Paths

Paths is a library written in pure C# for creating and editing waypoint systems in Unity. Paths uses the Catmull-Rom curve algorithm.

Supported versions of Unity

Paths supports any version of Unity that has support for the new UI Elements UI system.

Installation

All you need to do is import the unity-package into your project. Technically, the Paths library is just a Plugins/Numba/Paths folder with some scripts and other files.

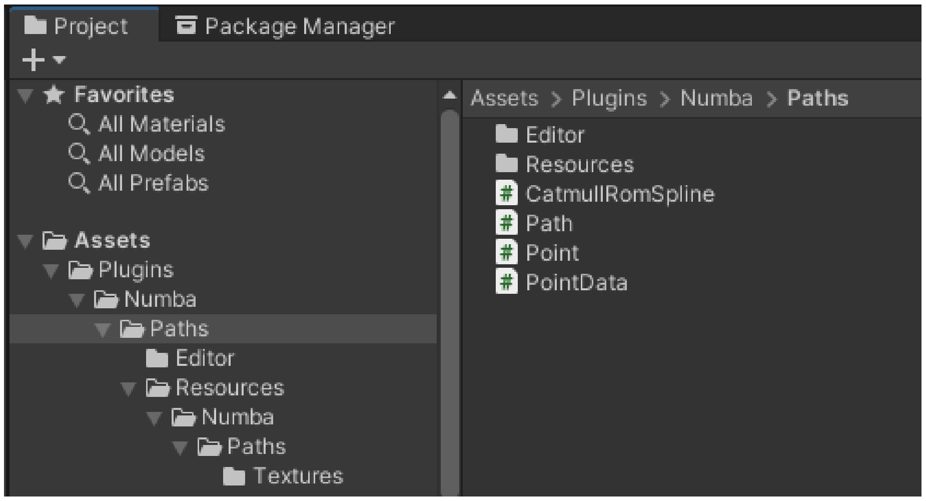


Figure 1: 2022-02-12 015235

Creating paths in the editor

To create a path in the editor in the Hierarchy tab, call the context menu and select `Path` > `Empty`.

This will create a blank object and add the `Path` script to it. Of course, you can create a blank and add the `Path` script through the components menu manually, but a ready-made menu is more convenient for this. In addition to the empty path you can create ready-made ones (line, triangle and others) but we'll come back to them a little later.

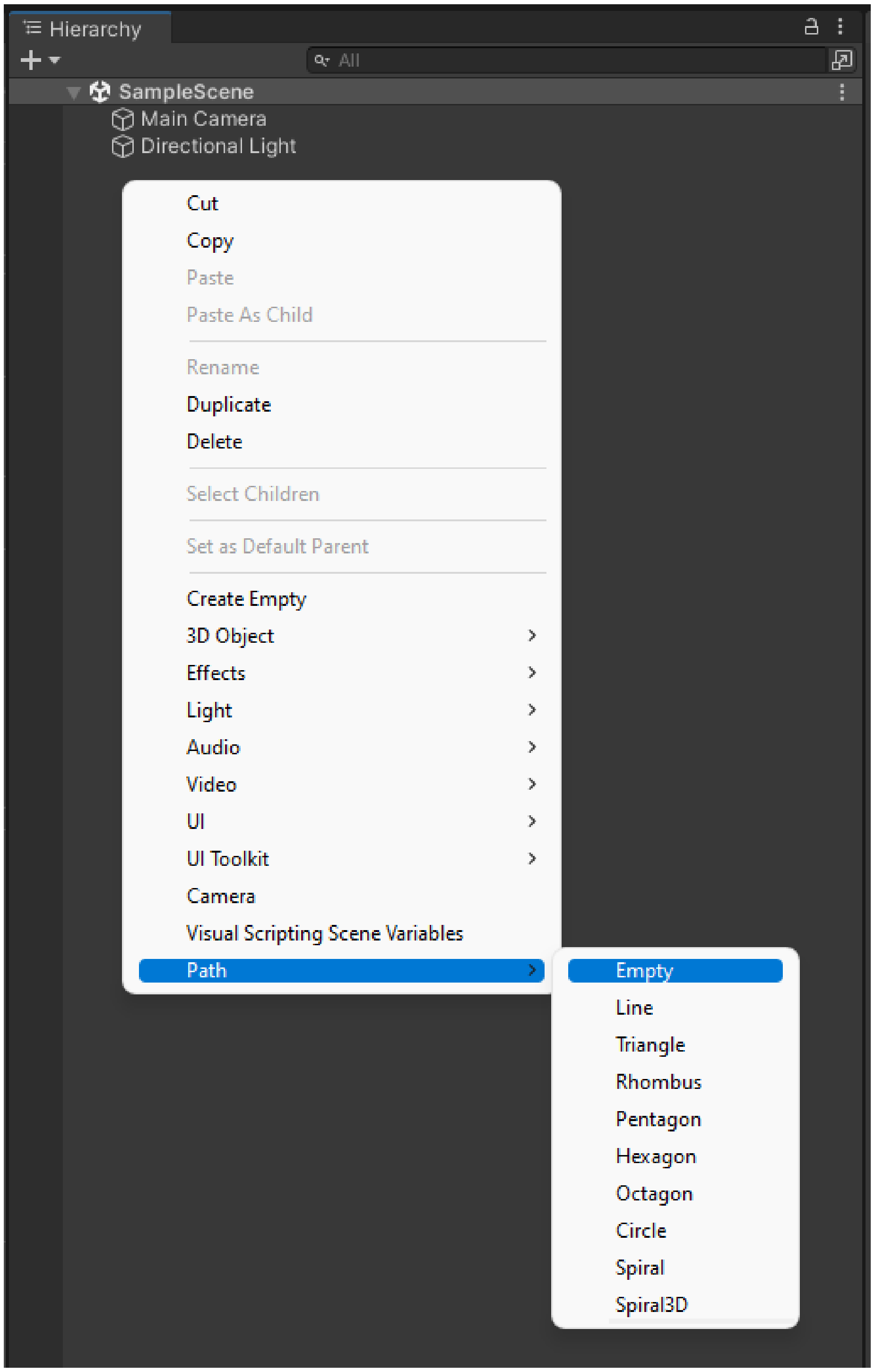
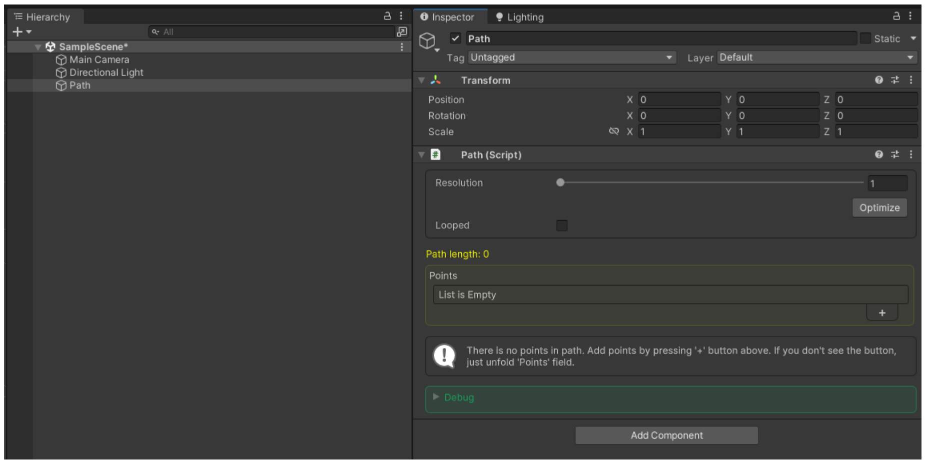


Figure 2: image

Setting the path in the editor



Path — Static
Tag: Untagged — Layer: Default

Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Resolution: 1
Looped: ☐

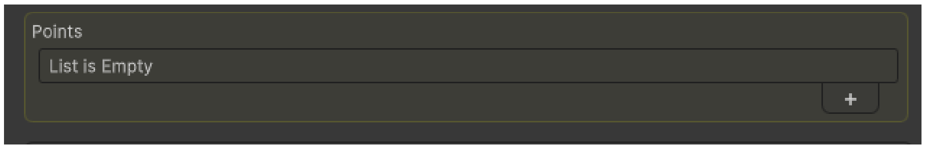
Path length: 0

Points
List is Empty

There is no points in path. Add points by pressing '+' button above. If you don't see the button, just unfold 'Points' field.

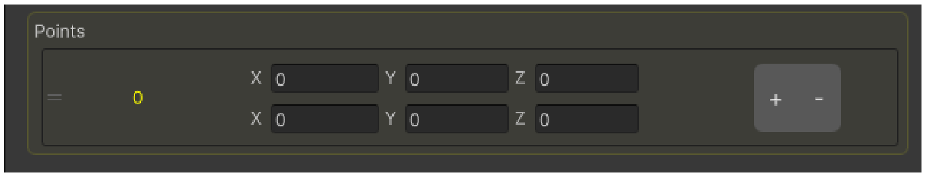
Figure 3: image

The script consists of several blocks. First of all, we are interested in the block of dots (yellow), it is currently empty. To add a point, click the `+` icon in the lower right corner of the block.



Points
List is Empty

Figure 4: image



Points

0 — X 0 Y 0 Z 0
X 0 Y 0 Z 0

Figure 5: image

As you can see there is now one point in the path. The yellow number on the left side is the position of the point in the whole list. Then there are two `Vector3` fields. The upper field is the position of the point, and the lower field is the rotation in the form of Euler angles. The position and rotation of the point are calculated relative to the game object to which the `Path` script is attached. So now point 0 coincides with the position of the parent object. This is also visible in the scene window.

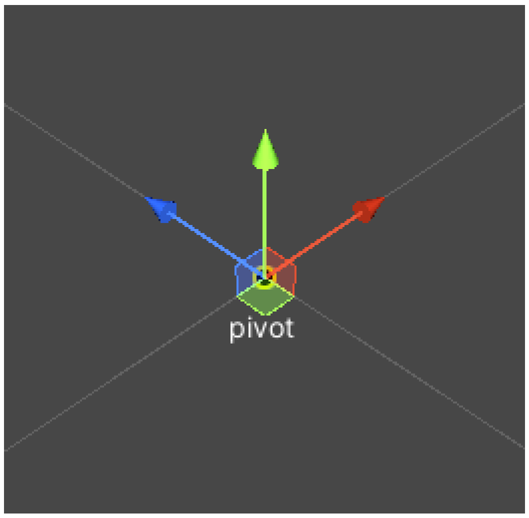


Figure 6: image

The `pivot` text indicates the position of the parent object. Change the point position in the inspector to [1, 0, 0] and switch the scene window camera to top view to make it easier to observe. You will see that the point and the pivot of the object are now in different places.

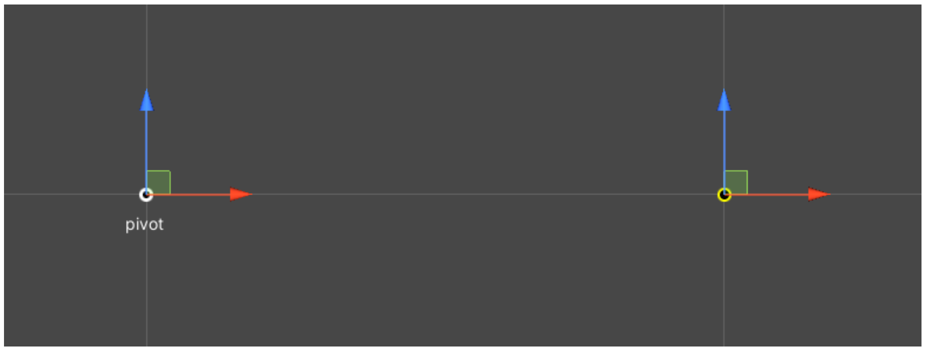
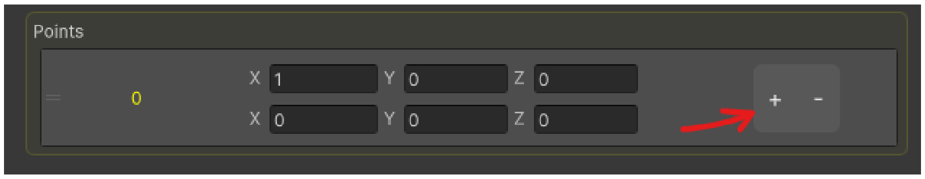


Figure 7: image

By moving, rotating and scaling the object to which the `Path` script is attached, you also move all its points.

