

# Procedural Indicators


**Version 2.0.0**

## Introduction

Procedural Indicators is a package that generates procedural meshes for 3D UI indicators. The result of the script is a standard unity mesh with vertex colors. Shaders and materials for Built-In, URP, and HDRP are provided but that does not stop you from applying your own materials, lightning, and post-processing on it. The package contains prefab examples that you can use right away by dragging and dropping them in the scene. Core features are generating 3D arrows, selections and grids, which are highly customizable.

## Introduction for version 2.0

Version 2.0 introduces SmartIndicators, a complete redesign of the original procedural indicators architecture. Old packages (ProceduralLibrary and ProceduralIndicators) are still included so that if you are already using them you can continue to do so while also using the new system. Old packages have only minor changes that enable them to work for unity 2020.3.+ versions. Smart indicators currently only support creating procedural arrows. Selection paths and grid indicators from the old package will also be transferred to the new architecture in future versions.



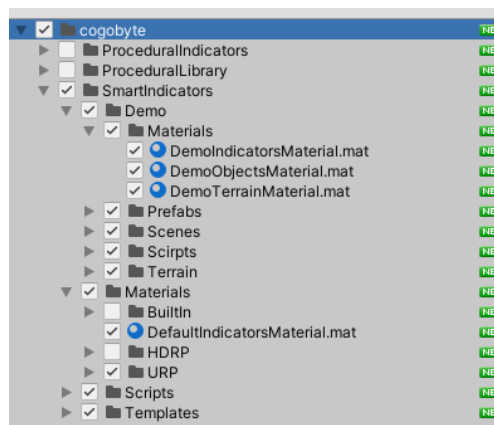
## Installation

### Importing package

You need to import the project using package manager. Old package files are deselected by default. You can include them if your project is already using the old package or you want to use selection indicators and grid indicators that aren't transferred to the new architecture yet. While choosing what to import you need to choose the render pipeline shaders for the render pipeline that you are using (BuiltIn, URP, HDRP).

### Render pipeline (BuiltIn, URP, HDRP)

During import, include the pipeline you want by choosing folder **Cogobyte - Smart Indicators - Materials - (BuiltIn/URP/HDRP)**. The standard pipeline (BuiltIn) is selected by default. After the folders have been added to the project, you need to select the DefaultIndicatorMaterial and switch to the shader you want to use.

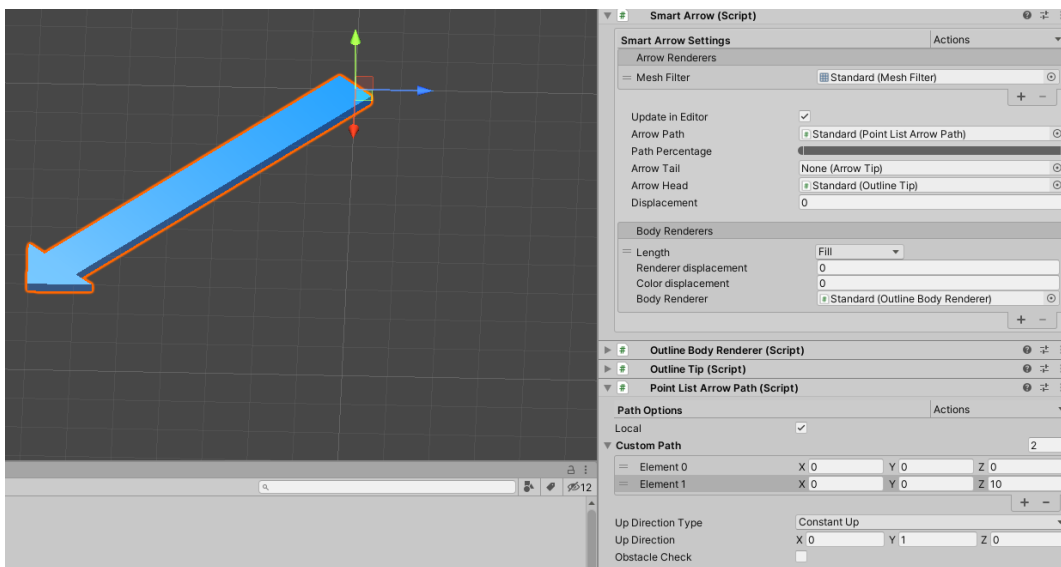


Package import options

In order for for demo scene to work you need to set the DemoIndicatorsMaterial to the BuiltInIndicatorShader/URPIndicatorShader/HDRPIndicatorShader, DemoObjectsMaterial to the default unity lit shader of the render pipeline and DemoTerrainMaterial to the default terrain shader of the unity render pipeline.

## Quick start

Drag and drop any prefab from **Cogobyte - SmartIndicators - Templates - IndicatorPrefabs** to the scene. You can use them right away. Each prefab comes with a main **Smart Arrow script**, a default **arrow path script** that defines the path through which the arrow will pass through and several optional **renderer scripts** for body and arrow tail and arrow head. The default path for the templates is the Point to Point Arrow Path. You can switch to other paths by removing the path script and in the Smart Arrow Script inspector options choose **Actions - Add Path** and choose the path you want to use. Available paths are Point To Point, Point List, Circle and Bezier.

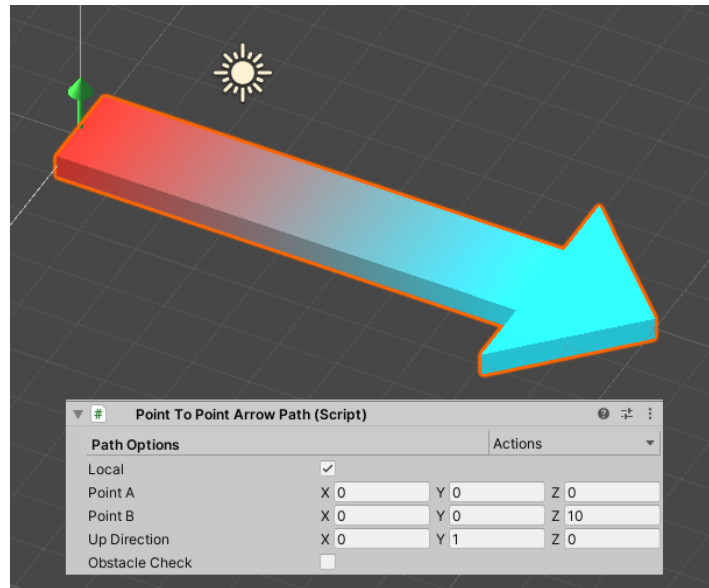


Simple arrow prefab

You can also view the ShowCaseDemo scene which shows how to use procedural indicators in various situations. Everything (including the handles) you see in the demo is created using procedural indicators. Each example has its own script so you can see how to edit the indicators using code and how it looks in the editor.

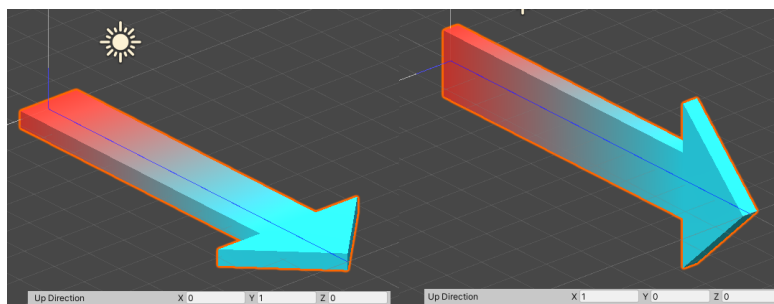
## Arrow paths

### Point to point arrow path



Point to point arrow options

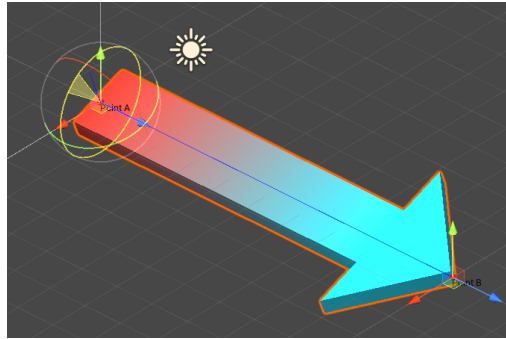
This path will render the arrow from vector field **point A** to vector field **point B**. Parameter **local** determines whether the specified points use the local or world coordinate system. **Up direction** is used to determine the roll around the path direction. The image below shows the arrow with Y up direction and X up direction.



Up direction set to world up and world right

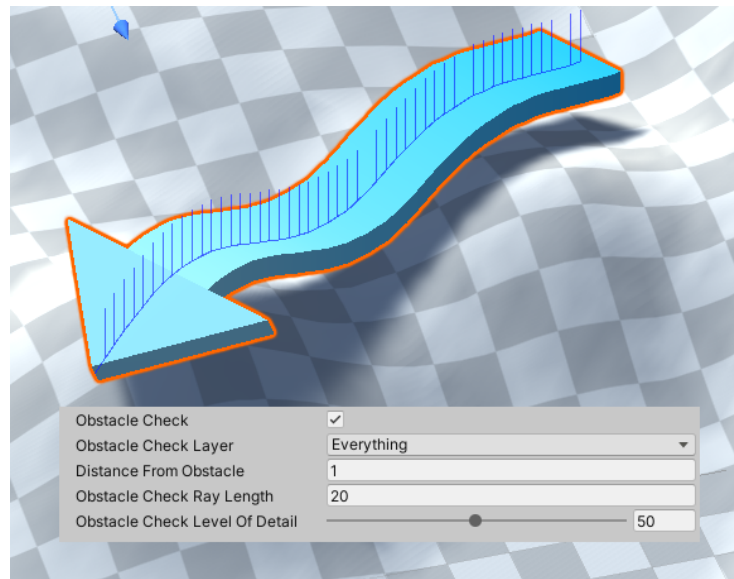
There are two gizmos to drag the pointA and pointB around and one gizmo to roll the up direction. A small blue line shows the up direction of the path and the final calculated path. You

can turn off control and path gizmos by choosing **Actions - Hide Control Gizmos and Hide Path Gizmos**.



