Modelling and Visualizing Sweep Objects

Ahmet Ayrancioglu, Mehmet Zorlu, Wouter Haneveer

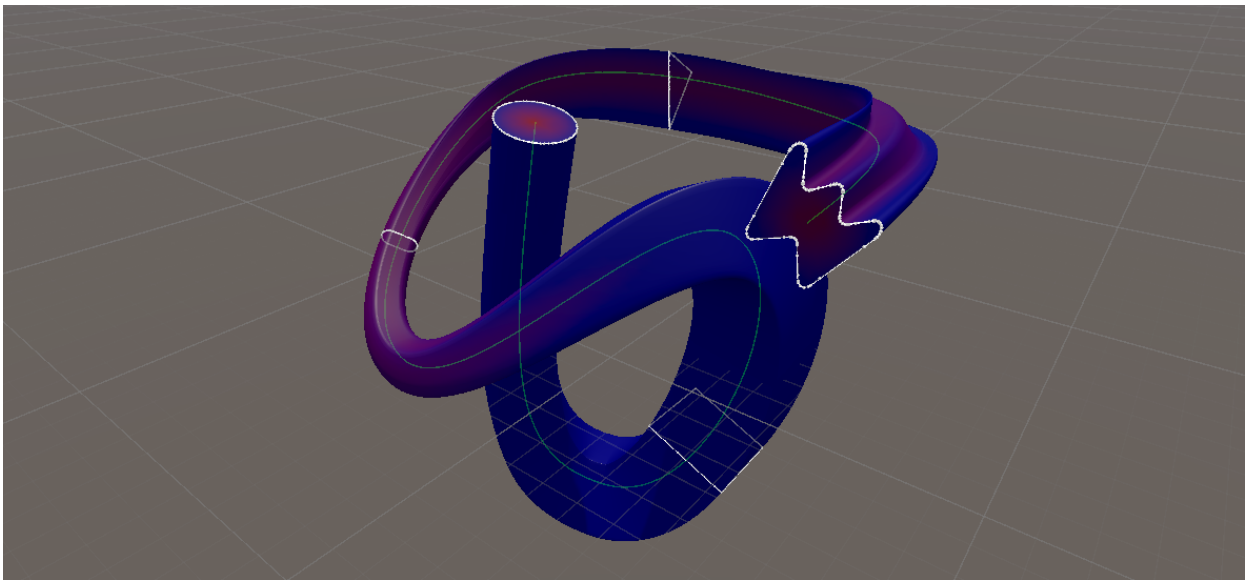

Fig. 1. 3D modelled sweep object with different cross sections that are morphed smoothly.

Abstract—In this paper we describe a Unity tool for modelling and visualizing sweep objects. To reach this goal we combine bézier curves together with shape morphing in order to create complex 3D shapes. With this tool Unity developers will be able to easily create and use complex 3D shapes in their applications. Finally we will discuss limitations of our approach.

Index Terms—Modelling, sweeping, 3D objects

1 Introduction

Both shape morphing and Bézier curves are widely researched topics, however, we still wanted our tool to be useful for some people. We decided to create a unity tool for game developers to be able to create complex 3D shapes easily. In this paper, we describe a Unity tool that can be used to create complex 3D shapes. To make our tool available for all unity game developers, we decided to implement it as a plugin, so within unity itself. It was also possible to import objects created using other software like Blender, however, for this the user would be required to first create and export the object in the other software as well as to import it into Unity. With our tool users can create the shape in Unity itself, allowing for easier modifications if the shape is not completely as expected. The tool itself is easy to use for Unity developers as the user interacts with the inspector and scene tabs built into Unity, so most of the interaction is very recognizable and intuitive for the developers.

2 MVP and the requirements

The minimum viable product (MVP) goal of this project is to achieve outputting a 3D model through the acceptance of two main inputs, a cross-section, and a sweep line. Following the MVP, there are further requirements determined to extend the functionality of the system. These are prioritized using the MoSCoW principle which uses the following keywords; MH - Must Have, SH - Should Have, CH - Could Have and WH - Won't Have. Below are the requirements that have been set during the start of the project and their state in the final product.

1. MH (Done) - Create the desired Bézier curve. Using an existing Unity compatible library for creating Bézier curves.
2. MH (Done) - Create desired cross-sections and place them on the curve in the desired position. Cross-sections can be defined using Scalable Vector Graphics (SVG) [2] paths, and the SVG file can be imported to Unity and used for the curve.
3. MH (Done) - Rotate and scale the cross-sections.
4. MH (Done) - Inspect the created object in a 3D shape.
5. MH (Done) - Interpolate between cross-sections.
6. SH (Done) - Apply various colours to the object. Any Unity material can be applied to the final 3D object, making it highly customizable.
7. SH - Clip parts of the object away by using a cutting plane.
8. SH (Done) - Apply various textures to the object.
9. CH (Done) - Export to 3D object format (e.g. obj), to image format (e.g. png).
10. CH - Import a .txt file to create the 3D shape automatically.
11. WH - Users won't be able to create branching paths.

3 Related Work

In this section, we discuss the research done before the implementations as well as various findings which influenced the course of the project. The main research is conducted on sweep objects, Bézier curves, and shape interpolation.

3.1 Sweep Objects

A sweep object is a 3D shape obtained by moving an object along a line. This can be as simple as creating a cylinder from a circle and a straight line. However, to create more complex shapes we need to be able to do more than just taking a straight line and a single 2D object to sweep along it. This includes but is not limited to: sweeping along the curved lines, interpolating between multiple cross-sections, rotating the cross-section around the line, and scaling the cross-section. One way of creating sweep objects is using volume models [5].

