M.S. THESIS

Geometry Kernel based on G^1-continuous Circular Arcs

G^1 연속 원호 기반의 기하학 커널

August 2022

서울대학교 대학원

컴퓨터공학부

정 하 선

Geometry Kernel based on G^1-continuous Circular Arcs

G^1 연속 원호 기반의 기하학 커널

지도교수 김 명 수

이 논문을 공학석사 학위논문으로 제출함

2022 년 4 월

서울대학교 대학원

컴퓨터공학부

정 하 선

정 하 선의 공학석사 학위논문을 인준함

2022 년 6 월

위 원 장 _____ 신 영 길 _____ (인)

부위원장 _____ 김 명 수 _____ (인)

위 원 _____ 서 진 욱 _____ (인)

Abstract

We discuss some technical issues in the design and analysis of geometry kernel for planar geometric models designed with G^1-continuous arc splines. In particular, we mainly focus on numerical instability in computing the intersection for planar curves, and how the usage of G^1-continuous arc splines can alleviate some of the instability problems inherent to geometric computations using floating-point arithmetic. With arc spline model's higher numerical stability, we further present algorithms for some essential operations of geometry kernel that require finding precise intersection points.

As an effort to improve the computational efficiency, we also present the data structures for the geometry kernel that is tailored to modeling with G^1-continuous arc splines. The presented data structures greatly simplify the bounding volume computation for planar curves through monotone segmentation and simplification of case analysis. Considering the 8-DOP bounding volume of planar curves, the BVH construction can be made with ease based on the support distances of each bounding volume. Finally, we consider applications of the aforementioned algorithms and data structures to the classical problems, such as convex hull and offset computations.

Keywords: Circular arcs, Arc spline, Geometry kernel, Self-intersections, Boolean operation, Bounding volume computation

Student Number: 2020-26241

Contents

List of Figures

List of Tables

Chapter 1

Introduction

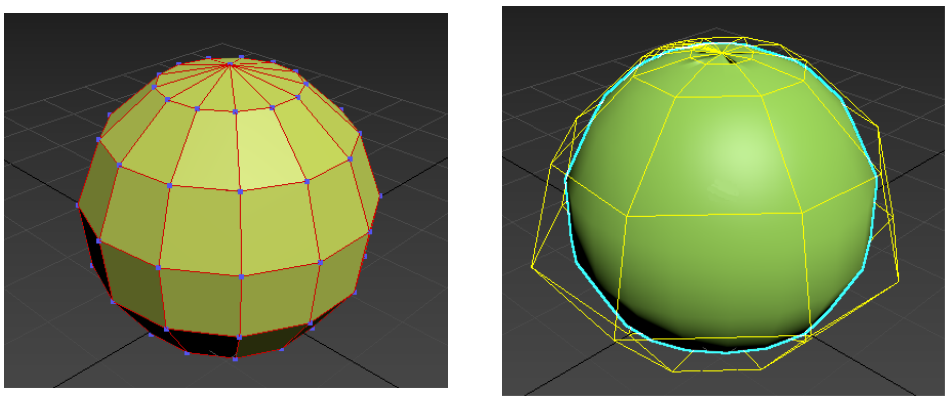
1.1 Background

Geometric modeling kernel is a set of core geometry functions used in 2D and 3D modeling applications. This software component defines the way of storing geometric shapes, and provides algorithms to execute operations such as Boolean operations, distance computation, bounding volume construction, and so on. Conventionally, geometric algorithms are mostly developed based on polygonal models, Non-Uniform Rational B-Spline (NURBS) curves and surfaces. Polygonal models defined by points, lines and triangles are used to represent objects for applications in digital media based on computer graphics technology. Additionally, as the de facto industry standards, commercial products are mostly designed using the NURBS models due to their representational and computational power that supports high precision not only in shape modeling but also in physical simulation.

Nonetheless, each conventional approach presents some limitations. Polygonal models are not G^1-continuous, and thus cannot avoid sharp corners and edges. Compared to NURBS representation, polygonal models show large approximation errors because of sharp edges and flat faces. Furthermore, polygonal models need to store information regarding vertices, edges, and faces, making the data size of objects become enormously large. On the other hand, NURBS models are often represented with high-degree rational polynomial curves and surfaces. When the degrees of the curves and surfaces become higher, the calculation tends to take considerably longer time and consequently can become numerically unstable.

Polygonal models and NURBS models present relative strengths and weaknesses compared to one another. Polygonal models are employed when the rendering time is more important than the precision. NURBS models are preferred when the precision is more important than the rendering time. Thus, polygonal models are mainly used for applications in video games, animations, and other forms of real-time computer graphics, while NURBS models play an important role in product design and engineering simulations.

As a compromise to these two mainstream modeling approaches, there has been some previous work done on arc-based modeling which suggests its efficiency and efficacy. Arc-based models offer a new stream where the advantages of both polygonal modeling and NURBS modeling can be maintained. In this thesis, we discuss some technical issues that must be considered in the development of a new geometry kernel which uses circular arcs to represent planar geometric objects.



(a) A polygonal model

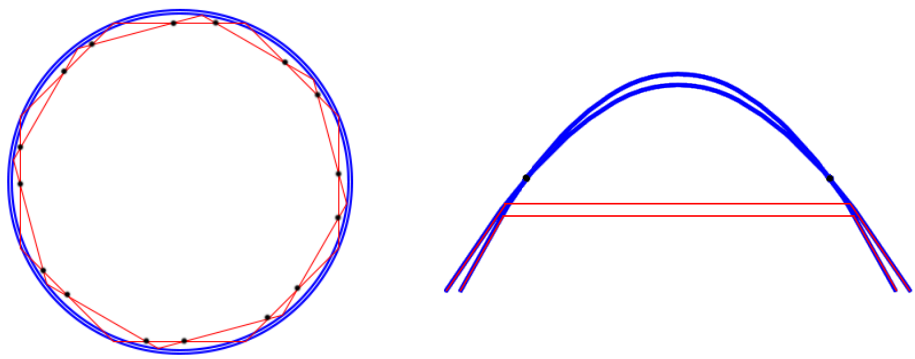
(b) A NURBS model

Figure 1.1: Polygonal model and NURBS model

1.2 Research objectives and main contributions

The geometry kernel based on G^1-continuous circular arcs is based on previous results developed for arcs. The main motivation is to put geometry knowledge about arcs altogether in one place. As a whole, the geometry kernel in this thesis defines how to represent objects using arcs maintaining G^1-continuity. In the course, we aim to demonstrate numerical stability of arc spline modeling through experiments and comparison. While some advantages of arc spline modeling compared to the conventional approaches are apparent, the analysis on numerical stability leads to some interesting new findings.

One of the comparable advantages of arc spline modeling over polygonal modeling is that the number of arc splines required to represent a curve is far fewer than the number of line segments required to represent a curve. This is because approximation error exhibits cubic convergence when an object is approximated by arc splines. On the other hand, quadratic convergence is



(a) False-positive intersection detection (b) False-negative intersection detection

Figure 1.2: Intersection detection using polygonal models

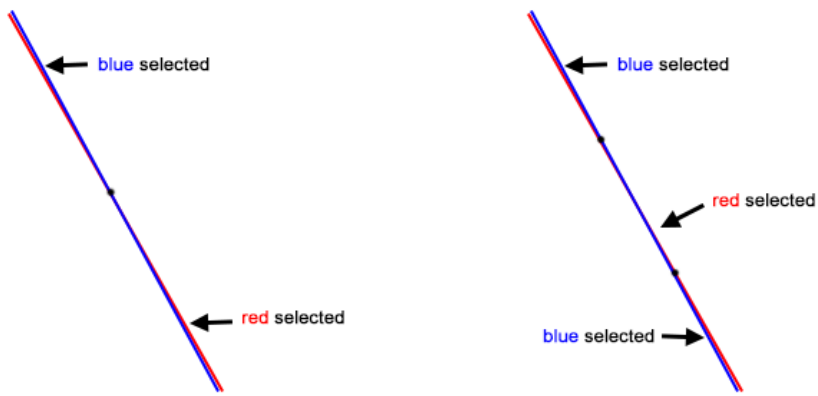
observed when an object is approximated by line segments [2]. This leads to an improvement in computing speed and accuracy for finding the intersection between two curves when they are represented in circular arcs. Furthermore, the arc spline modeling guarantees G^1-continuity, which is more compliant to the design specifications [5, 19].

Besides, polygonal models usually show larger error for finding the intersections. When polygons are used to approximate curves, there is a higher possibility of finding the intersections incorrectly. Intersections detected by polygonal curves often result in false-positive, or false-negative as shown in Figure 1.2. Two concentric circles do not meet each other because they share the same center, but different radii in Figure 1.2(a). However, polygons approximating the circles with 8 line segments meet each other at 16 points. On the other hand, the approximation error in polygonal models can lead to missing

some intersections. Furthermore, the precision in finding the intersections between almost tangential circular arcs is exceptionally high compared to finding the intersections between line segments that are almost tangential. Two circles can have at most two intersection points if their centers or radii are different. However, line segments that approximate almost tangential curves can result in detecting more intersections than the ground-truth because of approximation as well as numerical errors. This issue will be discussed in Chapter 4.2 with some experimental results.

For Boolean operations, the incorrect number of intersections can return wrong results when the algorithm flips the selection based on the intersection point. In Figure 1.3(a), a union operation determines that the blue line encloses the red line above the intersection point, and leaves the blue line segment while removing the red line segment. Then the determination is flipped below the intersection point, and the red line is selected. However, when one more intersection point is detected, the selection flips again, and the union operation returns a wrong result where the blue line is chosen instead of the red line in Figure 1.3(b). As such, some edge cases from polygonal modeling can be handled more reliably using arc-based modeling.