Gizmos on the pointA and pointB

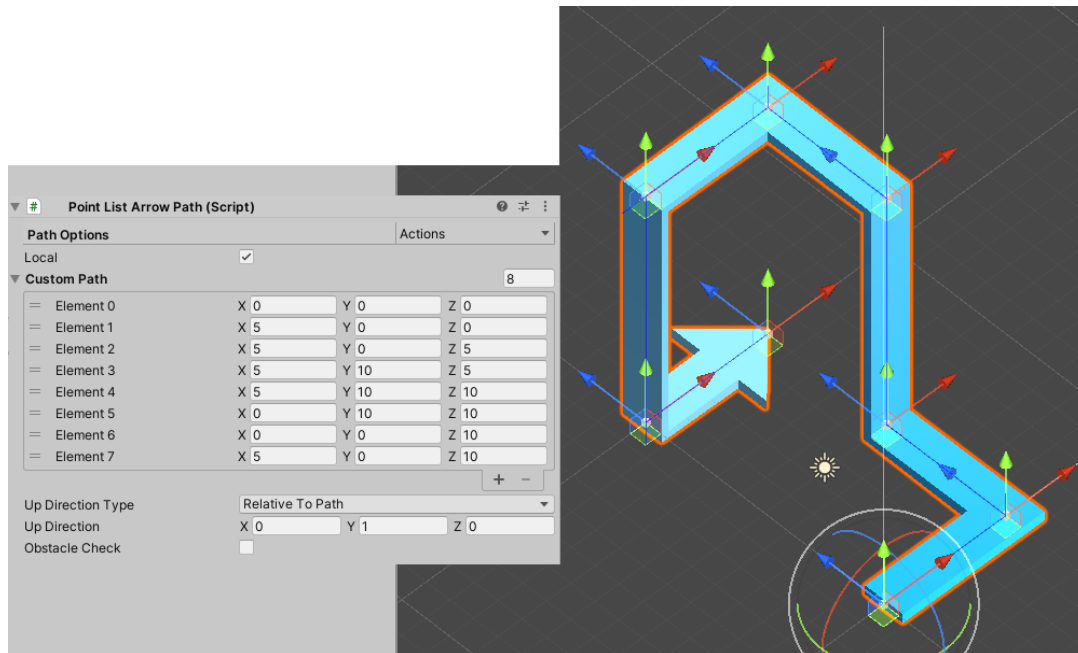
Each path has a mechanism that allows projecting the arrow above other objects. If **obstacle check** is turned on the arrow will project onto all objects with chosen **obstacle check layer**. **Obstacle check level of detail** determines the number of points for the obstacle check. Each point casts a ray of **obstacle check ray length** downwards. If the ray hits anything with chosen layers the point will be moved **distance from obstacle** length above the hit point.



Obstacle check options

## Point list arrow path

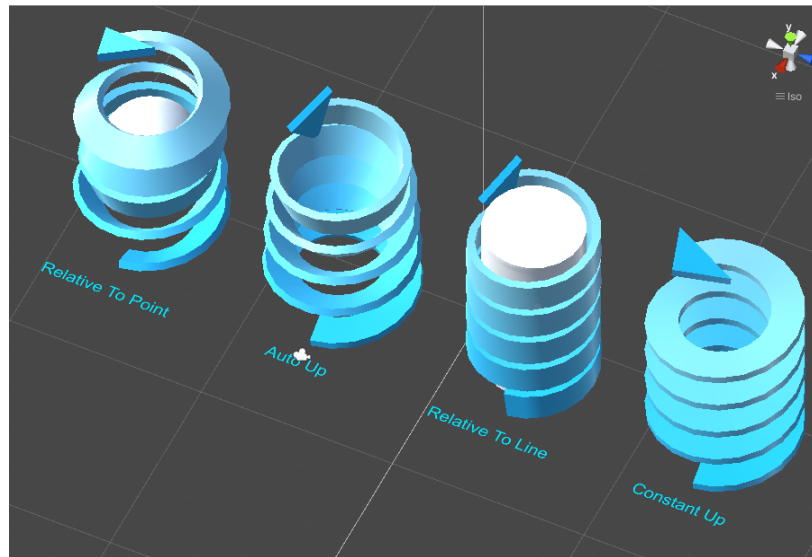
Point list arrow path contains a list of points called **custom path** that the arrow will pass through, jumping from one point to the next.



Point list arrow path options

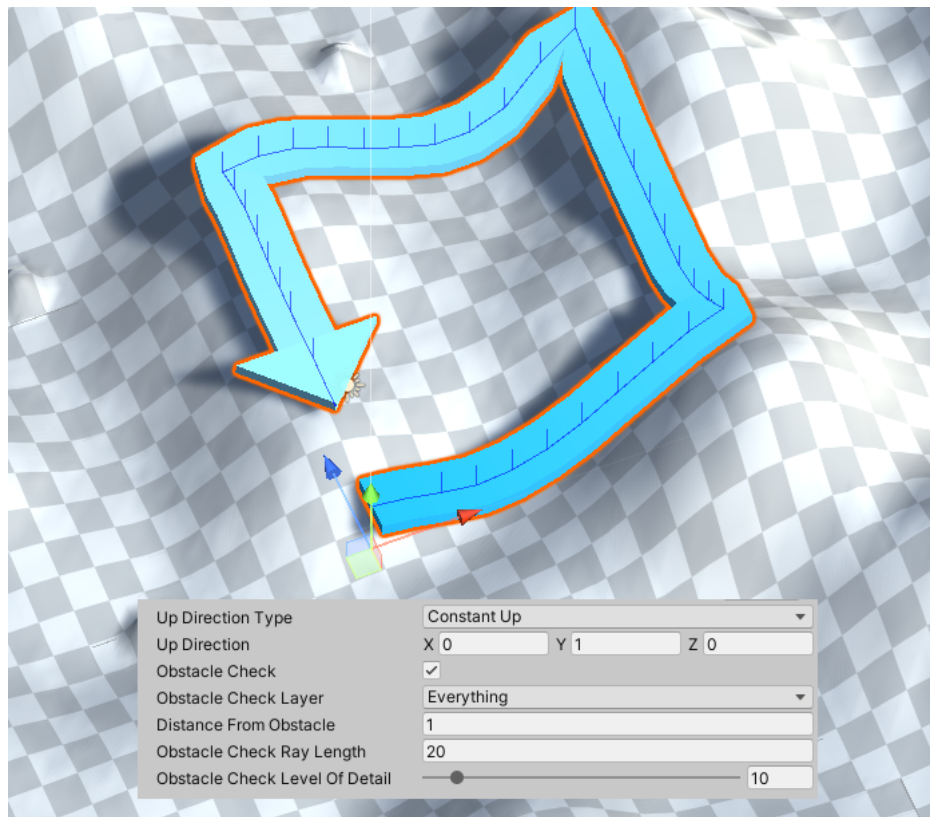
Up direction now has several modes:

- **Relative to path mode** will adjust the up direction whenever the arrow makes a turn based on the previous up direction.
- **Constant mode** will keep the same specified up direction whatever happens except when the arrow goes in the same direction as the up direction (it will be perpendicular instead).
- **Relative to point mode** is used when the arrow is circling around a sphere (Earth globe for example). It will always keep the up direction in the opposite direction of the reference point.
- **Relative to line mode** is used when you are making a spiral around a line. The up direction will always go from the closest point of line to the current calculated point.



Up direction modes

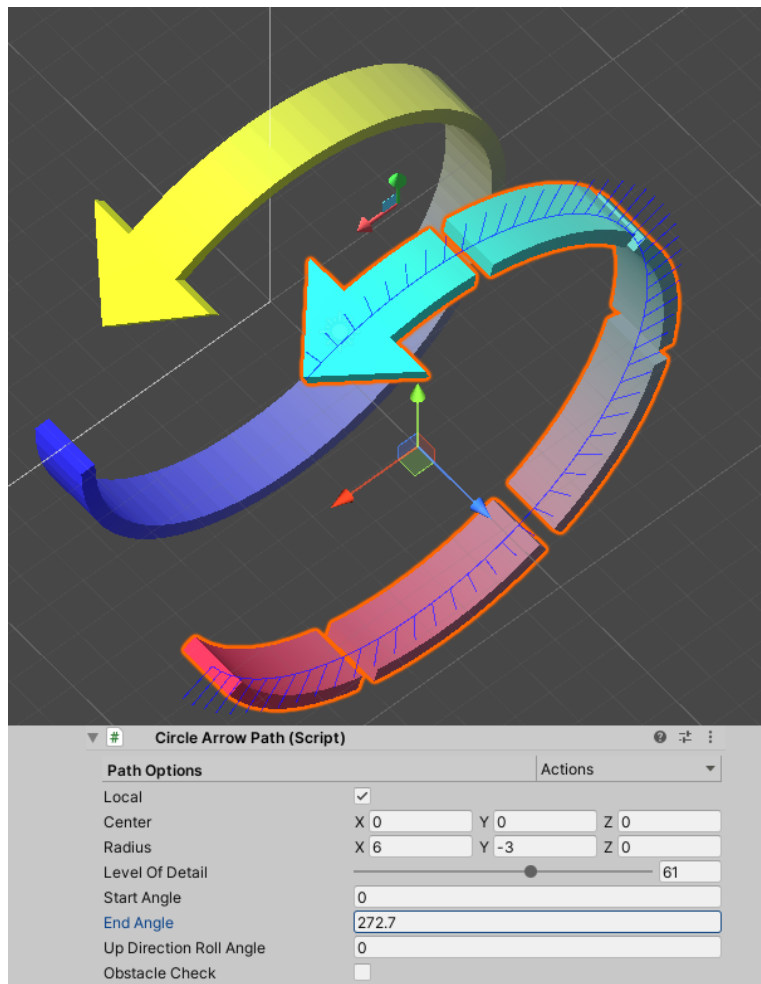
Obstacle check works the same as for the point to point obstacle check.



Point list obstacle check

## Circle arrow path

Circle arrow path is for calculating a circular path with a **center** and a **radius** vector. Up direction is relative to the center of the circle. It is possible to roll the up direction within the circle using the **up direction roll angle**. **Radius vector** determines the start of the circle. **Start angle** and **end angle** determine the used part of the circle for rendering arcs. Half circle arc would be from 0 (start angle) to 180 (end angle) degrees. **Level of detail** adds more points to the circle path. Around 40 points is needed for a path to be a smooth circle.

