Finding Junctions in Spline-based Road Generation

Danny Darwiche
Isak Nyström

Computer Science
Bachelor's Thesis
15 ECTS credits
Spring 2022
Supervisor: Carl Johan Gribel

Finding Junctions in Spline-based Road Generation

Isak Nyström
Computer Science, Game Development.
Malmö University
Malmö, Sweden
isaq.tokyo@gmail.com

Danny Darwiche
Computer Science, Game Development.
Malmö University
Malmö, Sweden
darwichedanny@gmail.com

***Abstract*—Splines are a common mixed-initiative technique for road generation. A designer draws the shape of the curve but the mesh can be procedurally generated along the spline. This relationship improves the workflow of building roads in virtual environments and video games without taking away all of the control of the designer. Whilst this technique is useful when building single roads such as race tracks, it unfortunately struggles when dealing with more complex road networks that feature intersections. These intersections struggle with overlapping meshes and flickering textures without a straightforward solution. This problem significantly limits the usefulness of spline tools when generating roads. This paper aims to solve part of this problem by suggesting a method for detecting intersections in splines that support procedural mesh generation.**

***Index Terms*—Bezier, Bezier Subdivision, Convex Hull, De Casteljau, Intersection Detection, Jarvis March, Procedural Geometry, Separating Axis Theorem, Splines, Sutherland-Hodgman**

I. Introduction

Procedural generation of roads is an apt solution when designing road networks in virtual environments and video games. One common technique for road generation is presented by Smelik et al. which employs user-assisted design with 3D splines [7]. This mixed-initiative approach to procedural generation of roads keeps the designer in control of the path and procedurally generates the road along the curve. This technique removes the need to manually place each road segment which allows for an efficient workflow when designing and updating roads. This is commonly used thanks to its ease of modification and because the designer's detailed influences are kept in the loop.

The problem with using curves for road generation is identifying intersections and adapting the procedural mesh generation at these points. It is not just a matter of curve-curve intersections. Because the curve represents the path of a mesh this shape needs to be accounted for in the intersection evaluation.

McCrae's sketch-based path design tool can solve intersections [4] but does not detail how to replicate that solution. The video game City Skylines is also another instance of splines being used for road generation with support for intersections but this one is not publicly accessible due to proprietary reasons.

Our research statement is to present a solution for identifying intersecting areas in spline-based road generation. Reparations of the mesh in intersections are outside the scope of this article. The proposed intersection detection algorithm is supported by Andersson's paper on Bezier and B-spline Technology [1], Jarvis' convex hull gift wrapping algorithm [2], Szauer's book on game physics in which he explains Separated Axis Theorem [9] which is used for polygon intersections and Sutherland and Hodgeman's paper Reentrant Polygon Clipping [8]. The research is conducted using the Design Science methodology presented by Peffers et al. [5] thus the aim is to develop an artifact using the game engine Unity [10]. The presented algorithm serves as an extension to standard Bezier tools that support mesh generation which can be considered a framework that the presented algorithm builds upon. There is no fixed way to build curve tools therefore any arbitrary tool could work as long as it is of the Bezier type of cubic degree. In this paper, we build our own. We expect this algorithm to serve as a stepping stone in solving the intersection problem for procedural road generation using splines.

II. Previous Work

A. Bezier Curves

As explained by Andersson in the paper Bezier and B-spline Technology [1], Bernstein Bezier curves interpolate between two anchor points using an arbitrary number of control points as guides. This allows for drawing of smooth non-linear curves. Splines are extremely versatile and find many use cases within the game development field. They can be found in non-linear camera movement, character and AI movement, object modeling, or simulation of soft objects such as cloth.

The Bezier curve was chosen for its convex hull property. As Anderson describes it, the convex hull can be illustrated by stretching a rubber band around all of the control points. The curve will always be enclosed within this convex hull. This fundamental property enables the collision detection used in this paper. Moving forward, all references to curves can be assumed to be a Bezier curve of the cubic degree meaning it features two anchor points and two control points.

B. Road Generation using Splines

Procedurally generating roads using spline technology is a common mixed-initiative approach to keep the designer in control of the results whilst road segments are defined by more than their start and endpoint.

This approach is featured in A Survey on Procedural Modelling for Virtual Worlds by Smelik et al. [7] and executed as a tool by McCrae in Sketch-Based Path Design [4]. McCrae's solution makes use of clothoid splines to allow users to directly draw curves with pen tools that can be translated into 3D geometry. It solves the intersection problem but unfortunately, the solution is not detailed enough to be recreated. Additionally, this paper will not use clothoid splines or support drawing with pen tools.

Generating 3D geometry along the curve requires the generation of various mesh data. At the least, vertices, triangles, normals, and UVs are required to have a mesh that can utilize lighting models and texturing. One way of achieving this is using the sweeping 2D curves technique proposed by Joshi [3] where 2D cross-sections are swept along an arbitrary 3D space curve. By defining a vertical "slice" or edge loop of the desired mesh and repeating it along a path, mesh extrusion is possible along a curve. By using a variation of this approach, it is possible to define a 2D cross-section of a road mesh, sweep it along a Bezier curve at a certain interval storing vertex points as mesh data, and connecting the vertices into triangles.

Nascimento et al. use an alternate approach that gets rid of the need to generate all the mesh data and instead uses a 3D curve to shape a terrain mesh [11]. The curve serves as a guide that can push and pull vertices on the terrain and flatten it to match a road. This unfortunately creates a dependency on a separate terrain mesh, forcing the roads to be projected along the terrain. Since the aim is to trace geometry along a curve without relying on additional terrain tools this method can't be used for mesh generation and can't solve mesh overlap problems at intersections.

C. Curve-curve intersections

There are different ways to solve curve-curve intersections. Sederberg and Parry present an approximate algorithm called Bezier Subdivision [6] which utilizes the previously mentioned convex hull property of Bezier curves. The curve can thus be simplified to a series of interconnected polygons that contain the curve. The intersection problem can now be solved by finding polygon collisions. To improve the accuracy of the convex hull, the Bezier Subdivision algorithm [6] subdivides the curve segments using the de Casteljau algorithm [1] which creates new convex hulls that more closely outline the curve.

