The `lua-tikz3dtools` Package

The lua-tikz3dtools Package, Version 2.0.1

https://github.com/Pseudonym321/TikZ-Animations/tree/master1/TikZ/lua-tikz3dtools

Jasper Nice

October 5, 2025

Dedicated to those who, like me, never found the tools they needed—and chose to build them.

“I long to accomplish great and noble tasks but it is my duty to accomplish small tasks as though they were great and noble.” —Sylvia Fedoruk

Contents

List of Tables

List of Figures

Preface

Purpose of This Work

I undertook this work because I was not satisfied with the existing tools for illustrating scenes with 3D parametric objects in TEX, such as surfaces. From the perspective of computer graphics as a whole, the ability to occlusion-order non-intersecting and non-cyclically overlapping 0–2-dimensional affine simplices in 3D—together with the systematic partitioning of simplices to eliminate intersections and cyclic overlaps—is a foundational advance. While my own motivation arose from the needs of a TEX illustrator, I recognized that this framework represents a paradigm shift in mathematics illustration and, more broadly, a landmark contribution to computer graphics. My goals were therefore twofold: to create the kind of illustration tool I had long envisioned, and to establish priority in implementing this transformative idea.

I have developed a systematic method for occlusion ordering 0–2-dimensional affine simplices in three dimensions. In addition, I have implemented a framework for eliminating intersections through partitioning, with the resolution of cyclic overlaps planned for the next version of the software. This work matters because it provides the only known approach for generating scenes with parametric objects that both intersect coherently and are occlusion-sorted robustly. In short, this is a rigorous and comprehensive software framework for handling intersections and occlusion ordering of affine simplices in three dimensions.

Context and Background

This book documents the LATEX package `lua-tikz3dtools`, which implements a groundbreaking algorithm for 3D illustrations composed of 0–2-dimensional affine simplices. Although the software is designed for creating illustrations in LATEX, the underlying method represents a significant advance for the broader field of computer graphics.

Acknowledgments of Guidance and Support

I am deeply grateful to the TEX Stack Exchange community, whose many contributions and generous assistance have been invaluable in helping me develop my skills as an

illustrator. I also acknowledge the Math Stack Exchange community for several insights that proved helpful along the way. This work would not have been possible without the years of learning made available by my peers on TEX Stack Exchange, nor without the foundational training in programming and linear algebra that I received from my professors at my local university.

Author's Contribution and Perspective

I am the principal designer of nearly all of `lua-tikz3dtools`, with a few specific components assisted by ChatGPT. My two primary original contributions are the occlusion-ordering algorithm and the simplex-partitioning algorithm. The portion produced with ChatGPT's assistance concerns the recursive traversal of segments that applies my algorithms. Apart from this limited assistance, the work was carried out independently, with valuable input and feedback from the TEX Stack Exchange community.

Structure of the Document

This book is organized into five chapters. Chapter 1 presents the problem statement, defining and framing the central challenge. Chapter 2 reviews the relevant literature. Chapter 3 describes the methodology, including the algorithms developed. Chapter 4 provides validation of the approach, and Chapter 5 presents the results and analysis.

Acknowledgements

This work was typeset using TEX, the typesetting system created by Donald E. Knuth, along with various extensions and packages developed by the TEX community. I am grateful to the vibrant TEX Stack Exchange community for their ongoing support and resources. For those interested, my contributions can be found at Jasper [Jas].

I also acknowledge the assistance of ChatGPT, which contributed to parts of the code and to the refinement of this documentation.

Jasper Nice

Chapter 1

Problem Statement

1.1 Background and Context

The field to which `lua-tikz3dtools` belongs is inherently multidisciplinary, combining elements of programming and mathematics, especially linear algebra. In my view, the current state of practice remains fragmented. Much of the existing literature focuses on isolated components of the broader problem; for example, publications may address tessellation of surfaces or the clipping of line segments by triangles, but seldom attempt a unified framework. I believe I am the first to orchestrate linear algebra and programming into a coherent synthesis that enables parametric object generation tessellated by affine simplices (points, line segments, and triangles). These simplices are then systematically clipped against one another using a novel algorithm that eliminates intersections. A second novel algorithm, also of my design, performs occlusion sorting of the resulting simplices via a transitive partial order.

This represents a genuinely groundbreaking achievement, with applications in computer graphics, computational geometry, and mathematical illustration.