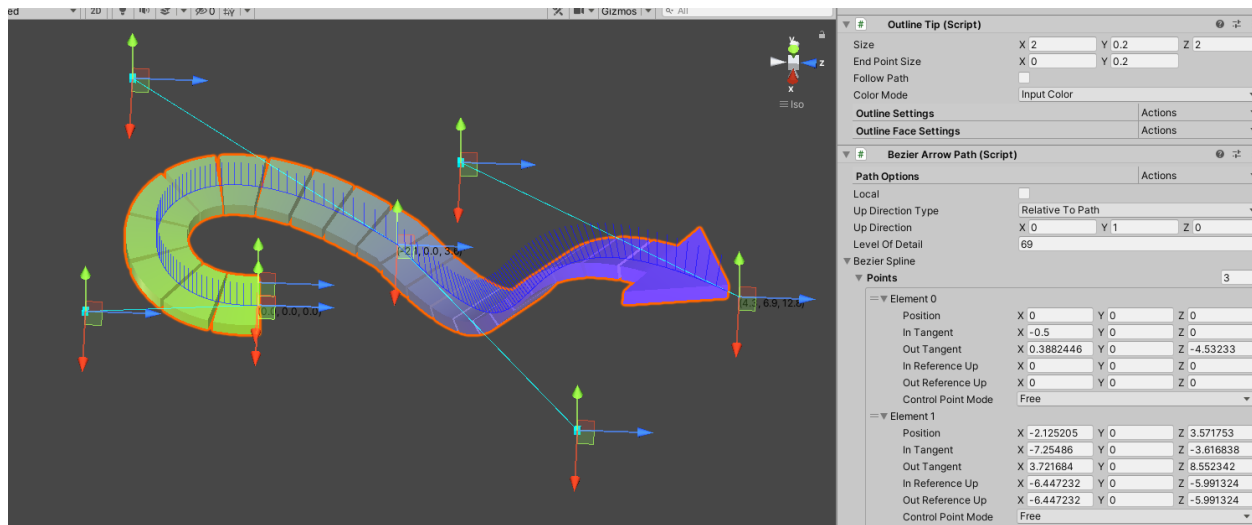


Circular arrow path



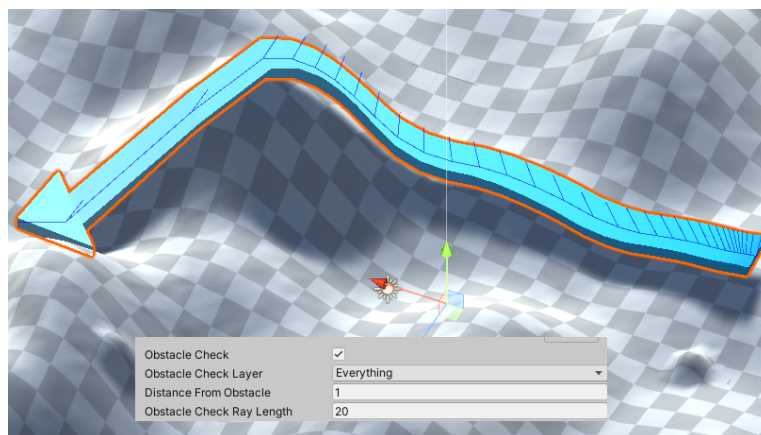
## Bezier arrow path

Bezier arrow path is composed of multiple **bezier splines**. It contains a list of bezier **points**. Two adjacent bezier points define a bezier spline.



Bezier arrow path

Each point has an **in tangent** control point and **out tangent** control point which define the in tangent vector and out tangent vector. Default **control point mode free** will not adjust control points, **aligned** and **mirrored** modes will keep the same direction for back and front control point. Mirrored will also keep the same in and out tangent vector length. Up direction modes are the same as the point list path except it has an additional mode called **define each point**. With this mode you can specify **in reference up** and **out reference up** for each spline point.



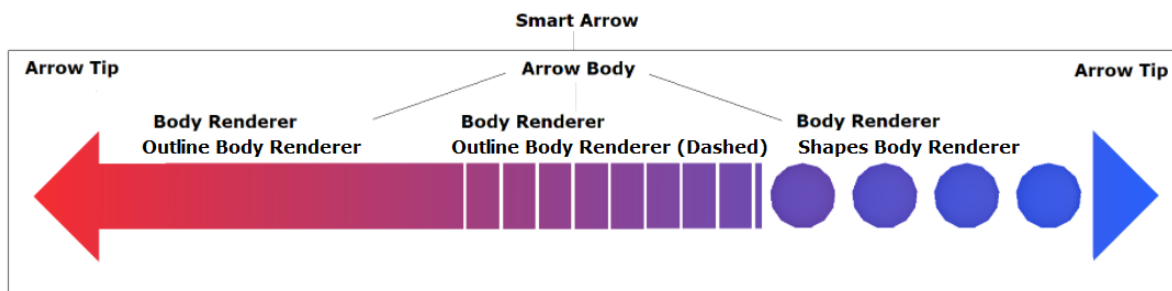
Obstacle check for bezier arrow path

## Creating the arrow from scratch

Next step is to create new arrows and customize them. Create an empty game object. Now add a **Smart Arrow** script as a new component.

### Smart arrow

SmartArrow is the main script that generates the final mesh of the arrow. It is composed of body renderers, two tip endpoints, and a path through which it passes through.



Smart arrow parts

The first parameter **arrow renderers** is a list of MeshFilter references to all mesh filters used to render the arrow. Arrow can render one or multiple standard unity meshes. The reason multiple meshes are used is to enable different materials to be used on the same arrow. You can either create mesh filters and renderers yourself or click the **Actions - Add mesh filter** option on the script and a GameObject with a mesh filter and a mesh renderer will be created and referenced for you automatically. Mesh filters can have any material attached to them, but if you want to use colors calculated by the smart arrow, you need to use a shader that uses the mesh vertex colors information. You can save the resulting meshes using **Actions - Save meshes** option. Whenever you copy paste or duplicate an arrow you need to call **Actions - Reset Mesh** to detach the reference to the old mesh. This will create a new mesh for the duplicate arrow and reference it. If you notice a weird behaviour that makes one of the copies disappear when the other copy is selected, you probably need to reset the mesh for the duplicate.

You need to choose the path of the arrow. You can add path scripts manually or use **Actions - Add Path** and then choose the corresponding path. Below the path reference is the path percentage parameter. It determines the range of the path that will be used for rendering the

arrow. This range is saved as two parameters: **start percentage** and **end percentage**. This can be used to animate the arrow's length by incrementing end percentage from 0% to 100% of the path over time to grow the arrow, or to push the arrow along a path over time (animating start percentage from 0 to 60% and end percentage from 40% to 100% will push an arrow that is 40% of path size along the path).

The next step is to choose the **arrow tail** and **arrow head** tips. Usually you don't want a tail unless you want a bidirectional arrow. If you are using tail and head, you can share the same arrow tip script for both of them. On the smart arrow script click on **Actions - Add Head** and then choose **outline tip**, **vertical Outline tip** or **mesh tip**.

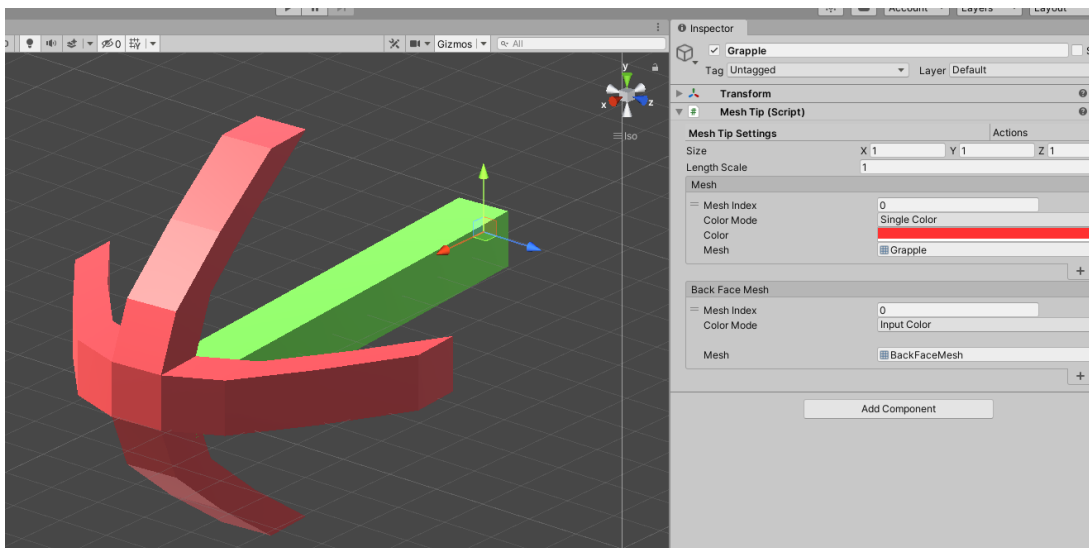
## Arrow tip

Arrow tips can be rendered on both ends of the arrow. There are three types of arrow tips: mesh tip, outline tip and vertical outline tip. All arrow tips have a Vector3 **size** parameter. The X,Y and Z components of the size are width, height and length of the tip. Arrow Path will take the length of the both tips and cut them off the calculated path and the remaining space will be used by the arrow body. If there is not enough space for the tips, they will shrink until they can fit the path.

## Mesh tip

Mesh tip has a list of **mesh items** for the tip and the back face. Back face is rendered when there is no body renderer that is touching the tip. If there is a body renderer touching the tip back face will not be rendered and the body renderer will render its own connector instead.

The size.z component is used to reserve the space for the mesh tip on the arrow path without knowing the actual size of the mesh. To stretch the tip along the length use the **length scale** parameter. This way, the mesh can point to the center of the mass of the mesh or to the end of the mesh depending on what you want to achieve.