Unlike polygonal model, NURBS model guarantees G^1-continuity, and represents geometric objects more smoothly. But computation for NURBS models requires non-trivial mathematical tools. For arc spline models, the computing time increases proportionately to the number of arcs as the degree of polynomials is fixed to quadratic. However, the calculation becomes comparatively stable. For example, a quadratic equation has two discrete roots, while degree n polynomials can have up to n roots. Lastly, arc modeling allows classical construction using ruler and compass, and is thus quite intuitive.



(a) Correct Boolean operation (b) Wrong Boolean operation

Figure 1.3: Boolean union based on different number of intersections

We also describe data structures and algorithms for a minimal set of basic operations necessary for the geometry kernel. There are many mathematical properties that are useful for geometry modeling when it is based on circles, such as curvature information, or equidistant property. These properties are already utilized in many previous results as premises for efficient algorithms. All these algorithms begin with approximating planar geometric objects with biarcs. Offset computation exploits the radius, circle-circle intersection algorithm, and bounding circular arc, while convex hull computation only employs support distance and angles of the arcs. On the contrary, curvature monotonicity is used to construct Voronoi diagram in a stable way. Although the base models are the same, the information of arcs is used differently. We try to incorporate the necessary information of arcs, and support various operations by unifying the data structures.

We consider some technical issues in the design of spatial data structures. Well-designed data structures and construction rules for arc splines can improve the precision by reducing the numerical errors in floating-point arithmetic, and the number of arc splines to represent planar geometric models. It can also simplify some case analyses required for the algorithms. At the same time, we can make sure that all the necessary data are included efficiently.

To sum up, we introduce a new choice of geometry kernel other than the conventional polygonal modeling or NURBS modeling. With the cornerstone set, we hope to bring about more analyses to be done on arc spline modeling and exploit its advantages. The main contributions of this research are as follows:

- We analyze how arc based geometry kernel can maintain the respective advantages of conventional modeling approaches. We compare the precision error, running time, and the data size between our approach and the conventional approaches. In effect, we can model with arc based geometry kernel for applications in geometric modeling that require stable and precise computation.

- We explain the basic elements of spatial data structures that should be considered in order to support various geometric operations. Once we incorporate the advantageous traits of arcs in the data structure, we show how arc based modeling in this thesis can simplify existing solutions to some classical problems.

1.3 Thesis organization

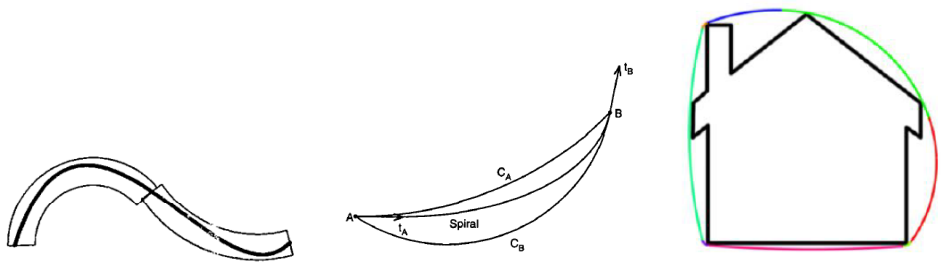
The rest of this thesis is organized as follows. Chapter 2 introduces previous work related to arc splines. Chapter 3 provides the data structures for arc representation and constructions. In Chapter 4, we present an algorithm for detecting intersections between two circular arcs. Chapter 5 addresses the Boolean operations such as union, intersection, and difference using the intersection finding algorithm presented in Chapter 4. In Chapter 6, we explain how to compute bounding volumes of planar objects composed of arcs. In turn, the bounding volumes of arcs is used to detect collisions and to compute distance between arc-based objects. Then Chapter 7 discusses applications of the arc based geometry kernel. Finally in Chapter 8, we conclude this thesis with possible extensions to 3D.

Chapter 2

Related Work

2.1 Biarc approximation

Previous geometric algorithms based on arcs approximate other forms of planar geometric curves with circular arcs, so that they can exploit computational advantages of circular arcs. Many algorithms approximate curves using biarcs to improve the efficiency of operations on planar curves. Especially, Meek and Walton [19, 20] demonstrated that the number of circular arcs required to approximate planar geometric curves is smaller than that of line segments to do the same. They also confirmed that biarc approximation has cubic convergence rate; on the other hand, polygonal approximation exhibits quadratic convergence. Another work of Šír et al. [26] showed how to maintain the G^1-continuity during the biarc construction. In this work, we use the same approximation method using biarcs to analyze the relative performances between the arc splines and line segments.



(a) Fat arc (b) Bounding circular arc (c) 8-arc

Figure 2.1: Bounding volumes using arcs

2.2 Bounding volume hierarchy

Bounding volume is a container that includes objects that users select. There are many forms of bounding volumes such as Axis-Aligned Bounding Boxes (AABB) [4], Oriented Bounding Boxes (OBB) [8], Discrete Oriented Polytopes (k-DOP) [14], and a family of sphere swept volumes [15]. Bounding Volume Hierarchy (BVH) is a data structure that organizes bounding volumes as nodes in a tree structure. BVH is used to accelerate many important operations by filtering out a subset of objects that are clearly excluded from further consideration. When arcs are proposed as the geometric primitives for bounding volumes, the applicability of bounding volumes can be expanded to non-convex objects as well. For planar curves, non-convex bounding volumes such as Bounding Circular Arcs (BCA) [20], n-arcs [21], and fat arcs [23] have been proposed.

2.3 Algorithms using biarc

There are several algorithms that use biarc to make algorithms efficient and stable. One application of biarc is offset curve trimming. Kim et al. [13] approximate planar rational curves with biarcs to trim offset curves efficiently. Furthermore, Lee et al. [16] detect self-intersection and trim offsets of deformable curves using BCA bounding volume. These works show how the usage of biarcs can accelerate intersection detection, and hence offset trimming.

Convex hull can also be computed efficiently using biarcs. Similar to offset trimming problem, Kim et al. [12] approximate curves with biarcs to compute the convex hull of planar freeform curves efficiently. When the curves are represented with biarcs, the interior culling can be done faster using the angle information of arcs. Then the convex hull is computed by concatenating the remaining arcs after the redundancies are eliminated.

Biarcs can accelerate the computation of Hausdorff distance as well. Kim et al.[14] approximate planar curves with biarcs to find the lower bound of Hausdorff distance efficiently. Then they remove redundant arcs based on the lower bound. Among the remaining arcs, they find a point with the maximum distance using a distance map. In the neighborhood around the point, they select candidate arcs. Finally, a precise Hausdorff distance is computed using the candidate arcs.

Biarc approximation is also used to enhance efficiency for the discovery of maximal disk, the computation of medial axis, and the construction of Voronoi diagram. Maximal disks are necessary to compute medial axis and construct Voronoi diagram, and they are easier to locate when planar geometric objects are represented with circular arcs [1]. Lee et al. [17] developed a highly efficient

algorithm to construct Voronoi diagrams, which uses BCA to speed up the process. We delve into these applications in Chapter 7, focusing on possible implementation and benefits with arc based geometry kernel.

Chapter 3

Preliminaries

3.1 Geometric models

The design of a geometry kernel starts with a formal definition of point and vector in two-dimensional space [18]. For vector computation, we shall use homogeneous coordinates to represent points and vectors as in Figure 3.1.

These basic data types are necessary for geometric computations. A point is encapsulated in an arc struct to represent the arc's center. A vector is used to represent the tangent or direction of an arc at a point, and calculated when necessary. We use one struct to represent all basic objects: vertex, line, and arc. These three types of objects are grouped to create a planar region, and can be changed to one another.

Arc struct contains type information, center point, squared radius, two angle structs, and convexity as illustrated in Figure 3.2. The type information stores whether an arc struct is a point, a line, or an arc. The angle struct

```
typedef struct
{
    float x, y, z;
} point;
```

(a) Basic data *point*

```
typedef struct
{
    float x, y, z;
} vector;
```

(b) Basic data *vector*

Figure 3.1: Basic data types

with quadrant information and angle information makes computations using arcs become somewhat quadrant-independent. Two angle structs represent angles at the start and end points. Note that in this thesis, we only consider affine transformation. So we assume one perspective where angle 0 represents 3 o'clock direction, and the angle increases counter-clockwise. Convexity is determined based on the direction in which the angle increases. An arc is concave if its starting angle is bigger than the ending angle. Convexity also indicates the interior of the boundary represented with arcs. We define the left side of the direction as the object interior that the arc bounds.

Then we have constructors to create vertex, line and arc. A vertex is a full circle with 0 radius, and with its center at the location of the vertex. As a result, the input for creating a vertex is only the location of a point.

