Happyville and Neon Escape

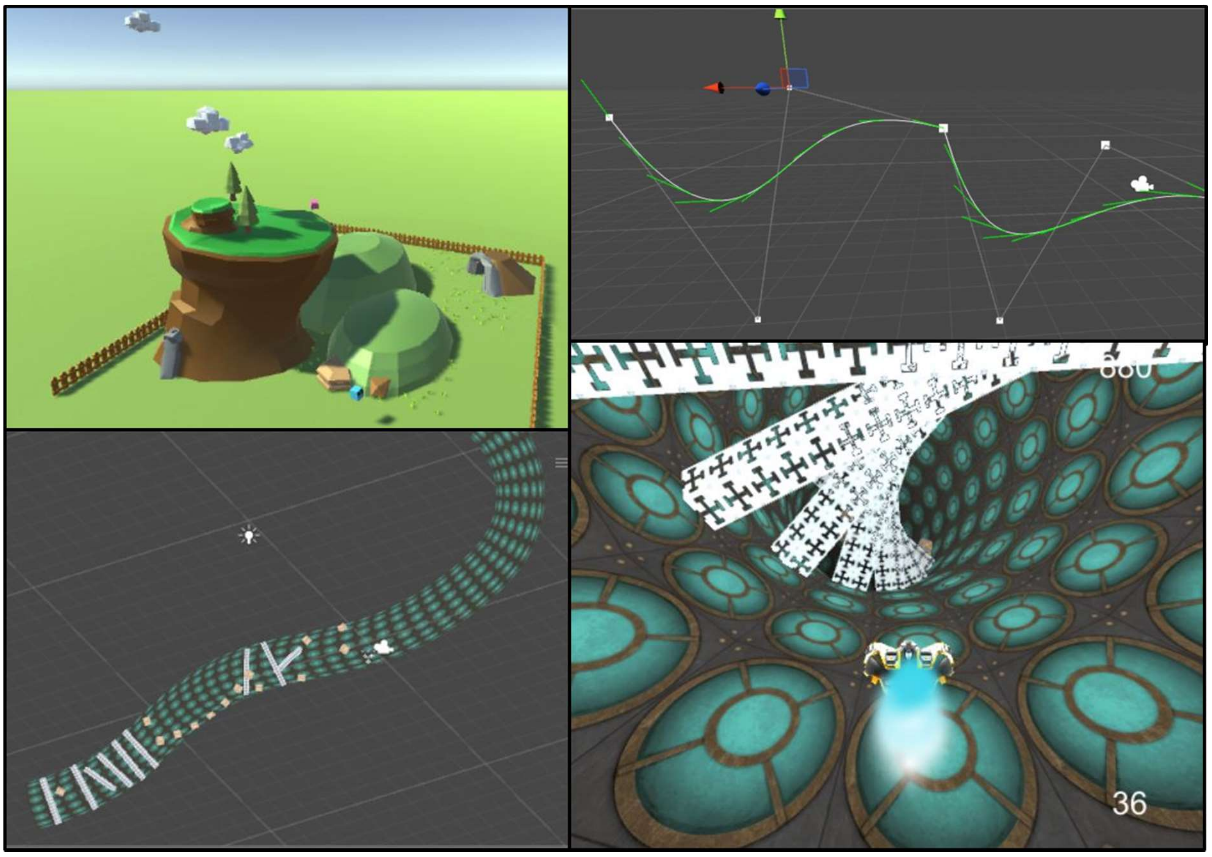
"Splines and Biarc techniques Implementation"

CW3 – Mathematics and Graphics for Computer Games II

Elio de Berardinis

Submission Date: 19-04-2016

No. of Pages: 12



INTRODUCTION and BACKGROUND

Bézier Curve

A Bézier curve is defined by a set of *control points* $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called its order (n = 1 for linear, 2 for quadratic, 3 for cubic, etc.) The first and last control points are always the end points of the curve; however, the intermediate control points (if any) generally do not lie on the curve. [1]

-Linear Bezier Curve

Given points $\mathbf{P}_0$ and $\mathbf{P}_1$, a linear Bézier curve is simply a straight line between those two points. The curve is given by:

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \text{ , } 0 \le t \le 1$$

and is equivalent to linear Interpolation.

-Quadratic Bezier Curve

A quadratic Bézier curve is the path traced by the function $\mathbf{B}(t)$, given points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$,

$$\mathbf{B}(t) = (1-t)[(1-t)\mathbf{P}_0 + t\mathbf{P}_1] + t[(1-t)\mathbf{P}_1 + t\mathbf{P}_2] \text{ , } 0 \le t \le 1,$$

which can be interpreted as the linear interpolant of corresponding points on the linear Bézier curves from $\mathbf{P}_0$ to $\mathbf{P}_1$ and from $\mathbf{P}_1$ to $\mathbf{P}_2$ respectively. Rearranging the preceding equation yields:

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2 \text{ , } 0 \le t \le 1.$$

The derivative of the Bézier curve with respect to t is

$$\mathbf{B}'(t) = 2(1-t)(\mathbf{P}_1 - \mathbf{P}_0) + 2t(\mathbf{P}_2 - \mathbf{P}_1).$$

from which it can be concluded that the tangents to the curve at $\mathbf{P}_0$ and $\mathbf{P}_2$ intersect at $\mathbf{P}_1$. As t increases from 0 to 1, the curve departs from $\mathbf{P}_0$ in the direction of $\mathbf{P}_1$, then bends to arrive at $\mathbf{P}_2$ from the direction of $\mathbf{P}_1$.