Sweep objects are a common tool inside modeling software. AutoCAD, Blender, and SolidWorks, for example, all have options to create a sweep object when given a line and a cross-section. This allows the users to easily create and edit smooth 3D objects without having to precisely edit each part of the object to get a smooth result.

One desirable aspect of sweep objects is that they are procedurally generated and easily editable. Designers can create shapes with precise measurements which are replicable and predictable. Another nice feature of procedural generation is that it is not destructive. Meaning that the underlying 3D object can still be changed even after layers of postprocessing steps are applied to the object, without needing to alter any layers that are applied above the base object. This makes sweep objects easy to work with and iterable.

3.2 Bézier Curves

A Bézier curve is a parametric line that is defined by a set of control points. These control points define a smooth line that is easy to use, store, and scale. See Fig. 2 for how control points affect the curve. Bézier curves are heavily used by many design software and it is almost synonymous with curves in many people's minds. Bézier curves provide an intuitive way for users to input complex curved lines that can then be processed and used for multiple purposes. One interesting post-processing done to Bézier curves that is also relevant to this project is converting a Bézier curve into an approximate polygonal chain. A polygonal chain (polyline) is a sequence of connected line segments that describes a jagged curve.

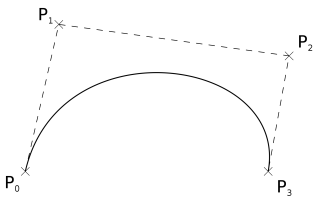

Fig. 2. A cubic Bézier curve with four control points.

There are various ways to convert a Bézier curve into a polyline. The most common approach is to use De Casteljau's algorithm [1]. The algorithm can be used to split a Bézier curve into 2 Bézier curves at an arbitrary place. It is used recursively until the control points are approximately collinear to create a polyline. A few steps of the process can be seen in Fig. 3.

3.3 Shape Interpolation/Morphing

Shape Interpolation (Morphing) is the process of smoothly transitioning from one shape into another. This process can be thought of as an animation that takes time T where the start shape slowly turns into the end shape. Then, this animation can be queried at any time t to get the intermediate object during the transition. An example of shape interpolation can be seen in Fig. 4.

To achieve a smooth shape interpolation between two cross-sections, we conducted in-depth research. In the end, we found that there are

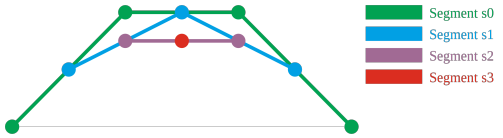

Fig. 3. Intermediate line segments obtained by recursively applying linear interpolation to adjacent points.

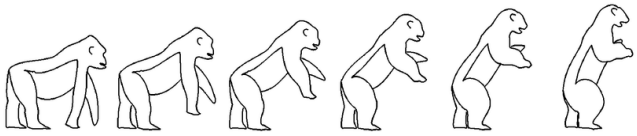

Fig. 4. Shape interpolation from an ape to a bear.

multiple ways of implementing shape morphing on 2D objects. For example, it is discussed that interpolation between similar shapes can be done easily via linear interpolation in between the coordinates of vertices [7]. However, it is also mentioned that this solution might lack in some cases and cause shortening of parts of the boundary during the transition. This means that the intermediate shapes during the transition will not be good enough to create a smooth animation.

An alternative method is to focus on interpolation of intrinsic boundary representation, meaning that instead of vertex coordinates, edge length and interior angles are used for calculations. This method aims to avoid the side-effects occurring in the initial method but it also comes with a caveat where the resulting polygon is open, so it has to be closed with additional computations which makes it questionable if it is a better method to use in our case.

4 Analysis and Solution

In this section, we will discuss the problems that we needed to solve to successfully implement the 3d Sweep Object. Additionally, we will explain in detail how we overcame the problems and which algorithmic techniques we used and implemented. On the way, we will give examples, pseudo-code, and figures to clearly describe our solution.

4.1 External Tool vs. Plugin

Right now, there are lots of tools to create complex 3d sweep objects provided by leading design software such as Blender, 3Dmax, AutoCAD, etc. These tools are extremely powerful and can meet the need of almost any designer. However, with the powerful capabilities of these tools, there comes a steep learning curve with them. Not everyone who needs these sweep objects can buy, download, or use these complex tools. We wanted to make our project as accessible and useful as possible and decided to implement it as a Unity plugin. Unity is a free game engine that is used both to create games and create product designs. With our tool, designers, and programmers can create complex 3d sweep objects without leaving Unity or being forced to learn another design software from scratch. Additionally, being tightly integrated with Unity means that people can quickly iterate over their designs and see the effects immediately. Finally, as Unity is a full-fledged 3d game engine, support for rendering, materials, textures, etc. is very good and can be used for our project too. Thus, we only needed to focus on the important parts of our project and not reinvent the wheel by doing everything from scratch.

4.2 Sweep object creator plugin for Unity

The GUI of our tool makes use of the GUI of Unity. Since our tool is a Unity plug-in we expect our users to know Unity already. Therefore, we will not explain common Unity controls and GUI elements that are used, and will mainly focus on the controls and GUI elements that are used to interact with the Sweep Object. Inside the Scene interface, as seen in Fig. 5, the user can pan around, zoom in and zoom out and rotate the camera using the same controls as always. Inside the

Hierarchy shown in Fig. 5 at the left side of the screen, the user can see all game objects currently added to the Scene. The two game objects that are necessary for our tool are the Output and Extruder game objects. The Output game object is needed to show a mesh while the Extruder game objects store the Bézier curve, cross-sections, and some other settings to create the sweep object. The Output game object exists of a Transform, Mesh filter, Mesh Renderer, and materials that are used by the Mesh Renderer. To add a sweep object to the scene the user needs to right-click inside the Hierarchy, go to the custom option inside the menu and click on Extruder. This will add both the Extruder and the Output game objects ready for use. To add further Cross sections the user can hover over the Extruder game object, right-click, go to custom and select the cross-section option which will add a cross-section to the sweep object.