In order to create a straight line with an arc, the circle containing the arc needs to be very big so that the displayed portion of the arc assimilates a straight line. Thus, the radius for this arc should be big enough. This leads to the center point being far away and out of the screen. But we know that the center point is located in the normal direction of the line. The length of the line will be calculated with the arc struct's angle. In this way, we can gradually shift a curve into line, and vice versa.

```
typedef struct {
    int quad;
    /* quad indicates which quadrant angle exists */
    float angle;
    /* angle is in between 0 to 90 */
} angle;

typedef struct {
    point center;
    float radius;
    angle startAngle;
    angle endAngle;
    bool concave = true;
    /* true if startAngle is smaller than endAngle */
} arcStruct;
```

Figure 3.2: Basic structs

Finally, an arc representation is quite straightforward. Convexity is implied within two angle structs, but we store this information for convenience. Note that the radius stores the squared radius so that we reduce the truncation errors from square root operations. To minimize the number of multiplications and the usage of trigonometry functions, we can store the starting point and the ending point of the arcs as well.

```
#define CREATE_VERTEX(x, y)
    ((arcStruct) { .center = point{x, y},
                   .radius = 0,
                   .startAngle = angle{0, 0},
                   .endAngle = angle{3, 90} })

#define CREATE_LINE(point x1, point x2)
    ((arcStruct) { .center = point at normal direction,
                   .radius = INF,
                   .startAngle = angle of x1,
                   .endAngle = angle of x2 })

#define CREATE_ARC(x, y, r, sAng, eAng)
    ((arcStruct) { .center = point{x, y},
                   .radius = r,
                   .startAngle = sAng,
                   .endAngle = eAng })
```

Figure 3.3: Basic constructors

3.2 Curve segmentation

Many geometric operations on a curve can be greatly simplified when the curve is cut into segments so that each segment changes monotonically [7]. This monotonicity is achieved by cutting the curve at x-extreme and y-extreme points, singular points, and inflection points. When the curve is of d-th degree, the polynomial equations for x- and y- extreme points are of degree $d-1$, and degree $2d-4$ for singular points, and $4d-7$ for inflection points.

Monotonicity along the x-axis or y-axis direction for example, ends when the respective extreme point is reached. In the planar case, these extreme points are calculated as follows:

$$x'(t) = 0,\ y'(t) = 0$$

We can also subdivide the curve at each inflection point, where the curvature is 0. This is a point where the convexity changes to concavity and vice versa. The equation for finding inflection points is given as follows:

$$x'(t)y''(t) - x''(t)y'(t) = 0$$

Finally, when the curve is segmented at curvature extreme points, the curvature is monotone within each segment of the curve. The curvature extreme points can be found by solving:

$$\begin{aligned} &\left(x'(t)y'''(t) - x'''(t)y'(t)\right)\left(x'(t)^2 + y'(t)^2\right) \\ -3&\left(x'(t)y''(t) - x''(t)y'(t)\right)\left(x'(t)x''(t) + y'(t)y''(t)\right) \quad = \quad 0 \end{aligned}$$

In this thesis, we segment the curve at each x-extreme, y-extreme, and inflection points to simplify the geometric operations.

Chapter 4

Arc-Arc Intersection

There are two types of self-intersections: local and global self-intersections. Global self-intersection is relatively more difficult to detect in general. It is also important to detect local self-intersections. A geometrically correct surface should be closed, and free of local and global self-intersections [11]. Like surface models, any planar model should be geometrically correct for stable computation. Furthermore, detecting the intersections between arc splines is a crucial part of many basic operations of the geometry kernel system such as Boolean operations, and collision detection. Local self-intersections cannot occur within a circular arc, but global self-intersections may occur among different arcs. In such case, the global self-intersections should be removed with a proper trimming process.

4.1 Algorithm

Bézier curves are not free of local self-intersections [6]. Compared to Bézier curves, we can show that the arc spline modeling detects the local self-intersections in a stable manner. To do so, we first approximate the Bézier curve with arc splines according to the algorithm of [26]. Then we find the intersections of the arc splines. Detecting the self-intersection for Bézier curves or any other forms of planar curves often uses binormal lines [22], or bounding boxes [25]. Theoretically speaking, the intersection detection algorithm using arc splines only requires some trigonometry, and Euclidean transformations. An algebraic result based on the trigonometry and simple mathematical operations is proposed below.

To find the intersection between the arc splines, we need to compare their arcs pairwise. Notably, G^1-continuity guarantees that the neighboring arcs cannot intersect with each other, so we can skip the comparison between the neighboring arcs. When comparing an arc against the other arc, we first screen them based on the circle-circle intersection to reduce the number of operations. Because an arc is part of a circle, if the circle corresponding to the arc does not intersect with the circle corresponding to the other arc, the arcs do not intersect. Given that r_1 and r_2 are the radii of two circles, and d being the distance between the centers of the circles, we determine whether the circles intersect using the following equation:

$$d \leq r_1 + r_2 \quad \text{and} \quad \mathrm{d} \geq |\mathrm{r}_1 - \mathrm{r}_2|$$

If there is an intersection between the corresponding circles of the arcs, we then move onto finding the intersection points. The computation becomes

much simpler by translating and rotating the centers of the circles to be located on the x-axis, and one of them to be on the origin point as in Figure 4.1(c). When the first circle's center is at the origin point, we have the following solutions for the intersection of the two circles in Figure 4.1(c):

$$x = \frac{d^2 + r_1^2 - r_2^2}{2d},\ y = \pm\frac{\sqrt{4d^2r_1^2 - (d^2 + r_1^2 - r_2^2)^2}}{2d}$$

In the above equation, we only need to know x in order to determine whether the intersection point lies on the arc. Note that x is equal to $r_1 \cdot \cos\theta_1$ and also to $d - r_2 \cdot \cos\theta_2$, where θ_1 and θ_2 are the angles between the x-axis and the intersection point centered on the respective circle's center. When the angle of rotation that places the second circle's center on the x-axis is denoted as θ, the real intersection point needs to be rotated back by θ. If $\theta_1 - \theta$ and $\theta_2 - \theta$ are both in between the start and end angles of the corresponding arcs, it means that the intersection point is on the arcs and that the arcs intersect with each other. The algorithm for detecting the intersection is given in Algorithm 1.

In order to minimize the numerical error, we tried to reduce the number of square root operations and the usage of trigonometry functions during the process of determining whether the intersection point lies on the arcs. We do so by calculating the actual intersection points instead of rotating and translating. Then the actual intersection points for Figure 4.1(a) are represented as follows:

$$x = x_1 + \frac{(d^2 + r_1^2 - r_2^2)(x_2 - x_1) \mp (y_2 - y_1)\sqrt{4d^2r_1^2 - (d^2 + r_1^2 - r_2^2)^2}}{2d^2}$$

$$y = y_1 + \frac{(d^2 + r_1^2 - r_2^2)(y_2 - y_1) \pm (x_2 - x_1)\sqrt{4d^2r_1^2 - (d^2 + r_1^2 - r_2^2)^2}}{2d^2}$$

Algorithm 1: Intersection detection between the two arcs

Result: boolean

input: arcStruct first, arcStruct second

$d \leftarrow$ distance between the center points

if *first.center.x* $>$ *second.center.x* **then**

    swap first and second

$r_1 \leftarrow first.radius$

$r_2 \leftarrow second.radius$

if $(d > r_1 + r_2$ *or* $d < |r_1 - r_2|)$ **then**

    `// the circles don't intersect`

    return false

$c_2 \leftarrow second.center - first.center$

$\theta \leftarrow \arctan(c_2.y, c_2.x)$

rotate c_2 by θ

$x \leftarrow \frac{d^2+r_1^2-r_2^2}{2d}$

$\theta_1 \leftarrow \arccos \frac{x}{r_1}$

$\theta_2 \leftarrow \arccos \frac{d-x}{r_2}$

if $(\theta_1 - \theta)$ *is on first arc* & $(\theta_2 - \theta)$ *is on second arc* **then**

    `// check for the positive intersection`

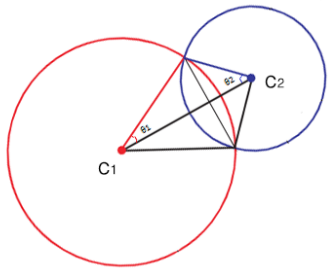
    return true

if $(-\theta_1 - \theta)$ *is on first arc* & $(-\theta_2 - \theta)$ *is on second arc* **then**

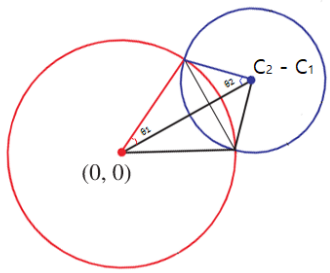
    `// check for the other intersection`

    return true

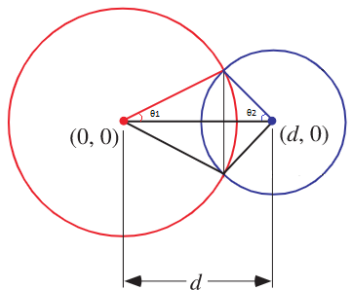
return false



(a) Original



(b) Translated to origin



(c) Rotated the center to x-axis

Figure 4.1: Two intersecting circles

We simply need to get the angles of the actual intersection point with respect to each circle's center. In order to determine if the intersection point lies on both arcs, we test whether those angles are in between the start and end angles of the respective arc's.

4.2 Experimental results

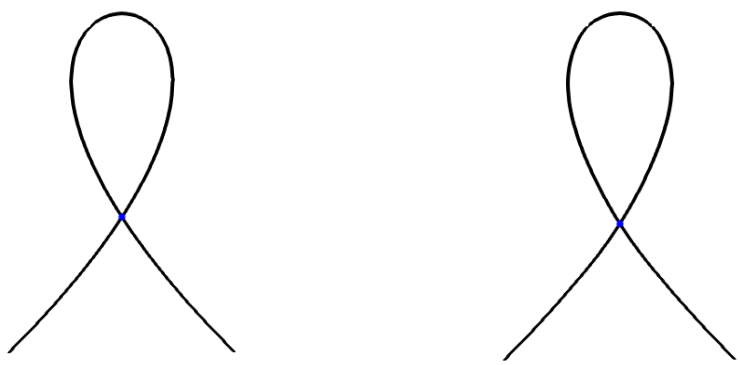
The algorithm is developed using C++ and OpenGL. The data type is in double precision. We have tested for transversal cases, almost identical cases, and almost tangential cases. The increase in the number of line segments does not improve the precision in almost tangential cases, but slows down the running time, and more seriously, the number of detected intersection points also increases. The tables show the number of splines used, running time, and the detected intersection points along with the illustrations. The four control points of a cubic Bézier curve are provided in the first row of the table. The first experiment shows that both line modeling and arc spline modeling detect the self-intersection reliably. But the number of segments for arc spline modeling is fewer, and thus the running time is shorter.