The second derivative of the Bézier curve with respect to t is

$$\mathbf{B}''(t) = 2(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0).$$

-Cubic Bezier Curve

Four points $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ in the plane or in higher-dimensional space define a cubic Bézier curve. The curve starts at $\mathbf{P}_0$ going toward $\mathbf{P}_1$ and arrives at $\mathbf{P}_3$ coming from the direction of $\mathbf{P}_2$. Usually, it will not pass through $\mathbf{P}_1$ or $\mathbf{P}_2$; these points are only there to provide directional information. The distance between $\mathbf{P}_0$ and $\mathbf{P}_1$ determines "how far" and "how fast" the curve moves towards $\mathbf{P}_1$ before turning towards $\mathbf{P}_2$.

Writing $\mathbf{B}_{\mathbf{P}_i,\mathbf{P}_j,\mathbf{P}_k}(t)$ for the quadratic Bézier curve defined by points $\mathbf{P}_i$, $\mathbf{P}_j$, and $\mathbf{P}_k$, the cubic Bézier curve can be defined as a linear combination of two quadratic Bézier curves:

$$\mathbf{B}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0,\mathbf{P}_1,\mathbf{P}_2}(t) + t\mathbf{B}_{\mathbf{P}_1,\mathbf{P}_2,\mathbf{P}_3}(t) \text{ , } 0 \le t \le 1.$$

The explicit form of the curve is:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \text{ , } 0 \le t \le 1.$$

For some choices of $\mathbf{P}_1$ and $\mathbf{P}_2$ the curve may intersect itself, or contain a cusp.

Any series of any 4 distinct points can be converted to a cubic Bézier curve that goes through all 4 points in order. Given the starting and ending point of some cubic Bézier curve, and the points along the curve corresponding to $t = 1/3$ and $t = 2/3$, the control points for the original Bézier curve can be recovered. [2]

The derivative of the cubic Bézier curve with respect to t is

$$\mathbf{B}'(t) = 3(1-t)^2(\mathbf{P}_1 - \mathbf{P}_0) + 6(1-t)t(\mathbf{P}_2 - \mathbf{P}_1) + 3t^2(\mathbf{P}_3 - \mathbf{P}_2) \,.$$

The second derivative of the Bézier curve with respect to t is

$$\mathbf{B}''(t) = 6(1-t)(\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0) + 6t(\mathbf{P}_3 - 2\mathbf{P}_2 + \mathbf{P}_1) \,.$$

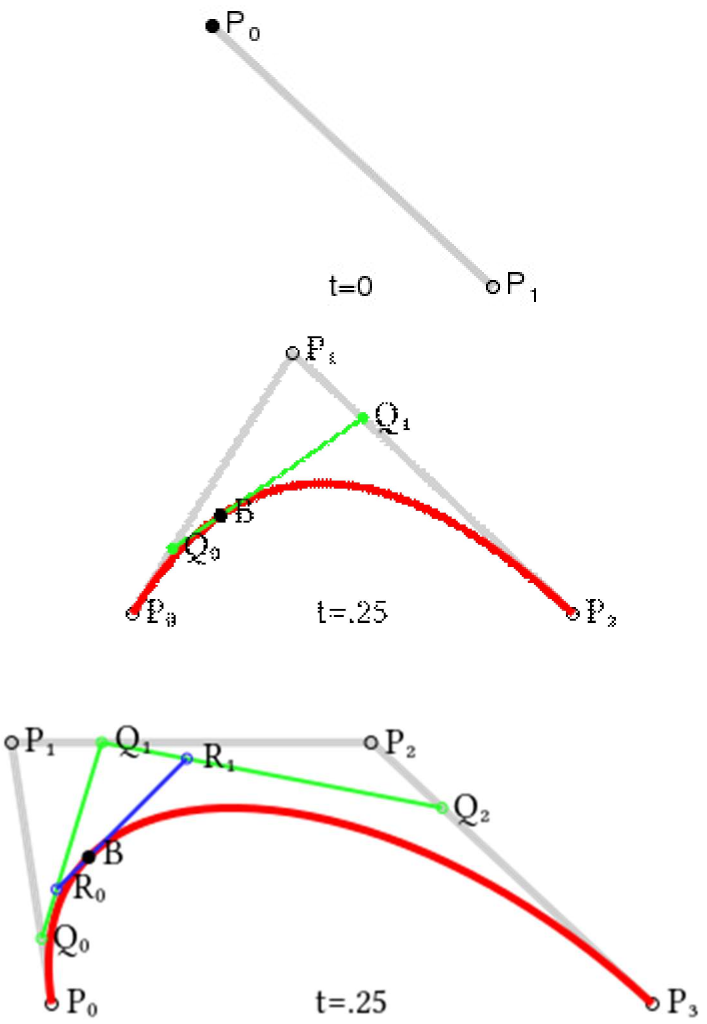


Figure 1 – From the top: Linear, Quadratic and Cubic Bezier Curves with Control points and Linear interpolation.

Applications in Computer Graphics

Bézier curves are widely used in computer graphics to model smooth curves. As the curve is completely contained in the convex hull [3] of its control points, the points can be graphically displayed and used to manipulate the curve intuitively. Affine transformations such as translation and rotation can be applied on the curve by applying the respective transform on the control points of the curve.

Quadratic and cubic Bézier curves are the most common. Higher degree curves are more computationally expensive to evaluate. When more complex shapes are needed, low order Bézier curves are patched together, producing a composite Bézier curve commonly known as "Path" or "Spline".