Mesh arrow tip

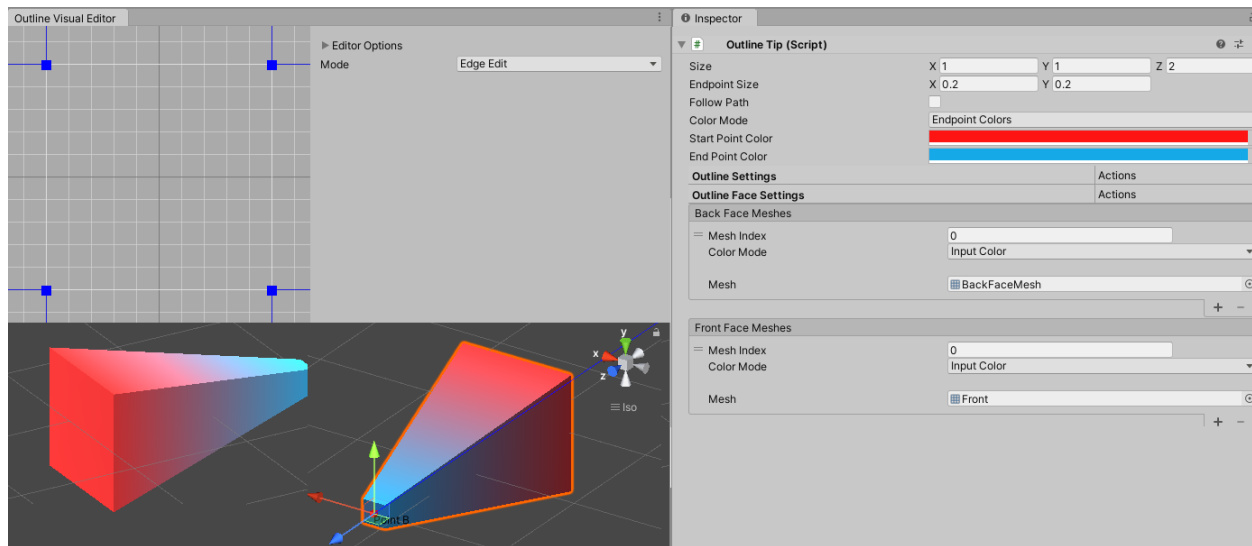
## Mesh item

Mesh item is a loaded unity mesh with additional properties and options and it is used often to configure custom meshes that are added to the arrow. **Mesh** property will load a standard unity mesh into the mesh item. **Mesh index** will add this mesh to the smart arrow renderer filter with the same index. It is used for multiple material meshes. Mesh items do not support submeshes. Submeshes need to be split into several meshes and loaded into the equal number of mesh items instead. The reason mesh items option is always a list is to support different materials. For the mesh item there are 3 color modes:

- **single color** which will add same color to all vertices,
- **vertex colors** which uses unity input mesh vertex colors and
- **input color** which uses the last color of the arrow body (for color along path).

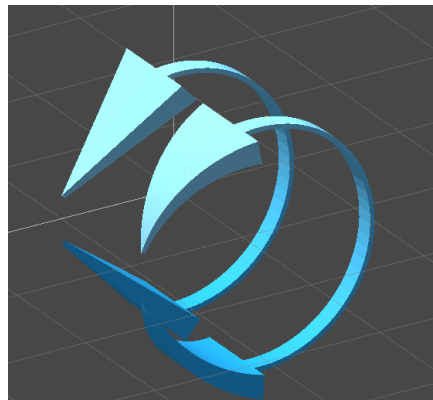
## Outline tip

Outline tip extrudes an outline from the start to the end of the tip. **Size** determines the width, height and length of the tip while **endpoint size** determines the width and height of the outline on the end of the tip. Color for the tip will be calculated from the **start point color** to the **end point color**.



Outline arrow tip

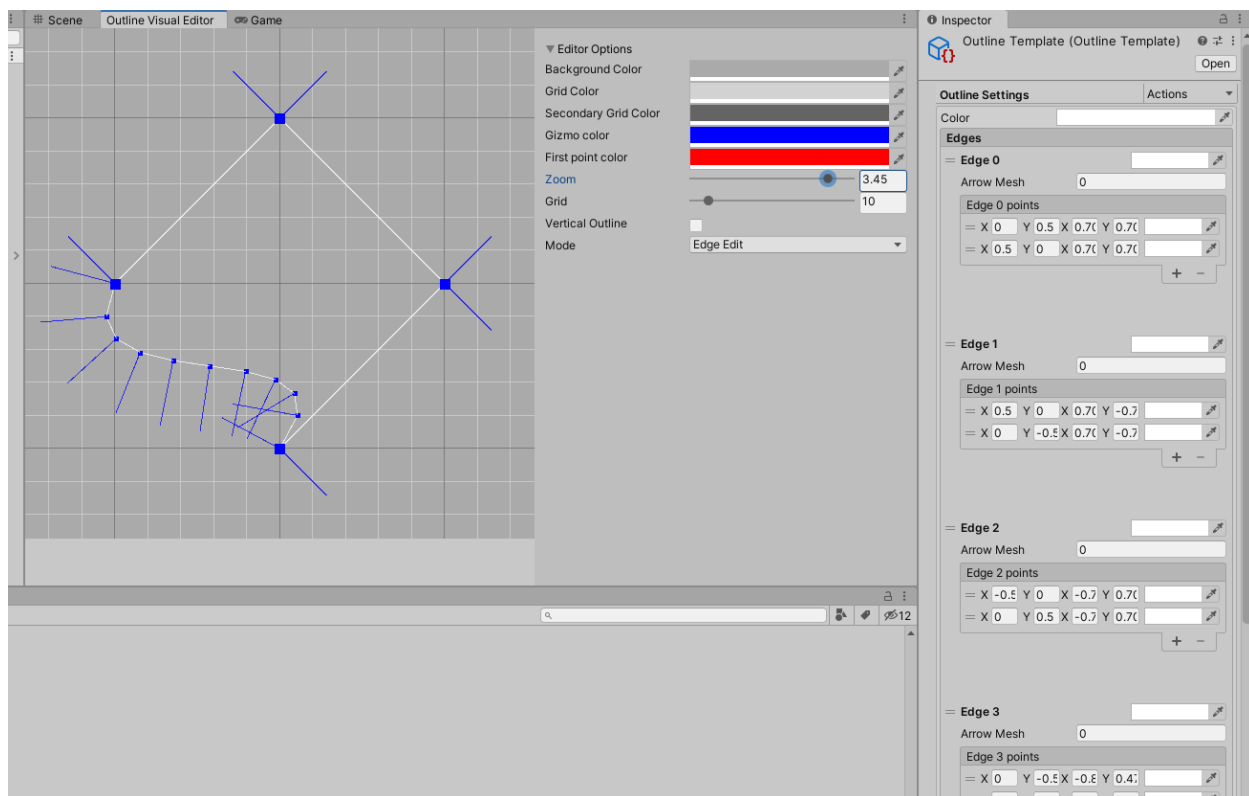
**Follow path** parameter will extrude the tip along the part of the arrow path following along if it is curved. If this parameter is set to false the tip will be rendered straight from start to end. It is mostly used when you want the tip to follow the arc of the path.



Follows path option off (left) and on (right)

## Outline

Outline is a scriptable object that is used to describe shapes used for extrusions. It can be saved or loaded using **Action - Load Outline** or **Action - Save Outline** option on the outline inspector script. You can create a new standalone outline using unity menu **Assets - Create - Cogobyte - Smart Indicators - Outline Template**. Outline is composed of a list of 2D Edges. Each edge can be in a different mesh (use different material) defined by the **arrow mesh index** parameter. This parameter needs to be the index of the arrow renderer in the smart arrow script that you want to render the vertices. Second parameter of an edge is a list of **points** of that edge. Each point has a 2D **position**, a 2D **normal** and a **color** for the vertex. Edge should be defined in a clockwise manner, or it will be extruded inverted. The reason there are multiple edges is to enable multiple outlines and sharp turns, because sharpness is defined by normals and you can have only one normal per edge point. For a square you need 4 edges and a circle can be made with only one edge since there are no points with 2 normals (no sharp edges).



Outline inspector settings

Multiple edges are also used when you want sharp color transition between vertices. For example for a circle with 2 colors (top and bottom) you need two edges each with its own color. If only 1 edge is used there is a color gradient transition between vertices.

It is also possible to specify the color of the whole outline and color per edge. Tip outline or body outline **color mode** will decide which of the specified colors are used.

Each outline has a front and back mesh that is of the mesh item type explained previously. You can auto generate the mesh using **Outline Face Settings - Actions - Generate Front Face** or **Generate Back Face**.

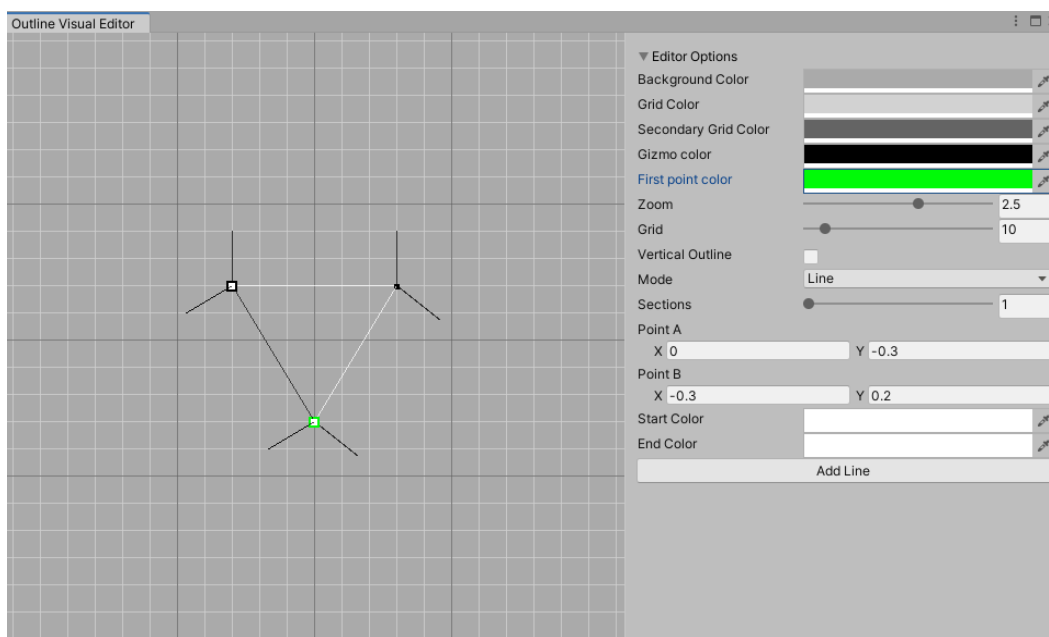
To preview and edit the outline numerically turn off **Actions - Hide Outline Settings** or to edit it with an outline visual editor use **Actions - Visual Editor**.