When detecting intersections between two different curves, polygonal approximation shows larger error compared to arc approximation. The test results can be found in Figure 4.3, 4.4, 4.5.

Figure 4.4 and Figure 4.5 use the same Bézier curve, but different number of line segments. These cases are for a scenario where solving the equations for the intersection of Bézier curves produces too many candidates as solutions. Figure 4.4 uses 50 lines segments to approximate each Bézier curve, while Figure 4.5 uses 100 line segments. While the approximation error got lower

Control points: (200, 100), (400, 300), (100, 300), (300, 100)		
	Arc	Line
Number of segments	24	50
Running time (ms)	0.4369	0.7037
Number of intersections	1 point	1 point

Table 4.1: Self-intersection detection



(a) Self-intersection detected by lines

(b) Self-intersection detected by arcs

Figure 4.2: Self-intersection detection

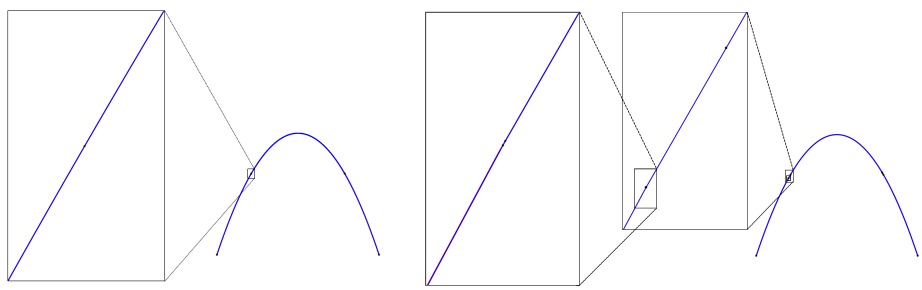
for Figure 4.5, the number of detected intersection points increased as the result of more line segments generated for better approximation. For example, point at (453, 524.91) is additionally identified as an intersection in Figure 4.5. However, with fewer line segments, the polygonal curve in Figure 4.4 does not pass the point at (453, 524.91). As a result, Figure 4.4 could not find (453, 524.91) as an intersection point that can be detected in Figure 4.5. Note that the line segments are more edged in Figure 4.5(b). Also, if a certain range is

detected as intersection, all line segments within that range can be identified as intersection. When more line segments are used, the number of line segments detected as intersection will increase. Nonetheless, both polygonal modeling and arc modeling detected the actual intersection point at (450, 525) reliably.

Overall, we can conclude that the polygonal modeling with lower approximation error tends to detect the intersections of planar geometric curves in excess, and take longer time to find the intersections compared to the arc modeling.

Curve 1 control points: (300, 300), (400, 600), (500, 600), (600, 300)		
Curve 2 control points: (300, 300), (400.01, 600.01), (499.99, 600.01), (600, 300)		
	Arc	Line
Number of segments	32	50
Running time (ms)	0.3321	0.4926
Number of intersections	4 points	7 points

Table 4.2: Intersection between almost identical objects

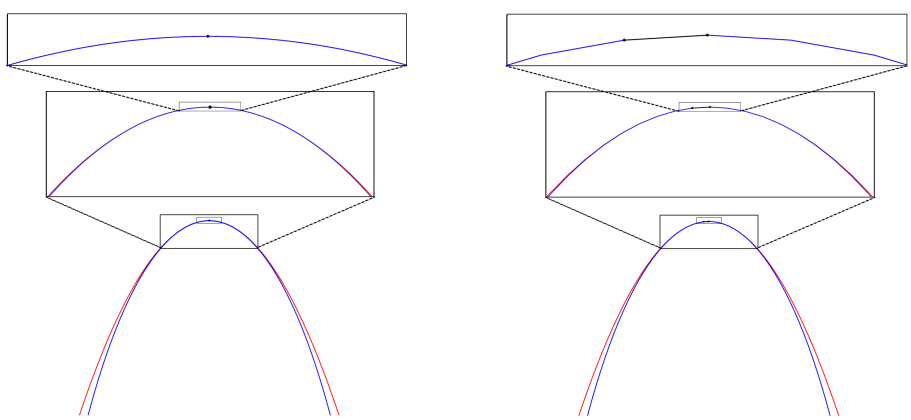


(a) Arc (b) Line

Figure 4.3: Intersection between almost identical objects

Curve 1 control points: (300, 300), (400, 600), (500, 600), (600, 300)		
Curve 2 control points: (310, 300), (390, 600), (510, 600), (590, 300)		
	Arc	Line
Number of segments	32	50
Running time (ms)	0.3309	0.5218
Number of intersections	1 point	1 line segment

Table 4.3: Intersection between almost tangential objects with fewer line segments

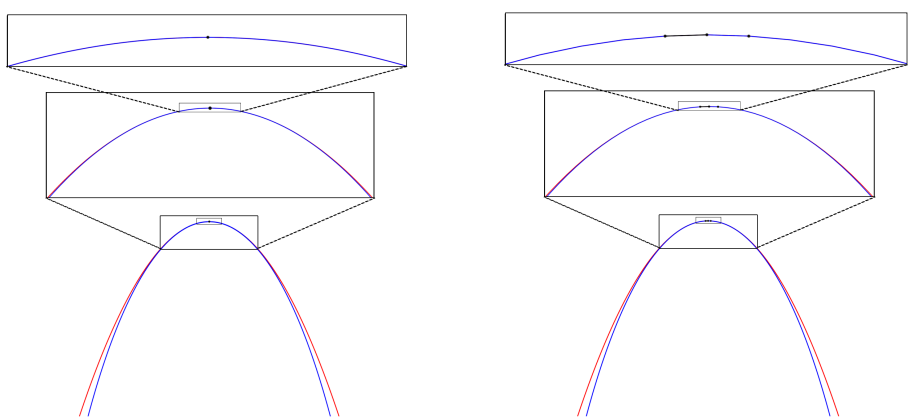


(a) Arc (b) Line

Figure 4.4: Intersection between almost tangential objects with fewer line segments

Curve 1 control points: (300, 300), (400, 600), (500, 600), (600, 300)		
Curve 2 control points: (310, 300), (390, 600), (510, 600), (590, 300)		
	Arc	Line
Number of segments	32	100
Running time (ms)	0.3288	1.863
Number of intersections	1 point	1 point + 1 line segment

Table 4.4: Intersection between almost tangential objects with more line segments



(a) Arc (b) Line

Figure 4.5: Intersection between almost tangential objects with more line segments

Chapter 5

Boolean Operations

Boolean operations are useful for merging two different objects, selecting the common section of the objects, and so on. There are Boolean operations other than union, intersection, and difference, but we will focus on the three basic operations because they are the most essential ones. In fact, other Boolean operations, and even the union and difference, can be reduced to the intersection operation [9].

Finding the intersections between the object boundaries is the first step in a Boolean operation for boundary representation geometric models. The conventional approaches to computing intersection points have limitations in the numerical precision and stability of the calculations. These problems become apparent especially when the objects are almost coincidental [9, 10], or almost tangential. When unnecessary intersections are detected, it negatively affects the performance and even the correctness of Boolean operations. With an increased number of intersections from polygonal models, the objects may

be excessively subdivided, and thus the higher the complexity of algorithms in topological decisions. This may also lead to Boolean operations erroneously removing some curve segments that are important. However, based on the analysis from Chapter 4, discovering intersection points with arcs is quite stable and robust.

The number of intersections between planar objects affects the size of list that keeps the end points and the intersection points. For example, there is only one intersection point between two non-parallel straight lines. The maximum number of possible intersection points between circular arcs increases to two. Now, intersecting two cubic Bézier curves can result in up to 9 possible intersection points. The computation time increases naturally as the size of list that we need to scan becomes larger. As a result, intersecting arcs is more manageable compared to intersecting NURBS curves.

5.1 Algorithm

In the preprocessing stage, the input curve is segmented at the intersection points using the intersection algorithm described in Chapter 4. If there is at least one intersection point, and the intersection is in the middle of the arc splines, the arc splines need to be segmented by adding new vertex at each intersection point. The end points of the arc segments are inserted in the list. The arc segments between the intersection points are removed if they have an improper relation with the other object. For example, the arcs that are determined to be inside the other object are returned for the intersection operation. Finally, the arc splines will be merged to form the boundary of the resulting object of the Boolean operation.

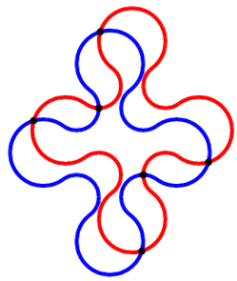
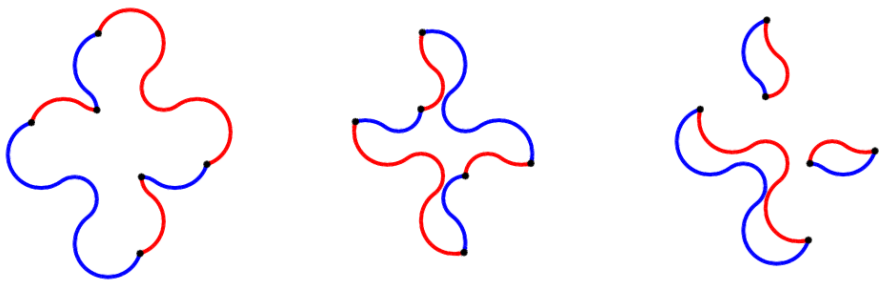


Figure 5.1: Two arc splines with intersection points



(a) Union (b) Intersection (c) Difference

Figure 5.2: Boolean result

Segmenting an arc spline is simple, as we know the location of each intersection point. The segmented arc splines will share many values with the original arc splines, such as one of the end points, center point, radius, and convexity. The intersection point is newly added as a new end point. If the intersection point is an end point of the curve, then we do not split the curve. One intersection point will divide the curve into two arc splines. An additional intersection point leads to one more arc segment. As a result, if there are k intersection points on a curve made of n arcs, the list will keep $n + k$ points. For the Boolean intersection operation, we need to determine which side of the intersection point is the common interior of both objects starting from

the first intersection point. If it is interior up to the next intersection point, the next intersection point exits the interior. The next intersection point will enter the interior of the objects again. Repeating the same procedure, we can construct the boundary for the intersection of the two planar objects.