This does not take the road mesh into account leaving parts of the mesh outside the convex hulls. To solve this some adjustments are required. Two copies of each control point are created and transformed using the positive and negative bounds of the 2D cross-section. The curves convex hull is now represented by eight control points that mirror the road mesh.

D. Convex Hull Gift Wrapping

At this point, the polygons are actually a collection of unordered points and if the polygon were to be drawn it could generate a concave shape with self-intersecting edges. Before the n-gon intersection can be solved, the points need to be re-sorted to only include points that lie on the convex hull of the polygon and are ordered in a counter-clockwise formation.

This is achievable using the Jarvis March algorithm presented by Jarvis [2]. The Jarvis March algorithm is also known as the convex hull gift wrapping algorithm and will be referred to as such for the remainder of this paper.

E. Intersection Detection using Separating Axis Theorem

Separating Axis Theorem, SAT for short, can be used to find collisions between two arbitrary convex shapes. This is another reason why the gift wrapping algorithm had to be used. Szauer offers an explanation of SAT in the book Game Physics Cookbook [9]. SAT aims to find one axis where the shapes are separated. If an axis of separation can be found, the polygons are not in collision. Each edge normal in each shape represents a potential axis of separation. The shapes are projected onto each axis and if they overlap on each axis the shapes are intersecting.

F. Polygon Clipping

It is now possible to find intersections between different curve segments represented by convex hulls. However, it is not enough to know whether or not they intersect. The part we are interested in is the area where these polygons overlap. Polygon clipping is used to find this overlap.

Polygon clipping is useful for cutting out the area where two shapes intersect. This is achievable using an algorithm proposed by Sutherland and Hodgman in 1974 which is appropriately named the Sutherland-Hodgman algorithm [8]. This algorithm also expects the two passed in polygons to be wrapped which is another reason for the gift wrapping algorithm.

III. Method

This research is carried out following the Design Science Research Methodology (or DSRM for short) presented by Peffers et al. [5]. The DSRM divides the workflow into six activities, all of which are covered in this paper. The following numbered list functions as a reading guide for this article, describing each activity's purpose based on Peffers et al. DSRM practices and under which chapter each activity is discussed.

1) *Problem Identification and Motivation*
   This initial activity has the goal of identifying the specific research problem and justifying the solution's value within the area of concern [5]. This activity is detailed in the next chapter (IV. Problem Identification and Motivation).
2) *Definition of the Objectives for a Solution*
   In this activity, the problem definition proposed in activity 1 is used to infer the objectives of the aforementioned solution [5]. This will be useful for managing the scope of the project in terms of what is a feasible execution. This activity is detailed in the fifth chapter (V. Objectives).

3) *Design and Development*
This activity concerns the development of the artifact and its functionality and architecture [5]. In this instance development of the artifact refers to the creation of an intersection algorithm for spline tools with support for procedural mesh generation. Thus the artifact can be divided into two components. The first component is a curve tool for creating roads with procedural mesh generation. The second component is an intersection detection algorithm for identifying and generating a polygon that encapsulates intersections. The intersection algorithm is an extension of a bezier curve drawing tool. Therefore a curve tool will be developed specifically with the purpose of extending its functionality. This curve tool is not the aim of this paper but rather a means to an end. It is a representation of what any arbitrary curve tool should support for this algorithm to work but it will not be evaluated for this very reason. It needs to be noted that the intersection algorithm is the presented artifact. Since it should be flexible enough that any arbitrary Bezier tool could implement it, the one developed and used in this paper could be substituted for any other curve tool of the Bezier type of the cubic degree. Other degrees could be supported but would require additional tweaks not presented in this paper. This activity is detailed in the sixth chapter (VI. Design and Development).
4) *Demonstration*
In the fourth activity, the artifact is demonstrated solving the research problem [5]. This activity is detailed in the seventh chapter (VII. Demonstration).
5) *Evaluation*
In the fifth activity, the artifact is evaluated by comparing its objectives (Activity 2) to the results of the demonstration (Activity 4) [5]. This paper will evaluate the solution based on two quality attributes:

a) **Correctness**
The correctness of the intersection detection needs to be evaluated by analyzing the accuracy of the identified intersection areas. This is necessary because the algorithm generates a close approximation of the intersection and features a collection of settings that can change the results of the detection. The evaluation is performed by visually observing and comparing the resulting intersection to the actual road.

b) **Performance**
The performance is evaluated by running a series of stress tests with the goal of achieving a minimum of 30 frames per second. The tests will be performed by us in Unity by sampling data from the Profiler. Performance is tested against CPU: i7-4790k, GPU: NVIDIA GeForce GTX 960, RAM: 16GB.
The time complexity of gift wrapping, De Casteljau, SAT, and polygon clipping are evaluated. The impact of different settings and features is also identified.

This activity is detailed in the eighth chapter (VIII. Evaluation).
6) *Communication*
In the final activity the problem, the proposed solution, its effectiveness, and future work are communicated to researchers [5]. This activity is split into two chapters, the ninth chapter (IX. Discussion) and the tenth chapter (X. Conclusion).

IV. Problem Identification and Motivation

In game development and design of virtual environments modelling interconnected structures (roads, train tracks, roller coasters) and placing them manually is a time-consuming task. It is also not friendly to fast changes. If a designer desires to change a road because of feedback the design process could require restarting and rebuilding the entire street.

Curve tools with procedural geometry are useful in significantly improving this workflow since a road segment can be shaped using curve tools in a matter of minutes and the road would dynamically build along the curve.

One problem with curve tools is finding intersection points. This is usually performed using approximate techniques such as those presented by Sederberg and Parry [6] and are suitable when the curve is a line segment.

This becomes increasingly problematic when procedural mesh generation along the curve is supported. Depending on the mesh, it can extrude from the curve in all directions. A road for instance would extrude perpendicular to the curve tangent. This means intersections may occur between road segments when the line segments do not overlap.

This does not begin to cover how to rebuild the mesh at points of intersections to adapt the geometry - a separate problem that is large enough to be its own topic.

Because of the problematic nature of intersections for curve tools that support mesh generation these tools are often not used for street roads. Instead, these types of tools can be limited to objects that wouldn't normally intersect, ropes for instance, or intersections are avoided, for instance, single-lane roads such as race tracks.