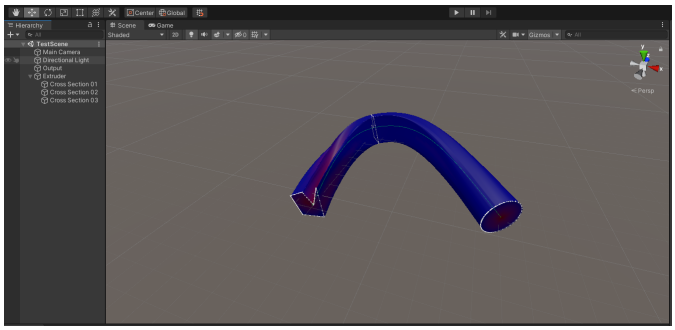

Fig. 5. Unity's Hierarchy and Scene interface, which displays the sweep object.

Most of the interaction for our tool is done by interacting with the Extruder game object. This game object contains a couple of children namely all the cross-sections that the user needs to create their sweep object. Once the user selects the Extruder in the Hierarchy a couple of things will change. First of all, the control points will appear on the Bézier curve. These are the points through which a Bézier curve will be drawn. By dragging these points around the user can change the curve and therefore the sweep object. If the user presses shift and presses somewhere inside the scene, a new point will be added to the end of the curve. Alternatively, by pressing the control key the user can add a point to the start of the curve. The user is also able to add a point between points by pressing shift and clicking on the desired place on the Bézier curve. To remove a point the user needs to press backspace while the point is selected.

Inside the Inspector the user can see the scripts that create the sweep object along with their settings as seen in Fig. 6. All of the settings under the Path Creator script belong to the creation of the Bézier curve and are explained by the creator of the asset creating the Bézier curves for our tool, S. Lague [3]. We recommend not to change any of the settings which change the Bézier curve since we found that these settings are the best while using our tool. However, under the display options menu, the user can find an option to increase the size of the control points on the Bézier curve which can make it easier to move these points around. Then, there are also some settings under the Extruder script in the same Inspector, Fig. 6. Here the user can hide the mesh of the sweep object to get a better view of the Bézier curve by deselecting the Show Mesh Renderer button. The Show Wire Mesh option allows the user to see how the triangles are formed to create the mesh. Then, there are two options for the cross-sections, the user can hide the cross-sections and add labels to the vertices of the cross-section to see which points of the cross-sections are being connected. Additionally, there are some options for the color of the mesh. The user can select two colors that are blended and applied to the mesh according to the distance between the mesh and the Bézier curve. This can scale according to the global 3d shape or can have a different scale per cross-section. Lastly, we have options to select a different interpolation curve between every two cross-sections which changes how fast cross-sections interpolate into the other cross-section. Inside this window, the user can find some standard interpolation curves however by double-clicking the user is also able to add a point to the curve to shape it to their own needs.

Then, by adding the optional Overlap Fix script the user can input an integer number for the number of iterations the overlap fix algorithm will do to try and fix any overlap in the mesh, as this algorithm changes the Bézier curve a little bit at each iteration if it finds any overlap. The algorithm can be run by pressing the Fix Overlap button. We advise the user to save the Scene before using this algorithm as it is not perfect and might ruin the shape created.

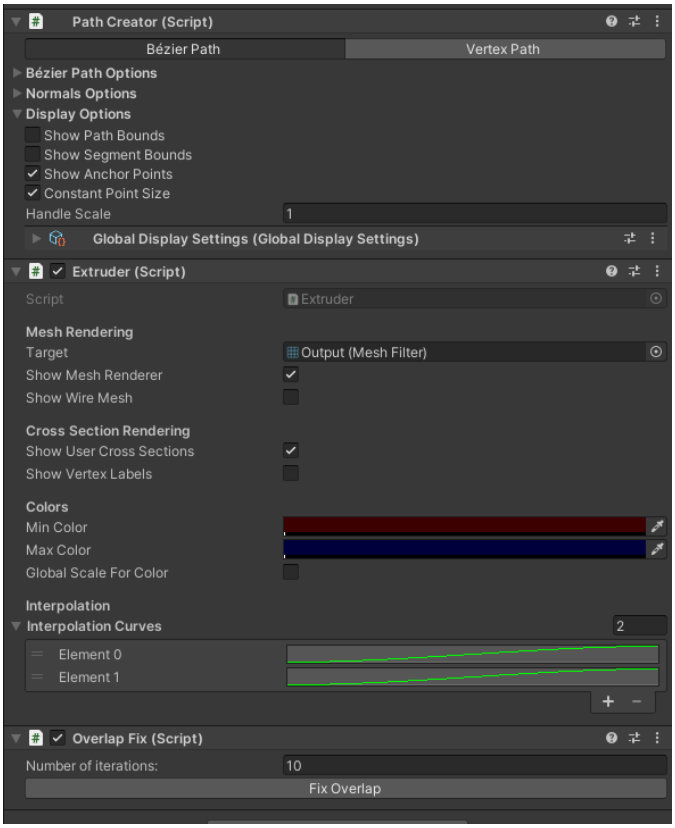

Fig. 6. Unity's Inspector interface, showing the scripts ran in the Extruder game object. The Overlap Fix script is an optional script the user needs to add themselves.