Lastly, the remaining arcs will be combined in a proper order. To maintain the G^1-continuity of the arc spline, a small arc that diffuses the rapid change of tangent needs to be added between the merged arcs.

5.1.1 No intersection case

We start with a case where there is no intersection point between the boundary curves of two objects A and B. When both A and B are connected objects with no holes, we have either (i) $A \subset B$, (ii) $B \subset A$, (iii) $A \cap B = \phi$. We can check if a sample point on the boundary of A is inside the region B to decide that A is included in B. Alternatively, we can determine the relationship by ray-casting from a point of A to B [3]. In general, A and B may have many connected components and with interior holes, e.g., the solids with holes as in Figure 5.3. Then each disconnected boundary curve should be tested against the other object for inclusion. The included object is returned for the Boolean intersection operation. We return nothing if there is no overlap between the two objects. Union operation basically reverses the result of the intersection operation. Difference operation subtracts the intersection region from the first operand. Alternatively, the difference can be expressed as an intersection between the first operand and the complement of the second operand. So the difference operation returns the first operand if there is no intersection and no inclusion. If the second operand is contained in the first operand, we return both objects but the second operand will change its orientation.

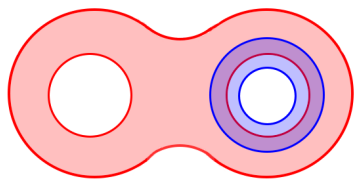


Figure 5.3: Two solids with holes with no intersection point

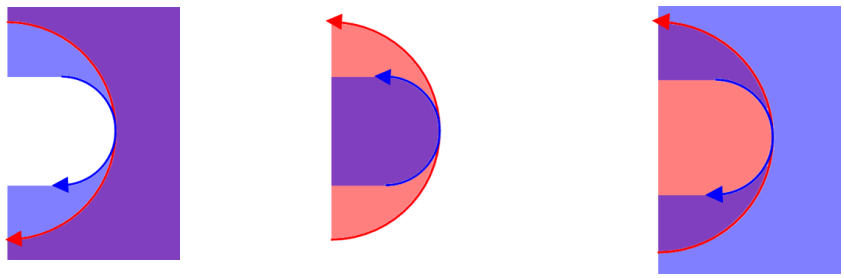
5.1.2 Tangential case

There are several cases we need to consider when arc-based objects intersect tangentially. Note that the arcs are tangential at the intersection point only when there is one touching point between the two curves, or when the two curves are identical. When the two curves are identical, they are regularized. If there is one intersection point where the arcs are tangential, the arc splines that meet at one intersection point may be either (i) facing in the same direction, (ii) facing against each other, (iii) locally tangential but globally transversal.

We compare the distance between the arc's center points and the difference between their radii to determine if the tangentially intersecting arcs are facing in the same way as in Figure 5.4:

$$d = |r_1 - r_2|.$$

In this case, the orientation of the arcs matter. When the direction of both arcs are concave, an arc with higher curvature is always included within another arc with lower curvature as in Figure 5.4(a). Thus, the arc with lower curvature is in the interior of the area bounded by the arc with higher curvature. On



(a) Both arcs are concave (b) Both arcs are convex (c) Different orientation

Figure 5.4: Tangential case when d $= |r_1 - r_2|$

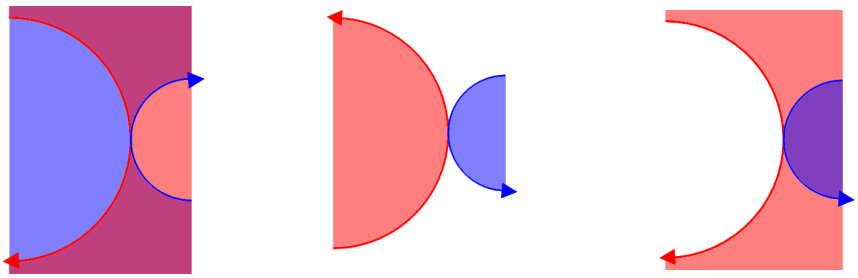
the other hand, if both arcs are convex as illustrated in Figure 5.4(b), the arc with lower curvature is included within the arc with higher curvature. Finally, when the directions are different, either the two arcs are in the interior of the union or the union area is bounded by both arcs. We can determine which case it is based on both the curvature and the direction, or by sampling a point in one arc, and testing whether the sample point is included in the other object.

Similarly, when the arcs are facing against each other as in Figure 5.5, the orientations of the arcs are used for the decision. This case occurs when the distance between the arc's center points is equal to the sum of the radii of both arcs:

$$d = r_1 + r_2.$$

In this case, the curvature does not matter. Instead, the directions tell us which arcs are included. For example, when both arcs are concave, both arcs bound the intersection area. However, when both arcs are convex, there is no intersection.

If the junction of two adjacent G^1-circular arcs is the intersection point as in Figure 5.6, the curvature can change after the intersection. Then the inclusion



(a) Both arcs are concave (b) Both arcs are convex (c) Different orientation

Figure 5.5: Tangential case when d $= r_1 + r_2$

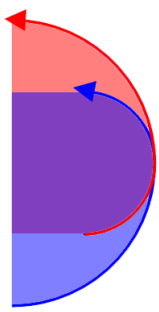


Figure 5.6: Transversal case where the arcs are tangential at the intersection

test for tangentially intersecting arcs may not be sufficient. Although the arcs are locally tangential at the intersection point, their intersection is transversal globally. Consequently, this case should be considered as a transversal case because it is a different arc after the intersection point. In other words, if it is a tangential case at an arc's end point, the inclusion tests for transversal case should be applied to check for a proper relation.

Note that the Boolean operation on tangential case becomes more complex for polygonal modeling. In the tangential case, we need to decide how much of the polygonal curve is tangential to the other curve. Then the line segment

that is tangential will be regularized. However, arc-based kernel detects more reliably the case of one intersection point, or more than one intersection point. The tangential arc segments are topologically decided based on the intersection point, and the coincidental arcs are regularized if there are more than one intersection point.

Also, consider the case in Figure 4.4 where unnecessary intersection points are identified. After deciding the direction for entry to the intersection area, additional intersection point will return reversed area for intersection region. This is the case described in Figure 1.3 of Chapter 1.

5.1.3 Transversal case

The transversal case is the basic case, where we identify the intersection area based on the intersection points. There are four arc segments separated by an intersection point. An arc segment needs to be compared against its adjacent arc segments to figure out which branches should be removed. The pairwise test needs to be done for all four possible pairs of adjacent arcs. It looks for three cases that determine which arcs are eliminated for Boolean operations: a case where both arcs share the same end points (Figure 5.7), a case where only the start points are the same (Figure 5.8), and a case where only the end points are the same (Figure 5.9).

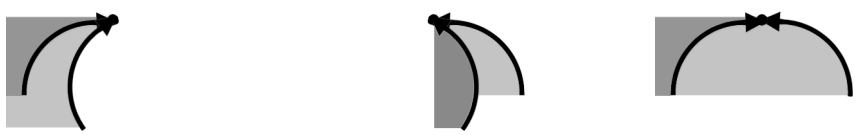
When two arc segments share the same start and end points, we first check the directions of the arcs. If the directions of the arcs are different as illustrated in Figure 5.7(c), the concave arc is the boundary arc for the intersection region and returned for the Boolean intersection operation. If the directions of both arcs are concave, then the arc with smaller radius is selected for the Boolean intersection operation. In general, the arc on the left is selected.

(a) Both arcs are concave (b) Both arcs are convex (c) Different orientation

Figure 5.7: Both adjacent arcs have the same start and end points

(a) Both arcs are concave (b) Both arcs are convex (c) Different orientation

Figure 5.8: Both adjacent arcs have the same start point



(a) Both arcs are concave (b) Both arcs are convex (c) Different orientation

Figure 5.9: Both adjacent arcs have the same end point

If only the end points are the same or only the start points are the same, the arc that lies on the left side of the other arc is the arc included in the other. This is stated in the Algorithm 2. If the intersection point is the start point of the first arc, then we check if the end point is on the left side of the second arc as in line 5 of Algorithm 2. If the intersection point is the end point of the first arc, then we check if the start point is on the left as in line 9.

After the entry point to the intersection region is decided, the exit point becomes the next intersection point. In this way, the arcs that are in between the exit point to the next entry point are in the exterior of both objects. The arcs that are in between the next entry point and the next exit point are the boundary for the intersection area.