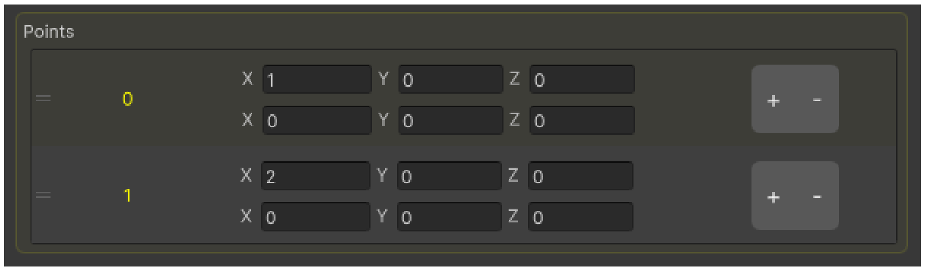
Now add one more point by pressing the `+` button near the point in the inspector window.

This will create another point right after the current one. Set it to position [2, 0, 0]. Our path now looks like this.


Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0

Figure 8: image


Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 0
	X 0	Y 0	Z 0

Figure 9: image

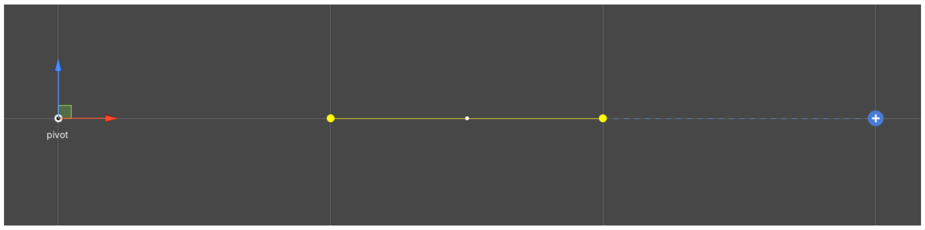


Figure 10: image

The yellow line between the two points is the path line. Later, we will figure out how to make an object move along this line. To delete a point, you can click the - button in the inspector window opposite to the point you want.

To see the point number in the scene window, just put the cursor on the desired point.

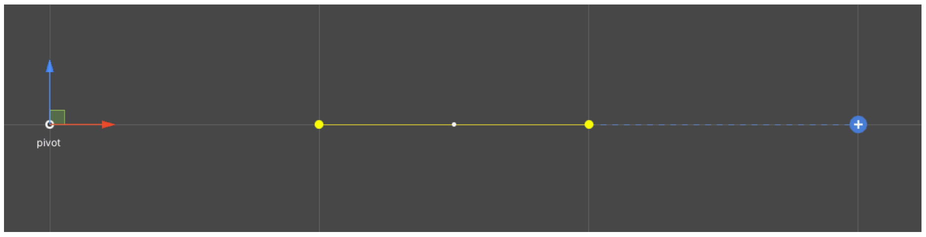


Figure 11:

As you can see, navigating with points in the scene window is not difficult. You can also add a new point directly in the scene window. To do that, press the blue + button. This will create a new point at that location and connect it to the previous one.

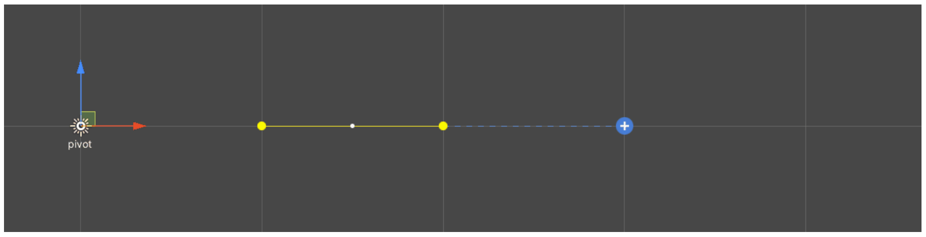


Figure 12:

Now the path looks a little unclear, let's move the point 1 up. You can do this directly in the scene window, to do it, click on point 1 to select it, turn on the move tool (hotkey W) and move the point up to position 1 on the z-axis (use CTRL to snap).

When a point is selected, a quick edit box for that point appears at the bottom right of the Stage window.

This block essentially just duplicates the point fields that are displayed in the inspector window. You can use it, or you can simply drag the point to the desired location using the gizmo tools.

You can add new points between the existing ones by moving the cursor over the small white circle between the two yellow points and a blue + button will appear, click on it to add a point.

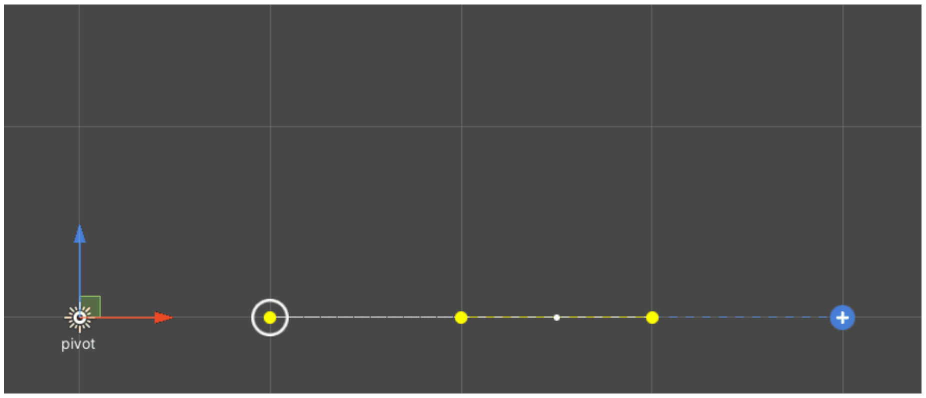
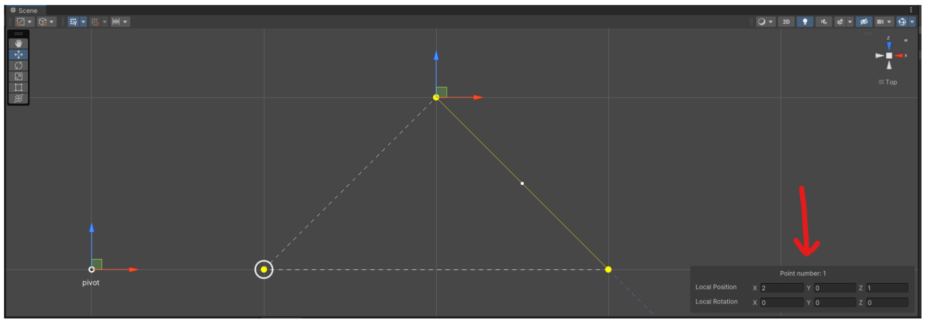


Figure 13:



Point number: 1

Local Position	X	2	Y	0	Z	1
Local Rotation	X	0	Y	0	Z	0

Figure 14: image

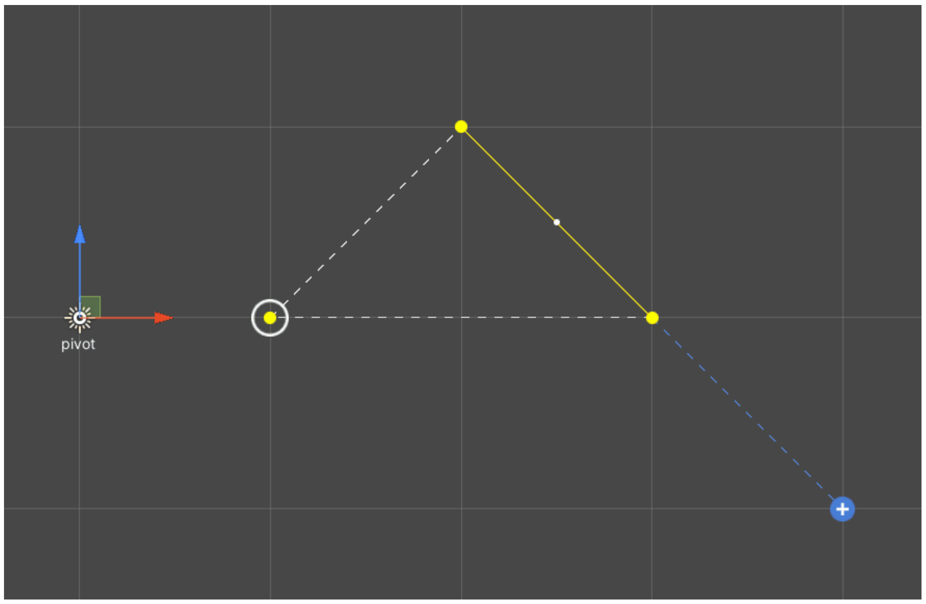


Figure 15:

Now the path consists of 4 dots. About the difference between the white dots and yellow dots, as well as between the dotted line and yellow line we will talk a little later, now we just need to learn how to edit the lines.

You can also delete unselected points directly in the scene window, move the cursor just above the point itself, a delete point button with an `x` sign will appear and double click on it to remove the point.

Undo the last action (deleting a point) by pressing CTRL+Z. `Paths` fully supports the undo/redo system built into Unity.

Working with points in the scene window is quite comfortable, but some parameters are only configurable in the inspector. The first block of the `Path` script in the inspector has 2 fields: `Resolution` and `Looped`.

The `Resolution` - path resolution, that is, this parameter is responsible for how many broken lines should be created between two points. When `Resolution` is 1, one line is drawn between two points, so the whole path is a broken line. Increasing the resolution value will make the line curve. The higher the resolution, the smoother the line becomes.

The second field, `Looped`, is responsible for the looped path.

You can find the optimal resolution for the path by clicking the `Optimize` button under the `Resolution` field.

Immediately below the `Looped` field there is information about how long the

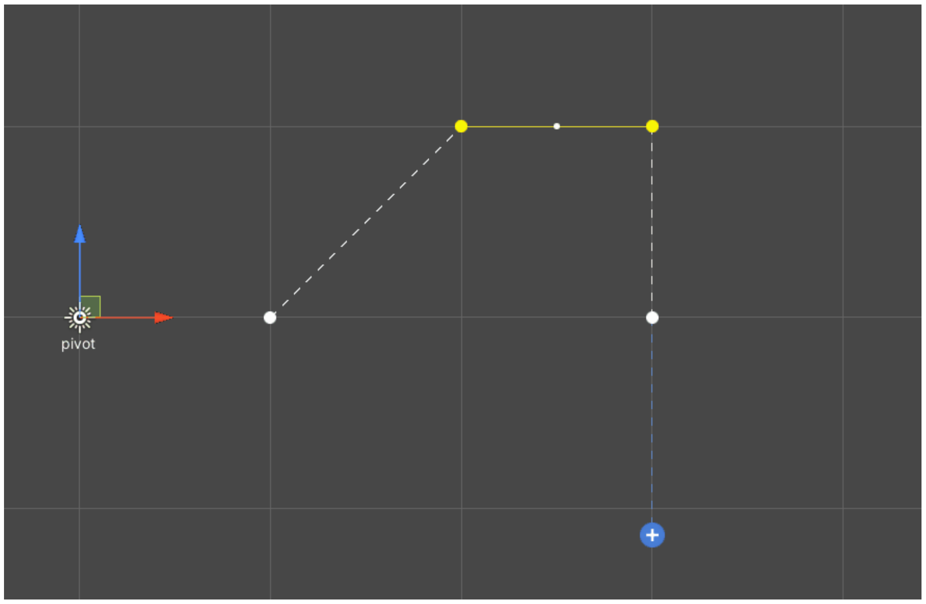
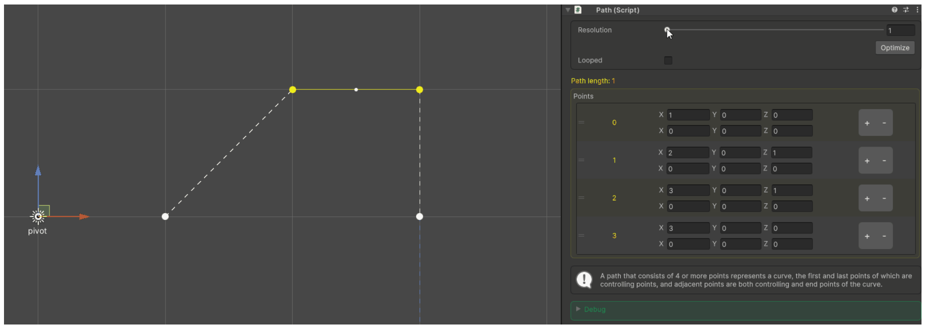