The last part of our tool that the user can interact with is the cross-section game objects. When one is selected in the Hierarchy the Inspector shows the Cross-Section script shown in Fig. 7. Here the user can change the place of the cross-section by selecting the desired T value. The user can also rotate and scale the cross-section, by changing the Rotation and Scale values. Finally, the user can change the shape of the cross-section by selecting an SVG file containing the desired cross-section.

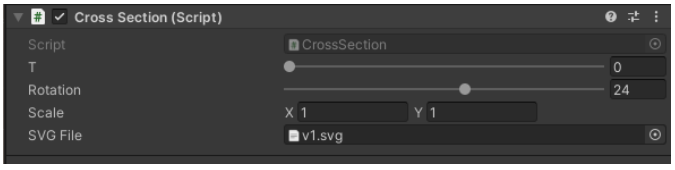

Fig. 7. Unity's Inspector interface, showing the scripts ran in the Cross Section game object.

4.3 Bézier curves

As Bézier curves are a solved problem and already widely implemented and used by many projects in many programming languages, we decided not to implement it again and use an existing Unity compatible library that provides us with the basic implementation of Bézier curves.

Bézier curves are used strictly for user interactions and cannot be used for creating the 3D object directly. We decided to use Bézier curves for user interactions as almost every design software currently available uses Bézier curves to create smooth curves and users are extremely acquainted with using Bézier curves, and we want this tool to be as user friendly as possible.

Using Bézier curves directly to create the 3D shape is not a feasible task. It is much more convenient to convert the Bézier curves into a polyline and work with discrete line segments and vertices. At first, to convert the Bézier curves defined by the user to a polyline we decided to use a simple approach of sampling uniform points along the Bézier curve. However, this approach has some downsides. The Bézier curve is not uniformly curved in all places, some parts of the curve can be similar to straight lines while some parts can be making U-turns. So if we were to uniformly sample points along the curve, we would either have to sample a lot of points to capture the resolution of the curves and lose performance, or sample a small number of points to improve performance but lose the resolution of the curves. Thus, a compromise has to be made to get the best of both worlds. So, we decided to sample more points from the parts of the curve that are more curved and sample fewer points from the parts of the curves that are less curved. A visual representation of this can be seen in Fig. 8.

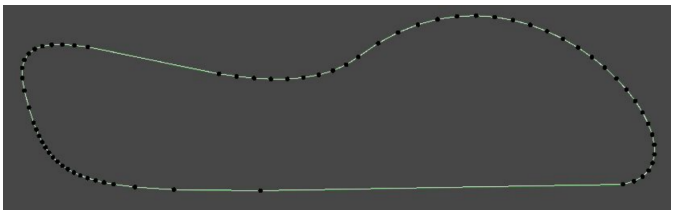

Fig. 8. Non-uniformly sampled points from Bézier curves w.r.t. the curvature.

This non-uniform sampling can be controlled by the user to choose the level of detail required for their 3D shape. There are 2 parameters that control the sampling. These are *Maximum Angle Error* and *Minimum Vertex Distance*. *Maximum Angle Error* determines the maximum angle that 3 consecutive vertices on the polyline can make. *Minimum Vertex Distance* determines the minimum distance that 2 consecutive vertices on the polyline can have. With these 2 parameters, the user can create very crude polylines or very smooth polylines as shown in figure Fig. 9.

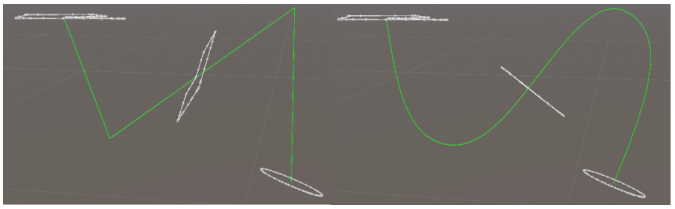

Fig. 9. Two polylines created from the same Bézier curve using different smoothing parameters.

4.4 Defining Cross Sections

Cross-sections are the second input that we get from the user for our program. The cross-sections give depth to the Bézier curve and make the sweep object what it is. We wanted the customization of the sweep object to be the main focus of how we define cross-sections, so we decided to use one of the most commonly used and powerful formats to define our cross-sections, Scalable Vector Graphics (SVG) [2]. By using SVGs we are giving the users the complete freedom to create the cross-sections they desire very precisely within the comfort of their own design tools. We believe that the compromise of going to another tool to define cross-sections is good, as most people who use Unity are already familiar with at least a 2D design tool to do even basic UI design in Unity. Additionally, by accepting a common format we are making creating cross-sections easier for our users as most of them would prefer not to learn a new UI and controls to create SVGs. Fig. 10 shows how easy it is to design various cross-sections using common design tools such as Figma [10].

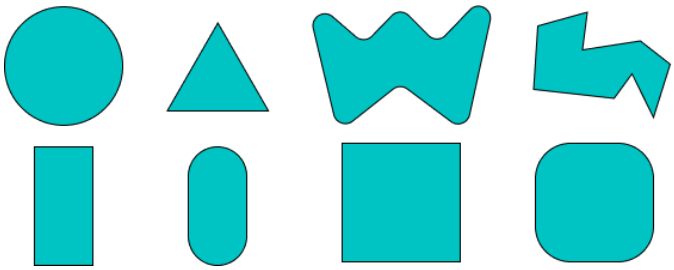

Fig. 10. Various SVGs that are used as cross sections for the sweep object made in Figma.