Algorithm 2: In-out test which checks whether the first arc before the intersection is included in the second arc

Result: boolean

input: arcStruct first, arcStruct second, Point intersection

1 Point s_1=first.start, s_2=second.start, e_1=first.end, e_2=second.end;

2 **if** $s_1 ==$ *intersection* **then**

3 // start point of the first arc is the intersection:

4 // check if the end point is on the left

5 Return $((e_2.x - s_2.x) \times (e_1.y - s_2.y) - (e_2.y - s_2.y) \times (e_1.x - s_2.x)) > 0$;

6 **else if** $e_1 ==$ *intersection* **then**

7 // end point of the first arc is the intersection:

8 // check if the start point is on the left

9 Return $((e_2.x - s_2.x) \times (s_1.y - s_2.y) - (e_2.y - s_2.y) \times (s_1.x - s_2.x)) > 0$;

Algorithm 3: Boolean operation for union and intersection

Result: array of arcs

input: arcStruct first, arcStruct second, char opcode

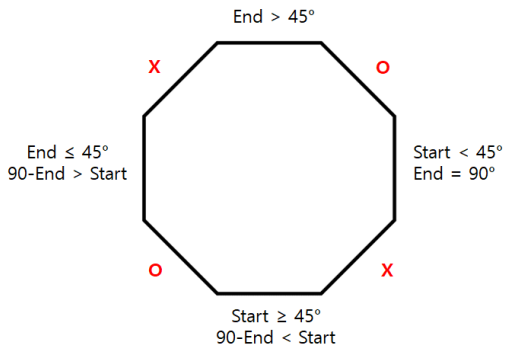
```
1  intersections = Find the intersections using Algorithm 1;
2  if intersections.size == 0 then
3  |   return {first, second} for union;
4  else if d^2 == (r_1 − r_2)^2 || d^2 == (r_1 + r_2)^2 then
5  |   if d^2 == 0 then
6  |   |   Return {first, second}; // Identical case
7  |   else
8  |   |   // Tangential case
9  |   |   Find the entering arcs around the intersection point;
10 else
11 |   Split the curve at the intersection if intersection is not end point;
12 |   Find the entering arcs around the intersection using in-out test;
13 |   Return the arcs that are before entering the interior for union or
14 |     the arcs that are after entering the interior for intersection;
```

Chapter 6

Bounding Volume Computation

In this chapter, we present how to compute the bounding volume of arc splines. As an arc is part of a circle, we can use the whole circle of an arc as the bounding volume. But for more tightness, we explore AABB and k-DOP bounding volumes. Although there are bounding volumes made with arcs, which are effective in handling non-convex objects, usage of conventional bounding volumes can still be effective for collision detection and minimum distance computation. We need fewer arcs to represent a planar object compared to polygons, because the approximation order of arcs is higher. Thus, there are fewer nodes in the BVH and the culling is faster for arc-based objects.

First, we need to know x-extreme and y-extreme points in order to compute the bounding volumes. When an arc spline is segmented monotonically, the end points of the arcs become x-extreme and y-extreme points. We can create AABB with the end points of arc splines, but we will analyze k-DOP bounding volume, which is more complex, but tighter.

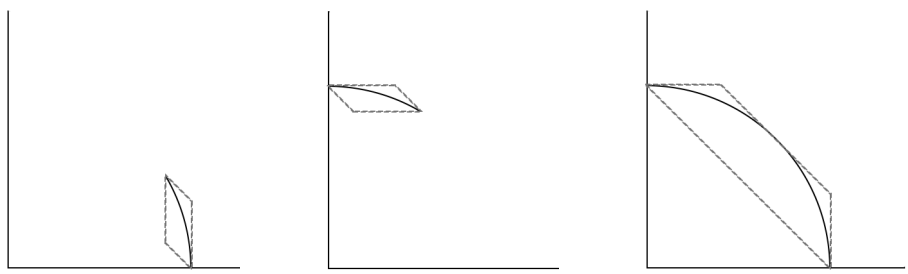


j	Support distance
0	$\max(s_x, e_x)$
1	StartAngle < 45°: $s_x\cos\theta + s_y\sin\theta$ 45° ≤ StartAngle: $r_i + x_i\cos\theta + y_i\sin\theta$
2	$\max(s_y, e_y)$
3	
4	$\min(s_x, e_x)$
5	90°-EndAngle < StartAngle: $s_x\cos\theta + s_y\sin\theta$ StartAngle < 90°-EndAngle: $e_x\cos\theta + e_y\sin\theta$
6	$\min(s_y, e_y)$
7	

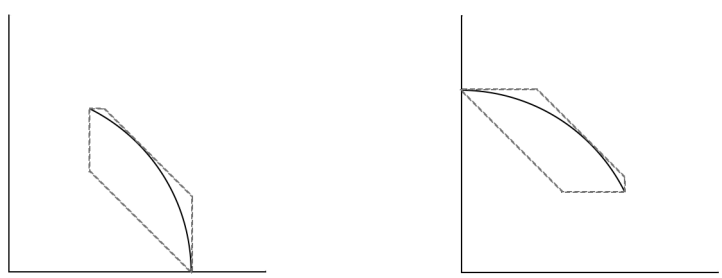
Figure 6.1: Conditions for each direction to exist for 8-DOP and support distances

Assuming that the curves are segmented monotonically, the end points of the arc always reside in the same quadrant. This simplifies the case analysis for the computation of k-DOP bounding volume. We use 8-DOP in which the cases are divided at every $\frac{\pi}{4}$ starting from $\frac{\pi}{8}$. Figure 6.1 shows which boundary should exist for the convex arc segment that is located in the first quadrant. The cases for the second quadrant are simply rotated versions of the first cases by $\frac{\pi}{2}$ counter-clockwise. Figure 6.2 elaborates on some of the specific examples.

Then we store the support distance of each direction in an array. To bound the smaller bounding volumes, we compare the support distance of the same index in the array, and extract the maximum support distance for each direction. The support distance for fifth direction in Figure 6.1 depends on whether the start angle is smaller than the difference between 90° and the end angle.



(a) $start, end < 45°$ (b) $start, end > 45°$ (c) $start < 45°$ & $end > 45°$



(d) 6.2(c) & $(90° - end > Start$ (e) 6.2(c) & $(90° - end < Start)$

Figure 6.2: Examples of 8-DOP in the first quadrant

6.1 Collision detection

Like many collision detection algorithm, we proceed to make a bounding volume hierarchy [3]. The BVH is constructed with the bigger bounding volumes found by comparing the support distances of smaller bounding volumes of the same index. We can tell whether the bounding volumes overlap by checking $\frac{k}{2}$ directions for k-DOP. Two k-DOPs D_1, D_2 do not overlap if none of the $\frac{k}{2}$ directions of D_1 overlaps the corresponding direction of D_2 [14]. For 8-DOP in Figure 6.2, checking the overlap for only East, Northeast, North, Northwest direction is enough. If the bounding volumes are disjoint, then there is no collision. If the bounding volumes overlap with each other, and is not a leaf node in the BVH, we traverse down the tree.

If the bounding volumes overlap and both are leaf nodes, it means that there is a collision between the two bounding volumes. For higher precision, the algorithm for intersection detection in Chapter 4 can be used to find the precise intersection between the arcs corresponding to the overlapped bounding volumes. As collision detection is a closed question with positive and negative answer, we can conclude that there is a collision if there exists an intersection. Note that the number of arc segments used to represent the curves in Figure 4.2 is 24, and the number of line segments used is 50. The resulting BVH in each case has height of 6(47 nodes) for arc representation, and height of 7(99 nodes) for polygonal representation. This means that we have to traverse down further to locate the leaf node for polygonal representation, and thus takes longer time to detect the collision.

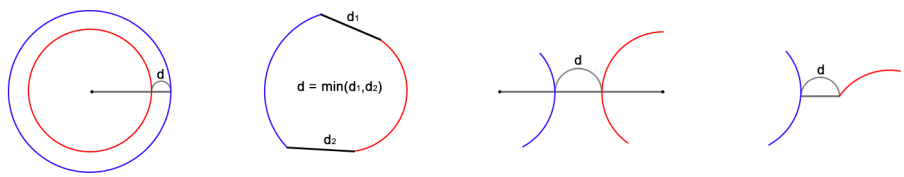
6.2 Minimum distance computation

Similar to the collision detection, minimum distance computation can use the bounding volumes to accelerate the process. The minimum distance is approximated by finding the lower bound and upper bound of the distances between the bounding volumes [15]. This approach is also used to evaluate the minimum distance between solids of revolution [24]. To efficiently compute the minimum distance between the solids of revolution, they also use biarc approximation and circles.

We start from calculating the distance between the root nodes of the BVH. The distance is stored as the priority for the pairs of children in the priority queue. We go through the hierarchy repeating the same operation between the children until we reach the leaf nodes. When we calculate the distances between the leaf nodes, we take off the bounding volume and calculate the distance between the arc splines. The minimum distance between the arc splines becomes the upper bound of the minimum distance between two planar objects. As we stored the distance between the bounding volumes for every pair of nodes in the BVH, except for the leaves, we can extract the first element in the priority queue as the lower bound of the minimum distance.

To get the upper bound, we can sample multiple random points on each boundary of the arc splines, and use the distance between them. Or we can calculate the distance between the arc splines based on the scenarios given in Figure 6.3. With the lower bound and upper bound of minimum distance obtained, we can approximate the minimum distance between the arc splines by averaging the lower bound and the upper bound.

When the arcs are concentric as in Figure 6.3(a), we know the minimum



(a) Concentric (b) Facing against (c) Facing toward (d) Arc-Point

Figure 6.3: Minimum distance between two arcs

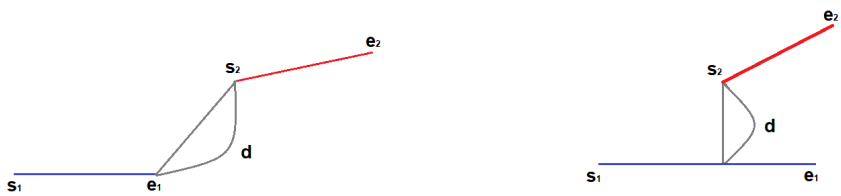
distance between them is $r_1 - r_2$ with each arc's radius being r_1 and r_2. We can check whether the arcs are facing against each other by the quadrant information of the angles. The minimum distance for Figure 6.3(b) occurs at the end points of each arc. Say the s and end points are denoted as $(s_1.x,\ s_1.y)$, $(e_1.x,\ e_1.y)$ for the first arc, and $(s_2.x,\ s_2.y)$, $(e_2.x,\ e_2.y)$ for the second arc. Then we need to calculate 4 pairs of distances between the end points, and select the minimum distance.