Bézier Spline

In geometric modelling and in computer graphics, a composite Bézier curve or "Bézier Spline" is a piecewise Bézier curve that has at least C0 continuity [4]. In other words, a composite Bézier curve is a series of Bézier curves joined end to end where the last point of one curve coincides with the starting point of the next curve. To guarantee smoothness, the control point at which two curves meet must be on the line between the two control points on either side. Depending on the application, additional smoothness requirements (such as C1 or C2 continuity) may be added [5]. C2 continuous composite cubic Bezier curves are known and defined as cubic B-splines [6].

A C0 continuous composite Bézier is also called a **polybezier**, by similarity to polyline, but whereas in polylines the points are connected by straight lines, in a polybezier the points are connected by Bézier curves. A **beziergon** (also called **bezigon**) is a closed path composed of Bézier curves. It is similar to a polygon in that it connects a set of vertices by lines, but whereas in polygons the vertices are connected by straight lines, in a beziergon the vertices are connected by Bézier curves.

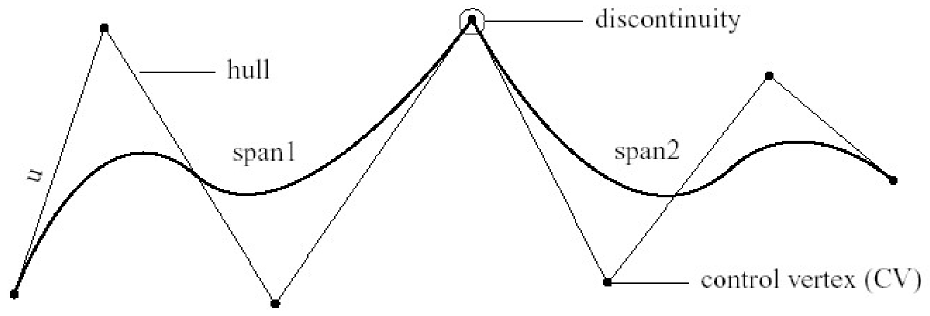


Figure 2 – A Bézier Spline with C0 continuity. Contains mathematical discontinuity on the first derivative.

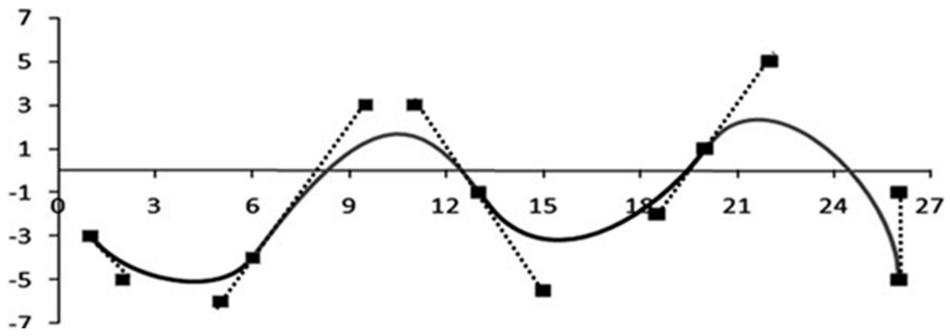


Figure 3 – A Bezier Spline with C1 Continuity. Smoothness is imposed on the first derivative.

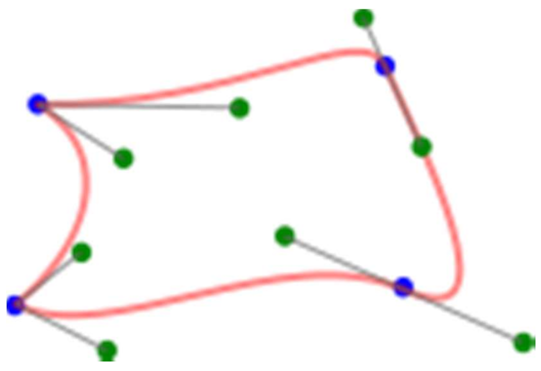


Figure 4 **-** A Beziergon.

Biarc

A Biarc is a model commonly used in geometric modeling and computer graphics that is composed of two consecutive circular arcs with an identical tangent at the connecting point. Since the tangents at the connecting node are the same, the G1 continuity property is preserved. They can be used in approximation of splines and other plane curves by placing the two external endpoints on the curve, each with the same tangent as the curve's at that point, and finding the middle node according to some criteria. Thus a sequence of biarcs can be formed such that it is G1 continuous throughout and is tangential to the original curve (i.e. sharing its location and direction) at each biarc's end points. Shortening biarcs will improve the approximation's closeness to the original curve.

To approximate a Bézier curve with a biarc, take the two endpoints, A and B, of the Bézier. Find the point C which is the intersection of the tangents at A and B, then take the incircle of triangle ABC. Its centre, which lies on the bisectors of the three angles of the triangle, is the intersection point of the two arcs of the biarc. Their endpoints are A and B. The closer the centre lies to the curve, the better the approximation - if the distance is over a threshold, subdivide the Bézier.

In case circular arc primitives are not supported in a particular environment, they may be approximated by Bézier curves [7]. Commonly, eight quadratic segments or four cubic segments are used to approximate a circle. It is desirable to find the length $\mathbf{k}$ of control points which result in the least approximation error for a given number of cubic segments.

In other words we can approximate a circular arc with a Bezier curve and a Bézier curve with a Biarc. Bezier curves give us more freedom but in some occasions its control points might be difficult to compute and control. On the other hand Biarcs give limited freedom but are easy to control and predict.