However, we are not supporting the entire feature set of the SVG format. We require the SVG to be saved as a path element and not as primitive elements such as rectangle, ellipse, etc. We choose this option because we wanted to support the most generic element that is available in the SVG specification and it is extremely simple to convert any primitive SVG element into a path element with a single click from any design software by flatting the object. With the path object, we can represent any 2d shape we want. Another constraint that we have on the cross-sections is that we only support cross-sections without any holes. In conclusion, we support any cross-section that can be defined as a sequence of vertices in 2D space.

4.5 Morphing Techniques

Following the results of research we have done on this topic, we tried implementing the shape morphing between two 2D cross-sections. The initial version of the code would accept the vertex coordinates of two cross-sections and a value t which is the transition (time) parameter represented by a single scalar ranging from 0 (source) to 1 (target). Using these inputs, the algorithm would initially find the cross-section with the smaller number of vertices (shape-A) and then duplicate the vertices of shape A that were closest to excess vertices of the other shape (shape-B) to match the number of vertices in both shapes. Then, it would pair-wise match the vertices of both shapes according to the vertices' direction vector in 2d space. Finally, it would do linear interpolation in between the vertices of the cross-sections. Although this method worked, it didn't provide a smooth morphing as it can be seen in Fig. 11.

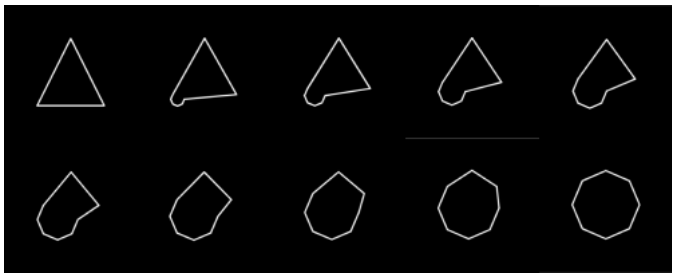

Fig. 11. Morphing from a triangle to a hexagon using the naive algorithm.

To smoothly morph from one shape to another, we needed to add vertices more smartly than just duplicating existing ones. To make the

interpolation easier, we implemented this vertex creation process as a preprocessing step for the user-defined cross-sections. The pseudo-code given in algorithm 1 first makes sure that every cross-section on the path is indexed in clockwise order from the point of the curve. Then, it aligns all shapes with each other so that index 0 of each shape roughly has the same direction in 2d space. Finally, it expands shapes with fewer vertices by introducing new vertices between existing vertices. This expansion process can be better seen in Fig. 12.

Algorithm 1: ExpandShapes

Input : Shapes: Ordered sequence of user defined cross sections.

Output : Shapes: Ordered sequence of user defined cross sections with same number of vertices.

```
1 ClockwiseShapes(Shapes);
2 while not all shapes in Shape have the same vertex count do
3     for shape in Shapes do
4         let referenceShape be the shape next to shape with a
            higher vertex count;
5         AlignShape(referenceShape, shape);
6         ExpandShape(referenceShape, shape);
```

As seen in Fig. 12, the new vertices that are added to certain points on Shape-A are added in a way that makes the vertex to vertex interpolation smoother. The relative distance of the vertex in shape-B to its left and right neighbor is preservers when added to shape-A.

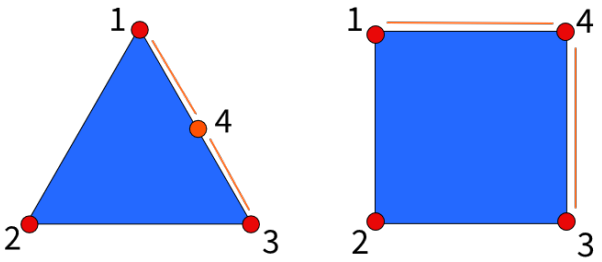

Fig. 12. Shape interpolation where closest vertices in cross sections match and new vertex (4) is inserted into the triangle proportional to its place in rectangle.

After this preprocessing step, it is trivial to morph between two shapes. We use a simple vertex-to-vertex interpolation strategy, as the number of vertices in each shape is identical, each shape is oriented in a clockwise fashion, and the vertex indices of shapes are aligned with each other. We simply, interpolate between one vertex coordinate to the other vertex coordinate.

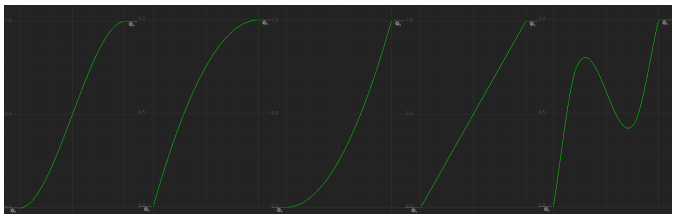

Fig. 13. Interpolation curve for controlling how to morph from one shape to another.

The default interpolation strategy is linear. However, this can be fully customized by the users. We provide interpolation curves for morphing between each consecutive pair of cross-sections. Users can choose from multiple preset interpolation curves or can define their own interpolation curve to have full control. These interpolation curves can be seen in Fig. 13. The effect of using different interpolation curves can be observed in Fig. 14 in an exaggerated fashion.

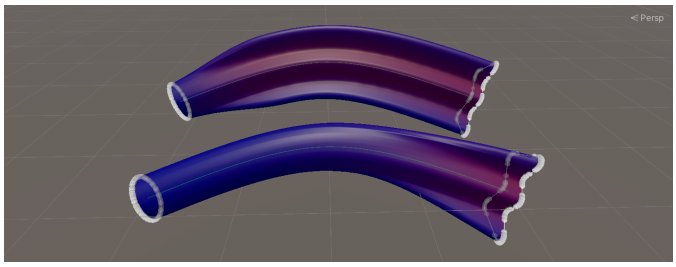

Fig. 14. Two simple sweep objects with different interpolation curves.