1.2 The Core Problem Being Addressed

The central problem addressed by `lua-tikz3dtools` is the illustration of 3D scenes composed of 0–2-dimensional affine simplices. Specifically, the package focuses on the representation of parametric objects tessellated into points, line segments, and triangles, including those that intersect. The primary outstanding challenge is the elimination of cyclic overlaps, which will be addressed in the next version of the software.

1.3 Scope and Boundaries of the Problem

The scope of the problem—excluding cyclic overlaps—is twofold: clipping intersecting simplices and partially transitive sorting of the resulting non-intersecting simplices.

This version of the package assumes that cyclic overlaps do not occur. It is designed for 3D diagrams composed of tessellated parametric objects, favoring exact partitioning over alternative methods to handle sparse objects without introducing occlusion errors.

1.4 Importance of the Issue

Until now, illustrators have often relied on high sampling to visually resolve occlusion errors arising from intersecting objects. Furthermore, to my knowledge, no software exists that explicitly sorts triangles correctly, let alone lower-dimensional affine simplices. Most existing approaches either produce improper sorting or rely on black-box methods. Solving this problem without using a z-buffer or ray tracing constitutes a significant contribution to both the theoretical and practical aspects of computer graphics, particularly in the domain of mathematical illustration. Based on years of studying 3D graphics, I can assert that this algorithm addresses a fundamental challenge for the future of the field.

From a theoretical standpoint, the method can illustrate 3D scenes composed of points, line segments, and triangles, demonstrating that a scene of triangles can be partitioned so as to allow a partial transitive ordering, resulting in correct visual occlusion.

This work represents a major theoretical advance in the conceptualization of 3D illustrations. Its synthesis of elementary linear algebra with computational methods is unique and represents a significant step beyond the current state of the art.

1.5 Current Limitations and Gaps

Previous approaches largely neglect intersecting simplices and often produce occlusion errors. In fact, many software packages do not even triangulate surfaces correctly. For example, drawing a Klein bottle in Mathematica, Asymptote, or pgfplots results in intersections that do not occlude properly. My approach resolves this issue systematically.

Not only does it eliminate intersections, but it also applies rigorous affine linear algebra for correct occlusion sorting. These gaps in existing methods persist largely because no one has previously pursued a comprehensive study of 3D mathematical illustration with this level of coherence. While many practitioners achieve impressive results, none combine triangulation, clipping, and occlusion sorting in a fully integrated framework. Unlike approaches that rely on splines or high sampling, my package ensures accurate visualization of sparse geometries.

High sampling is computationally expensive and often insufficient, since the human eye can detect very small errors. Achieving the necessary resolution via sampling alone is generally impractical. My software addresses this challenge directly by first clipping simplices and then performing occlusion sorting, ensuring correct visual results efficiently.

1.6 Objectives and Goals

The objective of this documentation is to present the software I have developed, demonstrate its correctness, and provide guidance on its use. Detailed explanations are provided for each of the novel algorithms, accompanied by examples illustrating how to apply the software in practice. In addition, practical use cases are discussed, and the software is validated through tests.

1.7 Intended Audience

This documentation is intended for 3D mathematical illustrators who are already familiar with basic programming, linear algebra, and Ti*k*Z. While the package primarily supports 3D mathematical illustrations in LaTeX, its underlying algorithms have implications beyond the LaTeX community. Computer graphics researchers will benefit from the first coherent synthesis of a transitive partial-order algorithm for occlusion sorting of affine simplices.

Mathematics textbook authors and illustrators will also benefit from a robust backend for occlusion handling and clipping, without relying on black-box software that produces incorrect results. Although some tools generate visually plausible output for users with limited 3D intuition, subtle errors often remain undetected. The approach presented here provides a rigorous and systematic solution, and it is likely that the software will have applications beyond these immediate use cases.

1.8 Summary of the Problem Statement

The central problem addressed by this documentation is the clipping and occlusion ordering of affine simplices. In other words, it concerns the visualization of 3D parametric scenes. This software is significant because it represents the first coherent synthesis for illustrating such scenes.

Chapter 2

Literature Review

2.1 Introduction to the Literature Review

The purpose of this chapter is to situate this work within the context of existing scholarship. The chapter is organized chronologically and reflects my personal journey along this path. Because my synthesis of the field is genuinely novel, most of the sources discussed are my own contributions, accompanied by testimony that contextualizes and motivates these developments.