Figure 16:



Resolution 1

Looped

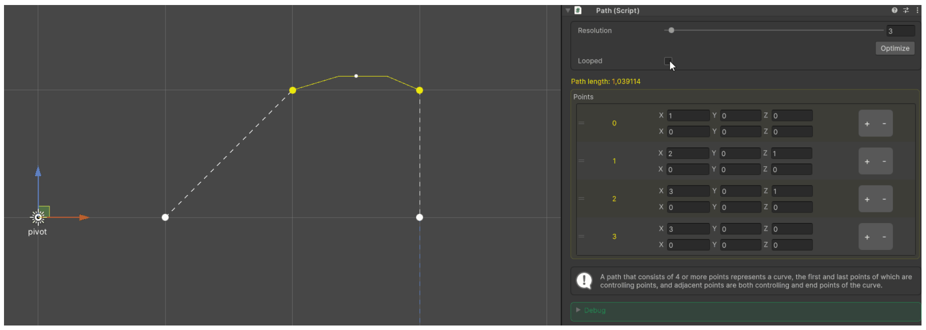
Path length: 1

Points

0	X	1	Y	0	Z	0
	X	0	Y	0	Z	0
1	X	2	Y	0	Z	1
	X	0	Y	0	Z	0
2	X	3	Y	0	Z	1
	X	0	Y	0	Z	0
3	X	3	Y	0	Z	0
	X	0	Y	0	Z	0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 17:



Resolution 3

Looped ☐

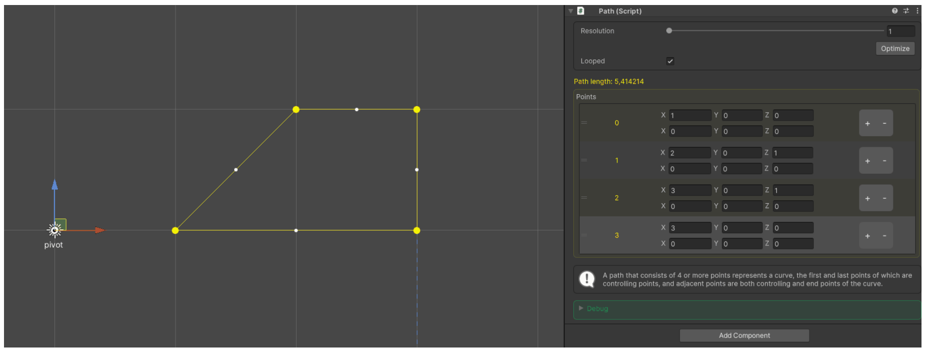
Path length: 1,039114

Points

0	X	1	Y	0	Z	0
	X	0	Y	0	Z	0
1	X	2	Y	0	Z	1
	X	0	Y	0	Z	0
2	X	3	Y	0	Z	1
	X	0	Y	0	Z	0
3	X	3	Y	0	Z	0
	X	0	Y	0	Z	0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 18:



Resolution 1

Looped ✓

Path length: 5,414214

Points

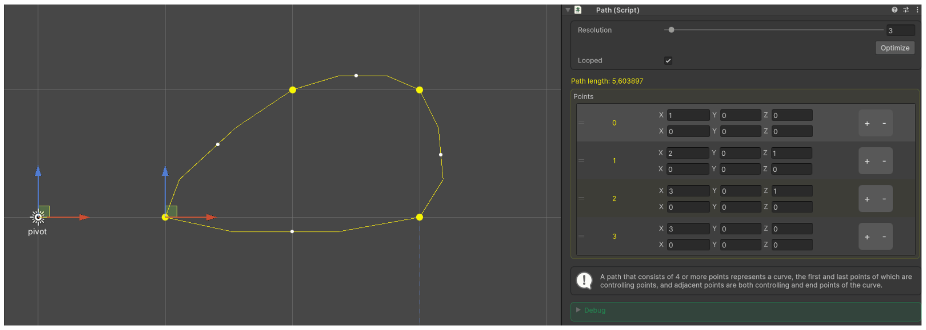
0	X	1	Y	0	Z	0
	X	0	Y	0	Z	0
1	X	2	Y	0	Z	1
	X	0	Y	0	Z	0
2	X	3	Y	0	Z	1
	X	0	Y	0	Z	0
3	X	3	Y	0	Z	0
	X	0	Y	0	Z	0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 19:

path is (yellow lines).

You can swap path points.



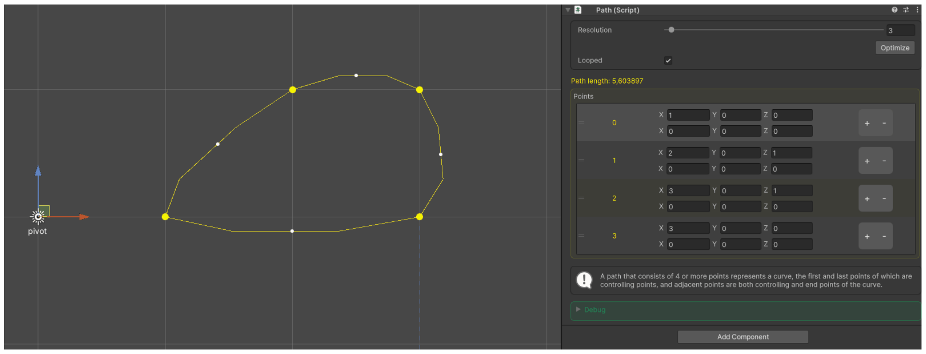
Resolution 3
Looped ✓
Path length: 5,603897
Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 3	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 3	Y 0	Z 0
	X 0	Y 0	Z 0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 20:

When the path is fully configured, you can see how the object will move along it. To do this, open the debug block (green), the scene will show the green edges of the imaginary cube, which will be at point 0. You can move the `Position` slider to specify on what percentage of the path you want this cube to be located.



Resolution 3
Looped ✓
Path length: 5,603897
Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 3	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 3	Y 0	Z 0
	X 0	Y 0	Z 0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 21:

By default, when debugging, the face of the cube is always directed according to the path vector.

However, you can change this behavior by turning off the `Use Path Direction` field. In this case the cube will match the rotations of the points. In our example, we didn't change the rotations of the points (only their positions), so the rotations of all the points coincide with the global axes.

To work with point rotations, turn on the `rotate` tool (hotkey E). You can set the desired point rotations in the points list in the Inspector.

Or rotate the point with the gizmo in the scene window.

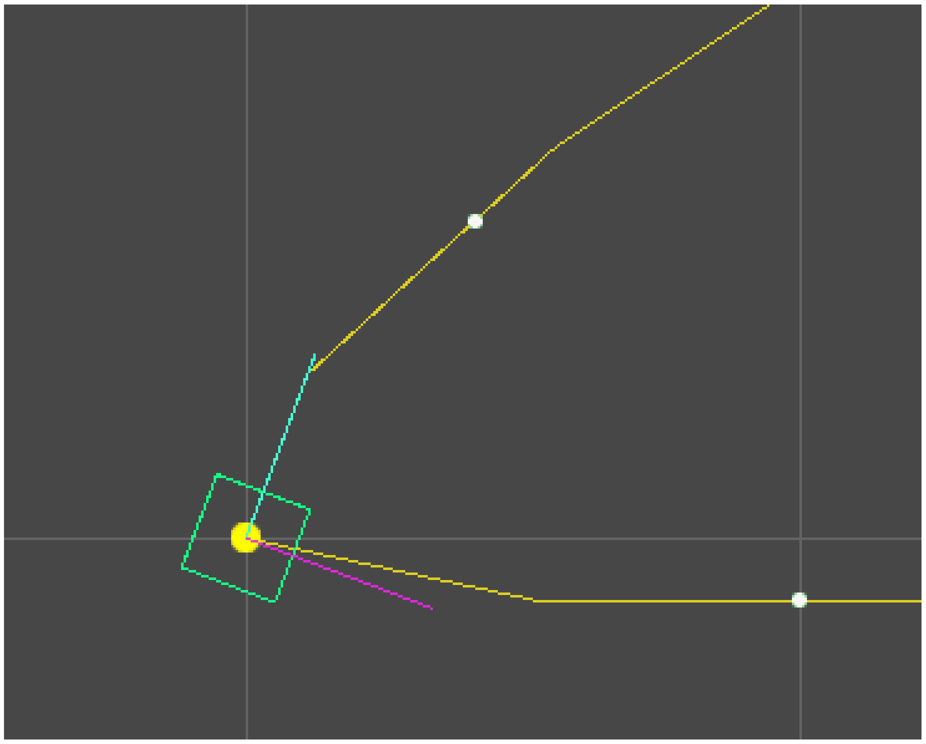
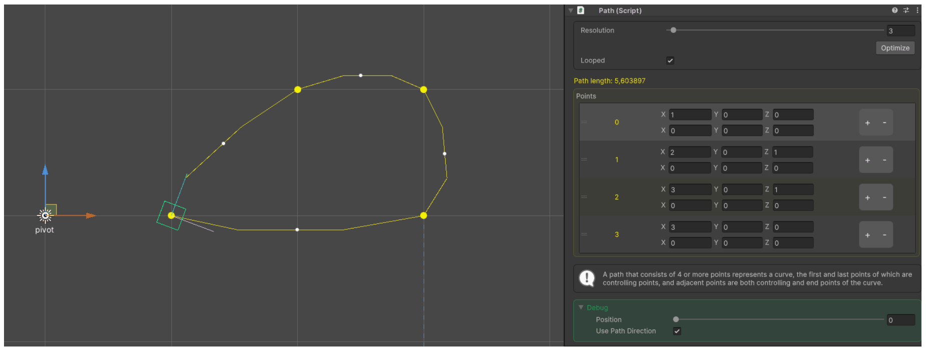


Figure 22:



Path (Script)

Resolution 3

Optimize

Looped ✓

Path length: 5,603897

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 3	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 3	Y 0	Z 0
	X 0	Y 0	Z 0

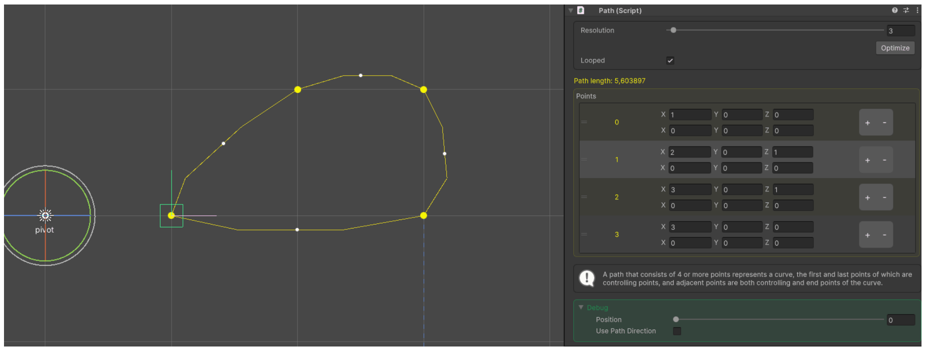
A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Debug

Position 0

Use Path Direction ✓

Figure 23:



Path (Script)

Resolution 3

Optimize

Looped ✓

Path length: 5,603897

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 3	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 3	Y 0	Z 0
	X 0	Y 0	Z 0