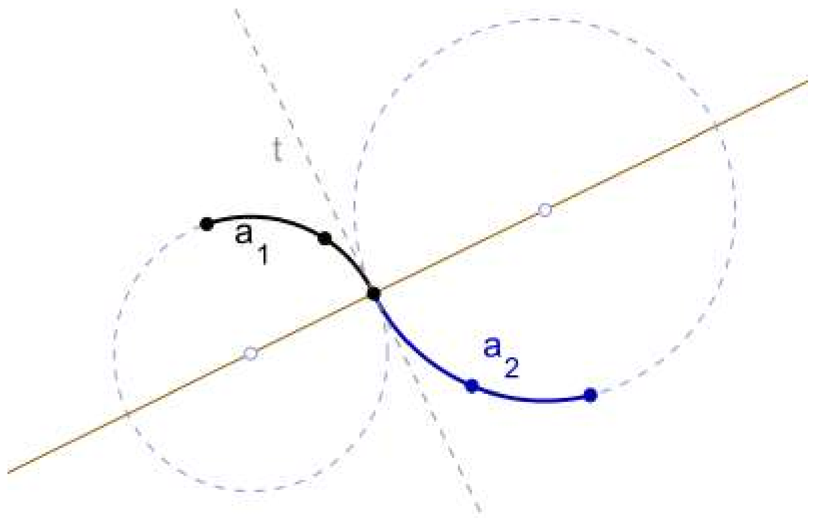


Figure 5 – Biarc technique to obtain a C1 curve

Objective of this work

In the previous paragraph we outlined the quasi-interchangeability of Bézier Splines and Biarcs. Although these techniques can be leveraged to obtain similar results, there are situations in which the freedom provided by Bézier splines is preferred. In other cases, smoothness (C1 continuity) and predictability (C2 continuity or Curvature) has instead the highest priority over freedom.

In this coursework we will provide two examples: one using Bezier splines and the other one using Biarcs. We will discuss their implementation in Unity 5 and explain why one technique is preferred over the other in the specific case.

IMPLEMENTATION and DEVELOPMENT

In this project the example dealing with splines is called "Happyville" and the one using Biarcs is called "Neon Escape". Specific Unity implementations, techniques and shortcuts have been drawn, adapted and optimized from pre-existing sources [8]. The code is heavily commented and for brevity will not be included in this report. Instead, the relevant C# scripts, their methods and their variables will be referenced here when appropriate.

Happyville is a simple and bucolic scenery simulation. Splines are used here to generate specific paths that objects in the scenes will follow simulating natural movement. The presence of very different objects such as atmospheric agents (clouds) and simple characters (animals) requires a lot of freedom in generating believable paths. Clouds move slowly and smoothly while animals might move fast, in a loop, back and forth or in a zig-zag fashion. In this case smoothness and predictability are not of highest priority and we can tweak and implement them as we like.

Neon Escape is a fast endless runner short game taking place inside randomly-generated curved cylindrical structures with various obstacles to avoid. Although we could implement Bézier splines to make any curve we want, they're not easy to control. It's not obvious how we could constrain random placement of control points such that we end up with acceptable shapes. And there is also no simple direct way to determine the length of a curve segment. In a fast endless runner, the transition from one random section of the pipe to the next should be smooth while at the same time offering enough variety. Presenting the player with a very sharp turn (a C1 discontinuity) would make the game unplayable and frustrating. Assuring such constraints are respected in this scenario is not trivial using Bézier splines. Instead, we can construct our pipe system with Biarcs. They are easily controlled by limiting the radius and arc length of the circles, they always respect C1 (and G1, Geometric) continuity and computing the length of each segment is simple.

Happyville

The first step in creating a Bézier spline is implementing the lines connecting the control points that will be used to adjust it to the desired curvature. The script "Line" contains the geometrical information (start and end point) and a custom Editor Script called "LineInspector" was created to actually draw the line through the function "OnSceneGui ()". This method also takes care of appropriate transformations of the control points between local (Line) and world Space. Creating custom editors allows us to control and display specific geometrical, visual and colour information without relying on Unity's default ones.

The second step was to build an actual Bézier curve. In this case we created a cubic Bézier curve. With the same methods described for the line component a "BezierCurve" script was created with its associated "BezierCurveInspector" custom editor. In BezierCurve the method "GetPoint(float t)" returns a specific point on the curve according to the parameter t. This parameter t is obtained by calling the function "GetPoint(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t)" in the "Bezier" script where the methods for cubic Bezier equation and its derivatives are calculated, normalized and converted to the appropriate t value. Other methods in the BezierCurve script deal with calculating its derivatives to show the rate of change along the curve and illustrate it with green lines (known as "velocity lines") or simply reset the curve to default values. Similar to the LineInspector script, BezierCurveInspector combines the information and actually draws the lines connecting the control points, the curve itself, the velocity lines and carries out the appropriate transformations.