Alternatively, when the arcs are facing toward each other, as illustrated in Figure 6.3(c), we can get the minimum distance using the distance between the center points and radii as follows.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - r_1 - r_2$$

for each arc's center points given as (x_1, y_1), and (x_2, y_2).

Otherwise, the minimum distance is calculated between the arc and the end point of the arc. In this case, which is Figure 6.3(d), we choose the minimum distance between the center point and the end point. Then the minimum



(a) Point-Point (b) Line-Point

Figure 6.4: Minimum distance between two lines

distance will be the smaller of

$$\sqrt{(s_2.x - x_1)^2 + (s_2.y - y_1)^2} - r1$$

or

$$\sqrt{(e_2.x - x_1)^2 + (e_2.y - y_1)^2} - r1$$

The minimum distance between the line segments occurs in two cases. Figure 6.4(a) is where we calculate the distances between two end points of the line segments. The minimum distance for Figure 6.4(b) is calculated by the formula given below:

$$\frac{(s_1.y - e_1.y)(s_2.x) + (e_1.x - s_1.x)(s_2.y) + (s_1.x \cdot e_1.y - e_1.x \cdot s_1.y)}{\sqrt{(e_1.x - s_1.x)^2 + (e_1.y - s_1.y)^2}}$$

Evaluating the minimum distance between the NURBS curves is much more complicated. As the minimum distance between the NURBS curves cannot be set to a certain degree of polynomials, we only present the approach to calculating it. We first need to get the parameters u, and v from solving the

dot products. Then we need to calculate the distance between the curves at these parameters. The distance between the curves is expressed as below:

$$d = \sqrt{||C(u) - D(v)||^2} \quad where\ u,\ v\ are\ the\ solutions\ of$$

$$< C(u) - D(v), C'(u) >= 0,$$

$$< C(u) - D(v), D'(v) >= 0$$

The numerical precision of minimum distance computed with arcs is similar to that of polygonal model, not to mention that it is much higher than NURBS model. The number of multiplications involved in the calculation of minimum distance between the arcs is at most 3 for each case. On the other hand, it is 7 for polygonal model. Also, while it is straightforward to calculate the distance between two concentric circles as in Figure 6.3(a), it can cause problem for both polygonal approach and NURBS approach. Both polygonal approach and NURBS approach will compare the minimum distances at each point on the boundary despite them being the same.

Chapter 7

Discussions on Applications

7.1 Offset computation

Offset computations highly depend on the performance of intersection finding algorithm. The offset curve is non-rational in general, even for a rational curve. So detecting and eliminating the redundant offset stood as challenges. This problem has been addressed with an algorithm using biarc approximation in Kim et al. (2012) [13] for rational curves. The algorithm that resulted in more efficient computation is presented in Lee et al. (2015) [16] for deformable curves. It is worth noting that the offset curves can be represented in arcs if the original curves are arcs. Thus, the offset curves and the original curves are geometrically closed.

Algorithm suggested in Kim et al. [13] starts with the curve segmentation. Then biarc trees are used to represent the distance map. Instead of solving a complex system of non-linear polynomial equations, Lee et al. [16] finds the lo-

cal self-intersection using osculating circles to the curve, and intersecting each pair of osculating circles. Also, local self-intersections of an offset curve can be detected where the radius of the arc is smaller than the offset radius.These are the main basis of algorithm's acceleration. The arc based geometry kernel takes the same approach to find the local self-intersection points. But the approximation step is unnecessary for the arc based kernel because the objects are already represented in arc. After the local self-intersections are dealt with, we need to handle the global self-intersection of the offset curve.

In order to find the global self-intersection of the offset curve, the previous algorithms of Kim et al. [13] and Lee et al. [16] compute the approximation error to narrow down the candidate offset curves, which may have global self-intersection. Then they check the redundant offset curve segments if they share a common normal in order to determine if there are multiple intersections, such as tangential intersection. Then, Kim et al. [13] tries to pinpoint a single intersection for the offset curves going down the biarc tree. However, arc spline based kernel has no approximation error. Both the global and local self-intersection points are found using arc-arc intersections. So we do not have to separate the cases between the global self-intersection and local self-intersection. Furthermore, we do not have to track down the single intersection point from a curve segment. As stated in Chapter 4, using line segments to detect intersection points results in finding more intersection points that the actual ones. However, this is not a problem in arc-arc intersection.

Lee et al. [16], tries to improve the performance by reducing the approximation error using different types of bounding volumes. In the two-level overlap test, AABB is applied first. When AABBs overlap, we have to test the overlap within these leaf nodes using the Bounding Circular Arc(BCA) and Line

Swept Circle(LSC). Also, the analysis suggests that the redundant offset curve segments occur as the redundant segments in the offset curve of a bigger arc as well. So the same redundant segments can be removed in multiple offset curves. In arc based geometry kernel, we use the k-DOP bounding volume for overlap test. When the arc spline is used as a base geometry to represent a planar model, it is easier to find the bigger arc than the arc that intersects because the radius is already known. Then we remove the redundant offset curve segments by checking the relative orientations of the two offset curve segments, just like in the Boolean operations.

7.2 Convex hull

Convex hull requires solving two bivariate equations. But when convex hull encloses multiple objects, it is difficult to achieve a real-time performance by solving the equations. To achieve a real-time performance, biarc approximation of a curve plays an important role again [12]. The interior arcs are detected faster using the support distance to the edges that share the same direction as the arcs, the number of which is at most two. This is possible because of the monotone segmentation of curves, all the more reason to segment the curves so that monotonicity is maintained. In fact, the same thing cannot be applied to curves represented by lines because we cannot tell the angle of the line strips. But arc based geometry kernel can handle this algorithm.

Then the remaining arcs, after the trimming of interior arcs, have to be merged in a way that each pair of arcs share a common tangent line. The proposed approach finds the common tangent line recursively using the osculating circle. This can be quite exhaustive for the polygonal modeling system,

because multiple osculating circles should be found for the same curve. For arc based kernel, all the arcs have their osculating circles known. So the algorithm from Kim et al. [12] can be used in arc based kernel to compute convex hull efficiently.

7.3 Voronoi diagram

We first look into the computation of medial axis, because it requires finding a maximal disk which is also necessary for computing the Voronoi diagram. O. Aichholzer et al. [1, 2] introduced a divide-and-conquer algorithm for medial axis computation. According to their results, medial axis convergence is not guaranteed for polygonal approximations. But it is not the same for the approximations using G^1-continuous arc. The curvature extrema is a necessary feature for the medial axis to converge into one. Naturally, the arc based geometry kernel has the curvature information ready at hand.

First, we select a random arc spline segment of the planar curve as a starting point. Then we find the disk that is tangent to this segment at the midpoint, or the end point of the arc. This process is repeated until the valid maximal disk is found. The disk is considered a valid maximal disk if it does not intersect or overlap with any part of the curve. The only information that arc kernel system does not have is the midpoint. Yet, the calculation of the midpoint of an arc is simple because we know the angle of the arc. By rotating one of the end point by half of the angle difference toward the other end, we get the midpoint. But more importantly, we already know the disk that is tangent to the arc segment in arc based kernel, the whole circle of the arc segment itself. But this is only the case for the convex arc segment, so we still

need to find the tangential disk for concave segments. This step is repeated until the planar curve is divided into three basic cases. Finally, the medial axis is a set of all centers of maximal disks in the conquer step.

A more efficient result is presented by Lee et al. [17]. The main difference of this work is that Lee et al. segments the input freeform curves to monotone spirals. The spiral curves are further segmented at the points where maximal disks touch. Then the maximal disks are computed at the end points. They use Möbius transformation to search for the maximal disks more efficiently. Then the Voronoi diagram is constructed by detecting the bifurcation point, or by inserting bisector depending on the relationship between the spirals that share a common maximal disk. Spirals are basically a chain of G^1-continuous arc splines that change curvature monotonically. So if we further segment the arc splines to form a spiral, at a point where the derivative of the curvature is 0, we can also support this efficient algorithm to construct a Voronoi diagram.

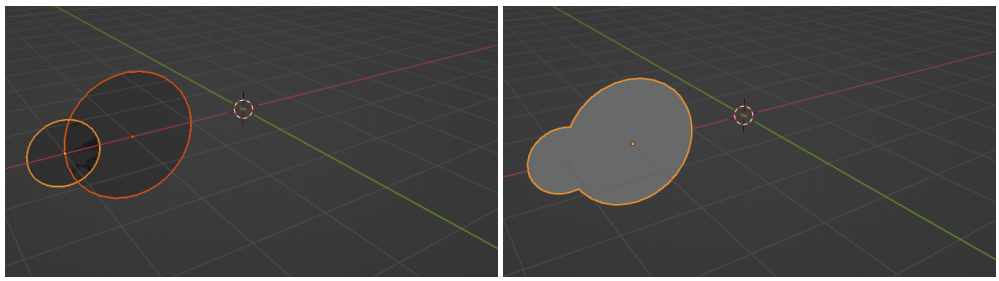
Chapter 8

Conclusion

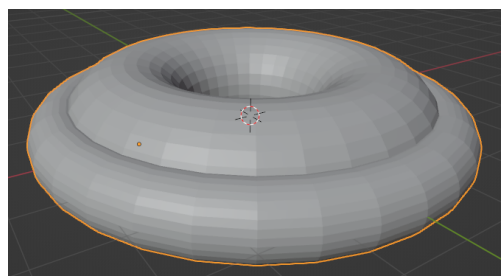
We have discussed stability and efficiency issues in the design of geometry kernel based on G^1-continuous circular arcs, and some potential benefits in modeling with arcs over the conventional approaches. Planar objects modeled with circular arcs have demonstrated improved stability and efficiency for operations between one another, such as intersection detection, Boolean operation, and bounding volume computation.