You can create an arrow that has 1 body and 1 tip with the same chosen outline using unity menu **Tools - Cogobyte - Smart Indicators - Generate Outline Arrow**.



## Outline visual editor

Visual editor provides a 2D editor for outlines. It is composed of two parts: the **visual editor area** that draws outline edges and gizmos and the **menu area** for input options. Visual editor area has a grid for snapping positions. You can change the grid size and zoom in and out using the **zoom** and **grid** properties. A secondary grid is drawn to show a square of size 1. Center is in the (0,0) and corners are in range from -0.5 to 0.5. It is recommended to design outlines to be within the square since you can scale the outline in arrow properties. The **vertical outline** option should be checked when editing the vertical outline which will be explained later. You can also change the color of the background, secondary grid, and gizmos to better suit your needs.

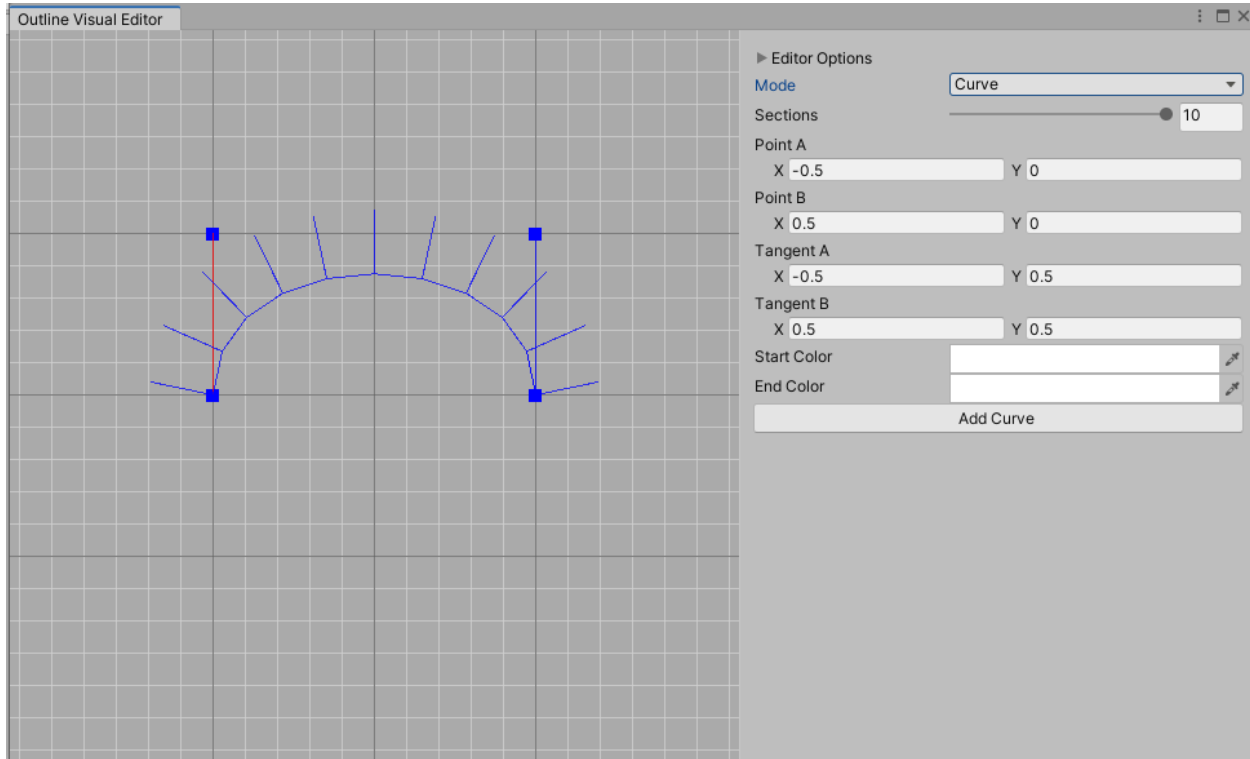


Outline visual editor

Visual editor has a **mode** for editing existing edges and modes for creating lines, curves, and circles.

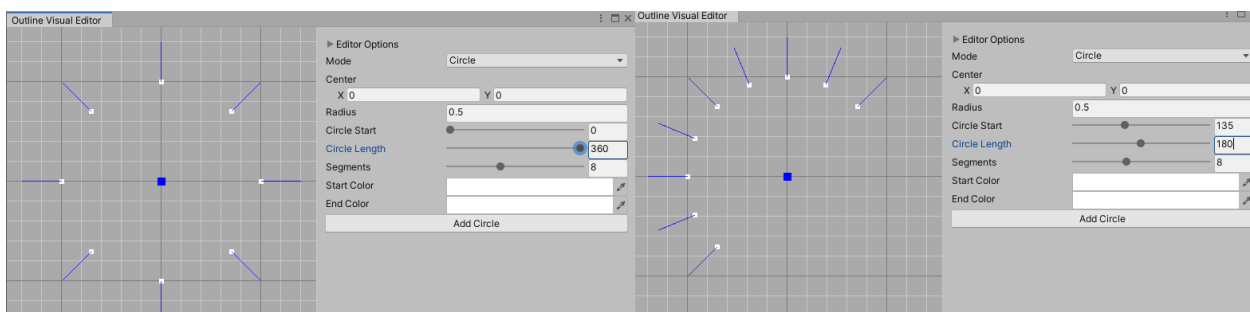
**Line mode** is for adding lines from **point A** to **point B**. **Sections** parameter splits the line to multiple vertices that have a color from the **start color** to the **end color** along the line. Point A is painted with the first point color so it's easier to keep track of what is point A and what is point B. Pressing keyboard shortcut **P** will add the point where the mouse is hovering for faster adding of points.

**Curve mode** is for adding bezier curves from **point A** to **point B** with two control points (**tangent A** and **tangent B**). **Sections** and **color** parameters work the same as for line mode. If you rotate the curve the control points will also rotate preserving the shape of the curve.



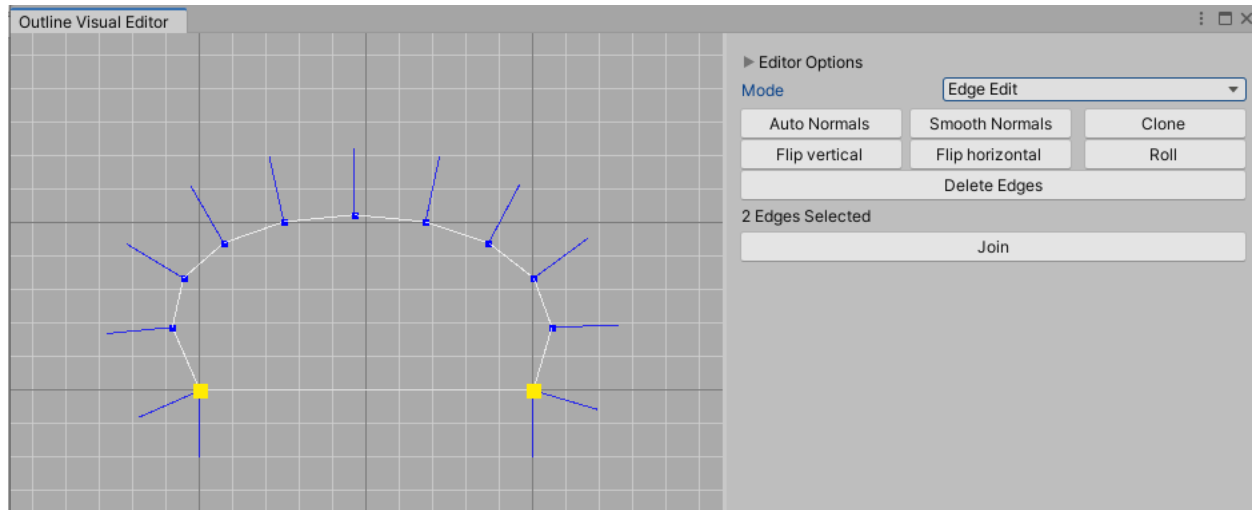
Curve mode

**Circle mode** is for adding circles with parameters **center** and **radius**. **Circle start** determines the start angle in degrees of the circle and **circle length** defines the angle from start to end of the circle and can be used to make arcs.



Circle mode

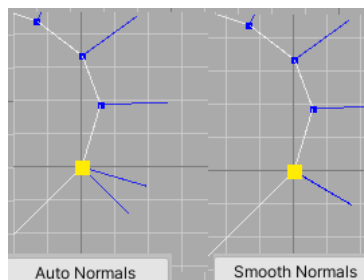
**Edge edit mode** is for selecting and editing edges.