2.2 Historical Background and Evolution of the Field

I began as a complete amateur, initially motivated by a desire to create mathematical textbook illustrations. Over time, I developed a particular interest in 3D illustrations, but quickly grew dissatisfied with the state of the available tools and resolved to build them myself. My first attempt involved tessellating a surface into quadrilaterals, but I soon realized that the occlusion was incorrect. At the time I was taking my first course in linear algebra, and with the help of ChatGPT [Ski25], I recognized that the dot product could be used to approximate depth-ordering by midpoint comparisons. I later refined this idea through a Mathematics Stack Exchange inquiry [httb]. By intuition, I also realized that tessellating into triangles was more coherent than quadrilaterals, which are often non-coplanar. Initially, I attempted to sort the triangles by their midpoints, but the results were still inadequately occluded.

This led me to another inquiry on occlusion ordering of non-intersecting, non-cyclically overlapping triangles [htta], where a pivotal suggestion was made in the comments: to sort only simplices whose orthogonal projections on the viewing plane overlap. This insight proved to be the breakthrough. From this seed grew the present occlusion algorithm. In the final system, simplices are first projected onto the viewing plane; when an overlap is detected, sorting is resolved by using the inverse orthogonal projection back onto both shapes. Although it took months of refinement, this ultimately yielded a coherent and general solution.

The occlusion problem also naturally arises from intersecting and cyclically overlapping simplices. My algorithm addresses the first case by completely eliminating intersections through partitioning, thereby establishing a foundation alongside the occlusion algorithm. The resolution of cyclic overlap is left for future work.

2.3 Existing Approaches in Practice and Academia

Until now, no comprehensive solution to the problem of clipping and occlusion ordering of affine simplices has existed. Existing practices in 3D mathematical illustration and computer graphics address only fragments of the problem. For example, many software systems triangulate surfaces and order the resulting triangles using primitive heuristics, but they do not perform clipping of intersecting simplices, and their occlusion methods are often approximate or incorrect.

Much of the literature in both academia and practice focuses on partial approaches—such as tessellation, polygon clipping, or depth-sorting—but not their coherent integration into a rigorous framework. This fragmentation explains why current tools, despite producing visually plausible illustrations, frequently introduce occlusion errors when tested under precise mathematical scrutiny.

The work presented here unifies these scattered threads into a single coherent system, addressing both intersection elimination and rigorous occlusion ordering. This synthesis has not been achieved previously in the literature or in practice.

2.4 Comparative Analysis of Approaches

When comparing existing approaches to the method implemented in `lua-tikz3dtools`, a fundamental distinction emerges. Existing tools and frameworks for 3D illustration—whether in academic research or practical software—typically handle only parts of the problem. Some systems focus on tessellation, others on clipping, and still others on heuristic depth-sorting. However, none integrate these processes into a coherent framework capable of systematically handling intersections and providing rigorous occlusion ordering of affine simplices.

By contrast, the algorithm presented here addresses both of these challenges directly. It introduces a systematic clipping procedure to eliminate intersections and a transitive partial-order approach to occlusion sorting. No existing approach in either the academic literature or applied software achieves this combination. In short, while prior methods provide partial or approximate solutions, the present work provides the first complete and rigorous solution to the problem of visualizing 3D parametric scenes composed of affine simplices.

2.5 Strengths and Contributions of Current Work

The present work successfully demonstrates the systematic illustration of 3D scenes composed of affine simplices. This was achieved through the use of affine linear algebra to explicitly define simplices and to traverse their spans. The span of a simplex is the

set of points obtained by taking linear combinations of its basis vectors, originating from a given point.

A further breakthrough came from reducing the problem to its simplest building blocks: the intersection and occlusion of the lowest-dimensional simplices. From there, higher-dimensional simplices, such as triangles, were treated in terms of these more elementary cases. This reductionist approach provided a clear and coherent foundation for solving the problem at scale.

2.6 Limitations and Open Challenges in Current Methods

This documentation does not, in its current form, address cyclically overlapping simplices; this challenge is reserved for future work. The decision to defer this aspect was intentional, in order to secure authorship priority on the core contribution before extending the framework further.

Future versions will also introduce the ability to clip parametric objects—that is, collections of simplices belonging to the same object—by themselves. This will enable the computation of intersection sets as new, customizable parametric objects, thereby broadening the applicability of the package.

2.7 Emerging Trends and Future Directions

Recent developments in computer graphics emphasize the pursuit of photorealism, with ray tracing technologies at the forefront. These methods simulate lighting, reflection, and refraction effects at a high degree of physical accuracy. While such techniques are powerful, they are computationally intensive and primarily focused on rendering dense, visually realistic scenes rather than sparse, mathematically rigorous ones. Nevertheless, the conceptual framework of ray tracing demonstrates the importance of light-based visibility models, which may offer inspiration for extending methods of occlusion ordering in the future.