4.6 Creating the 3D mesh

The last piece of the puzzle for creating the 3D sweep object is to combine the cross-sections and the interpolated shapes between the cross-sections into a mesh. The way we do this is straightforward but works well. First, we create a polyline from the user-defined Bézier curve as mentioned in Sect. 4.3. Then, for every vertex in the polyline, we create another cross-section by interpolating between the two user-defined cross-sections that the vertex lies between. The interpolation is done as described in Sect. 4.5. Following that, we connect these cross sections by connecting the vertices of each shape that have the same index. (As indices are aligned) After this procedure, we are left with a bunch of rectangles connecting the whole 3D shape. However, to render the 3D shape we need to turn it into a proper mesh with triangles. So, we convert each of the rectangles into 2 triangles by connecting the diagonals. Finally, we triangulate the two closed ends of the 3D shape by using a Delaunay triangulation [8] algorithm, to get the final mesh. An example of the created mesh structure can be seen in Fig. 15.

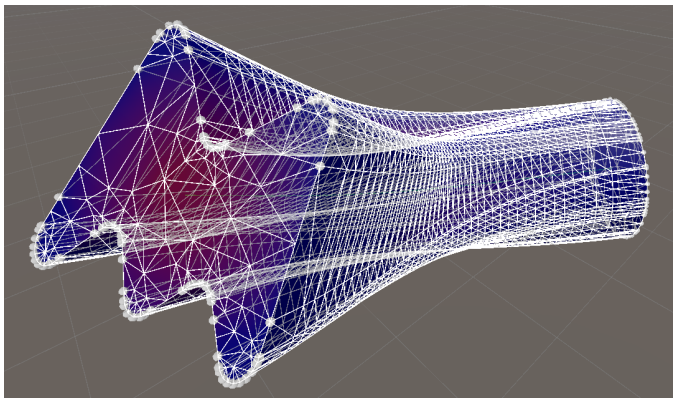

Fig. 15. Mesh structure of a simple 3d sweep object.

To help the user with creating non-overlapping meshes, we created an algorithm that searches for places where the mesh overlaps and adjusts the Bézier curve a little bit such that the mesh does not overlap anymore, an example can be seen in Fig. 16. This algorithm works in two steps, first, we find a place where the mesh overlaps and then solve the overlapping issue by changing the Bézier curve.

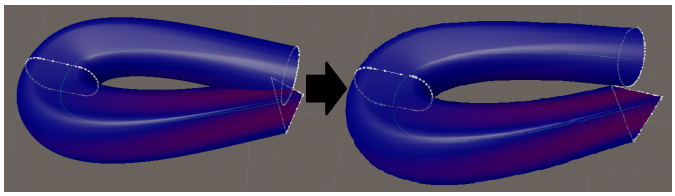

Fig. 16. Mesh overlap algorithm solving mesh overlapping issue.

In the first step of this algorithm, we use the cross-sections generated at each vertex of the polyline and check if any two of them overlap.

However, since we have 2D cross-sections in a 3D space this was quite the challenge. Looking at all segments in a cross-section to see if they intersected a segment of another cross-section took way too much time, so we had to find another solution. The solution we found was that since all cross-sections are actually in 2D, we could find the 3D plane that contained all points of the cross-section. Then, we could find the line where two planes from different cross-sections intersected and see if any of the segments inside the cross-section intersected with this line. If from both cross-sections a segment intersects with this line then we know that the cross-sections intersect. Once we know which cross-sections intersect we can find the closest anchor points of the Bézier curve to adjust it around the place where it causes an overlapping mesh. There are two cases: either the closest anchor point for both cross-sections is the same or different. If the closest anchor points are the same, then most likely the problem is that the Bézier curve takes a turn that is too sharp. In this case, we need to bring this anchor point more in line with its neighbors to make the turn less sharp. If the two closest anchor points are not the same, then it is most likely the case that these points are too close to each other for the mesh to fit between them. Therefore, we should make the distance between these two anchor points larger such that there is enough space for the mesh. To not change the Bézier curve too much, the algorithm works with multiple iterations. In each iteration, the algorithm changes the curve a little and then checks again if there is any overlap before changing it again. This way the algorithm changes the curve as little as possible and keeps the sweep object as close to the actual user input as possible.

4.7 Shading, Color & Texture

As this project is implemented as a Unity plugin, the created sweep object can use the powerful features of the Unity engine for shading, coloring and texturing. However, to achieve this compatibility we had to make sure that we created our sweep object mesh properly. For shading to work properly the mesh has to have the correctly calculated Normals, Tangents, and UVs in addition to correctly calculated triangles. While procedurally generating the mesh we also kept in mind these extra constraint and made sure that these values were calculated correctly.

The user can assign any material created in the Unity engine to the sweep object like it was a regular Unity mesh. Fig. 17 shows some example materials with various properties applied to the same object. Users can take full advantage of Physics-Based Rendering (PBR) [6] materials to design their sweep object exactly as they want it.

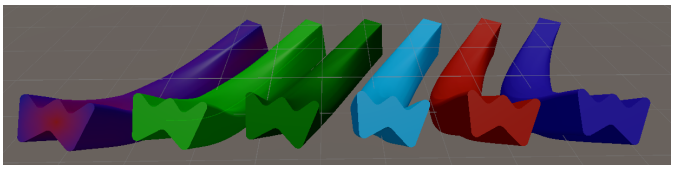

Fig. 17. Sweep object with various materials assigned to it.