By solving the intersection problem for curve tools with procedural mesh generation their use cases can increase which in turn would improve the workflow of designing interconnected structures in virtual environments.

There are video games that use spline technology for path generation, City Skylines being one example. These solutions are either proprietary or unpublished therefore making them inaccessible.

There exists research that uses curves to generate roads that support intersections. Nascimento et al. proposed using 3D Bezier curves to edit terrains [11] but this creates a dependency on additional terrain tools. McCrae's sketch-based design tool for drawing roads [4] is the desired solution but does not detail how the intersection detection works.

V. Objectives

The objective of this research is to propose an algorithm for identifying intersections for Bezier tools that support procedural mesh generation. The mesh considered in this paper is roads.

In this thesis, the problem is discussed from an intersection point of view because it is the easier problem to visualize. However the solution is not limited to only these cases and is still applicable in all junction type scenarios, hence the title.

The intersection detection algorithm is an extension of Bezier tools which will serve as a framework which this solution builds upon. There is no definite curve tool for generating roads therefore any arbitrary tool could serve as the framework. For this paper, a custom one is built and necessary features and limitations are discussed in the second chapter (II. Previous Work). Because the Bezier tool serves as the base and insight into algorithm requirements, the Bezier part is not designed for testing and will not be evaluated. Additionally, both the Bezier tool with the intersection algorithm should be considered as a proof of concept and not a finalized build.

The Bezier curve has been limited to a polynomial of the cubic degree and restricted to the XZ plane. Thus the curve is in 2D space. Whilst the mesh is generated in 3D space and will extrude in all three directions, only the width is considered in the algorithm effectively projecting the road onto the XZ plane.

The proof of concept is built inside of the game engine Unity [10]. This artifact is built into Unity's editor window as an editor script making it possible to run the artifact offline. The algorithm however is not dependent on the game engine and can be implemented in other engines.

Finally, the aim of the intersection detection is not to solve reparations of the mesh and texturing at intersections. Only to find a technique for identifying intersecting areas.

VI. Design and Development

Table[I] explains what the colors and shapes represent in all the figures. Figure[1-4] are challenges that were encountered during development. What the complete implementation looks like is covered in the next chapter (VII. Demonstration).

ITEM	COLOR + SHAPE
Anchor Point	Black Dot
Control Point	Black Dot
Bezier Curve	Green Curve
Non-intersecting Segment	Red Polygon
Intersecting Segment	Blue Polygon
Intersecting Points	White Dot
Intersection Hull	White Outline

TABLE I: In the figures, the following colors represent the following items.

The artifact this paper sought to develop can be separated into two parts, the spline itself and the proposed intersection detection.

The reason for developing the spline part instead of just using an already existing spline creator was to understand which prerequisites and dependencies the intersection method would end up having. This also highlighted what kind of tools that the intersection detection would work with.

Since the results should be applicable for game developers, the artifact is produced in the game engine Unity [10]. It is built into the Unity scene editor directly and updates every time the GUI event Repaint is called.

A. Spline

The Bezier spline was chosen because of its convex hull property [1] that allows for easier intersection detection.

For mesh generation, a 2D cross-section was defined and was then repeated at evenly spaced points along the spline and connected into triangles [3]. This makes it so that the mesh is the same size over the whole spline and that the length from the mesh can be easily attained for the intersection detection.

B. Intersection Detection

The intersection detection can be separated into a couple of different problems:

1) Subdividing the spline.
2) Creating a hull that covers the whole mesh instead of just the spline.
3) Finding which subdivided segments that intersect.
4) Creating a new polygon by clipping it out of the two intersecting hulls.
5) Merging neighbouring clipped polygons.
6) Creating a convex hull over the merged polygons so that they can be outlined and compared.

The convex hull property of Bezier curves is used to turn the curve-curve intersection problem into a simple but approximate polygon collision problem. The De Casteljau algorithm [1] is used to subdivide the curve into smaller segments so that the intersection detection will look at smaller more accurate areas. Following the principles of the Bezier Subdivision technique in [6] the curve segments are turned into convex hulls.

If only the points gained from De Casteljau's algorithm were used it would only cover the spline and not cover the edges of the mesh. This is evident in Figure[1] where a hull is only based on a segment's control points. It accurately captures the curve but misses large chunks of the mesh. To solve this the points are offset to the left and the right by the 2D cross-section's width according to the direction of the spline which returns eight points instead of the original four. These eight are used to create the convex hull.

The convex hull now covers all of the mesh no matter where our control points are. These points are randomly ordered and do not necessarily outline a convex polygon without self-intersecting edges. Figure[2] shows an inflated but unordered hull that is clearly not convex. The gift wrapping algorithm

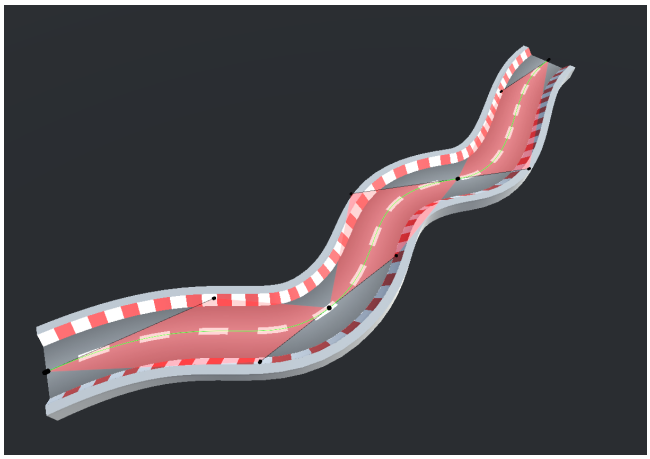

Fig. 1: Convex hulls without mesh offset.