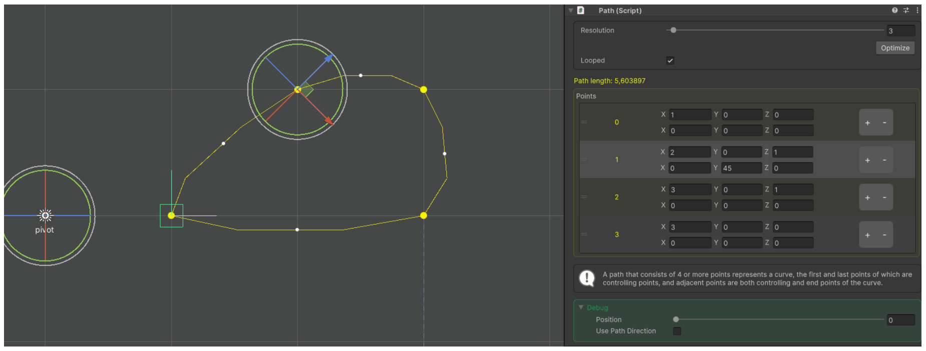
A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Debug

Position 0

Use Path Direction

Figure 24:



Path (Script)

Resolution 3

Optimize

Looped ✓

Path length: 5,603897

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 2	Y 0	Z 1
	X 0	Y 45	Z 0
2	X 3	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 3	Y 0	Z 0
	X 0	Y 0	Z 0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

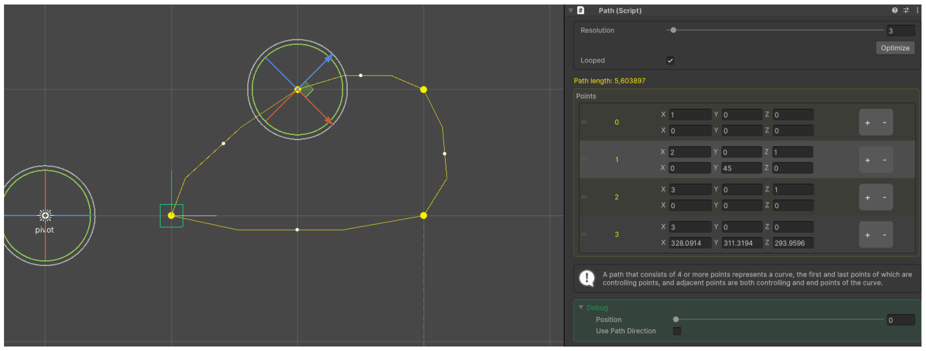
Debug

Position 0

Use Path Direction

Figure 25:

To make it easier to understand the rotation of a point, the local axes of the point are displayed inside the `rotation` tool gizmo. Once all the desired points are rotated, you can debug the path, i.e. see how the special cube will behave as it traverses that path. Passing through the points, the rotation of the cube will correspond to the rotations of these points.



Path (Script)

Resolution 3

Optimize

Looped ✓

Path length: 5,603897

Points

0	X 1	Y 0	Z 0	+ -
	X 0	Y 0	Z 0	
1	X 2	Y 0	Z 1	+ -
	X 0	Y 45	Z 0	
2	X 3	Y 0	Z 1	+ -
	X 0	Y 0	Z 0	
3	X 3	Y 0	Z 0	+ -
	X 328.0914	Y 311.3194	Z 293.9596	

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Debug

Position 0

Use Path Direction

Figure 26:

Creating ready-made path templates

In addition to an empty path, you can create paths with points already in them. To do this, use the context menu `Path > ...` in the hierarchy window. For example, if you want to create a circle, use `Path > Circle`.

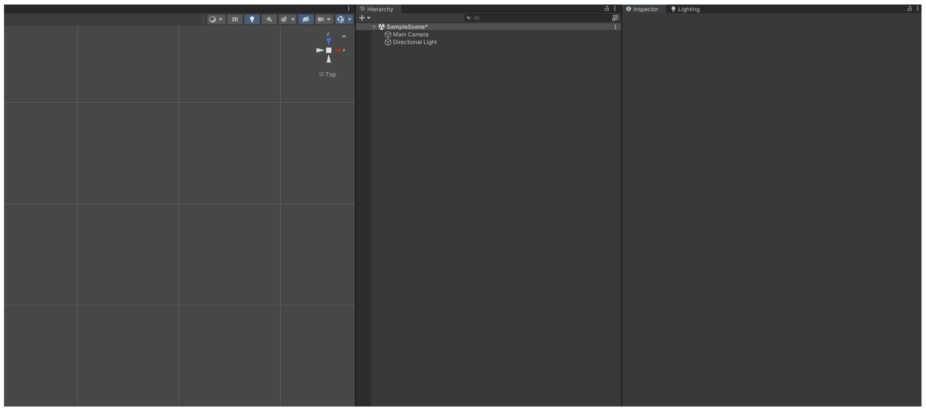


Figure 27:

Or for example, you want a 3D spiral.

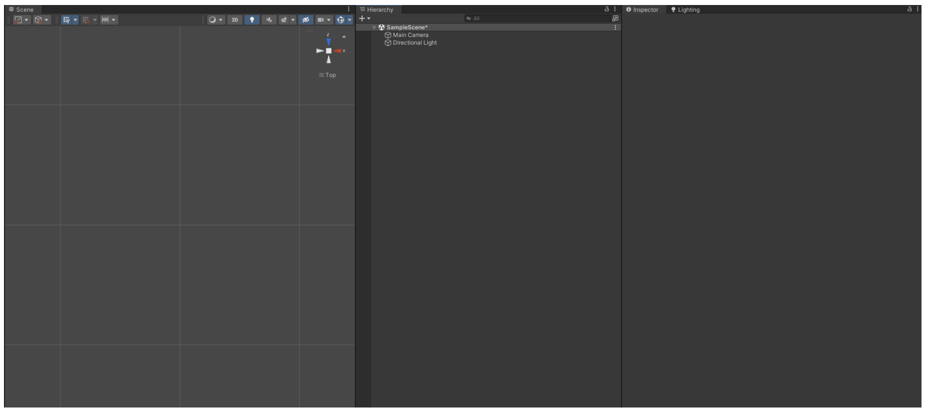


Figure 28:

Path special cases

Now that you know how to control lines in the editor, it is necessary to deal with special cases of path construction, namely when the path consists of 0, 1, 2, 3 and more than 3 points. The point is that the Catmull-Rome algorithm knows how to work with at least 4 points, with the first and last being the controlling ones, and those in the middle - between them will draw the line.

Path of 0 points.

In this case the path is nothing. Calling methods (we'll talk about them later) to retrieve data on the path will result in an exception.

Path from 1 point

A path from 1 point will always return a value at that point when calling methods to get data on the path, whether the path is looped or not. This single point is used as 4 points in this location.

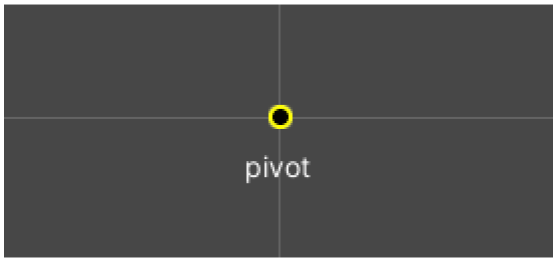


Figure 29: image

Path from 2 points

The Catmull-Rom algorithm for 2 points can only construct a straight line, so regardless of the value of the `Resolution` field between two points will always be a straight line. If the path is looped, it will consist of two segments, the first from point 0 to point 1, and the second vice versa. This is because each point is both a controlling point and an end point (that is, the total will be 4 points).

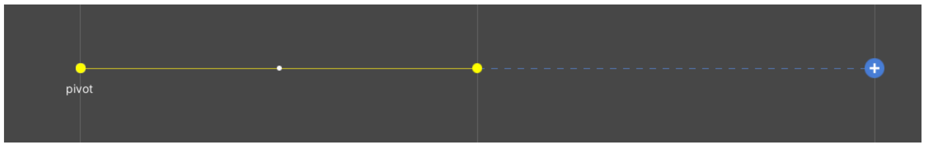
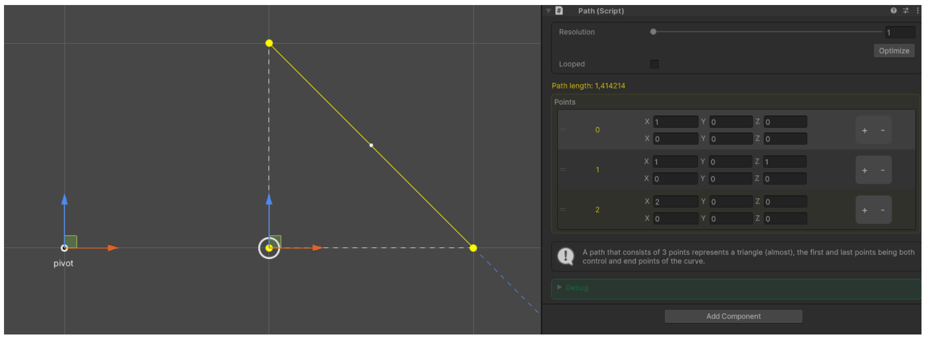


Figure 30: image

Path of 3 points

Three points allow you to describe one segment of a curved line. Point 0 in this case is the controlling point for points 1 and 2, and points 1 and 2 are the end points, i.e. the ones between which the path is drawn. The `Resolution` field affects the smoothness of this line. The first point is both the controlling and the end point, which again together with the other two gives 4 points.



Resolution 1

Looped ☐

Path length: 1,414214

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 1	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 2	Y 0	Z 0
	X 0	Y 0	Z 0

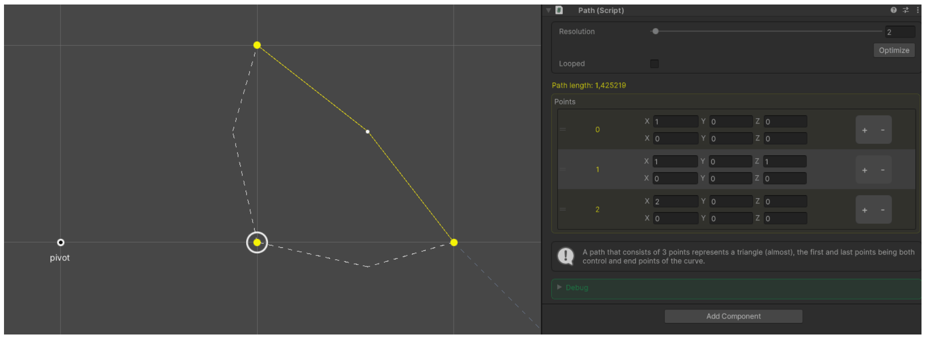
A path that consists of 3 points represents a triangle (almost), the first and last points being both control and end points of the curve.

Figure 31:

At the same time the path can be looped.

A path of 4 or more points

When a path consists of 4 or more points, the first point (with the index 0) and the last one are controlling points and all other points are end points. You can control the line starting from point 1 separately thanks to the control point 0. It is the same with the penultimate point. The controlling points are drawn in white and connected by a white dotted line. These points do not participate



Path (Script)

Resolution 2

Optimize

Looped

Path length: 1,425219

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 1	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 2	Y 0	Z 0
	X 0	Y 0	Z 0

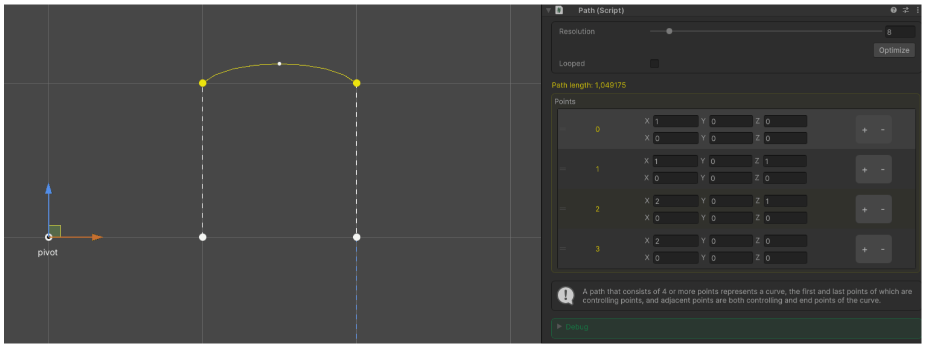
A path that consists of 3 points represents a triangle (almost), the first and last points being both control and end points of the curve.

Debug

Add Component

Figure 32:

in the traversed path, but only influence it. Moreover, with 4 points you can achieve the same result as with 1, 2 or 3 points.