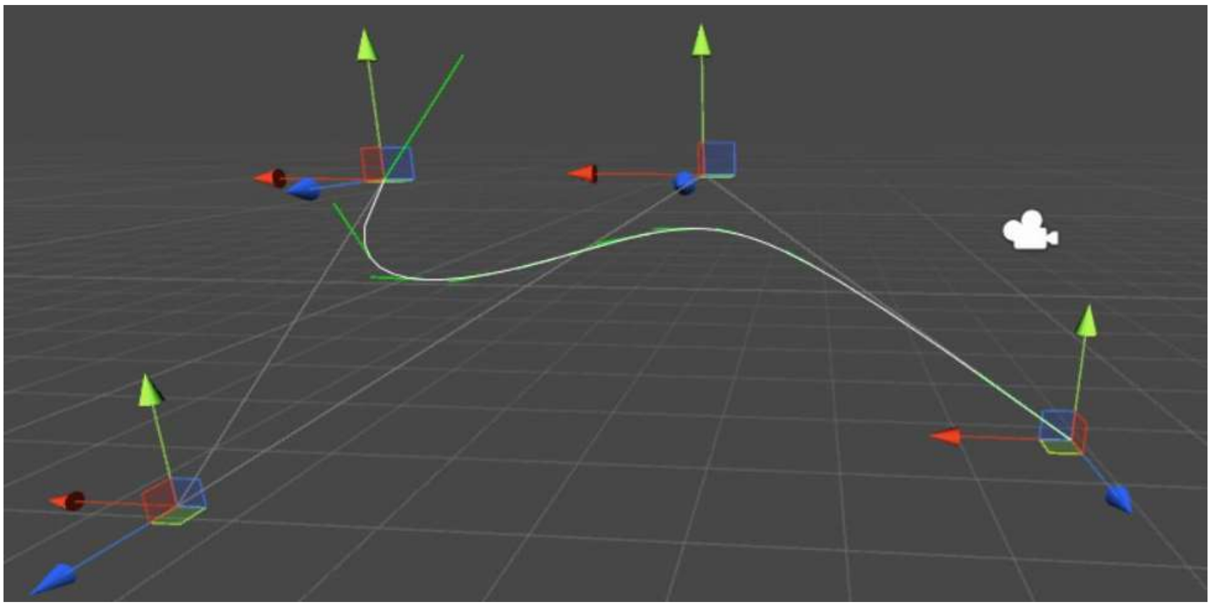
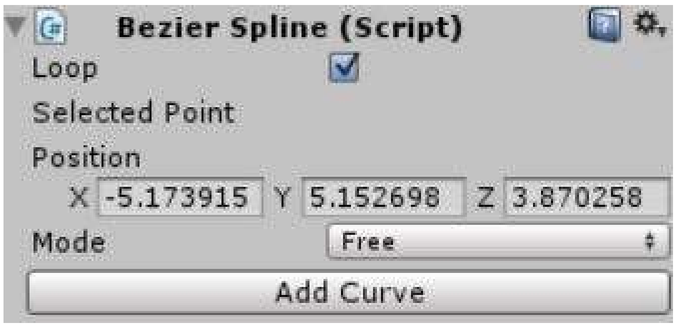


Figure 6 – Cubic Bezier Curve in Unity with control points (axis) and velocity lines (green).

Once the curve has been created, the final step to produce a spline requires the appropriate concatenation of multiple curves and the possibility of adding certain restrictions on the endpoints to obtain continuity should it be required by the situation. To do so we created two other scripts following the same logic of the previous ones: "BezierSpline" and "BezierSplineInspector". The methods in these scripts allow us to draw the cubic Bezier curves and concatenate them. Moreover, the custom Inspector allows the user to simply press a button to add more curves, close the spline in a loop (or Beziergon) and decide if and which constraints to enforce on control points in order to achieve continuity (Mirrored or Aligned with previous and next control point).



Bezier Spline (Script)

Loop ✓

Selected Point

Position

X -5.173915 Y 5.152698 Z 3.870258

Mode Free

Add Curve

Figure 7 – Custom Inspector for a Bezier Spline with Loop option, Control Points mode and option to add extra curves.

The remaining methods in these two scripts perform housekeeping functions such as changing the colour of enforced control points, highlighting only the control point selected to avoid clutter or reset the spline to default values.

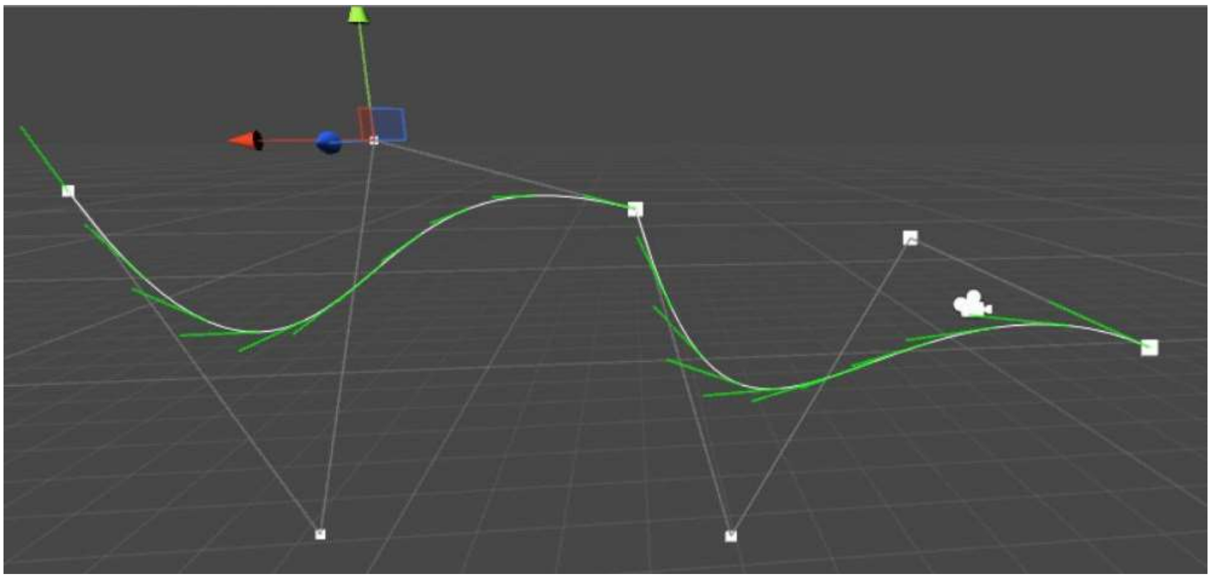
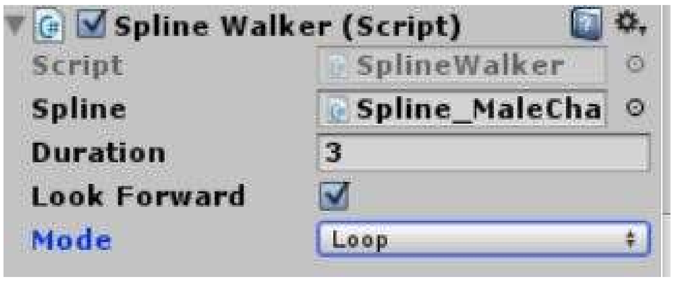


Figure 8 – A complete Bezier Spline with control points, velocity lines and a tidy, uncluttered interface.