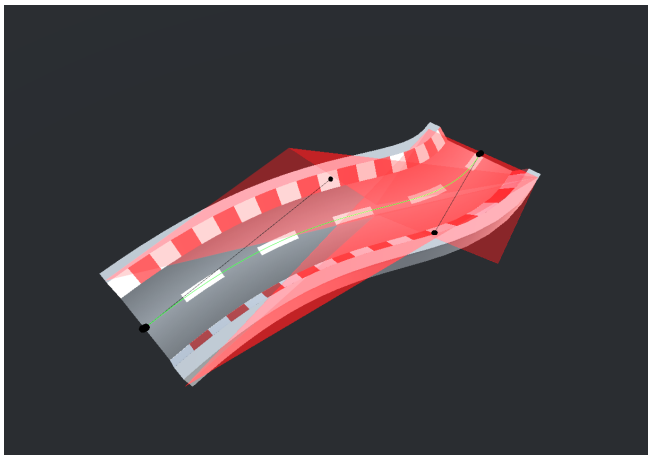

Fig. 2: Convex hull without gift wrapping.

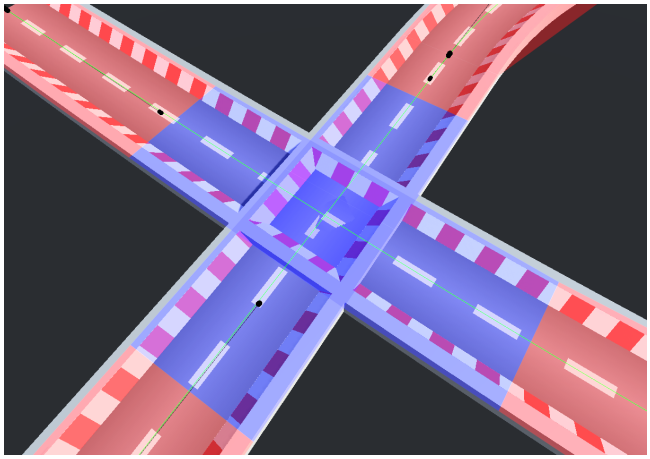

Fig. 3: SAT alone is not enough to identify the intersecting area.

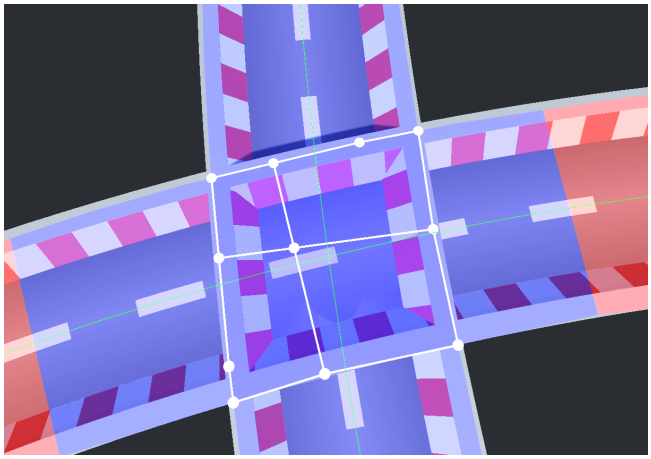

Fig. 4: Without merging polygons in intersections each segment becomes a separate polygon.

in [2] solves this problem and ensures the polygon is convex which is necessary for the next step.

To find intersections between polygons the Separate Axis Theorem [9] is utilized. In Figure[3] intersecting areas has been colored blue. Whilst SAT can tell whether segments overlap that alone is not enough because it considers the entire blue area as an intersection which is not the case. The excess blue outside the overlap needs to be clipped.

After an intersection is detected, polygon clipping [8] is used which returns a resulting polygon that only includes points that are part of the union of the two starting polygons.

These polygons are then checked if they are connected to the other newly created polygons. This is done by checking if the two polygons contain any of the same points. If they do then the first polygon gains the second polygon's points, and the second polygon is deleted, ultimately merging neighbouring polygons. Without this step, the intersections would consist of multiple polygons which would make it difficult to work with. One intersection should be represented by one polygon. Figure[4] shows this problem.

Since the points are now no longer in order the list of points has to be reordered to visualize them as polygons. This is done by using the gift wrapping algorithm again. This has the added benefit of removing any points that are in the middle of the newly created polygon leaving only the hull.

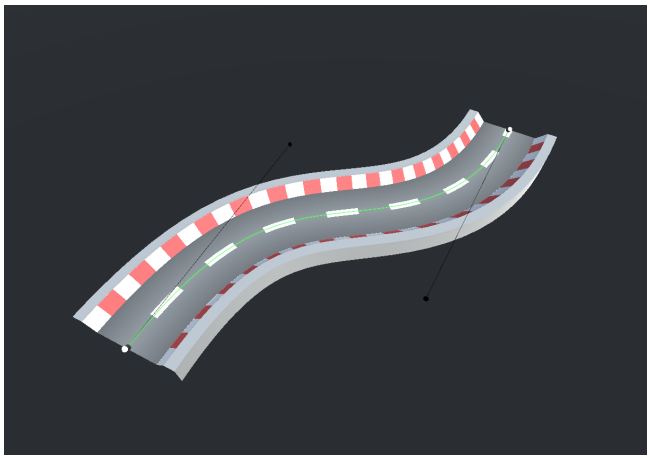

Fig. 5: Non-intersecting curve segment with the road mesh enabled and the intersection algorithm disabled.

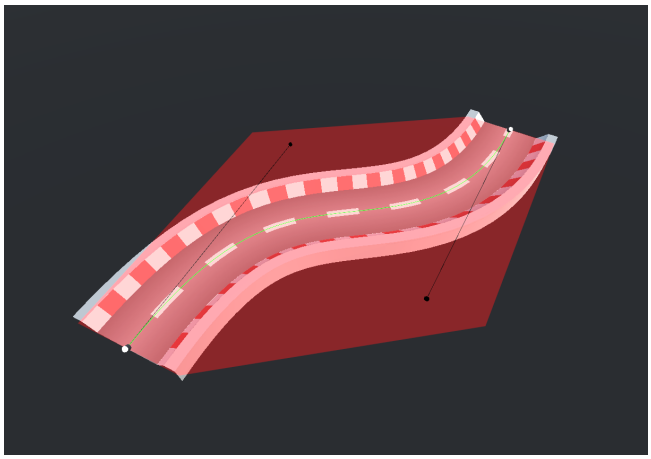

Fig. 6: Same curve as Fig.[5] but with the intersection algorithm enabled at a subdivision count of 0.