Path (Script)

Resolution 8

Optimize

Looped

Path length: 1,049175

Points

0	X 1	Y 0	Z 0
	X 0	Y 0	Z 0
1	X 1	Y 0	Z 1
	X 0	Y 0	Z 0
2	X 2	Y 0	Z 1
	X 0	Y 0	Z 0
3	X 2	Y 0	Z 0
	X 0	Y 0	Z 0

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Debug

Figure 33:

And this is what a path of 7 points might look like.

API

To work with paths via code, a convenient API has been developed.

Path creation

To create a path use the static method `Path.Create()`. It will create a game object named "Path" at position zero, add a `Path` component to it and return it as a result. This method has 4 overloads: 1. `Path Create()` 2. `Path Create(Vector3 pivotPosition)` 3. `Path Create(Vector3 pivotPosition, bool useGlobal, params Vector3[] points)` 4. `Path Create(Vector3 pivotPosition, bool useGlobal, IEnumerable<Vector3> points)`

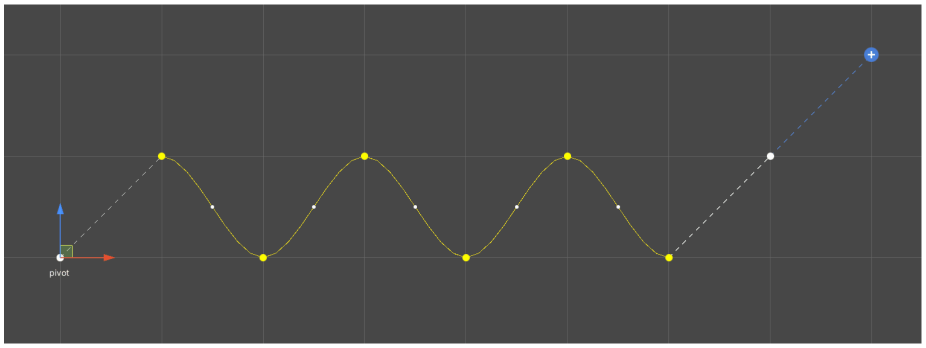


Figure 34:

Where: * `pivotPosition` - path position. * `useGlobal` - specifies whether the points (parameter `points`) are transmitted in global (`true`) or local (`false`) space. * `points` - a collection of points.

For example:

`var path = Path.Create(new Vector3(1f, 0f, 1f), true, Vector3.zero, new Vector3(1f, 0f, 0f));`

will create a new path located at position [1, 0, 1] with two points at the global positions [0, 0, 0] and [1, 0, 0].

You can also use the `Path.CreatePolygon` and `Path.CreateSpiral` methods.

The `CreatePolygon` creates a polygon path (triangle, rhombus, pentagon and others). There are 3 overloads available: 1. `Path CreatePolygon(int sideCount, float radius)` 2. `Path CreatePolygon(Vector3 pivotPosition, int sideCount, float radius)` 3. `Path CreatePolygon(Vector3 pivotPosition, Vector3 normal, int sideCount, float radius)`

Where: * `sideCount` - the number of faces. * `radius` - distance from the center of the figure to any corner. * `pivotPosition` - position of the path in space. * `normal` - the normal of the figure, that is a vector representing where the face of the figure is directed in space. * `up` - a vector representing the direction of the face of the figure when rotated towards `normal`.

For example:

`var path = Path.CreatePolygon(5, 1f);`

this code will create a pentagon with a radius of 1 meter.

`CreateSpiral` creates a path-spiral (Archimedean). There are 3 overloads: 1. `Path CreateSpiral(float offsetAngle, int coils, float step, int pointsCountPerCoil, bool use3D = false)` 2. `Path CreateSpiral(Vector3 pivotPosition, float offsetAngle, int coils, float step, int`

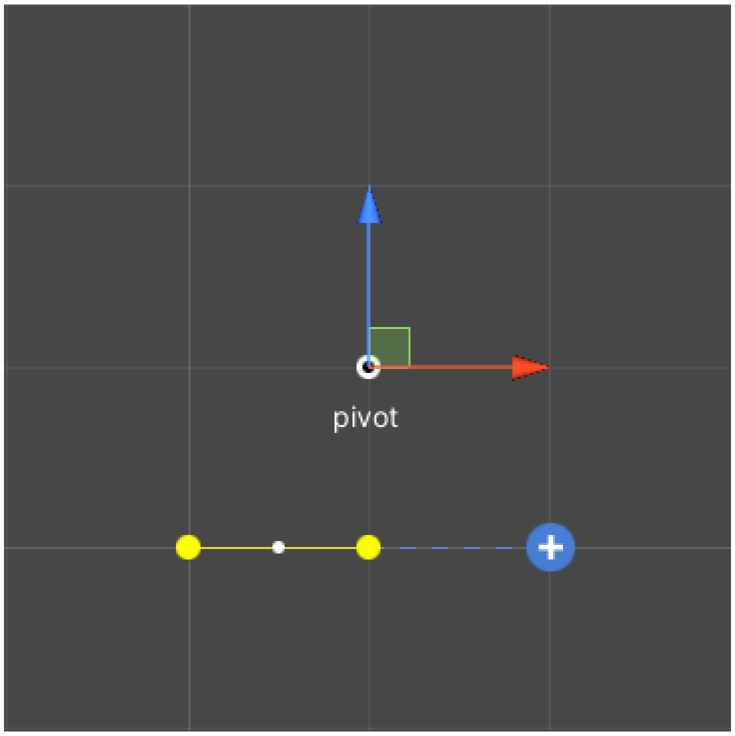


Figure 35: image

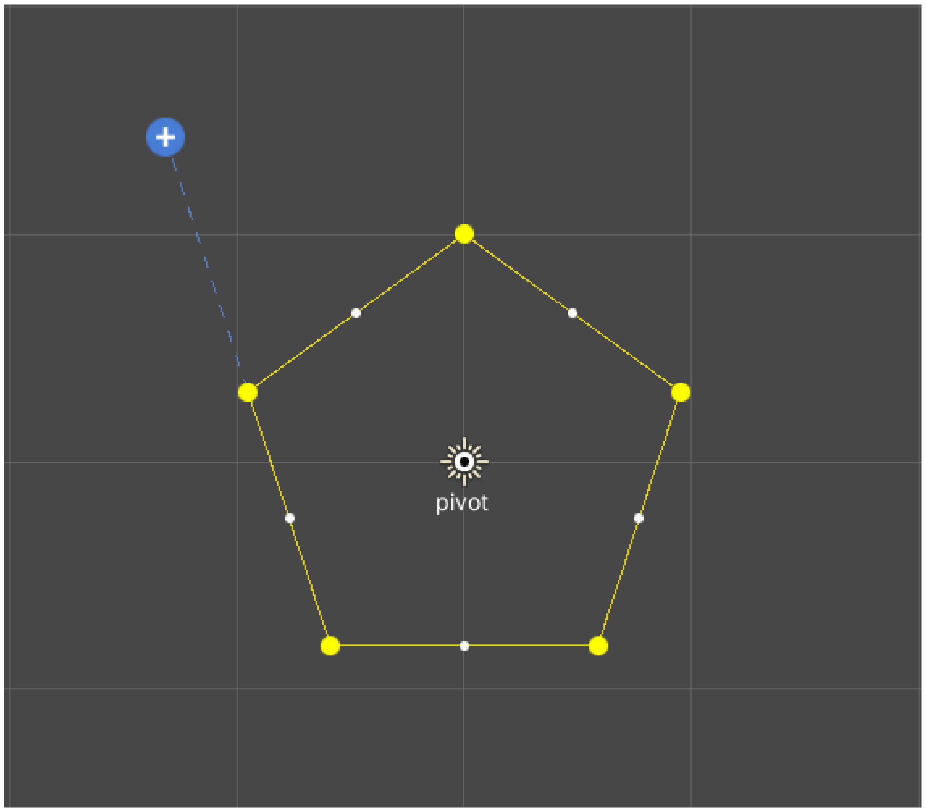


Figure 36: image

`pointsCountPerCoil, bool use3D = false)` 3. `Path CreateSpiral(Vector3 pivotPosition, Vector3 normal, float offsetAngle, int coils, float step, int pointsCountPerCoil, bool use3D = false)`

Where: * `offsetAngle` - angular displacement (in degrees) of the helix, i.e. rotation of the helix around its center. * `coils` - number of coils of the spiral. * `step` - spiral pitch, i.e. the distance between two coils. * `PointsCountPerCoil` - number of generated points per coil. * `use3D` - Is it necessary to create a three-dimensional spiral? * `pivotPosition` - position of the path in space. * `normal` - the normal of the figure, that is, a vector representing where the face of the figure is directed in space. * `up` - a vector representing the direction of the face of the figure when rotated towards `normal`.

For example:

```
var path = Path.CreateSpiral(0f, 3, 1f, 8, true);
```

this code will create a 3D spiral with 0 degree offset, 3 turns, 1 meter distance between turns, and 8 points per turn.

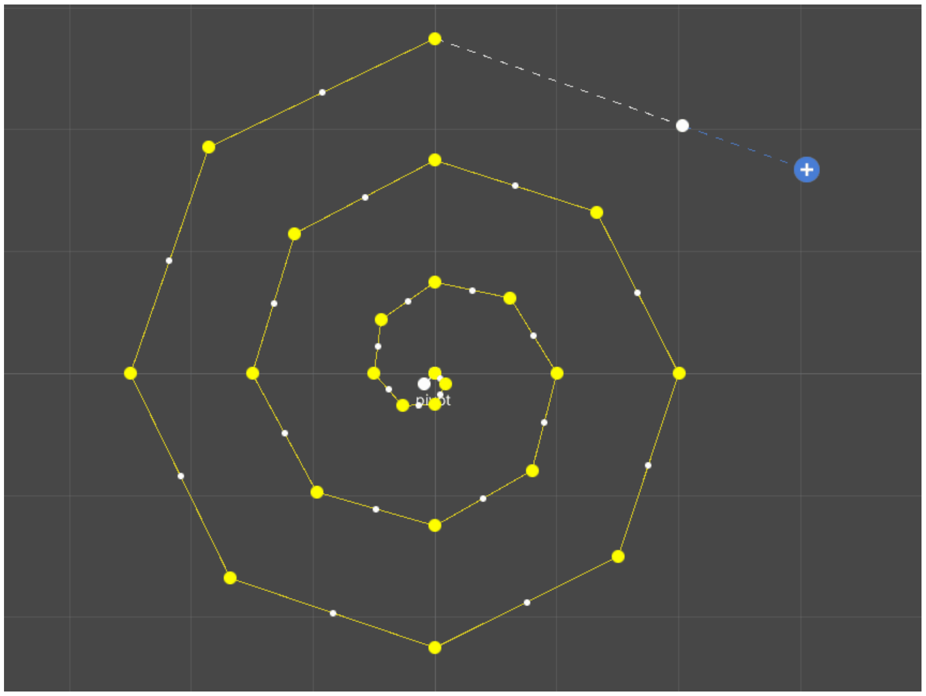


Figure 37:

The `CreateArc` creates a path-arc. There are 3 overloads: 1. `Path CreateArc(float width, float height, int sideCount)` 2. `Path CreateArc(Vector3 pivotPosition, float width, float height, int` `sideCount)` 3. `Path CreateArc(Vector3 pivotPosition, Vector3 normal, Vector3 up, float width, float height, int sideCount)`

Where: * `width` - width of the arch in meters. * `height` - height of the arch in meters. * `sideCount` - number of sides in the arch (minimum 3). * `pivotPosition` - track position in space. * `normal` - the normal of the figure, that is a vector representing where the front side of the figure is directed in space. * `up` - a vector representing the direction of the face of the figure when rotated towards `normal`.

For example:

```
var path = Path.CreateArc(2f, 4f, 8);
```

this code will create an arch 2 meters wide, 4 meters high and 8 sides.