Intersecting two circular arcs is slightly more difficult than that of intersecting two line segments. However, the overall algorithmic stability is improved because fewer arcs are intersected with each other. At the same time, the intersection points identified by using arcs reflect the actual result more faithfully than the case of intersecting two line segments.

The number of cases that we need to consider for topological decision of Boolean operation increases for arc based modeling compared to polygonal modeling. Nonetheless, the increase in number is manageable, and the in-out



(a) Two circles (b) Union between two circles



(c) Solid of revolution made out of (b)

Figure 8.1: Union between two circles

test which identifies the arcs in the interior is considerably simpler than the multitudinous cases of the arrangement of NURBS curves. Furthermore, the stability improvement from intersection detection enhances the stability of Boolean operation as well.

The computation of 8-DOP bounding volumes can be done fairly easily, and the BVH can be constructed more efficiently using circular arcs and their support distances. The resulting BVH is more compact than when the same object is represented in polygons. Accordingly, the operations using BVH can be done efficiently. We can improve the stability at the leaf level by calculating the intersection point or the distance between the arcs.

In conclusion, G^1-continuous arc geometry kernel can be a compromise between the two conventional approaches: polygon based approach, and NURBS based approach. While we mainly considered planar objects in this paper, the applications of arc spline modeling can be extended to the 3D case because circular arcs become torus in 3D.

Bibliography

[1] O. Aichholzer, W. Aigner, F. Aurenhammer, T. Hackl, B. Jüttler, and M. Rabl. Medial axis computation for planar free-form shapes. *Computer Aided Design*, 41(5):339–349, 2009.

[2] O. Aichholzer, A. Franz, H. Thomas, J. Bert, R. Margot, and Z. Šír. Computational and structural advantages of circular boundary representation. *International Journal of Computational Geometry & Applications*, 21(1):47–69, 2011.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time Rendering*. A.K. Peters, 2008.

[4] G. v. d. Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of graphics tools*, 2(4):1–13, 1997.

[5] P. Bo, H. Pottmann, M. Kilian, W. Wang, and J. Wallner. Circular arc structures. *ACM Transactions on Graphics (TOG)*, 30(4):1–12, 2011.

[6] G. Farin and D. Hansford. *The essentials of CAGD*. A.K. Peters, 2000.

[7] R. T. Farouki. *Hierarchical Segmentations of Algebraic Curves and Some Applications*. Elsevier, 1989.

[8] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180, 1996.

[9] C. M. Hoffmann. *Geometric and Solid Modeling : an Introduction*. Morgan Kaufmann, 1989.

[10] C. M. Hoffmann, J. Hopcroft, and M. Karasick. Robust set operations on polyhedral solids. *IEEE computer graphics and applications*, 9(6):50—59, 1989.

[11] T. Ju. Fixing geometric errors on polygonal models: A survey. *Journal of Computer Science and Technology*, 24(1):19–29, 2009.

[12] Y.-J. Kim, J. Lee, M.-S. Kim, and G. Elber. Efficient convex hull computation for planar freeform curves. *Computers & Graphics*, 35(3):698–705, 2011.

[13] Y.-J. Kim, J. Lee, M.-S. Kim, and G. Elber. Efficient offset trimming for planar rational curves using biarc trees. *Computer Aided Geometric Design*, 29(7):555–564, 2012.

[14] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[15] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Technical report, Technical Report TR99-018, Department of Computer Science, University of North Carolina, 1999.

[16] J. Lee, Y.-J. Kim, M.-S. Kim, and G. Elber. Efficient offset trimming for deformable planar curves using a dynamic hierarchy of bounding circular arcs. *Computer Aided Design*, 58:248–255, 2015.

[17] J. Lee, Y.-J. Kim, M.-S. Kim, and G. Elber. Efficient voronoi diagram construction for planar freeform spiral curves. *Computers Aided Geometric Design*, 43:131–142, 2016.

[18] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[19] D. S. Meek and D. J. Walton. Approximation of quadratic bézier curves by arc splines. *Journal of Computational and Applied Mathematics*, 54(1):107–120, 1994.

[20] D. S. Meek and D. J. Walton. Approximating smooth planar curves by arc splines. *Journal of Computational and Applied Mathematics*, 59(2):221–231, 1995.

[21] R. Ollington. Using piecewise circular curves as a 2d collision primitive. *Asia-Pacific journal of business*, 9(2):1–13, 2018.

[22] D. Pekerman, G. Elber, and M.-S. Kim. Self-intersection detection and elimination in freeform curves and surfaces. *Computer Aided Design*, 40(2):150–159, 2008.

[23] T. W. Sederberg, S. C. White, and A. K. Zundel. Fat arcs: A bounding region with cubic convergence. *Computer Aided Geometric Design*, 6(3):205–218, 1989.

[24] S.-H. Son, S.-H. Yoon, M.-S. Kim, and G. Elber. Efficient minimum distance computation for solids of revolution. In *Computer Graphics Forum*, volume 39, pages 535–544. Wiley Online Library, 2020.

[25] S. Suri, P. Hubbard, and J. Hughes. Analyzing bounding boxes for object intersection. *ACM transactions on graphics*, 18(3):257–277, 1999.

[26] Z. Šír, R. Feichtinger, and B. Jüttler. Approximating curves and their offsets using biarcs and pythagorean hodograph quintics. *Computer Aided Design*, 38(6):608–618, 2006.

초 록

본 논문에서는 G^1-연속 원호 스플라인으로 구성된 평면 모델에 대한 기하 커널의 설계 및 분석에서 몇 가지 기술적인 문제를 논의한다. 특히 평면 곡선에서의 교차점 계산이 수치적으로 불안정할 수 있는 점을 보이며, G^1-연속 원호 스플라인의 사용이 이러한 컴퓨터 프로그래밍 고유의 불안정성을 완화할 수 있는 방법에 주로 초점을 맞춘다. 이에 더해, 원호 스플라인 모델의 더 높은 수치적 안정성을 토대로, 정밀한 교차점을 찾아야 하는 기하 커널의 일부 필수 연산에 대한 알고리즘을 제시한다.

또한, 계산 효율성을 향상시키기 위한 노력의 일환으로 G^1-연속 원호 스플라인 모델링에 맞게 구상된 기하 커널에 대한 데이터 구조를 제시한다. 제시된 데이터 구조는 각 곡선 조각들이 단조롭게 변화하게끔 분할한 후, 분석된 케이스들을 간단화시켜 평면 곡선에 대한 경계 볼륨 계산 역시 간단하게 만들 수 있다. 평면 곡선의 8-DOP 경계 볼륨을 사용하면, 각 경계 볼륨의 지지 거리 사이의 비교를 통해 경계 볼륨 계층의 상위 노드를 쉽게 계산할 수 있다. 마지막으로 볼록 껍질 및 오프셋 계산과 같은 기존의 문제들에 본 논문에서 소개하는 알고리즘과 데이터 구조를 적용하는 방법을 제시한다.

주요어: 원호, 원호 스플라인, 기하커널, 자가교차, 불리안 연산, 바운딩 볼륨 계산
학번: 2020-26241

Acknowledgments

이 자리를 빌어 끝까지 저를 믿고 지도해주신 김명수 교수님께 깊은 감사의 말씀 드립니다. 2년이란 짧다면 짧고 길다면 긴 시간동안 컴퓨터 그래픽스 분야의 이론에 대해 무지했던 제가 기하 커널에 관한 논문을 쓰게 되었다는 사실이 스스로도 놀라울 뿐입니다. 해박하신 교수님 덕분에 저에게는 너무나 유익하고 정말 많은 것들을 배울 수 있는 시간이었습니다. 은사라고 부를 만한 사람이 없던 제게 김명수 교수님은 졸업 이후에도 늘 감사할 분으로 남을 것입니다.

제가 대학원에 입학했을 시기에는 이미 코로나가 터져 많은 것들이 비대면이 되었고 여러 기회들이 사라졌음에도 불구하고, 낯선 학습 환경에 적응할 수 있도록 도와주신 박영진 박사님, 홍규연 박사님, 손상현 석사님, 정민규 석사님, 함유경 석사님께도 많은 감사를 드립니다. 선배님들 덕분에 외롭지 않은 대학원 생활을 할 수 있었고, 학문적으로도 많이 이끌어주셨기에 제가 도태되지 않고 버틸 수 있었습니다.

뿐만 아니라 학사 행정에 대해 도움을 주신 행정실 직원분들께도 감사한 마음을 전합니다. 친절하고 유능하게 안내해주셨기에 제가 어느 하나도 빠트리지 않고 무사히 여기까지 올 수 있었습니다.

서울대학교에서 석사 생활을 할 수 있었던 것은 정말 여러모로 저에게 매우 좋은 기회였고 앞으로 두 번 다시 없을 행운이었다고 생각합니다. 제 주변에서 도움을 주신 분들과 더불어 최적의 환경 덕분에 지금의 발전한 제가 있을 수 있다고 생각합니다. 다시 한 번, 모든 분들에게 감사드립니다.