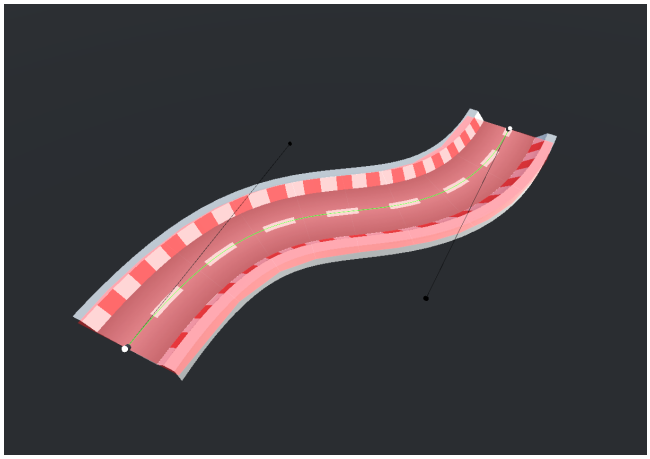

Fig. 7: Same curve as Fig.[6] at a subdivision count of 3.

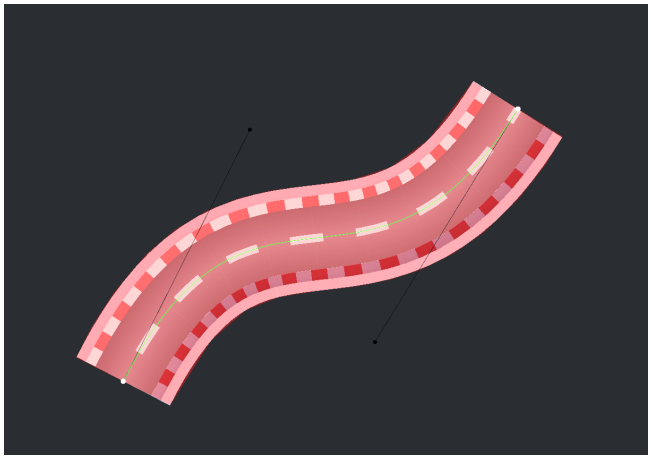

Fig. 8: Same curve as Fig.[7] but from a top-down view.

VII. Demonstration

In Figure[5] the curve is a single segment of two anchor points and two control points. The 2D cross-section is extruded along the curve creating a seamless mesh. The intersection detection algorithm is disabled.

In Figure[6] the curve is the same as the previous one but with the intersection detection algorithm enabled. The red polygon represents the convex hull of the curve after it has been inflated to fit the mesh. It detects no intersections yet. The curve has not been subdivided which is why it covers a large portion of space outside of the road.

In Figure[7] the curve is the same as the previous one but the subdivision count has been increased from 0 to 3. This subdivision is enough to clip the excess space captured in the previous figure. A subdivision count of 3 is used in all further demonstrations.

Because of the camera perspective, it might look like the non-intersecting segment hulls are not as accurate as they are. Because the road mesh is extruded in all directions but the convex hulls are projected onto the XZ plane the perspective is warped even further. To prove that the hulls in fact cover the road, Figure[8] shows the same curve as before from a top-down perspective.

In Figure[9] two curves cross each other. This image clearly communicates what the problem statement was referring to. If this was intended for a racing game the track would become unplayable. The texturing is also incorrect because the roads overlap at the intersection area therefore the texture appears to be flickering in this spot.

Using the same situation as before but enabling the intersection detection, Figure[10] shows it capturing the intersecting area. The white dots and outline represent the polygon that encapsulates the overlap.

By disabling the mesh in Figure[11], the overlap becomes clearer.

In Figure[12] the curves are no longer intersecting. Only the meshes overlap and whilst not as bad as before it is an undesirable result.

In Figure[13] the case is the same as before but with the algorithm enabled. The green curves are not touching and only

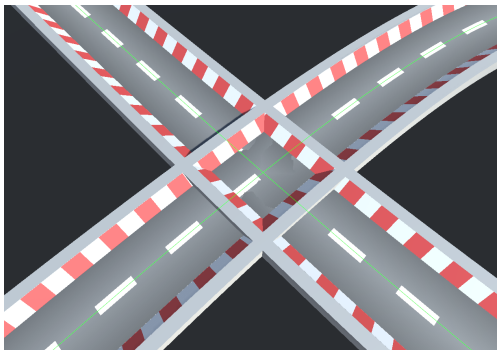

Fig. 9: Intersecting curve segments with the road mesh enabled and the intersection algorithm disabled.

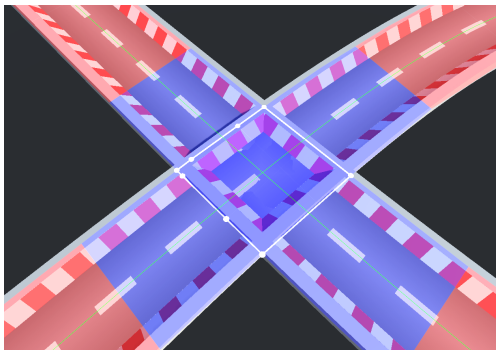

Fig. 10: Same curve as Fig.[9] but with the intersection algorithm enabled.

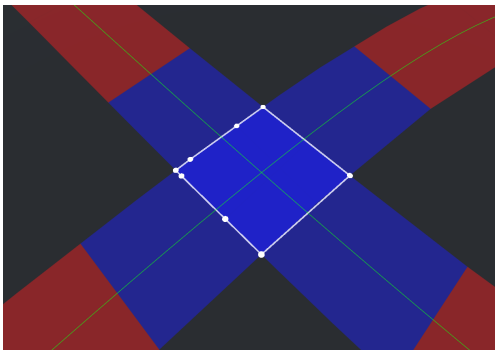

Fig. 11: Same curve as Fig.[10] but with the road mesh disabled.

