be used to create the ellipse, with a minimum of 3.

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.

MakeRectInLine

```
static function MakeRectInLine (line : VectorLine,
                                rect : Rect,
                                index : int = 0) : void
```

Creates a rectangle in the Vector2 or Vector3 array for VectorLine **line**. 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.

```
static function MakeRectInLine (line : VectorLine,
                                topLeft : Vector2,
                                topRight : Vector2,
                                index : int = 0) : void
```

Creates a rectangle in the Vector2 or Vector3 array for VectorLine **line**. The supplied **topLeft** and **topRight** 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.

MakeTextInLine

```
static function MakeTextInLine (line : VectorLine,
                                text : String,
                                position : Vector2,
                                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, and is **size** pixels in height. 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.

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. The default font contains no lowercase characters, but the character set can be edited in the VectorChar file in Standard Assets/VectorScripts.

```
static function MakeTextInLine (line : VectorLine,
                                text : String,
                                position : Vector3,
                                size : float,
                                characterSpacing : float = 1.0,
                                lineSpacing : float = 1.5,
                                uppercaseOnly : boolean = true) : void
```

As above, except **position** is a Vector3 in world space coordinates, and **size** is in world units. If the points array for **line** is resized to fit the line segments needed for the string, any new references should point to line.points3.

ResetTextureScale

```
static function ResetTextureScale (line : VectorLine) : void
```

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

SetCamera

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

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.

SetCamera is called automatically with the default parameters the first time Vector.DrawLine or Vector.SetLine 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 **Vector.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.

SetCamera3D

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

Sets the camera up for drawing lines with Vector.DrawLine3D. The optional **camera** is set to the first camera in the scene found tagged "Main Camera" by default. In contrast to Vector.SetCamera, the culling mask for the supplied camera is not altered, and the vector camera normally used for drawing lines is not created. Use this function if only DrawLine3D will be used (it saves a little bit of overhead by not having two cameras). If DrawLine will be used at any point, use SetCamera instead.

SetColor

```
static function SetColor (line : VectorLine,
                          color : Color) : void
```

Sets all the line segment colors in **line** to the supplied **color**. The line has its color changed immediately without having to call Vector.DrawLine.

SetColors

```
static function SetColors (line : VectorLine,
                           color : Color[]) : void
```

Sets all the line segment colors in **line** to the supplied **color** array. The line has its colors changed immediately without having to call Vector.DrawLine. 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 **line** if using a discrete line, or the length of the Vector2 or Vector3 array minus one if using a continuous line.

SetColorsSmooth

```
static function SetColorsSmooth (line : VectorLine,
                                 color : Color[]) : void
```

Sets all the line segment colors in **line** to the supplied color array. Additionally, vertex colors are smoothly blended. The line has its color changed immediately without having to call Vector.DrawLine.

SetDepth

```
static function SetDepth (line : VectorLine,
                          depth : int) : void
```

Sets the line depth of **line** to the value supplied by **depth**. This is an integer clamped between 0 and 100. 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. Vector.DrawLine must be called for the line to change depth on-screen.

```
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.

SetLayer

```
static function SetLayer (line : VectorLine,
                          layer : int) : void
```

Sets the layer for **line** to the value specified by **layer**. Used primarily for setting the layer of a line drawn with Vector.DrawLine3D, if Vector.SetCamera was used instead of Vector.SetCamera3D, so that the 3D line will be visible to the standard camera.

SetLine

```
static function SetLine (color : Color,
                         params points : Vector2[]) : VectorLine
```

Creates a VectorLine using the supplied **points**, and draws it immediately using the supplied **color**. The points use screen-space coordinates. “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.

If Vector.SetCamera has not been called, it’s called the first time SetLine is used, using the default parameters.

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

As above, but uses a series of Vector3s instead of Vector2s, and the points use world-space coordinates.

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 Vector.SetCamera3D has not been called, it’s called the first time SetLine 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 the Vector.MakeLine shortcut. The color, material, line width, end cap length, depth, LineType, and Joins are then used whenever constructing a line with Vector.MakeLine. When this is called again, any further lines will then use the new defaults, but already-created lines will be unaffected.

SetTextureScale

```
static function SetTextureScale (line : VectorLine,
                                 textureScale : float,
                                 offset : float = 0.0) : void
```

Changes the UV mapping for **line**, so that the texture, instead of being stretched to fill a line segment, is uniformly scaled such that its width in the direction of the line segment is **textureScale** times that of its height. Using 1.0 for square textures results in the width and height being the same. How many times the texture repeats along the line segment therefore depends on the length of the line segment. The optional **offset** value offsets the texture along the line a certain percentage of the scale, with 1.0 being 100%. This is the same visual effect as altering the material's mainTextureScale.x value, though the material in this case is not affected. Once Vector.SetTextureScale has been used, Vector.ResetTextureScale can be used to reset the line back to its original appearance.

SetVectorCamDepth

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

Sets the depth of the vector cam used to show lines drawn with Vector.DrawLines. By default, when Vector.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.

SetWidths

```
static function SetWidths (line : VectorLine,
                           lineWidths : float[]) : void
```

Sets the pixel widths of the line segments in **line** to the values supplied by the **lineWidths** float 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. Vector.DrawLine 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.

ZeroPointsInLine

```
static function ZeroPointsInLine (line : VectorLine,
                                  index : int = 0) : void
```

Sets points in the Vector2 or Vector3 array in **line** 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 (which must be less than the length of the Vector2 or Vector3 array or else an error is generated).

use3DLines

```
var use3DLines : bool = false
```

Tells the VectorManager routines to use Vector.DrawLine3D if VectorManager.use3DLines is set to true, or Vector.DrawLine if set to false, which is the default. See Vector.DrawLine3D for more details about 3D lines.

DestroyObject

```
static function DestroyObject (line : VectorLine,
                               gameObject : GameObject) : float
```

Properly disposes of **gameObject** that has a corresponding 3D VectorLine object created using VectorManager.ObjectSetup and **line**. Note that this is not the same as Vector.DestroyObject. This is called automatically in OnDestroy when VectorManager.ObjectSetup is used, so generally it's not something that needs to be called manually.

GetBrightnessValue

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

Given the distance of **position** from the non-vector camera used in Vector.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) : 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. The first two use OnBecameVisible and OnBecameInvisible on **gameObject**’s renderer as optimizations, so that the lines aren’t computed when the object isn’t visible. This depends on an enabled mesh renderer. The object’s mesh can be a normal mesh, or an invisible bounds mesh created by the BoundsMaker editor script, which consists only of vertices and no triangles, so it adds no draw calls.

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 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, so drawing must be handled by the user.

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. Brightness.Normal (or Brightness.None) doesn’t add any component, so the vector line’s color won’t be altered. In order for Brightness.Fog to work, the line must contain segment colors. Currently only the first color in the array is used for the entire object.

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.