Edge edit mode

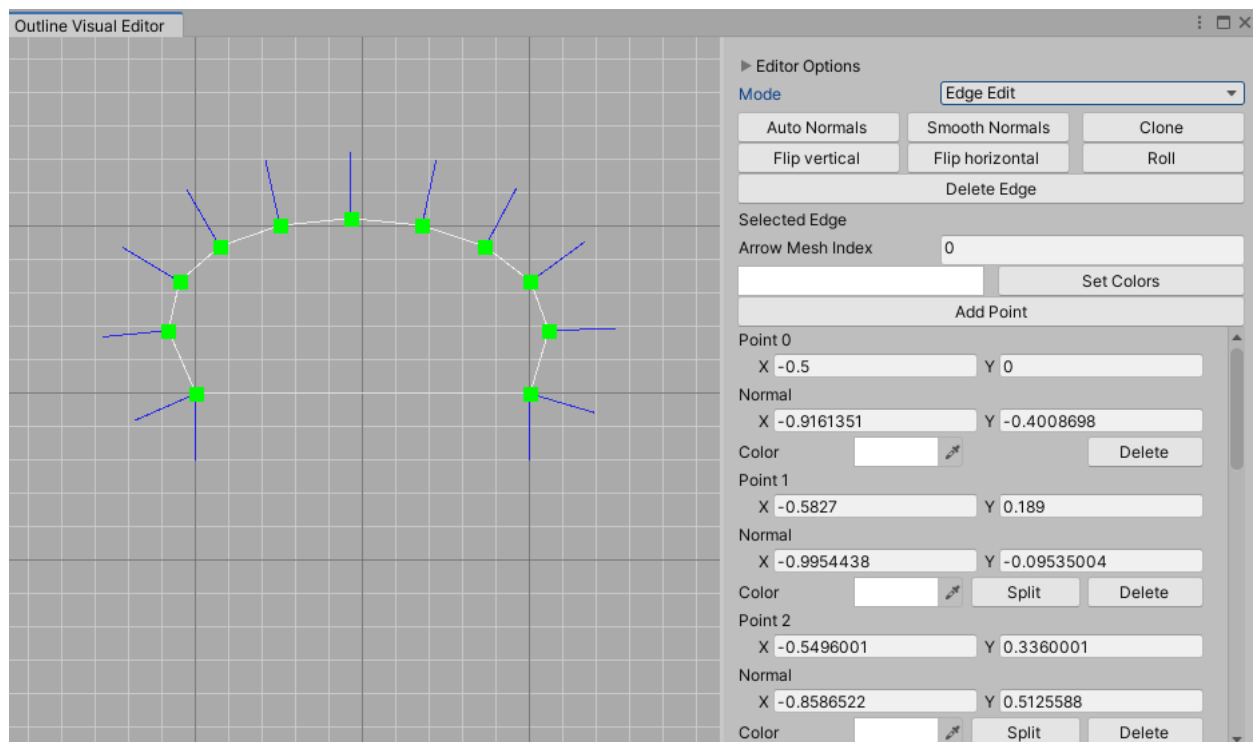
Selecting multiple edges using edge edit allows you to **join** the edges into one, **flip the edges vertically** and **horizontally**, **swap** the edges x and y which has a 90 degree **roll** effect around the center, duplicate the edges (**clone**), **delete** the edges and **calculate normals**.

**Auto normals** will calculate normals for each edge and **smooth normals** will make smooth points between edges that share a point.



Auto and smooth normals

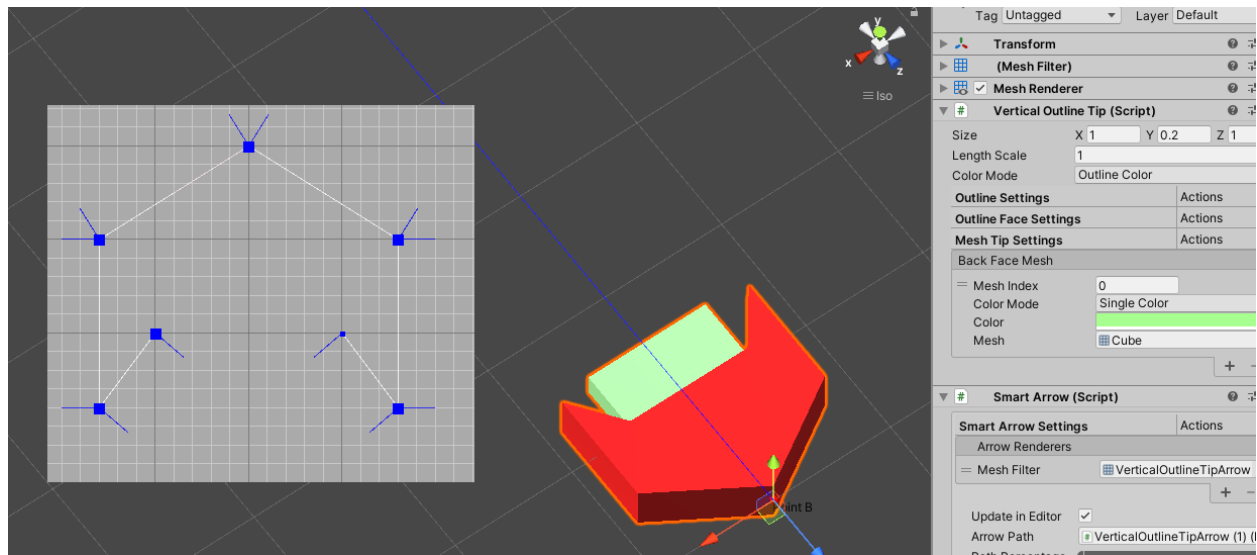
Selecting a single edge will give you a **list of points** of that edge and a button to **add new points** or change the **mesh index**. You can move around each point by dragging it. You can specify **position**, **normal** and **color** for each point individually. The point can also be used to **split** the edge into two edges.



Single edge edit mode

## Vertical outline tip

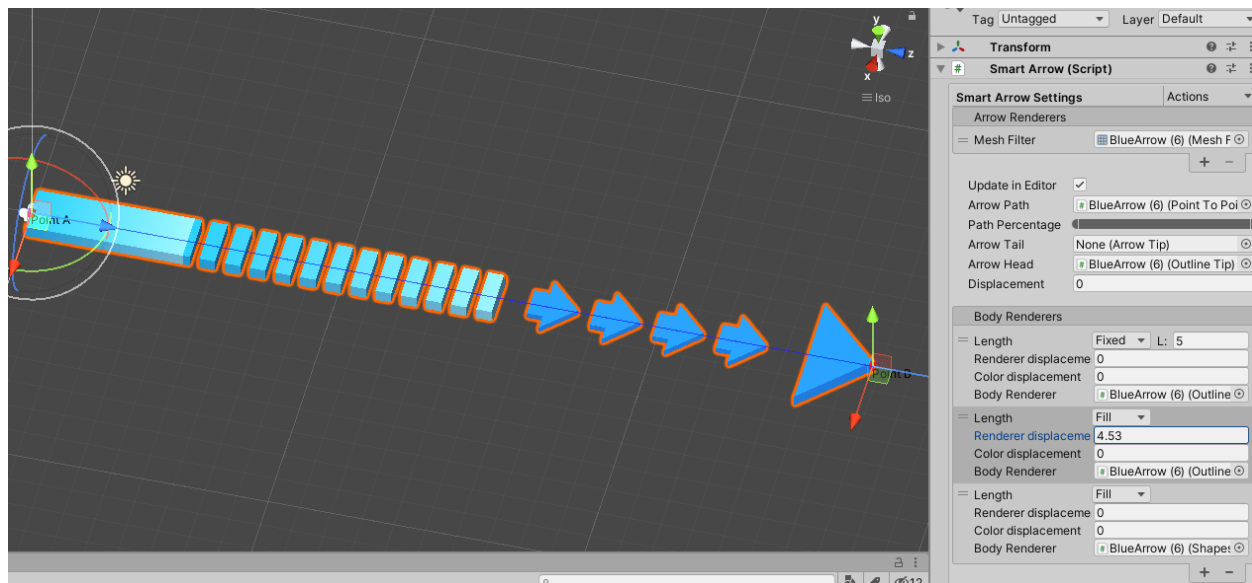
Vertical outline tip renders the outline along the Y axis of the tip (upwards). **Length scale** acts the same as for the mesh tip. It uses the same outline like outline tip with one difference: The outline should start at Vector2(-0.5,0) and end at Vector2(0.5,0) so it is easier to connect it with the outline of the body. Checking the Vertical Outline option mentioned previously in the visual editor will move the grid to a Y range from 0 to 1 instead of -0.5 to 0.5 for easier editing.



Vertical outline tip

## Body renderers

Last parameter of the Smart Arrow script is the list of body renderers. Each body renderer has its own **length** which can be defined using **fixed** values, **percentages** of the path or **fill** which fills the remaining space of the path with that body renderer. If there are multiple renderers with fill option the remaining space will be distributed evenly. If there are no body renderers with fill mode, body renderers will loop again from the first path.



Multiple body renderers

**Displacement** is calculated for each body renderer. It offsets the whole body, which is used for dashed and shape body renderer to animate movement of shapes along that path either forward(+) or backward(-). Displacement can be set for all body renderers at once and for each body renderer individually. **Color displacement** acts the same but for colors along the path.

Each body renderer in the list should have a body renderer script assigned. If the body renderer is not assigned, that part of the path will render empty space. Available body renderers are the **outline body render** and **shapes body renderer**. You can either create a script and reference it or you can do this automatically using **Actions - Add Body Renderer** then choose the body renderer, and smart arrow will create the arrow body renderer and reference it automatically.

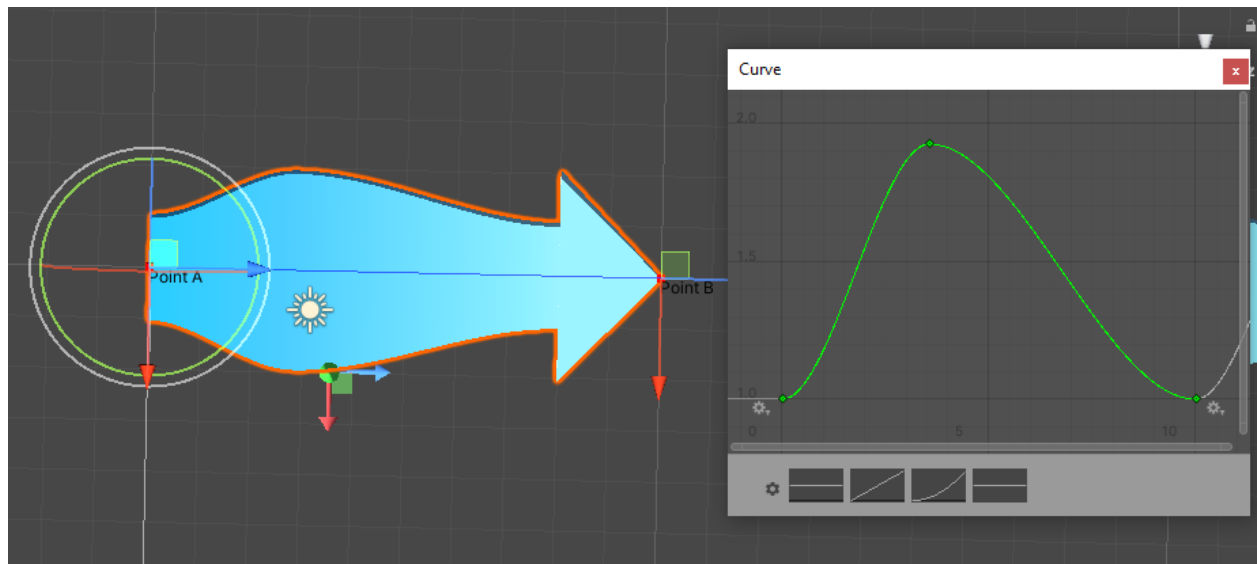
## Body renderer

Body renderer adds vertices and triangles related to the body of the arrow. Currently available renderers are outline body renderer and shape body renderer.