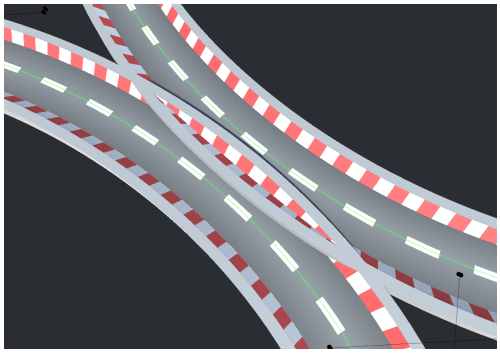

Fig. 12: Non-intersecting curve segments with overlapping mesh with the road mesh enabled and the intersection algorithm disabled.

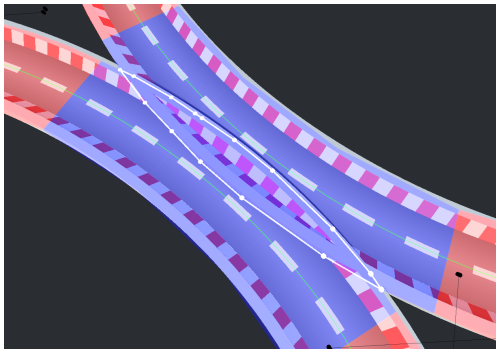

Fig. 13: Same curve as Fig.[12] but with the intersection algorithm enabled.

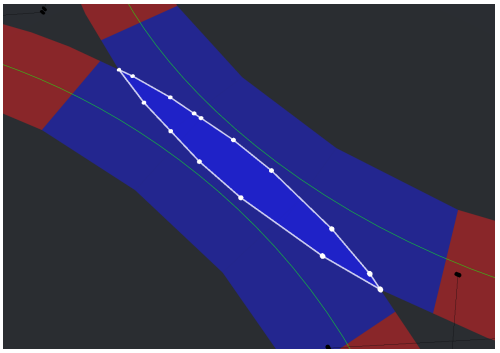

Fig. 14: Same curve as Fig.[13] but with the road mesh disabled.

the part where the mesh collides is outlined.

Figure[14] is the same case but with the mesh disabled to make the overlap clearer.

A. *Performance Tests*

Several tests were conducted to evaluate how different settings impact the performance. The data is presented in Table[II-IV]. The headers represent different curve settings and statistics and are explained in Appendix A. The results are an average value based on multiple test results. In all the tests the curve is static meaning it isn't manipulated and moved when the data is collected.

Table[II] shows the data from the tests concerning the subdivision of the curve. In these tests, the curve path is never changed. It is only subdivided which means the curve is split into multiple new segments that are used internally to create hulls. The subdivision happens evenly over the entire curve. These tests feature no intersections. The Mesh property was disabled for tests at 5, 80 and 320 segments to evaluate how re-building the mesh every frame impacts the performance.

SubDiv	Seg	FPS	ms	Tris	Mesh	Length
0	5	150	6.67	5k	O	220
0	5	200	5.00	5k	X	220
1	10	145	6.90	5k	O	220
2	20	135	7.41	5k	O	220
3	40	115	8.70	5k	O	220
4	80	83	12.05	5k	O	220
4	80	99	10.10	5k	X	220
5	160	42	23.81	5k	O	220
6	320	15	66.67	5k	O	220
6	320	17	58.82	5k	X	220

TABLE II: Data from testing subdivision counts impact on performance.

Table[III] shows the data from the tests concerning adding segments by extruding the curve with new points. Unlike the tests in Table[II], these tests change the curve path and make it longer by adding new points at the end of it. Because the curve becomes longer, the length impacts the mesh generation by requiring more 2D cross-section repetition to cover the entire road. This is why Tris increase proportionally to the Length. To highlight the effect of the length on mesh generation, each test

was tried again with the mesh generation disabled as indicated on the white rows. These tests feature no intersections.

SubDiv	Seg	FPS	ms	Tris	Mesh	Length
0	10	125	8.00	5k	O	250
0	10	155	6.45	5k	X	250
0	20	105	9.52	6k	O	280
0	20	130	7.69	6k	X	280
0	40	77	12.99	7k	O	380
0	40	98	10.20	7k	X	380
0	80	47	21.28	8k	O	530
0	80	58	17.24	8k	X	530
0	160	24	41.67	11k	O	810
0	160	30	33.33	11k	X	810

TABLE III: Data from testing curve segments' and curve length's impact on performance.

Table[IV] shows the data from testing the intersection detection algorithm. SubDiv count is not present in the table but it was at one throughout all tests. Every other test (white rows) has the algorithm disabled to measure the change in FPS. This curve was extruded in the same fashion as the tests in Table[III] but with intersections. It is worth noting that the road built for these tests doesn't reflect realistic road networks and was built in order to fit as many intersections as possible in the smallest length achievable.

Seg	FPS	ms	Tris	IntAlg	Length	IntSec
30	80	12.50	7k	O	760	5
30	103	9.71	7k	X	760	5
50	55	18.18	15k	O	1270	10
50	79	12.66	15k	X	1270	10
92	31	32.26	25k	O	2270	20
92	55	18.18	25k	X	2270	20

TABLE IV: Data from testing the intersection detection algorithm's impact on performance.

VIII. Evaluation

A. Correctness

Figure[6] shows that the polygon outlining the road is not entirely accurate. The subdivision count is too low to capture the curve and would be prone to detect illegal collisions. In figure[7 8] the curve has been subdivided three times which significantly improves the accuracy. This resolution is enough to properly outline the road.

Figure[10 11] captures the 4-way intersection despite using approximate measures to achieve this result. Considering this the result is good enough to identify the intersecting area.

Figure[13 14] proves that the algorithm can solve intersections when only the mesh overlaps. Despite the curves not overlapping, the intersection area is correctly traced.

B. Performance

Multiple different algorithms are run after each other. These algorithms have different time complexities:

1) De Casteljau = O(n^2) where n is the number of points on the spline.
2) Gift wrapping = O(nh) where n is the number of points in the polygon and h is the number of points on the convex hull.
3) SAT = O(n^2) where n is the number of points from the polygon with the most points of the two polygons that were used.
4) Polygon clipping = O(nm) where n and m are the number of points on the respective polygon.

In the code, there is a double for loop that runs over all the segments. It runs the three algorithms SAT, polygon clipping, and gift wrapping.