The `CreateWave` creates a path-wave. There are 3 overloads available: 1. `Path CreateWave(float height, float frequency, int repeat, bool startToUp = true)` 2. `Path CreateWave(Vector3 pivotPosition, float height, float frequency, int repeat, bool startToUp = true)` 3. `Path CreateWave(Vector3 pivotPosition, Vector3 normal, Vector3 up, float height, float frequency, int repeat, bool startToUp = true)`

Where: * `height` - wave height in meters. * `frequency` - frequency of the wave per 1 meter. * `repeat` - the number of repetitions of the wave. * `startToUp` - in which direction the wave starts, up (true) or down (false). * `pivotPosition` - position of the path in space. * `normal` - the normal of the figure, i.e. a vector representing where the face of the figure is directed in space. * `up` - a vector representing the direction of the face of the figure when rotated towards `normal`.

For example:

```
var path = Path.CreateWave(1f, 2f, 3);
```

this code will create a wave with a height of 1 meter, a frequency of 2 waves per meter and repeat it 3 times.

Path properties

- `PointsCount` - total number of points in the path
- `SegmentsCount` - number of segments in the path. A segment is a curve between two points. In a looped path, the number of segments is always equal to the number of points.
- `Resolution` - path resolution. Affects all segments.
- `Looped` - is the path looped?
- `Length` - total length of the path (sum of lengths of all segments).

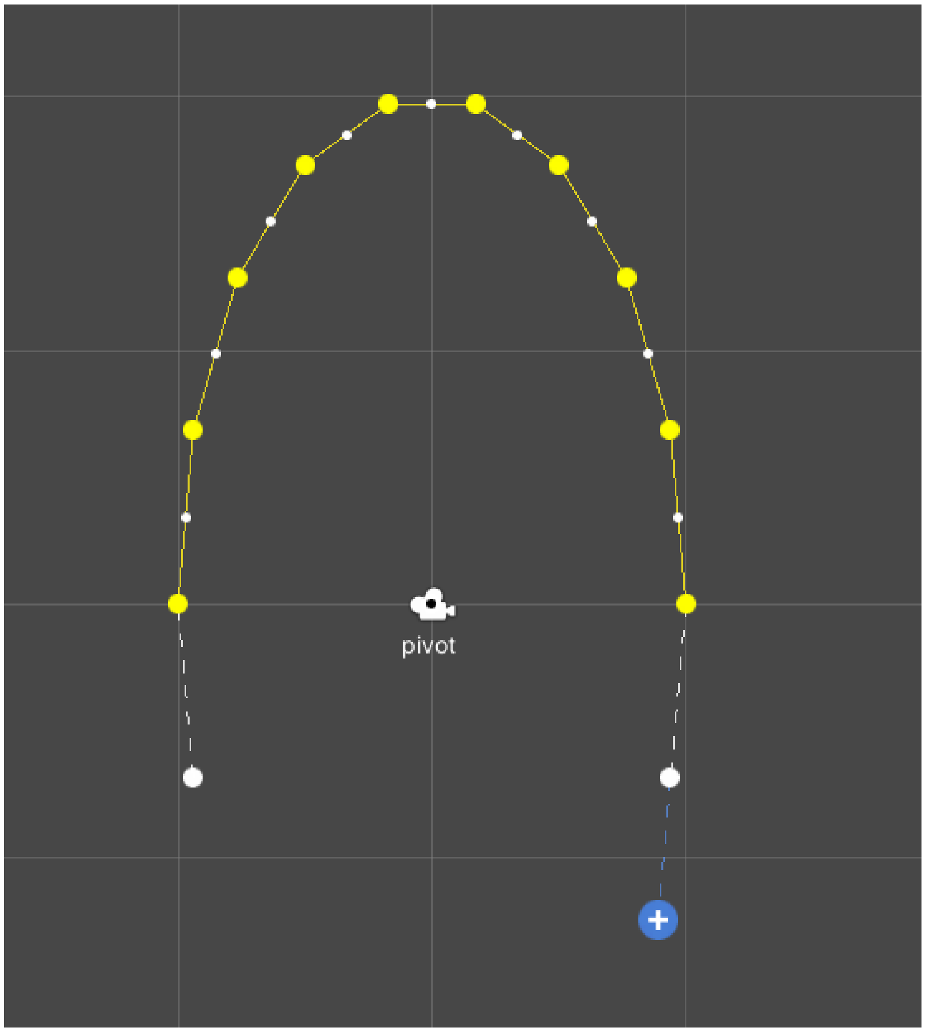


Figure 38: image

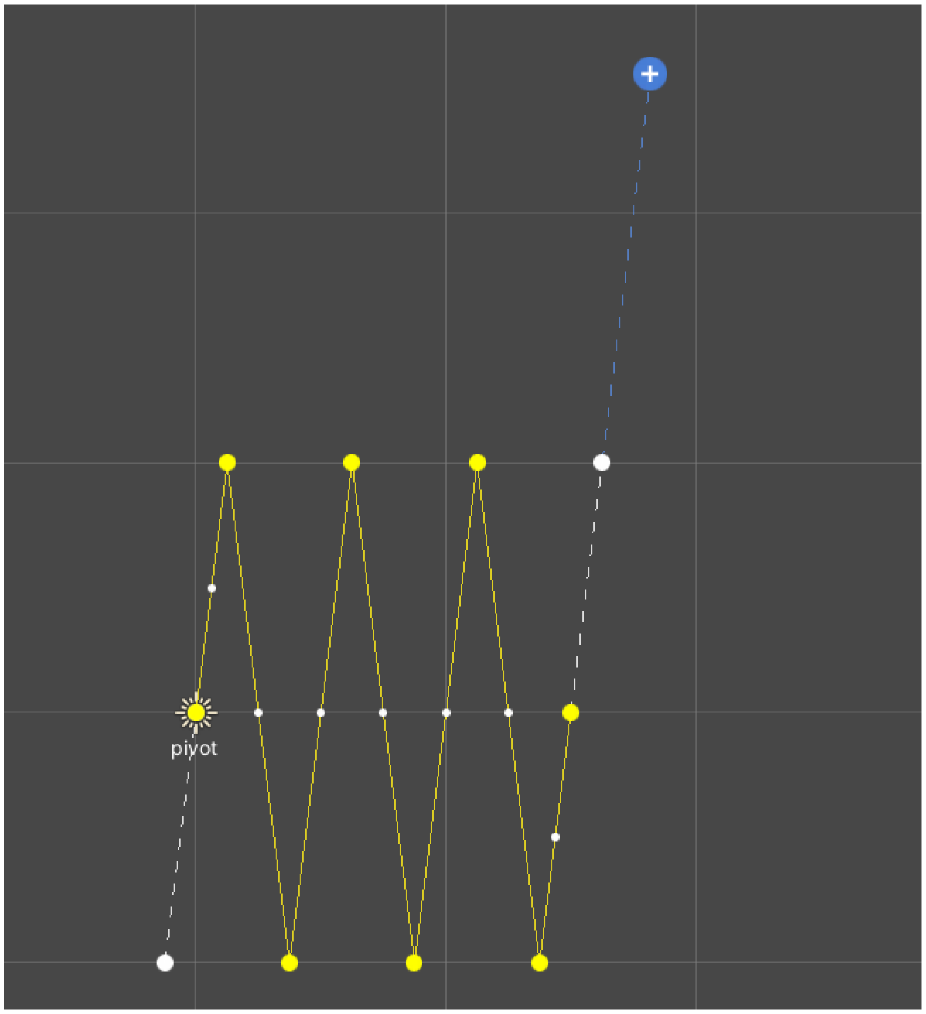
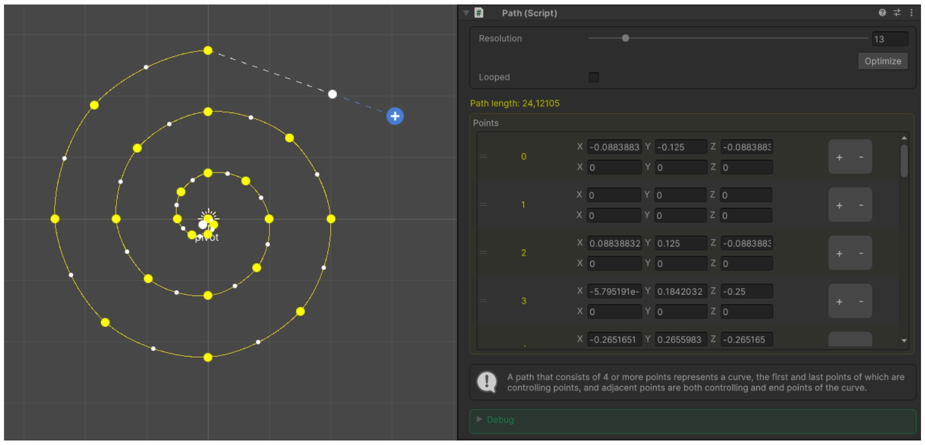


Figure 39: image

Path optimization

You can use the `Optimize` method to optimize a path. The algorithm will find the most appropriate value for the `Resolution` field, such that the path looks moderately smoothed.

```
var path = Path.CreateSpiral(0f, 3, 1f, 8, true);
path.Optimize();
```



Resolution 13

Looped ☐

Path length: 24,12105

Points

0	X -0.0883883	Y -0.125	Z -0.0883883
	X 0	Y 0	Z 0
1	X 0	Y 0	Z 0
	X 0	Y 0	Z 0
2	X 0.08838832	Y 0.125	Z -0.0883883
	X 0	Y 0	Z 0
3	X -5.795191e-	Y 0.1842032	Z -0.25
	X 0	Y 0	Z 0
	X -0.2651651	Y 0.2655983	Z -0.265165

A path that consists of 4 or more points represents a curve, the first and last points of which are controlling points, and adjacent points are both controlling and end points of the curve.

Figure 40: image

Segment length

Use the `float GetSegmentLength(int segment)` method to get the length of a particular path segment.

```
var path = Path.CreatePolygon(4, 1f);
var segment1Length = path.GetSegmentLength(1);
Debug.Log(segment1Length);
```

In the figure above, the indexes of the segments formed by the dots are shown in red. Below you can see the result of calling the `GetSegmentLength` method.

Point type

To work with points through code, there is a data type Point. It is a structure with properties `Position` and `Rotation`. You can create a point using the default constructor or an overloaded version that takes the position and rotation of the point in space. All points inside a `Path` object are calculated locally, but as you'll see later, almost all methods allow you to work with points globally as well if you want to.

```
var point1 = new Point(); // Create a point at position [0, 0, 0] and with zero rotation.
```

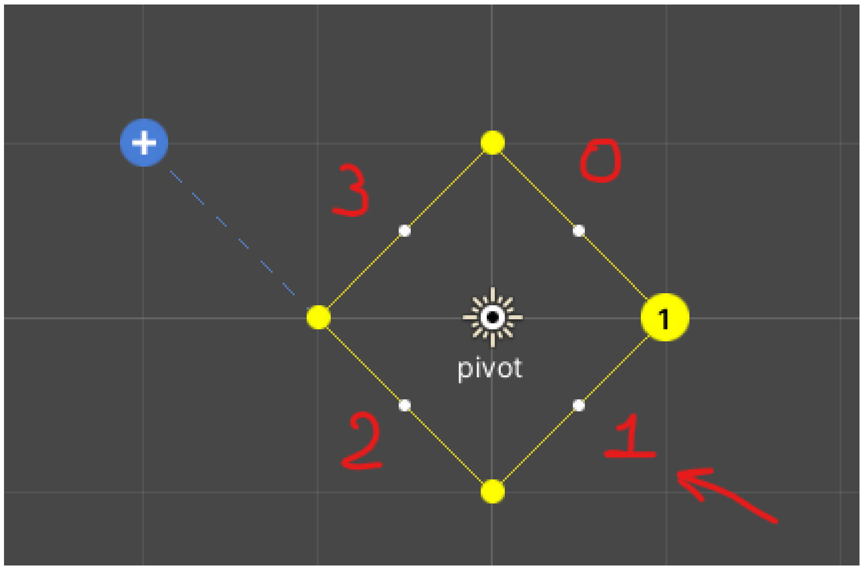
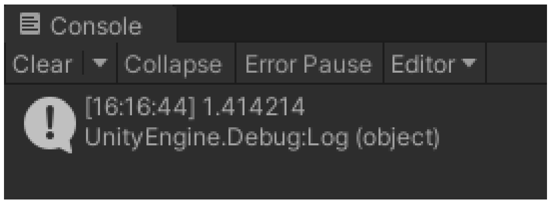


Figure 41: image



```
[16:16:44] 1.414214
UnityEngine.Debug:Log (object)
```

Figure 42: image

```
point1.Position = new Vector3(1f, 2f, 3f); // Set the position of this point to [1, 2, 3].
point1.Rotation = Quaternion.Euler(0f, 90f, 0f); // Set the rotation of this point in the di

var point2 = new Point(1f, 2f, 3f, Quaternion.Euler(0f, 90f, 0f)); // Do the same thing, but
```

PointData type

This type is an extended version of the `Point` structure, which in addition to the `Position` and `Rotation` properties contains also `Direction`. The `Direction` is the direction in space that corresponds to the movement along the path at the point `Position`, that is, it is a vector representing the direction of movement along the path. You don't have to create `PointData` structure objects manually, instead the methods that we'll explore later in this documentation will calculate this direction themselves and return `PointData` to you instead of `Point` where it makes sense.