Depth Shader

We additionally wrote a special depth shader to display the nature of how the sweep object is generated. The depth shader calculates how far each vertex is from the user defined Bézier and assigns a color depending on how far the vertex is. The assinged colors are sampled accordingly from a linear color gradient defined by the user. This distance calculation is done per cross section so it highlights the contours of the complex sweep object. An example of this shader can be seen in Fig. 18 and Fig. 19.

Another feature of the shader is that it has 2 modes: global, and local. The local mode limits the gradient calculations to within cross section only, meaning that sweeping from a small circle to a large circle will not change the color of the sweep object. The global mode looks at the whole object and applies the gradient accordingly, meaning that in the above example the small circle will look more pinkish and the large circle will look more blueish. An illustration of the differences between the local mode and the global mode can be seen in Fig. 19.

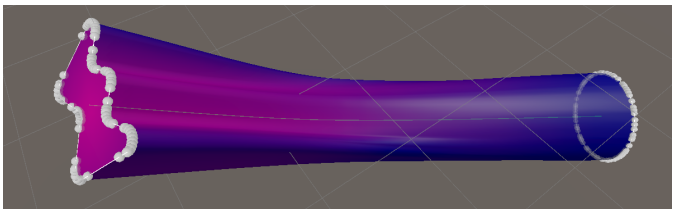

Fig. 18. Sweep object with our depth shader applied in local mode. Vertices close to the Bézier curve are colored pink and vertices far from the curve are colored blue.

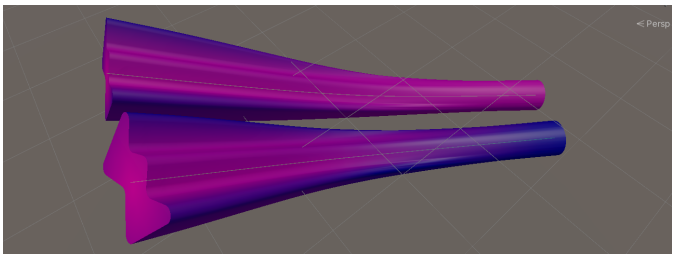

Fig. 19. Two sweep object with our depth shader applied. The above object is in global mode, the other is in local mode.

Finally, we can texture the generated mesh. However, as the generated mesh can be arbitrary complex, calculating the ”correct” UV coordinate is not a trivial task. This task does not even have a ”correct” solution as users might prefer different applications of the same texture depending on the UVs. Thus, we provide a simple UV mapping, but this mapping can be edited by the users using various design software. An example of our simple UV mapping can be seen in Fig. 20.

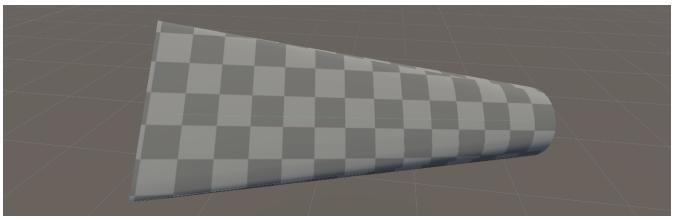

Fig. 20. A sweep object with default texture applied to show how the simple UV mapping works.

4.8 Exporting the Mesh

Users can export the mesh of the created sweep object as an .fbx file [9] to use their creations in other design software. This is beneficial as the users might want to modify the sweep object further manually using sculpting tools, etc. Additionally, the exported mesh can be imported back into Unity and be used exactly as the original sweep object. This can be thought of baking the mesh, which removes any overhead of the procedural generation steps.

4.9 Implementation Details

Unity uses C# as programming language so our plugin is also written in C#.

To create the Bézier curve we used a Bézier Path Creator asset made by Sebastian Lague [3]. We did have to modify this asset for us to make full use of it as we needed to store information about cross sections on the points in the Bézier curve.

We use an open source SVG parser library [11] written in C# for Unity to parse the SVG files into an ordered sequences of 3d vertex coordinates.

We use an open source 2D triangulator library [4] written in C# for Unity to triangulate the end cap shapes of the sweep object. The library provides the widely used Delaunay Triangulation algorithm [8].

5 Results and Evaluation

In this section, we show the results achieved at the end of our project, followed by the evaluation of the requirements. Finally, we discuss the limitations encountered as well as some ideas on how they can be approached in the future.

5.1 Results

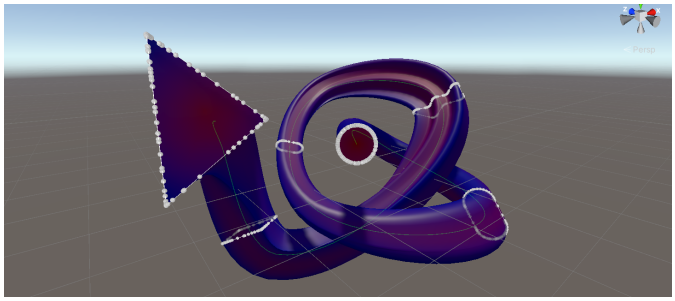

Fig. 21. A sweep object showing the feature set of our project.

To demonstrate the results of our project we created a sample sweep object as seen in Fig. 21. The sweep object starts with a circular cross-section and smoothly transitions into a capsule cross-section. Then it transitions to the same capsule cross-section which is scaled down and rotated to show the available parameters in cross-sections. Then the capsule cross-section transitions into a complex concave shape without any issues. Then the object transitions to a rectangle shape followed by a transition to a triangle which is scaled up. As can be seen from the image, these transitions work on any user-defined Bézier curve no matter how complex the curve is. Finally, we can see where the sweep object gets thinner and thicker through the shader we provided.