This double for loop has the highest time complexity in the code. In this loop, the SAT is run every time to check for collision. If a collision is detected then the other two algorithms are run.

All three of these algorithms however have a cap on how long they will take since the polygons that are used have a fixed number of points, between four to eight.

The total time complexity becomes O(n^2x^2) where n is the number of segments and x is the number of points on the polygon which is between 4-8 points.

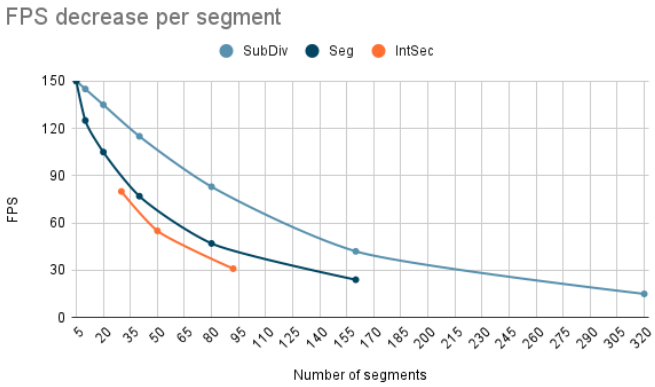

Fig. 15: Graph showing how the fps drops as the segment count increases for Table[II-IV]. Data with Mesh and IntAlg disabled is not plotted.

The graph is a visualization of the cost of the three test cases from the previous chapter.

SubDiv (light blue) corresponds to Table[II] and performs the best being able to reach over 200 segments before reaching the 30 FPS threshold.

Seg (dark blue) corresponds to Table[III] and sees a significant drop in performance. There are two primary reasons both related to curve length. These are discussed later on in this chapter.

IntSec (orange) corresponds to Table[IV] and performs the worst. This is because intersections are also added which require more actions to be performed after SAT returns positive collisions.

In all three cases the intersection detection algorithm is active. Whilst no intersections are present in SubDiv or Seg, SAT still needs to run on each segment to reach that conclusion. That means that that the number of segments and the algorithm cost are related. More segments means more hulls which require SAT to perform collision checks for each one. When intersections are also present, the cost goes up further because polygon clipping, merging and more gift wrapping is required.

The time it takes to rebuild the mesh is tied to the length of the curve. As stated, a longer curve increases the number of 2D cross-section repetitions required to extrude the mesh and as a result the number of Tris. This is part of the reason why SubDiv performs better than Seg. In Table[II] the curve retains the same Length and Tris no matter how many segments it gets. The cost of generating the mesh is therefore the same at all segment counts. Yet disabling the mesh cost at 5, 80 and 320 segments saves 50, 16 and 2 FPS respectively. Initially this doesn't seem like constant cost. The reason being that as the segment count increases so does the SAT collision checks. The SAT cost is negligible at 5 segments compared to the Mesh cost. At higher segment counts this is reversed so the mesh cost becomes negligible compared to the SAT cost which is why we save successively less by disabling the Mesh property.

This effect can also be spotted in Table[III]. Disabling mesh updates saves 30 FPS at 10 segments but only 6 FPS at 160 segments, incrementally saving less the more segments are present. Since the Length is changed in these tests, the mesh cost is not static, however it still becomes negligible as the SAT cost becomes dominant. This relationship shows how expensive the SAT part of the intersection detection algorithm is.

The impact of drawing the curve (see green curve in figures) and control points (see black dots in figures) to the screen was not included in the testing. This is because visualizing the curve to the user is an essential part of any curve tool and exclusion is not an option because it makes it impossible to work with. This integral part naturally has an associated cost that can't be avoided. Normally the cost for subdivision and extrusion should be about the same. However, because extrusions increase the curve length and the number of control points, the cost increases with the amount of things that needs to be drawn to the screen. Subdivisions generate more segments along the curve but these aren't used when visualizing the curve. As a result the cost of drawing is not changed.

This cost can be seen in the results between Table[II] and Table[III]. Despite having the same number of segments, the results differ by a large margin.

Table[III] reaches its FPS cap of 30 at 160 segments with mesh updating disabled. Table[II] is at 42 frames with the same segment count but with the mesh updating still enabled. This discrepancy is due to the high cost of visualizing the curve to the user.

Table[IV] shows that 20 intersections is a soft limit to maintain 30 FPS. At 92 segments with 20 intersections, 31 FPS can be achieved with mesh updating and the intersection detection algorithm enabled. This value is low but affected by several other factors. As previously stated Length affects both mesh generation and requires a longer curve to be drawn on screen. This coupled with 25k Tris and 1 subdivision drastically lowers the FPS before the intersection detection algorithm runs. At 92 segments, the FPS is already reduced to 55 without running the intersection detecting algorithm. Enabling it costs an additional 24 FPS. Two conclusions can be drawn from this. Firstly, multiple external factors affect performance. Secondly, the intersection detection algorithm is very costly to run. Thus optimizations are suggested in the next chapter.

Since the tests were performed on static curves, performance can be more volatile when the curve is being manipulated.

The thing that decreases the performance the most is creating more segments either through adding them to the spline or increasing the number of subdivisions De Casteljau's algorithm does because this generates more hulls for the intersection detection algorithm to check.

IX. Discussion

This paper is the first step in understanding and solving the much bigger issue of adapting the road mesh to fit intersections and junctions. As demonstrated the intersection detection algorithm can successfully carve out road intersections at points where the mesh overlaps.

The algorithm proved to be expensive to run every frame. Since the artifact is built to be used inside the editor, real-time performance isn't as stressed of a requirement. For very long curves, the algorithm execution time is slow but this is on a per object basis. Since the artifact is a component placed on game objects, its scripts don't run if the object is deselected. The road remains in view but the road isn't rebuilt every frame.

Assuming the designers isn't building the entire virtual world's road network on a single object, performance can remain stable at around 100 segments per object. Thus, without any performance improvements, the algorithm can be a feasible solution to detect all types of junctions in roads including intersections. For real-time use, as perhaps an in-game level editor tool, this isn't as feasible. The curve can't be rebuilt every frame and excessive subdivision isn't an option since SAT would perform too many unnecessary collision checks.