In contrast, the approach taken in this work focuses on exact algebraic partitioning and sorting of affine simplices, prioritizing mathematical rigor over visual realism. Future research may explore a hybrid direction, drawing selectively from photorealistic methods while preserving the precision needed for mathematical illustration. Such a synthesis would represent a promising new trajectory: combining the clarity and correctness demanded by pedagogy with techniques inspired by the broader field of computer graphics.

2.8 Proposed Approach and Its Advantages Over Existing Work

The limitations of existing methods are clear: they either neglect intersections altogether, rely on approximate sampling, or obscure their logic within black-box imple-

mentations. As a result, mathematical illustrators are left without a reliable way to render sparse parametric geometries in three dimensions without visible errors. This is the precise gap that motivates the present work.

The approach developed in `lua-tikz3dtools` addresses this gap by providing a systematic method for clipping intersecting simplices and establishing a rigorous transitive partial order for occlusion sorting. Unlike prior tools, this method is fully transparent in its logic, grounded in affine linear algebra, and designed specifically to ensure correctness even for sparse configurations where high-sampling techniques fail.

The principal advantage of this work is its ability to produce illustrations that are both mathematically precise and computationally efficient. By reducing the problem to its most fundamental constituents—simplices—and resolving both intersections and occlusion ordering at that level, the method achieves a coherence that existing approaches lack. This framework not only serves the immediate needs of mathematical illustration, but also establishes a foundation that can be extended to broader areas of computer graphics.

2.9 Summary of the Literature Review

The review of existing tools and practices makes one thing clear: the available methods were inadequate for producing precise 3D mathematical illustrations. They either ignored intersections, introduced occlusion errors, or relied on approximate techniques such as high sampling. Bound by the limitations of the time, I was compelled to invent my own approach. While this work does not yet address the final challenge of cyclic overlaps, it establishes a rigorous framework for clipping intersecting simplices and systematically ordering them by occlusion. The next chapter will present the methodology by which this is achieved.

Chapter 3

Methodology

3.1 Introduction to the Methodology

This chapter explains how the research was carried out and how the core algorithms were developed. The synthesis did not arise from a gradual accumulation of partial solutions, but from a decisive insight gained after years of experience with 3D illustration. That insight was to frame the entire problem in terms of affine linear algebra, which provided the clarity and rigor needed to resolve clipping and occlusion systematically.

The implementation was carried out in LaTeX and Lua, not by accident but because I am a mathematics textbook illustrator, and these are the tools I know and use daily. This choice ensured that the algorithms were embedded directly in the environment where they are most relevant, while also demonstrating their applicability beyond it. The following sections present the design, implementation, and validation of the algorithms, showing how they directly address the fundamental gaps identified in the Problem Statement and Literature Review.

3.2 Research Design and Overall Approach

The research design of this work is computational and developmental in nature. Its objective was not to test an existing theory, but to create and validate a new one. The approach is best described as exploratory, since the central algorithms were developed from first principles through problem-driven experimentation, guided by years of practical experience with 3D illustration.

This design is well-suited to the research objectives, which required the invention of novel methods for clipping and occlusion ordering of affine simplices. Rather than relying on preexisting frameworks, the work advances a new synthesis, implemented and tested in Lua and LaTeX. The emphasis throughout is on demonstrating the feasibility and rigor of the approach, supported by concrete computational validation.

3.3 Detailed Description of the Proposed Approach

Users are provided with commands that automatically tessellate, clip, and occlusion-sort parametric objects with projective transformations applied. The following subsections describe each component of the system in detail.

3.3.1 How Users Interact with the Software

Command Name: `\displaysegments`

This command clips intersecting simplices produced by the tessellation of parametric objects and performs occlusion sorting on the resulting set. It ensures that overlapping simplices are rendered correctly according to their depth and spatial relationships.