Finally, a component was created in order to attach an object or "Walker" to the spline and have it move along the desired path. The script "SplineWalker" contains these functions. The "Update ()" method in this script updates the position of the object along the spline at every frame. On top of that the user can decide the speed and mode of walking along the curved path: In a loop, Back and forth (ping pong) or only once and decide whether they want the character to look in the direction it is facing by setting the bool variable "LookForward" to True. These parameters can be easily adjusted by the user in the inspector.



Spline Walker (Script)

Script	SplineWalker
Spline	Spline_MaleCha
Duration	3
Look Forward	✓
Mode	Loop

Figure 9 – Custom Inspector for the spline walker. Adjust the speed through the duration parameter, choose the modality and make your character look forward.

Unity 5 contains many free assets in the store. In order to contextualize these concepts and present them in a meaningful way, a low-poly, colourful, bucolic scenario was downloaded from the store. Splines were associated with clouds and simple cubic characters to demonstrate their use for different purposes. The clouds move smoothly and slowly while the characters move faster, in a zig zag fashion and face the direction they move as shown in **Figure 10** (refer to the youtube video for the scene in motion).

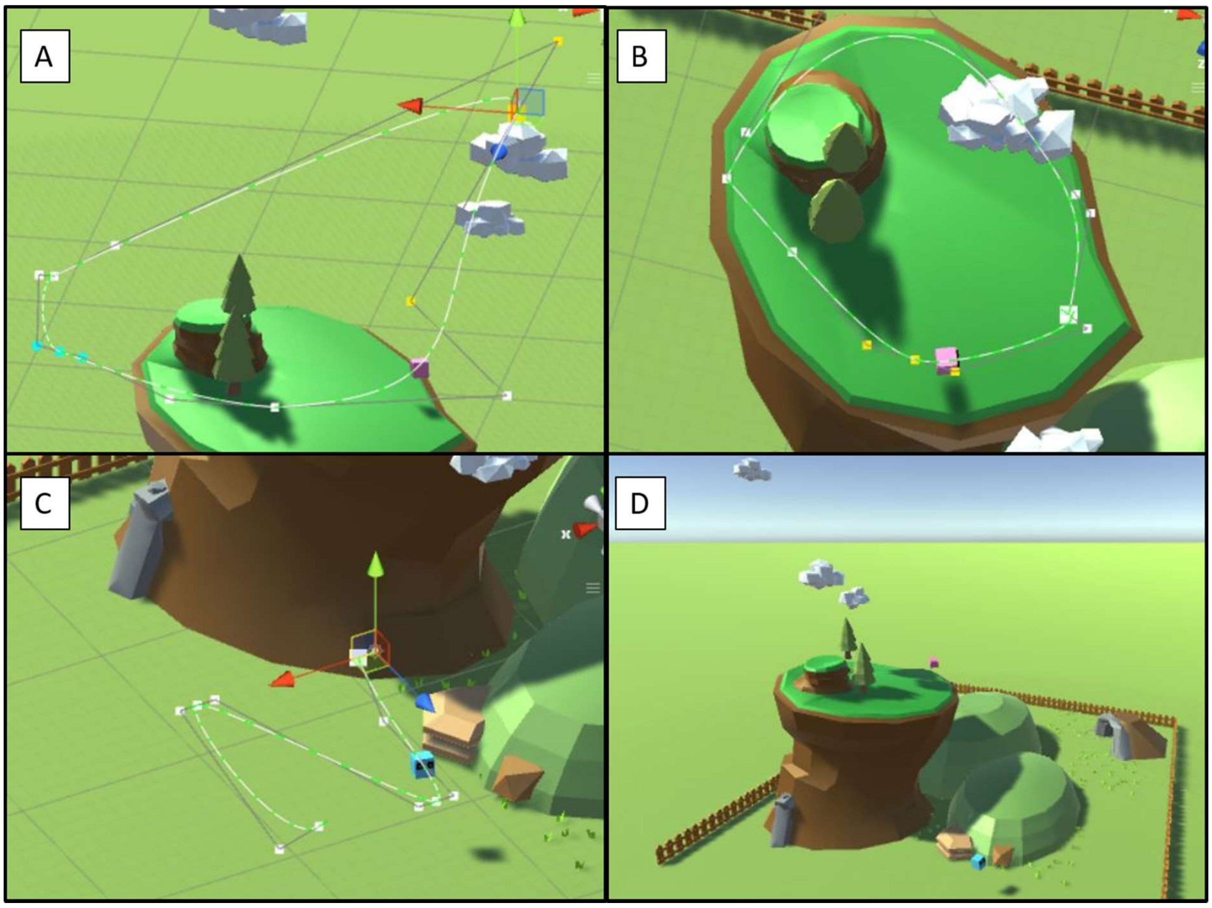


Figure 10 – **(A)** The Clouds move in a circular fashion at slow pace. Most of the control points are restricted to create smoothness across different curves in the spline (light blue squares for mirrored, yellow for aligned). **(B)** The female character on top of the hill also moves in a loop. Here though, most of the control points have no restrictions (white squares control points) so that the character can demonstrate sharp turns. **(C)** The Male character at the bottom of the hill moves in a zig-zag fashion and ping-pong mode with no restrictions on control points. **(D)** In-Game Scene.