There are still areas in which the artifact can be improved both in terms of its limitations and algorithm cost which could result in feasible real-time execution.

A. Limitations

The intersection detection was specifically designed with Bezier curves of cubic degree in mind to fully utilize the convex hull property. The solution will likely not function with other curves that don't have the same properties. The same is true for Bezier curves of other degrees but this is likely easier to add support for.

B. Improvements

There is a clear connection between the number of curve segments and the performance. This is obvious when reaching hundreds of segments during which the program slows down. Various planned improvements were never implemented due to time constraints.

Currently, the curve is recreated and re-skinned in every Unity event Repaint call which is triggered by almost any other event. That means that the curve is cleared, calculated, skinned, subdivided, gift wrapped, SAT, polygon clipped, merged and gift wrapped every frame. This is a problem that primarily concerns implementations in Unity's scene editor. This is however still an area in which the solution could be improved so that update calls are only triggered when the actual curve is updated.

One drawback of the current curve subdivision is that the entire curve is subdivided even in areas where, at a subdivision count of 0, no intersections would be detected. This is a waste of resources so an improvement could be dynamic subdivision. The thought behind it was to start by checking for intersections at the lowest resolution, 0, and only in hulls that returned positive would the curve be subdivided recursively until no more intersections were found or a subdivision count limit was reached. This strategy limits the number of hulls created which keeps SAT from running excessively.

The SAT algorithm is run the most and is one of the more expensive algorithms. If another more efficient polygon intersection solution could replace SAT it could result in a lower time complexity and improved performance. This goes for all the algorithms. There are other solutions for gift wrapping and clipping polygons that were not tested. SAT is still the largest choke point because it determines if the other algorithms are called inside of the double for loop that runs over all the segments.

X. Conclusion

This paper set out to find a method for identifying intersections in Bezier curves that generate a road mesh along them. Through Bezier technology, Separating Axis Theorem, and Polygon clipping we have found a method that can accurately cut out a convex polygon that captures the entire intersection.

Addressing this is an important contribution to solving spline-based road generation that can create more complex road networks. While we have specifically focused on how to find junctions of the intersection type, the algorithm can identify all types of junctions. This solution is the first step to being able to merge meshes at intersections and increasing the use cases for spline-based road generation tools.

The intersection detection algorithm can be feasible under the right conditions. For real-time/ in-game use or more intensive road designs some changes are required notably on when to update and how to subdivide the curve. This is only true until mesh reparations are to be added. Fixing mesh overlap is likely also very expensive. Before reparations can take place, performance needs to see a significant boost.

A. Future Work

In terms of future research, we believe the next step would be to update the mesh and texture in the intersection area to remove overlap and flickering.

This, however requires the algorithm cost to be reduced first. One effective way of achieving this would be to implement dynamic subdivision and limiting the number of update calls. The algorithms used to build the intersection detection are only proposed solutions. Due to time constraints, alternatives couldn't be tried and evaluated. If faster options can replace SAT, which is the biggest time sink, or polygon collisions can be calculated in a more efficient way, the performance could see a substantial increase.

One could also add support for other curve types so they too can benefit from the intersection detection algorithm. This takes second priority to the aforementioned suggestions according to us.

XI. Acknowledgement

We would like to thank Carl Johan Gribel for his excellent supervision of this thesis and for the many valuable discussions and insights he provided.

We would also like to thank all the reviewers for their constructive feedback.

References

[1] Fredrik Andersson and Berit Kvernes. "Bezier and B-spline Technology". In: *Umea university Sweden* (2003).

[2] R.A. Jarvis. "On the identification of the convex hull of a finite set of points in the plane". In: *Information Processing Letters* 2.1 (1973), pp. 18–21. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(73)90020-3. URL: https://www.sciencedirect.com/science/article/pii/0020019073900203.

[3] Pushkar Joshi. "Curve-Based Shape Modeling a Tutorial". In: *IEEE Computer Graphics and Applications* 31.6 (2011), pp. 18–23. DOI: 10.1109/MCG.2011.96.

[4] James Palmer McCrae and Karan Singh. *Sketch-based path design*. University of Toronto, 2008.

[5] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. URL: https://doi.org/10.2753/MIS0742-1222240302.

[6] Thomas W Sederberg and Scott R Parry. "Comparison of three curve intersection algorithms". In: *Computer-Aided Design* 18.1 (1986), pp. 58–63. ISSN: 0010-4485. DOI: https://doi.org/10.1016/S0010-4485(86)80013-6. URL: https://www.sciencedirect.com/science/article/pii/S0010448586800136.

[7] Ruben M. Smelik et al. “A Survey on Procedural Modelling for Virtual Worlds”. In: *Computer Graphics Forum* 33.6 (2014), pp. 31–50. DOI: https://doi.org/10.1111/cgf.12276. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12276. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12276.

[8] Ivan E. Sutherland and Gary W. Hodgman. “Reentrant Polygon Clipping”. In: *Commun. ACM* 17.1 (Jan. 1974), pp. 32–42. ISSN: 0001-0782. DOI: 10.1145/360767.360802. URL: https://doi.org/10.1145/360767.360802.

[9] Gabor Szauer. *Game Physics Cookbook*. Packt Publishing Ltd, 2017.

[10] Unity Technologies. URL: https://unity.com/.

[11] Bruno Torres do Nascimento, Cesar Tadeu Pozzer, and Flávio Paulus Franzin. “Procedural Editing of Virtual Terrains Using 3D Bézier Curves”. In: *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 2019, pp. 135–143. DOI: 10.1109/SBGames.2019.00027.

APPENDIX

A. *Table 2-4 Definitions*

SubDiv	Refers to the subdivision count which determines how many times each segment should be subdivided.
Seg	Refers to how many segments make up the curve.
FPS	Refers to frames per second.
ms	Refers to number of milliseconds required per frame.
Tris	Refers to the number of triangles in the scene.
Mesh	Refers to whether or not the mesh should be reconstructed every frame. O means on and X means off.
Length	Refers to the length of the curve.
IntAlg	Refers to whether or not to calculate the intersections every frame. O means on and X means off.
IntSec	Refers to the number of intersections in the curve.

Explanations of the headers in data tables II-IV.