Command Name: `\appendpoint`

This command appends a projectively transformed 0-dimensional affine simplex (a point) to the list of simplices, also called the list of segments. An example of its usage is as follows:

```
\appendpoint[
    x = {cos(tau/6)}
    ,y = {tau/6}
    ,z = {0}
    ,fill options = {
        fill = red
        ,fill opacity = 0.7
    }
    ,transformation = {euler(pi/2,pi/3,pi/6)}
]
```

Command Name: `\appenlabel`

This command appends a label positioned at a projectively transformed 0-dimensional affine simplex (a point) to the list of simplices. An example of its usage is as follows:

```
\appendlabel[
    x = {cos(tau/6)}
    ,y = {tau/6}
    ,z = {0}
    ,name = {Hi!}
    ,transformation = {euler(pi/2,pi/3,pi/6)}
]
```

Note: there are still struggles with adding math mode to labels.

Command Name: `\appendsurface`

This command tessellates a projectively transformed 2-dimensional parametric surface into triangles, and adds those triangles to the list of segments. An example of its usage is as follows:

```
\appendsurface[
    ustart = {0}
    ,ustop = {1}
    ,usamples = {18}
    ,vstart = {0}
    ,vstop = {1}
    ,vsamples = {9}
    ,x = {sphere(tau*u/36,pi*v/18)[1][1]}
    ,y = {sphere(tau*u/36,pi*v/18)[1][2]}
    ,z = {sphere(tau*u/36,pi*v/18)[1][3]}
    ,fill options = {
        preaction = {
            fill = green
            ,fill opacity = 0.8
        }
        ,postaction = {
            draw
            ,line join = round
            ,line cap = round
        }
    }
    ,transformation = {euler(pi/2,pi/3,pi/6)}
]
```

Command Name: `\appendcurve`

This command tessellates a projectively transformed parametric curve into line segments, and adds those line segments to the list of segments. An example of its usage is as follows:

```
\appendcurve[
    ustart = {0}
    ,ustop = {1}
    ,usamples = {36}
    ,x = {cos(tau*u)}
    ,y = {sin(tau*u)}
    ,z = {0}
    ,draw options = {
        draw
        ,red
    }
```

```
    ,transformation = {euler(pi/2,pi/3,pi/6)}
    ,arrow tip = {true}
    ,arrow tip options = {fill = green}
    ,arrow tail = {true}
]
```

Command Name: `\appendsolid`

This command tessellates a projectively transformed parametric solid into boundary triangles, and adds those triangles to the list of segments. An example of its usage is as follows:

```
\appendsolid[
    ustart = {0}
    ,ustop = {1}
    ,usamples = {18}
    ,vstart = {0}
    ,vstop = {1}
    ,vsamples = {6}
    ,wstart = {0}
    ,wstop = {1}
    ,wsamples = {2}
    ,x = {w*sphere(tau*u/36,pi*v/18)[1][1]}
    ,y = {w*sphere(tau*u/36,pi*v/18)[1][2]}
    ,z = {w*sphere(tau*u/36,pi*v/18)[1][3]}
    ,fill options = {
        preaction = {
            fill = green
            ,fill opacity = 0.8
        }
        ,postaction = {
            draw
            ,line join = round
            ,line cap = round
        }
    }
    ,transformation = {euler(pi/2,pi/3,pi/6)}
]
```

3.3.2 How Parametric Objects Are Tessellated into Simplices with Transformations Applied

A zero-dimensional parametric object requires only one sample and is tessellated by a single zero-dimensional affine simplex (a point). A one-dimensional parametric object (a curve) often requires more, and is tessellated by one-dimensional affine simplices (line segments). This is done using an even subdivision of the parameter

into samples. Similarly, a two-dimensional parametric object is sampled along both input parameters to obtain a mesh, which is then triangulated. Triangles are used instead of quadrilaterals because quads are often not coplanar and thus ambiguous. A parametric solid is tessellated along three parameters in the same way.

3.3.3 How Intersections Between Simplices Are Detected and Resolved

Intersections are eliminated through minimal partitioning of simplices. To identify the cases, a bottom-up approach is taken: we first determine the meaningful ways to partition lower-dimensional simplices, and then extend this to higher dimensions.

There is no meaningful way to partition a point by another point, so that case is omitted. A point can divide a line segment into two if they intersect, so this case is retained. A point and a triangle have no meaningful partitioning with respect to each other.

A line segment can partition another line segment, and a line segment can also be partitioned by a triangle. Finally, a triangle can be partitioned by another triangle. For reasons of minimality, only one of the two intersecting simplices is partitioned in each case.

Partitioning a Line Segment by a Point

We first check whether a point lies on a line segment, and if it does, we partition the line segment at that point. To perform this test, the line segment is expressed as a one-dimensional affine basis by replacing the second endpoint with its difference from the first. We then compute the vector from the affine origin to the point.