### Outline body renderer

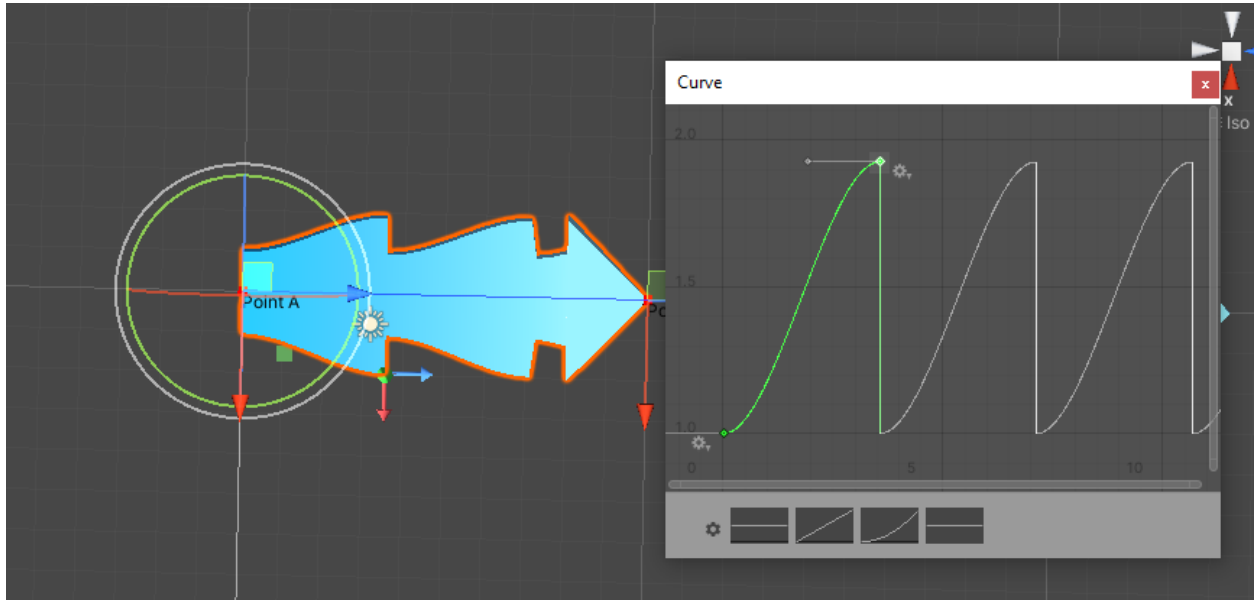
Outline body renderer extrudes an outline along the path. Displacement has no effect if the **dash** option is turned off since the same outline is on the whole path. It uses the same outline as outline tip which can be loaded or saved to .asset object. Outline back face and front face are used as front and back cap during extrude. First parameter is the width and height multiplier of the outline. It is best to define the outline in 1x1 space and then change the width and height parameter if you want a wider or higher path. Arrow can change its **width**, **height**, **color** and **roll** along the path. **Width**, **height** and **twist** functions are used to change the width, height and roll along the path. There are three modes for these functions:

- **Fixed mode** will have a constant value along the whole path.
- **Function mode** uses a unity animation function. To render the function smoothly a **lod** value needs to be set higher.



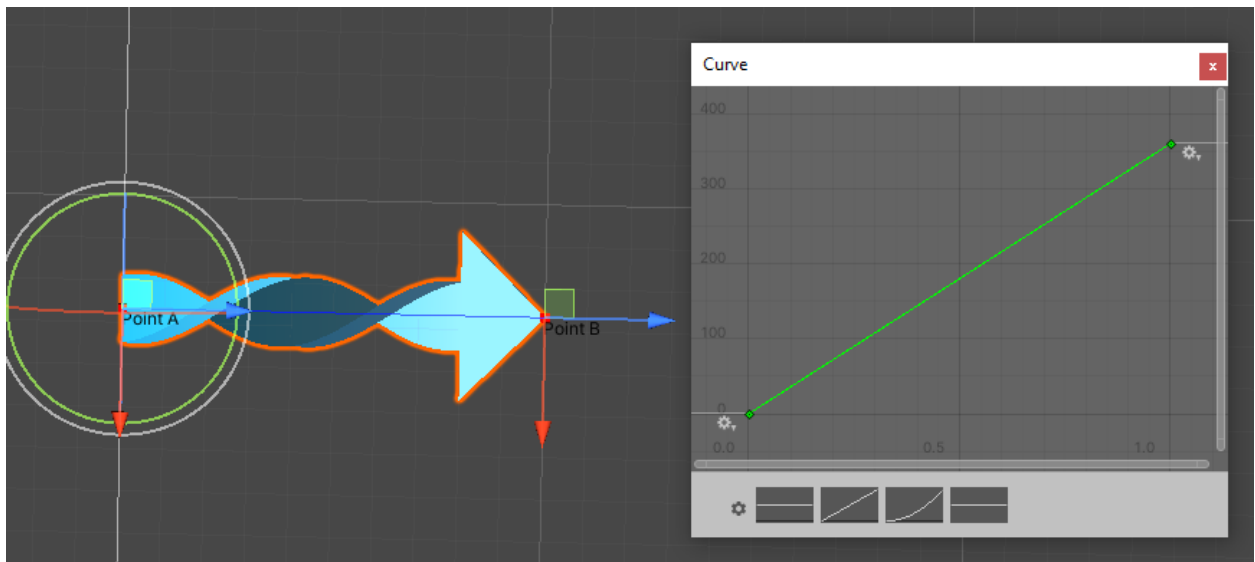
Width function mode

- **Function length mode** enables repeatable function by using exact time with the current length of the path. Function option will stretch the path to the maximum time of the function.



Width function length mode

**Roll function** should mostly use linear growth for best look. The one in the image starts from the angle 0 to 360 making a full circle along the path.

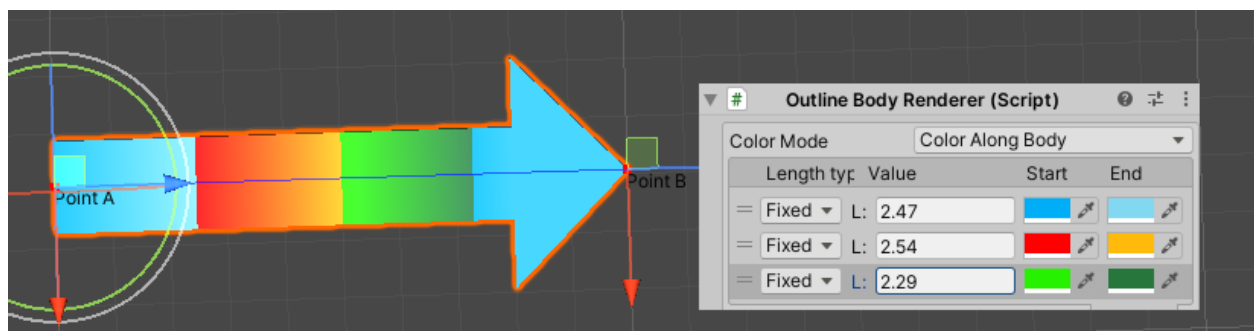


Roll function mode



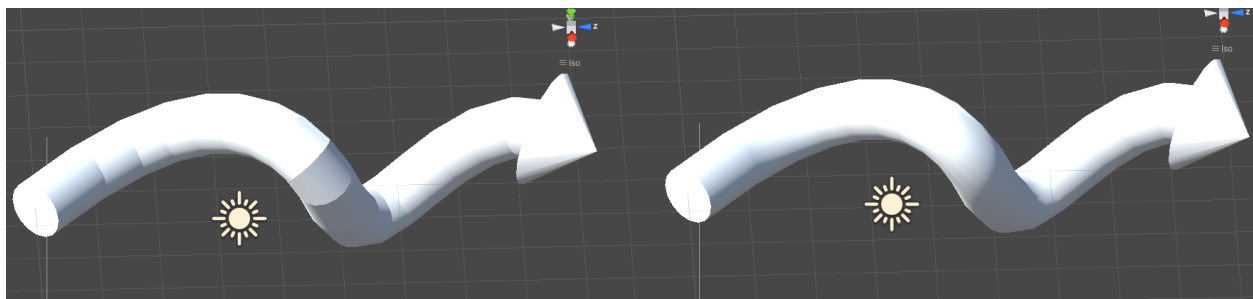
Color along the path has several modes:

- **Outline color mode** uses the outline color defined in the outline properties
- **Color per edge mode** uses outline edge colors
- **Color per vertex mode** uses the outline edge point colors.
- **Color along body mode** generates colors along the path. It can be defined using a unity gradient property.
- **Color list mode** gives more possibilities since unity gradient has a limit of 8 colors and it is possible to set the fixed lengths of the colors. Color List is a list of path colors. Each color has a **length** mode which can be **fixed** value or **percentage**. Each color item has a **start color** and **end color** for the given length. When the path consumes all colors it will loop from the first color.