Neon Escape

In Neon Escape we used Biarc technique to procedurally generate an endless, curved, cylindrical path with obstacles to avoid. The Biarc technique will allow us to add enough variety to the track without the technical issues of assuring continuity along different sections as this is intrinsically assured by the geometrical characteristics of this technique.

The first step in building our procedural track is creating its basic component: a Pipe. For this purpose the script "Pipe" was created. A single pipe segment is simply a partial circle. Or as we're working in 3D, a partial torus. A torus is defined by two radii. In our pipe's cases these are the radius of the pipe, and the radius of the curve it's following.

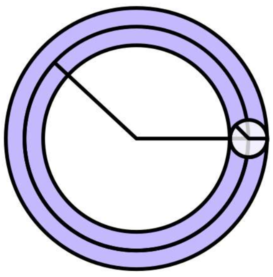


Figure 11 – A Pipe is a partial torus defined by two radii.

In order to draw our pipe we need to define the vertices along the torus and create a first ring. Iterating this ring along the curve will generate our torus. To place vertices we need to be able to find points on the surface of our torus. A torus can be described with a 3D sinusoid function.

$x = (R + r \cos v) \cos u$

$y = (R + r \cos v) \sin u$

$z = r \cos v$

In the above function r is the radius of the pipe. R is the radius of the curve. The u parameter defines the angle along the curve, in radians, so in the 0–2π range. The v parameter defines the angle along the pipe. The method "GetPointOnTorus(float u, float v)" implements this in our Pipe script.

Once the points on torus are correctly obtained we can create a mesh to connect them appropriately with the methods "SetVertices()", "SetTriangles()" and initialize it in "Awake()". By doing this for the whole circle we obtain a completely textured torus. In order to obtain only a fraction of the torus (a Pipe) we can simply set a fixed distance between subsequent rings of vertices. Then, the curve's arc length (Pipe's length) will be determined by the number of segments to draw which is saved in the variable "curveSegmentCount" randomly determined within a specific range to avoid tangling or looping.

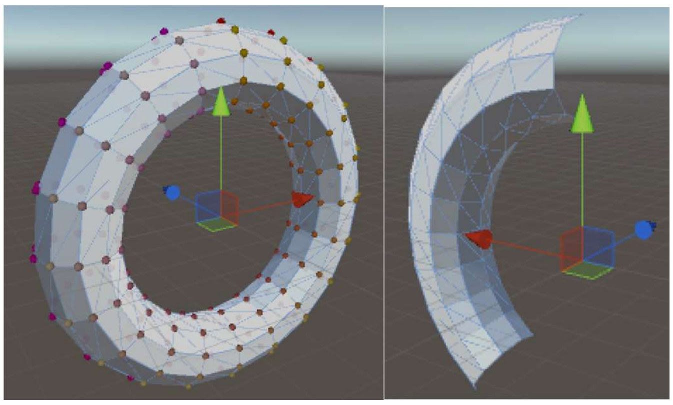


Figure 12 – **Left:** The entire textured Torus. **Right**: Pipe obtained by only drawing a certain number of segments along the curve.

Once the basic unit of our system (Pipe) has been created we need a way to concatenate multiple instances appropriately and create the track. The script "PipeSystem" takes care of that along with the script "Player" that simulates moving along the pipe system. In reality we will be moving the pipe system around the player (to which the main camera will be attached) in order to never leave the origin position. Placing everything at the origin is important because gives us great control and stability in our game. To achieve this we assign our pipe system to a world object set in the origin. The player object is also set in the origin and is given a rotator object and a simple avatar with rigid body and colliders to detect collisions with possible obstacles. The two scripts PipeSystem and Player make sure the pipe system correctly revolves around the player object (with the methods "Update()", "UpdateAvatarRotation()" and "SetupCurrentPipe()" in the Player script), detects the passage of the player between different pipe sections (with "Update()" again), generates, shifts and aligns the newly randomly generated pipe (SetupNextPipe(), ShiftPipes(), AlignNextPipeWithOrigin() in the PipeSystem script) while constantly taking into account the player's input. Other housekeeping methods make sure transformations (mostly rotations) between world and local space are computed correctly, delete passed pipe segments, check if the player is still alive ("Die()" method in Player script) keep track of the distance travelled in radians, and convert it to a score system to pass to the "MainMenu" and "HUD" scripts. These latter two scripts provide the user with an intro screen with the possibility of selecting 3 difficulty levels (based on speed) and provide information in game about the current velocity and distance travelled that will translate into a score.

Finally in order to challenge the player we inserted two simple obstacles inside the tracks (cubes and 3D rectangles). These obstacle should be randomly placed on the pipe's internal surface in order to provide variety while avoiding frustration. Items can be placed along a pipe's curve, with a rotator like the avatar to control where on the surface they appear. So positioning an item requires a pipe, a curve rotation, and a ring rotation. The scripts dealing with obstacles are "PipeItem", "PipeItemGenerator", "RandomPlacer" and "SpiralPlacer". The first two deal with creating instances of the obstacles inside the pipes while the last two deal with placing them either randomly (within certain ranges to avoid frustration and overlapping) or in a spiral fashion. The Pipe script contains an instance of PipeItemGenerator providing different combinations of the obstacles to randomly choose from. The obstacle combinations are made into prefabs (cube only, rectangle only or both) with different placing modalities (random or spiral). This way each pipe section will randomly choose the types of obstacles to display and how to arrange them. Other housekeeping methods in these scripts deal with properly deleting the obstacles when the relevant pipe has been passed. The last script, called "Avatar" is attached to the avatar object of the player and controls the particle system, detects collisions with obstacles and trigger the visual effect of explosion.