Next, we take the orthogonal projection of this vector onto the segmen's direction vector. If the sum of this projection with the affine origin is nearly coincident with the point, we proceed with testing; otherwise, the point is not on the segment. To confirm, we apply Gauss–Jordan elimination to express the point in terms of the line's affine basis. If the resulting coordinate lies within the unit interval, the point is indeed on the segment, and the segment is partitioned.

Partitioning a Line Segment by Another Line Segment

We first check whether two line segments intersect, and if they do, we partition one of them at the intersection point. To detect an intersection, each segment is expressed as a one-dimensional affine basis. The lines spanned by these bases are then intersected using Gauss–Jordan elimination. If the resulting parameters for both segments lie within their respective unit intervals, the segments intersect, and partitioning is performed.

Partitioning a Line Segment by a Triangle

If an intersection between the line segment and the triangle can be detected, we partition the line segment at that point. Both simplices are expressed as affine bases in

their respective dimensions, and the intersection is obtained by solving the resulting linear system via Gauss–Jordan elimination. If the line segment's coefficient lies within the unit interval, the candidate intersection is retained; otherwise, it is discarded.

When the line segment does intersect the plane of the triangle, we determine whether the intersection point lies inside the triangle using the cross-product method [httc]. In this approach, the triangle is tessellated into one-dimensional affine bases (its edges), and each vertex is connected to the point being tested, forming another one-dimensional affine basis. For every pair of affine bases emanating from a common vertex, we compute their rotationally ordered cross product. If all such cross products point in the same direction, the point lies inside the triangle. If the point lies outside, at least one rotationally ordered cross product will traverse its angle in the opposite orientation, producing an anticommutative result.

Partitioning a Triangle by Another Triangle

The partitioning of a triangle by another triangle builds upon the previous cases. The first step is to detect intersections between their edges. If two intersections are found (the expected outcome, though safeguards are included for degenerate cases), the cutting triangle partitions the other along the line defined by these two points. This produces a triangle and a quadrilateral, the latter of which is subdivided into two triangles. The intersections themselves are computed by treating each edge as a line segment and determining its intersection with the opposing triangle.

3.3.4 How Occlusion Ordering Is Determined

Occlusion ordering in our system is performed by first orthogonally projecting the two simplices onto the viewing plane. If a point of overlap is detected, we sort them according to the inverse orthogonal projection of that point back onto each shape.

Point Versus Point

If the points are not coincident but their projections coincide, then they are ordered by depth. Otherwise, the test remains inconclusive.

Point Versus Line Segment

This routine determines the occlusion relationship between a point and a line segment. It first expresses the line segment in terms of an affine basis, given by its origin and direction vector. The point is then projected orthogonally onto the line defined by the segment, producing a candidate projection. If the projection of the point onto the viewing plane (its xy-coordinates) is nearly identical to the projection of this candidate, the test proceeds; otherwise, the point and line segment are considered not to occlude each other. Next, the algorithm checks whether the projection lies within the bounds of the segment itself by comparing vector signs and computing a normalized coefficient. If the projection falls within the unit interval of the segment, the algorithm reduces the problem to comparing the depth of the point and its projection on the line. If these conditions fail, the test is inconclusive.

Point Versus Triangle

This routine compares the occlusion relationship between a point and a triangle. The point and the triangle are first projected orthogonally onto the viewing plane, where a cross-product test is applied to check whether the projected point lies inside the projected triangle. If this test fails, the point and triangle are considered not to occlude one another. If the point lies inside the projection, the algorithm then projects the point vertically onto the plane of the triangle. Using an affine basis for the triangle, the barycentric coordinates of the point with respect to the triangle are solved via Gauss–Jordan elimination. If the solution lies within the unit square (ensuring the projection is inside the triangle), the algorithm reduces the problem to a point-point occlusion comparison between the original point and its projection on the triangle. If any step fails to satisfy these conditions, the test remains inconclusive.

Line Segment Versus Line Segment

This routine determines the occlusion relationship between two line segments. First, both segments are projected onto the viewing plane, and their direction vectors are computed. If the direction vectors are not parallel, the algorithm solves for parameters t and s in the affine equations of the two lines using Gauss–Jordan elimination. If both parameters lie within the unit interval, the intersection point of the two segments is found, and the occlusion is reduced to a point-point comparison of the coincident intersection.

If the direction vectors are parallel, the algorithm instead falls back to endpoint tests: each endpoint of one segment is compared against the other segment using the point-line occlusion procedure. If any endpoint is found to occlude, the segments are ordered accordingly. If no consistent ordering can be determined, the test remains inconclusive.