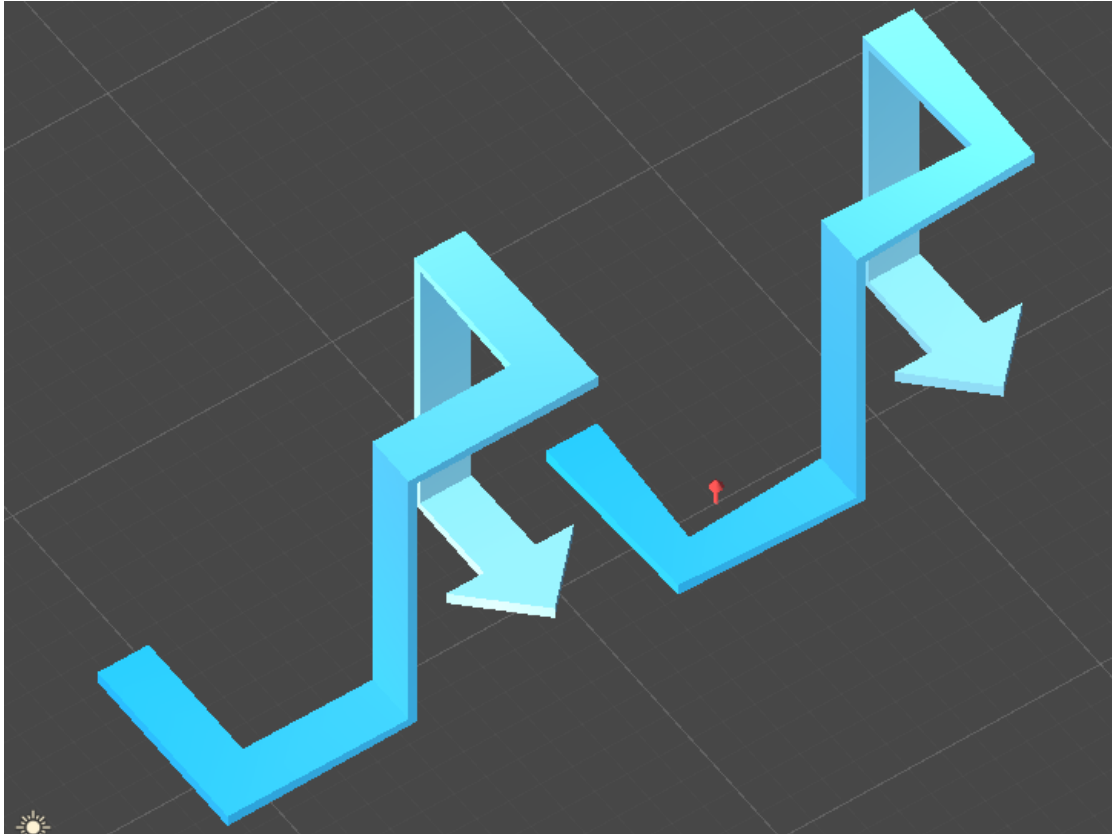
Color along body mode

**Smooth** option determines whether the path will render corners with flat or smooth normals.



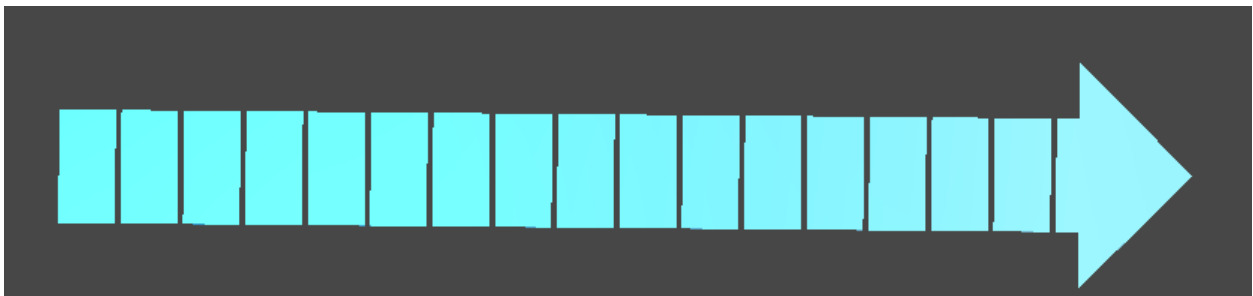
Flat (left) and smooth (right) normals

**Scale corners** will scale the outline if angles between points are sharp, meaning that if you make a 90 turn to the right up or left, the size of the arrow width will adjust so that the width looks the same in all directions.



With (left) and without (right) scale corners option

When the **dashed** option is turned on it has periodic empty spaces along the path. You can define the **length** of the dash and length for the **empty space**. Displacement will now have effect on the render by moving the empty parts according to displacement.

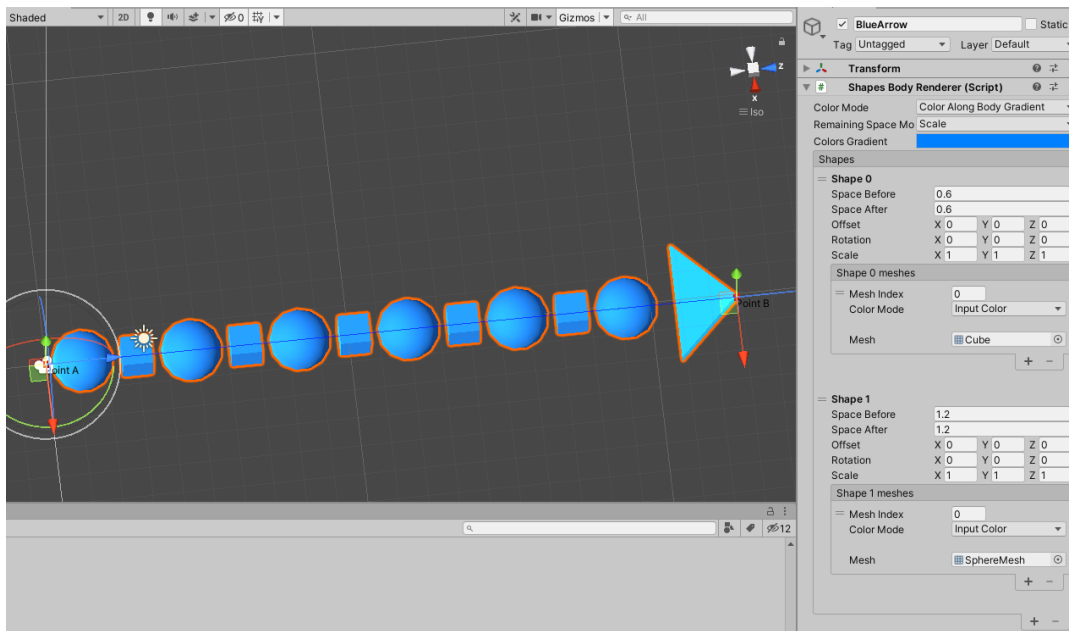


Dashed option

**Body tail face** and **Body head face** are mesh items used to connect the body and tips at the point where arrow body ends and arrow tip starts.

## Shape body renderer

Shape body renderer renders meshes along the path. It is composed of **shape items**. Each shape item has a list of **mesh items** that were previously explained. Each shape has empty space before and after the shape center. Each shape can be **offset** from the center, **rotated** and **scaled**. Shape body renderer can also have a **color along the path** like the outline body renderer.



Shape body renderer

Keep in mind that mesh is rendered at a point and space before and after should be equal or larger than mesh Z axis negative (space before) and positive (space after) size. Displacement of the body renderer will move the shapes along the path forward or backward. If there is not enough space for the shape it will scale according to the remaining space mode.

- **Scale mode** will shrink the shape relative to the remaining space
- **Hide mode** will not render the shape if there is not enough space to render it fully
- **Render mode** will render the shape in full scale even if there is not enough space. This should mostly be used if there is no arrow tip.

## Manipulating via script

You can manipulate all previously explained parameters to animate the arrow over time or to make a more complex value. Just be sure to update the arrow wherever you change something.

## Manipulating via animation

It is best to animate the arrow using scripts as you have more control over the arrow update and it is easier. You can also animate the arrow using the unity animation system by changing values over time. You just need to update the arrow every few frames depending on the speed of the change. The **ArrowUpdate** script will do that for you, you just need to assign it on the same object where the smart arrow script is located.

## Performance considerations and update modes

When selecting an arrow with a high count of path points, your framerate might drop. This is because the arrow is updated each frame while selected in the inspector with update option plus rendering all the gizmos for points takes a lot of the framerate plus unity inspector also has a significant drop in the performance if it is rendering a list with a high number of rows (for example case 1000 path points) which takes most of the rendering time. The effect will be doubled if the arrow is selected and also in runtime mode. Simply unselect the arrow and everything will be fine, or minimize the path points in the inspector. This won't affect your final build at runtime since there is no editor script code running while playing the game.

Although procedural indicators can work with a high count of vertices it is always good to reduce the number of vertices used whenever possible. Don't use an ultra high level of detail for the path. Each path point renders an `outline.edges.points.count` number of vertices. Lower the level of detail whenever you can if you can get the same look. Don't make an outline with a very high number of vertices unless the path point count is small.

Always use the outline body renderer smooth option if you can, as it renders half the number of vertices since it doesn't need double normals for sharp edges.

Whenever a value is changed in the script you should update the arrow by calling the update functions. For increased performance there are several update function options so you don't unnecessarily do calculations.

- **Update arrow** function will update everything.
- **Update arrow** with **update path false** will only update the arrow not the path. Call this if the **path hasn't changed** since the last update.
- **Update arrow** with **update body tip data false** will not update arrow body and tip relations to the path. Call this if the **path and tip length hasn't changed** since the last update.

If there are no changes to the path or the arrow between frames, try not to update the arrow. Reducing the arrow update call to every n-th frame will give a significant performance increase while you might not see the difference visually.

## Extending the package

You can make your own paths, body renderers and tips by extending the corresponding abstract class and implementing the functions. For extending tips you must implement the Arrow Tip script, for the paths extend the Arrow Path script and for bodies Body Renderer script. Each function that you need to implement is explained in the comments for that function within the code.