Also for this project we downloaded some free assets from the Unity Store in order to give the game a Sci-Fi feeling of racing through an alien spaceship as shown in **Figure 13**. (Check youtube video for the game in motion).

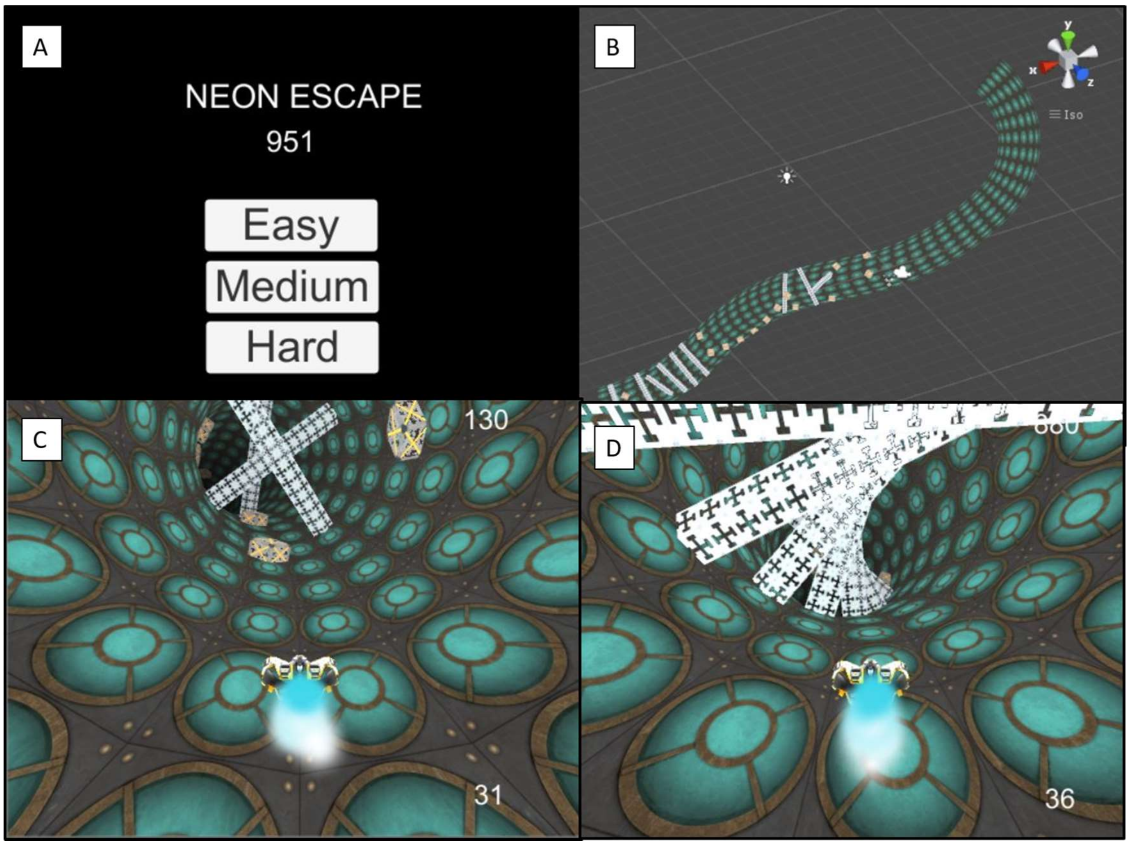


Figure 13 **–** **(A)** The Game's Intro Screen and difficulty selection. **(B)** The Unity Scene displaying the constantly evolving pipe system. **(C)** In game player's perspective. Facing randomly arranged obstacles. **(D)** Facing spirally arranged obstacles.

CONCLUSIONS and FUTURE DEVELOPMENTS

In this work we showed two powerful methods to generate paths and track systems: Bezier splines and Biarcs. Although they are theoretically interchangeable there are situation in which one is preferred over the other. When freedom has the highest priority over continuity Bezier splines are the way to go. This is the case in crowd simulations or characters animation/movements as seen in Happyville. On the other hand when continuity is a requirement to provide the player with an enjoyable experience such as in procedurally generated racing games like Neon Escape, biarcs provide more control and less technical issues.

Completing this coursework gave me the tools to fully understand how splines and biarcs are implemented in Unity. I plan on refining these two examples later on and to utilized the learned techniques in future projects.

FURTHER INFORMATION

A Demo video of the software can be found on Youtube: https://youtu.be/rG9HnAjE00M

GitHub Repository: https://github.com/eliodeberardinis/CW3_Math/tree/CW3_Math_Complete

REFERENCES

[1] Wikipedia: https://en.wikipedia.org/wiki/Bezier_curve

[2] John Burkardt. "Forcing Bezier Interpolation (from web.archive.org 2013-12-25)".

[3] Wolfram MathWorld. http://mathworld.wolfram.com/ConvexHull.html

[4] Smoothness. Wikipedia: https://en.wikipedia.org/wiki/Smoothness.

[5] Eugene V. Shikin; Alexander I. Plis (14 July 1995). *Handbook on Splines for the User*. CRC Press. pp. 96–. *ISBN 978-0-8493-9404-1*.

[6] Bartels, Richard H.; Beatty, John C.; Barsky, Brian A. (1987-01-01). *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann. ISBN 9781558604001.

[7] Stanislav, G. Adam. "Drawing a circle with Bézier Curves". Retrieved 10 April 2010.

[8] Catlike Coding: http://catlikecoding.com/