Line Segment Versus Triangle

This routine compares the occlusion relationship between a line segment and a triangle. The algorithm begins by testing each endpoint of the segment against the triangle using the point-triangle occlusion procedure. It then tests the segment itself against each of the triangle's edges using the line-segment occlusion procedure. If any of these comparisons establishes a definite ordering (occluded in front or behind), that result is returned. If no consistent conclusion can be drawn from the endpoint and edge tests, the routine returns inconclusive.

Triangle Versus Triangle

This routine compares the occlusion relationship between two triangles. The algorithm first tests each edge of the first triangle against the second using the line-segment–triangle occlusion procedure. If any edge establishes a definite ordering (in front or behind), that result is immediately returned. If these edge tests are inconclusive, the algorithm proceeds by testing the vertices of the first triangle against the second, and the vertices of the second triangle against the first, using the point-triangle occlusion

procedure. If any of these vertex tests determines a clear ordering, that result is returned. If neither the edge nor the vertex tests establish a consistent relationship, the routine concludes that the occlusion status is inconclusive.

3.3.5 How the Final Illustration Is Rendered in LaTeX

Once the simplices are ordered, the command `\displaysegments` identifies the type of each simplex along with any assigned options, and prints empty path statements that the user may customize with their desired options.

3.4 Rationale and Development Process of the Approach

I chose this method over alternatives because I wanted a reliable way to render low-resolution parametric objects that intersect accurately. Existing graphics software in the TEX ecosystem often handled these cases incorrectly, which motivated me to build a system of my own. The goal was not only to address this gap within TEX but also to contribute a more principled solution to computer graphics more broadly.

3.5 Data, Tools, and Resources Used

This project does not rely on external datasets; it operates entirely on geometric primitives generated within the system itself. The implementation is written primarily in Lua, chosen for its seamless integration with LuaLATEX. All major libraries for vector arithmetic, cross-product tests, and affine-basis manipulations were developed independently by the author. A few minor helper functions, notably the Gauss–Jordan solver and the topological sort, were assisted by ChatGPT. For visualization and rendering within LATEX, the Ti*k*Z package was used to draw and display all simplices.

3.6 Implementation Details

For the implementation details, please see the source code.

3.7 Validation Strategy

To validate the algorithm, all clipping and occlusion cases will be tested systematically. This includes points, line segments, and triangles in intersecting and non-intersecting configurations, as well as triangle-triangle and line-triangle clipping scenarios. Practical examples will be provided to illustrate correctness. Validation is primarily internal, but results can be cross-checked against existing computational geometry tools for external verification.

3.8 Challenges Encountered and Solutions Implemented

Being the first to coherently orchestrate linear algebra for the clipping and occlusion sorting of 0–2-dimensional affine simplices, I encountered several failed attempts before finally achieving a correct solution. The major breakthrough came when I realized that starting with low-dimensional cases and progressively working upward was the key to success.

3.9 Limitations of the Current Methodology

The current methodology does not account for cyclic overlap; this aspect is deferred to a future edition.

3.10 Remaining Challenges and Directions for Future Work

The main remaining challenge is the elimination of cyclic overlap. Additionally, future work may include clipping objects by other objects and potentially working with their intersection sets.

3.11 Summary of the Methodology

In this chapter, a computational and developmental approach was presented for systematically tessellating, clipping, and occlusion-sorting zero- to two-dimensional affine simplices. The methodology leverages affine linear algebra to detect intersections and determine depth ordering, implemented entirely in Lua and LaTeX with visualization via Ti*k*Z. By addressing low-dimensional cases first and progressively building up, the system ensures accurate handling of points, line segments, and triangles, while all major clipping and occlusion scenarios are implemented.

With the methodology fully specified and operational, the next chapter focuses on validating the approach: testing all relevant cases, examining correctness, and demonstrating the effectiveness of the algorithms in practice.

Chapter 4

Validation

4.1 Introduction to Validation

The purpose of this validation chapter is to systematically evaluate the algorithms developed for tessellation, clipping, and occlusion sorting of affine simplices. Validation ensures that the approach functions as intended, accurately detects intersections, correctly orders overlapping simplices, and produces reliable visual outputs. Success is measured in terms of correctness, consistency across all simplex types, and the faithfulness of rendered illustrations to the underlying geometric models.