5.2 Evaluation

As mentioned earlier, MoSCoW principle is used to sort out the priorities of the features. In the end, we have managed to implement all the Must Have features which were critically important. This not only meant that the Minimum Viable Product was ready but also the project had strong foundations to further enhance it. To make it even more functional and attractive, we have implemented 2 of the 3 Should Have features and 1 of the 2 Could Have features.

5.3 Limitations

Firstly, although the tool creates the objects instantaneously in a what you see is what you get format, it lacks in terms of performance. This is because the mesh is created from scratch in every frame, so it is computed over and over again even if there is no change in most of the object. The limitation of this problem is that it uses too much unnecessary computation power and causes Unity to lag while inspecting and making changes to the object. To optimize this performance issue, we have to make sure that whenever there is a change in a certain section of the mesh, only the relevant parts are updated instead of creating the mesh from scratch.

A couple of limitations with the mesh overlap algorithm were found, first of all, the algorithm sometimes computes that there is overlap in the mesh while there is none. This is because the intersection line between the two planes of the cross-section can intersect both cross-sections without the cross-sections intersecting. This can be solved by looking at the coordinates where the intersections take place and seeing if they are at the same places. Another limitation is that the algorithm only looks for overlap at the vertices in the created polyline, therefore if there is any overlap in the places where the polyline has no vertices the algorithm will not detect them. And lastly, the way the algorithm tries to resolve overlapping is not always correct. This is because the algorithm uses little data to compute a solution, with more research into resolving the overlapping of the mesh this algorithm could be improved to make better decisions on how to solve the mesh overlapping.

6 Conclusion and Future Work

In this section, we discuss the summary of work done including the features accomplished, as well as additional features and improvements that can be introduced to the project in the future to make it better.

6.1 Summary of the work done

To sum up, the project resulted in being a plugin tool that we believe will be useful for game developers both in terms of time and effort put into creating a 3D shape. Instead of an external application, we decided that it's more logical to make it a plugin due to benefits such as ease of use and accessibility. Throughout the project, all of the must-haves and some of the should/could-have features have been achieved.

One of the most fundamental features is the creation of the Bézier curve which we chose to use a library for its core. To generate the 3D shape in as high resolution as possible while balancing the performance, we are using a non-uniform sampling of the points along the curve. The user can control this via 2 parameters which are Maximum Angle Error and Minimum Vertex Distance. Another fundamental feature is the definition of cross-sections which is used in the extrusion along the curve to create the 3D shape. This input is captured through Scalable Vector Graphics (SVG) format to provide the user with the flexibility of creating custom and even complex shapes. With these two features, the minimum viable product was achieved.

In addition to these, we have implemented morphing between cross-sections. This was achieved with detailed research and in a few iterations where the technique was improved significantly at each one, resulting in a smooth interpolation between two different shapes of cross-sections. Another important and essential implementation in generating the 3D shape was to combine the cross and the intermediate sections into a mesh. This allowed the rendering of the whole object smoothly. Finally, we implemented a depth shader that highlights the intricacies of the 3d sweep object.

6.2 Future Work

The features that were planned initially and weren't implemented can be introduced to the project. Clipping parts of the object by using a cutting tool would be a useful feature for the developers where they would need to export a more customized object for their games. Additionally, the option to import a text file to create the 3D shape in an automated way would also be a flexible and time-saving way. Finally, although game developers might not find it useful, a way to animate the creation of the object could be introduced. This can help the developers look at the object from different perspectives and also to have a bit more fun.

Some features that are implemented can also be improved upon. In addition to applying different colors to the object, our tool allows textures to be applied to the object. This currently only works for simple shapes, and thus could be improved to work for more complex shapes as well. Secondly the mesh overlap algorithm has some limitations as seen before allowing for improvements to be made for this algorithm by researching how to adapt the Bézier curve to solve mesh overlapping, or by improving the mesh overlap detection.

References

[1] Finding a Point on a Bézier Curve: De Casteljau's Algorithm.

[2] Scalable Vector Graphics (SVG) 2.

[3] S. Lague. Bézier path creator. `https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082`.

[4] N. Masatatsu. unity-triangulation2d. https://github.com/mattatz/unity-triangulation2D, 2016.

[5] G. Sealy and G. Wyvill. Representing and rendering sweep objects using volume models. In *Proceedings Computer Graphics International*, pp. 22–27, 1997. doi: 10.1109/CGI.1997.601264

[6] U. Technologies. Unity - Manual: Material charts.

[7] Unknown. Interpolating corresponding shapes. https://tuprints.ulb.tu-darmstadt.de/213/5/diss4a.pdf.
[8] Wikipedia contributors. Delaunay triangulation — Wikipedia, the free encyclopedia, 2022. [Online; accessed 14-April-2022].
[9] Wikipedia contributors. Fbx — Wikipedia, the free encyclopedia, 2022. [Online; accessed 14-April-2022].
[10] Wikipedia contributors. Figma (software) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 14-April-2022].
[11] S. Yoshihiro. Svgmeshunity. https://github.com/beinteractive/SVGMeshUnity, 2018.

A CONTRIBUTIONS

Name	Student ID	Contribution
Ahmet Ayrancioglu	1678620	Presentation & Report & Implementation
Mehmet Zorlu	1753673	Report & Presentation & Interpolation
Wouter Haneveer	1300334	Report & Interpolation research & Mesh Overlap Algorithm