Operations on points

Adding to the end

- `void AddPoint(float x, float y, float z, bool useGlobal = true)` - creates a new point with specified position and adds it to the end of the path.
- `void AddPoint(Vector3 position, bool useGlobal = true)` - creates a new point with the specified position and adds it to the end of the path.
- `void AddPoint(Point point, bool useGlobal = true)` - adds a point to the end of the path. ### Insertion by index
- `void InsertPoint(int index, float x, float y, float z, bool useGlobal = true)` - creates a new point with the specified position and inserts it into the path at the specified index.
- `void InsertPoint(int index, Vector3 position, bool useGlobal = true)` - creates a new point with the specified position and inserts it into the path at the specified index.
- `void InsertPoint(int index, Point point, bool useGlobal = true)` - inserts a point into the path at the specified index. ### check for content
- `bool ContainsPoint(float x, float y, float z, bool useGlobal = true)` - checks if there is a point with the specified position in the path.
- `bool ContainsPoint(Vector3 position, bool useGlobal = true)` - checks if there is a point with the specified position in the path.
- `bool ContainsPoint(Point point, bool useGlobal = true)` - checks if there is a point in the path. ### Index search
- `int IndexOfPoint(float x, float y, float z, bool useGlobal = true)` - finds point with specified position in the path and returns its

index

- `int IndexOfPoint(Vector3 position, bool useGlobal = true) -` finds point with specified position in the path and returns its index
- `int IndexOfPoint(Point point, bool useGlobal = true) -` finds a point in the path and returns its index ### Remove from the path
- `bool RemovePoint(float x, float y, float z, bool useGlobal = true) -` removes a point with the specified position from the path. Returns `true` if the point has been removed.
- `bool RemovePoint(Vector3 position, bool useGlobal = true) -` removes the point with the specified position from the path. Returns `true` if the point has been removed.
- `bool RemovePoint(Point point, bool useGlobal = true) -` removes the point from the path. Returns `true` if point has been deleted. ### Remove by index
- `void RemovePointAt(int index) -` removes a point from the path at the specified index. ### Complete clearing.
- `void ClearPoints() -` removes all points from the path.

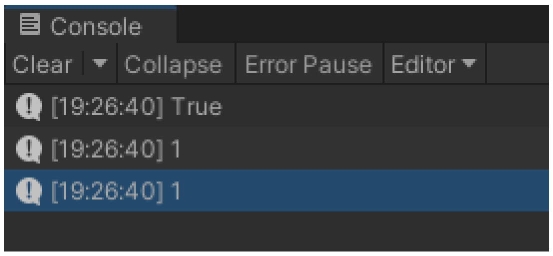
Example of using these methods.

Almost all of the methods described above have a `useGlobal` parameter, which tells the method in which space (local? or global?) the position and/or rotation are passed to it. A value of `true` means that the data is passed in global space.

```
var path = Path.Create();
path.AddPoint(Vector3.zero);
path.AddPoint(1f, 0f, 0f);

var point = new Point(2f, 0f, 0f, Quaternion.identity);

path.InsertPoint(0, point);
Debug.Log(path.ContainsPoint(point));

var index = path.IndexOfPoint(Vector3.zero);
Debug.Log(index);

path.RemovePoint(point);
path.RemovePointAt(index);

Debug.Log(path.PointsCount);
```

Read and change points

To read point values, use the `PointData GetPoint(int index, bool useGlobal = true)` method. This method will return you a `PointData` object representing the point and its direction in the path.



```
[19:26:40] True
[19:26:40] 1
[19:26:40] 1
```

Figure 43: image

```
var path = Path.CreatePolygon(4, 1f);
print(path.GetPoint(0));
```

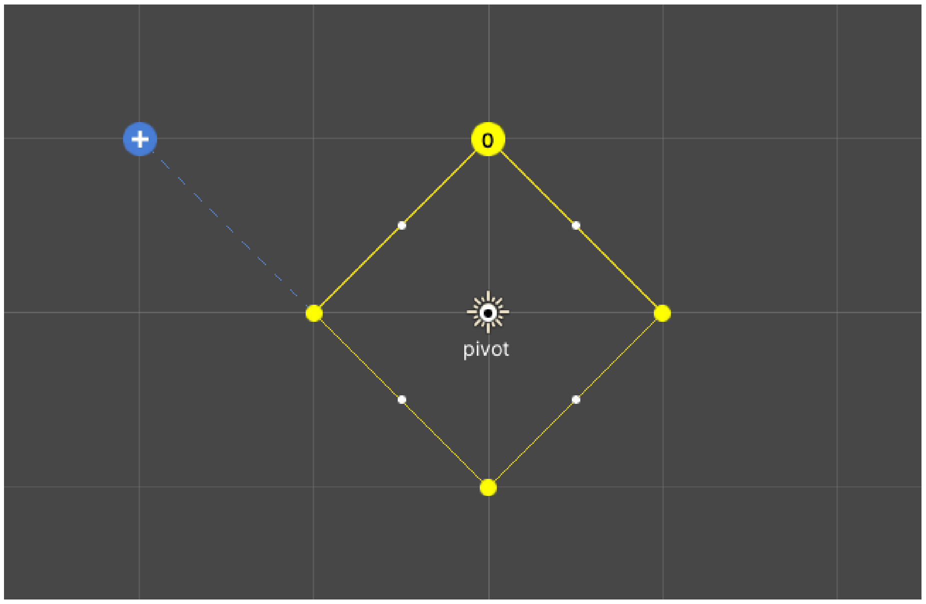
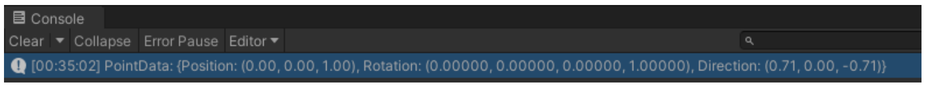


Figure 44: image

The second line reads the value of point 0. The figure shows that the direction of movement from point 0 to point 1 (to the right and below it) in global space is [1, 0, -1], if we normalize this value, we get [0.71, 0, -0.71], as we see in the 'Debug' tab in the second screenshot.

Use one of the overloaded variants of the `SetPoint` method to write points: * `void SetPoint(int index, float x, float y, float z, bool useGlobal = true)` - change point position by index `index`. * `void SetPoint(int index, Vector3 position, bool useGlobal = true)` - change point position by index `index`. * `void SetPoint(int index,`



```
[00:35:02] PointData: {Position: (0.00, 0.00, 1.00), Rotation: (0.00000, 0.00000, 0.00000, 1.00000), Direction: (0.71, 0.00, -0.71)}
```

Figure 45: image

`Quaternion rotation, bool useGlobal = true)` - change point rotation by index `index`. * `void SetPoint(int index, float x, float y, float z, Quaternion rotation, bool useGlobal = true)` - change position and rotation of the point by index `index`. * `void SetPoint(int index, Vector3 position, Quaternion rotation, bool useGlobal = true)` - change position and rotation of the point by index `index`. * 'void SetPoint(int index, Point point, bool useGlobal = true)` - change point by index `index`.

For example:

```
var path = Path.CreatePolygon(4, 1f);
path.SetPoint(0, 0f, 0f, 2f);
```

This changes the position of the point with index 0 to [0, 0, 2f].

Calculate a point on a path

Use the `Calculate` method if you need to calculate a point on a path (on the yellow line). There are several overloads of this method. * `PointData Calculate(int index, bool useGlobal = true)` - calculates the point lying on the path by the index. Unlike the `GetPoint` method, this method takes into account only those points that lie on the path (the yellow line in the editor). You can use the `useGlobal` parameter to specify in which space you want to calculate the point; if `true` is passed, the calculation will take place in the global space. * `PointData Calculate(int segment, float distance, bool useNormalizedDistance = true, bool useGlobal = true)` - calculates the point on the specified segment. If the `useNormalizedDistance` parameter is `true`, then the `distance` parameter will be used as a normalized value (that is, from 0 to 1), otherwise `distance` is treated as meters. * `PointData Calculate(float distance, bool useNormalizedDistance = true, bool useGlobal = true)` - calculates the point on the whole path. The parameters behave the same as in the previous method.

For example, we have this way.

And we want to get a point that lies on a segment with the index 1. The point must be at a distance of 1 meter from the beginning of the segment. To understand how to solve this problem, let's first understand the segment indices.

In the figure above, point indices are marked in red, and segment indices are marked in blue. Segments always start at the first point of the path (the yellow point, not the white one). Note that the segment with index 1 is at the bottom

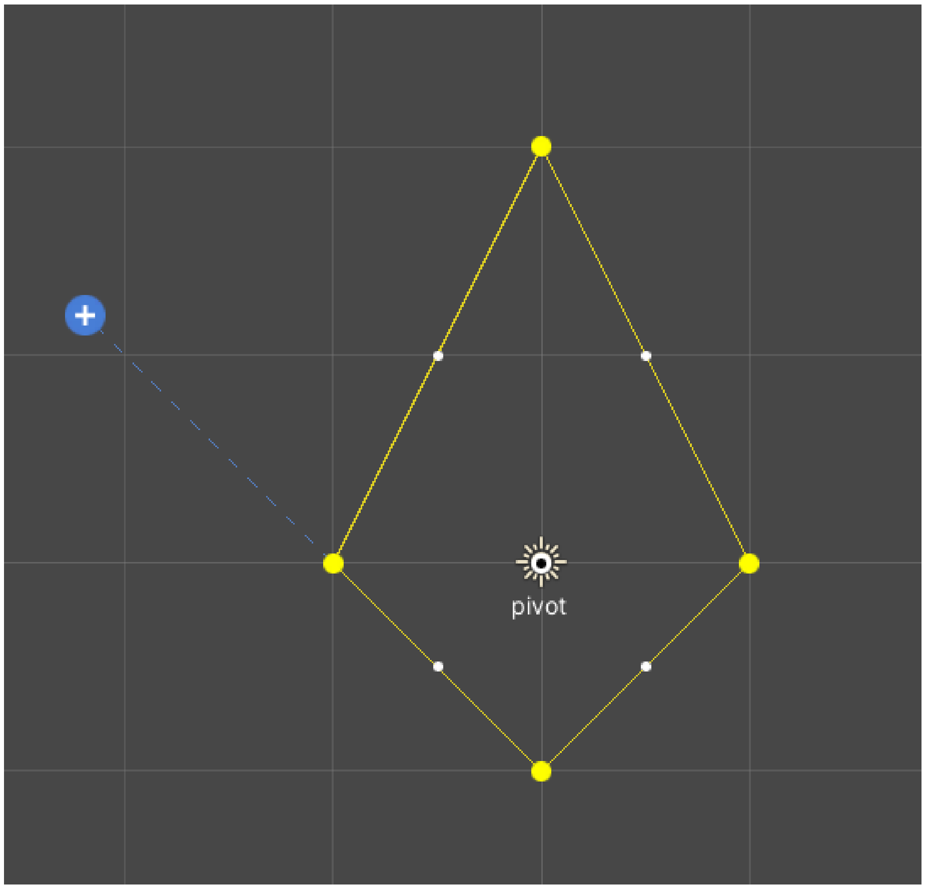
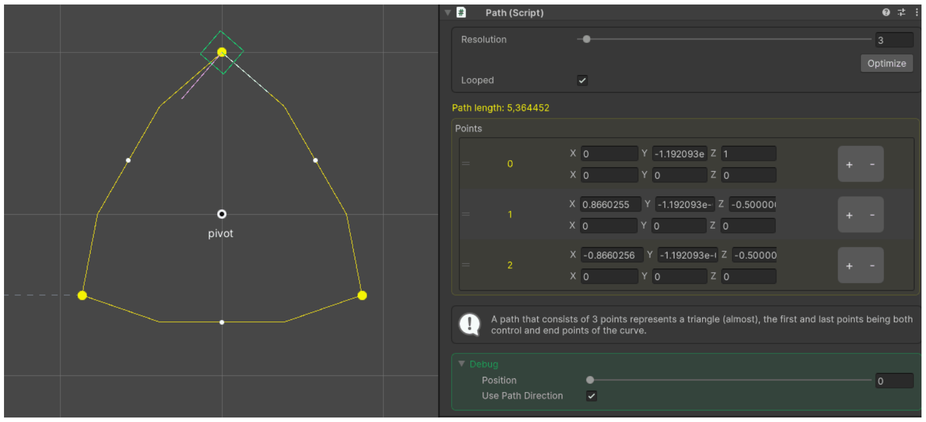


Figure 46: image



Path (Script)

Resolution 3

Optimize

Looped ✓

Path length: 5,364452

Points

```
0   X 0           Y -1.192093e   Z 1
    X 0           Y 0            Z 0
1   X 0.8660255   Y -1.192093e-  Z -0.50000
    X 0           Y 0            Z 0
2   X -0.8660256  Y -1.192093e-  Z -0.50000
    X 0           Y 0            Z 0
```

A path that consists of 3 points represents a triangle (almost), the first and last points being both control and end points of the curve.

Debug

Position 0

Use Path Direction ✓

Figure 47:

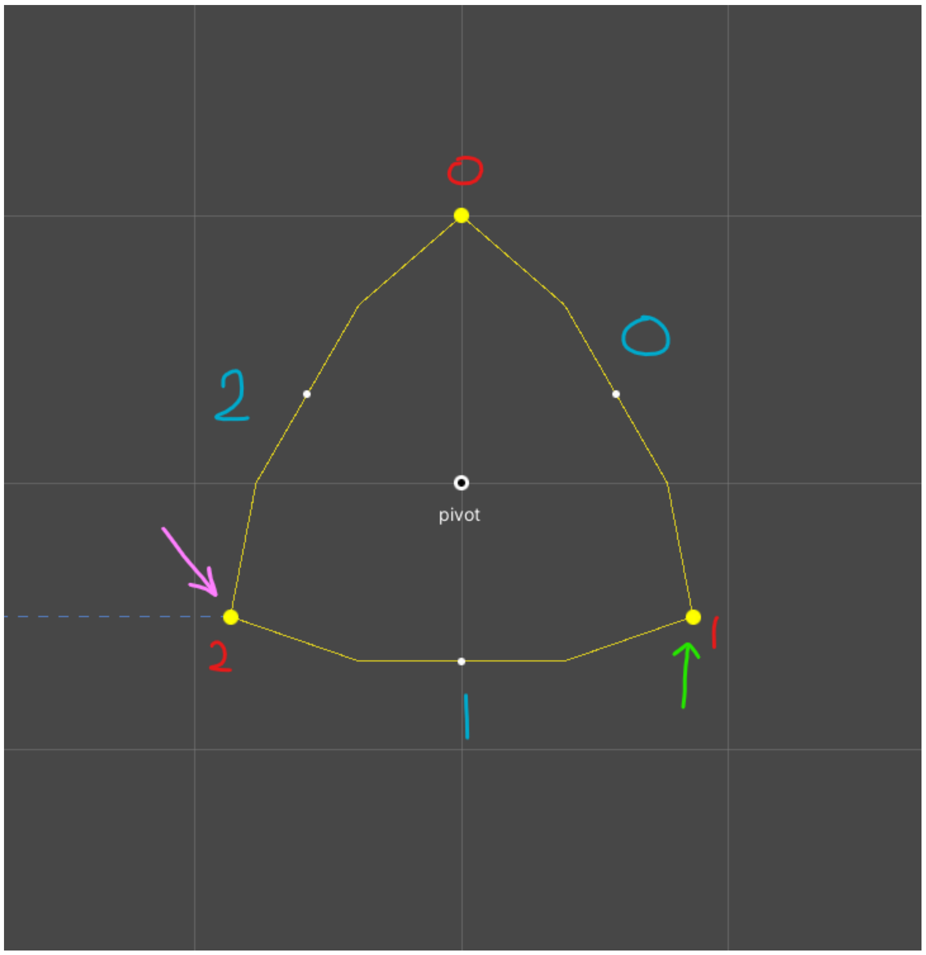


Figure 48: image

of the triangular path. Its beginning is indicated by a green arrow, and its end by a pink one. According to the problem, we need to find a point lying on the path at a distance of 1 meter from the green toward the pink. The second overloaded method `Calculate` from the list above is suitable for this purpose.

```
var data = _path.Calculate(1, 1f, false); // Calculate a point on the segment with the index

var sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere).transform;
sphere.position = data.Position;
sphere.localScale *= 0.1f;
```

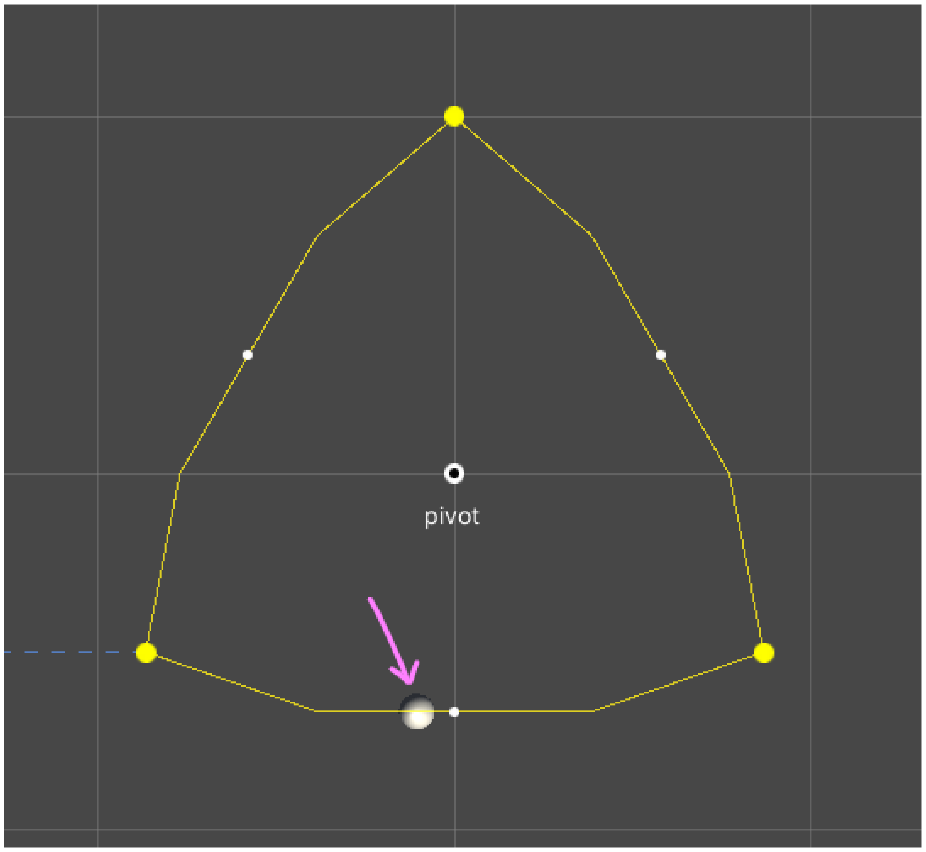


Figure 49: image

As you can see, the point has been successfully calculated.

If you need to calculate a point not on a particular segment, but on the whole path, just skip the segment index in the `Calculate` method;

```
var data = _path.Calculate(1f, false); // Calculate the point on the entire path, at a dista
var sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere).transform;
sphere.position = data.Position;
sphere.localScale *= 0.1f;
```

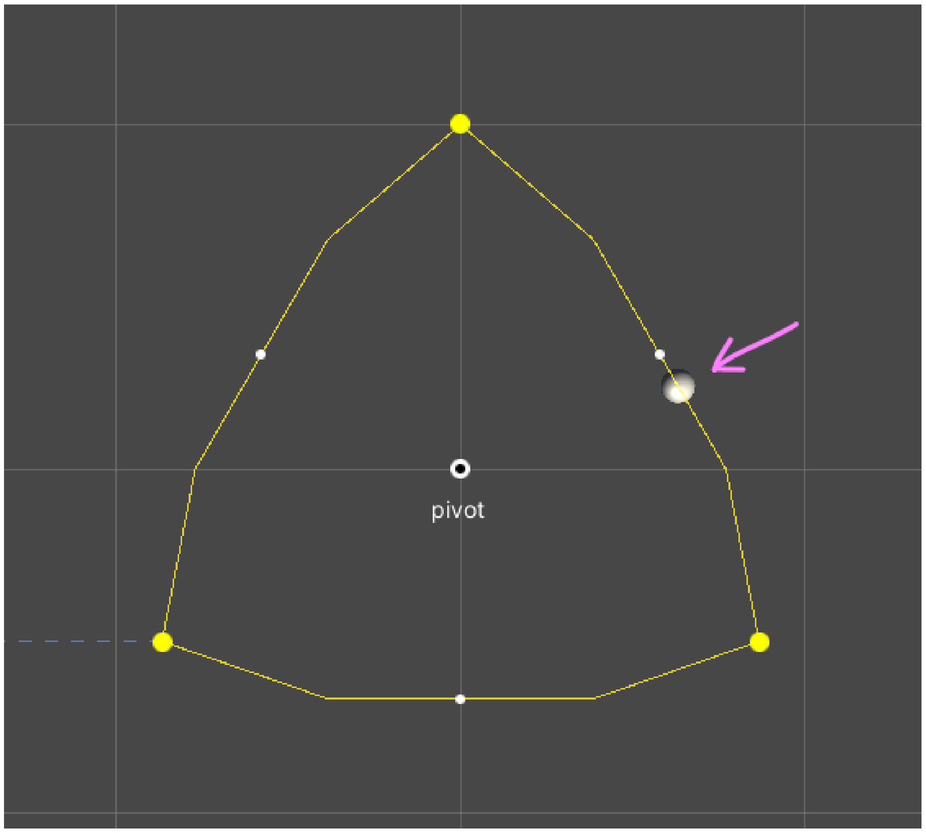


Figure 50: image

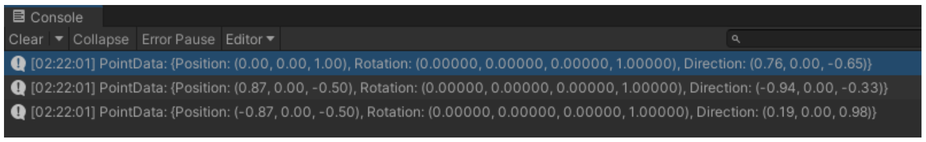
As you see now the point is calculated on the whole path. If the specified distance from the beginning is greater than the length of the path, the point will be bounded at the end of that path.

Point Cycle

To create a point cycle you can use the `PointsCount` property and the `GetPoint` method.

```
for (int i = 0; i < _path.PointsCount; i++)
  print(_path.GetPoint(i));
```

Well, that's all for now! We hope you will find the PATH bibliography helpful.



```
[02:22:01] PointData: {Position: (0.00, 0.00, 1.00), Rotation: (0.00000, 0.00000, 0.00000, 1.00000), Direction: (0.76, 0.00, -0.65)}
[02:22:01] PointData: {Position: (0.87, 0.00, -0.50), Rotation: (0.00000, 0.00000, 0.00000, 1.00000), Direction: (-0.94, 0.00, -0.33)}
[02:22:01] PointData: {Position: (-0.87, 0.00, -0.50), Rotation: (0.00000, 0.00000, 0.00000, 1.00000), Direction: (0.19, 0.00, 0.98)}
```

Figure 51: image

Good luck!