This chapter is structured to present internal tests for each class of simplex (points, line segments, triangles), followed by combined scenarios that exercise clipping and occlusion in increasingly complex configurations. Each section provides detailed examples, examines algorithmic behavior, and highlights both successes and limitations, offering a thorough assessment of the methodology's practical effectiveness.

4.2 Internal Evaluation

4.2.1 Occlusion

Point Versus Point

4.2.2 Point-Point Occlusion Test

To validate the point-point occlusion routine, we use the following example. In this test, a green point is expected to occlude a red point, while a blue point remains independent and does not participate in the occlusion ordering. This setup allows us to verify that the depth comparison logic correctly identifies which point lies in front.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
```

```
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendpoint[
            x = {0}
            ,y = {0}
            ,z = {0}
            ,fill options = {fill = red}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendpoint[
            x = {0}
            ,y = {0}
            ,z = {1}
            ,fill options = {fill = green}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendpoint[
            x = {1}
            ,y = {1}
            ,z = {0}
            ,fill options = {fill = blue}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

When rendered, the green point should appear above the red point, demonstrating that the occlusion sorting functions as intended, while the blue point remains unobstructed.

Point Versus Line Segment

For testing the point-line occlusion routine, we set up a scenario where a red point should appear on top of a blue line segment. This checks that the algorithm correctly identifies when a point occludes a segment based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendpoint[
            x = {0}
            ,y = {0}
```

```
            ,z = {1}
            ,fill options = {fill = red}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {u}
            ,z = {0}
            ,draw options = {
                draw = blue
                ,line cap = round
            }
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the red point should visually appear above the blue line segment, confirming that the point-line occlusion comparison is working correctly.

Point Versus Triangle

Point Versus Triangle

For testing the point-triangle occlusion routine, we set up a scenario where a blue point should appear on top of a red triangle. This checks that the algorithm correctly identifies when a point occludes a triangle based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendpoint[
            x = {0}
            ,y = {0}
            ,z = {1}
            ,fill options = {fill = blue}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendsurface[
```

```
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {2}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,transformation = {translate(0,0,-3)}
            ,fill options = {fill = red}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the blue point should visually appear above the red triangle, confirming that the point-triangle occlusion comparison is working correctly.

Line Segment Versus Line Segment

For testing the segment-segment occlusion routine, we set up a scenario where a red line segment should appear in front of a blue line segment. This checks that the algorithm correctly identifies when one segment occludes another based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {u}
            ,z = {1}
            ,draw options = {draw = red, line cap = round}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
```

```
            ,y = {-u}
            ,z = {0}
            ,draw options = {draw = blue, line cap = round}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the red line segment should visually appear in front of the blue line segment, confirming that the segment-segment occlusion comparison is working correctly.

Line Segment Versus Triangle

For testing the segment-triangle occlusion routine, we set up a scenario where a red line segment should appear in front of a blue triangle. This checks that the algorithm correctly identifies when a segment occludes a triangle based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {u}
            ,z = {1}
            ,draw options = {draw = red, line cap = round}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {2}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,transformation = {translate(0,0,-3)}
```

```
            ,fill options = {fill = blue}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the red line segment should visually appear in front of the blue triangle, confirming that the segment-triangle occlusion comparison is working correctly.

Triangle Versus Triangle

For testing the triangle-triangle occlusion routine, we set up a scenario where a red triangle should appear in front of a blue triangle. This checks that the algorithm correctly identifies which triangle occludes the other based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {2}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,transformation = {translate(0,0,-3)}
            ,fill options = {fill = blue}
        ]
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {2}
            ,x = {u}
            ,y = {v}
            ,z = {1}
            ,transformation = {
```

```
                matrix_multiply(zrotation(pi/4),translate(0,0,-
3))
            }
            ,fill options = {fill = red}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the red triangle should visually appear in front of the blue triangle, confirming that the triangle-triangle o

4.2.3 Clipping

Line Segment By Point

For testing the line-segment clipping routine, we set up a scenario where a red point clips a blue line segment. This verifies that the algorithm correctly identifies the portion of the line segment that lies in front of or behind the point in depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendpoint[
            x = {0}
            ,y = {0}
            ,z = {0}
            ,fill options = {fill = red}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {u}
            ,z = {0}
            ,draw options = {
                draw = blue
                ,line cap = round
            }
            ,transformation = {translate(0,0,-3)}
        ]
```

```
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the blue line segment should be divided by the red point, confirming that the line-segment clipping routine handles point-based occlusion correctly.

Line Segment By Line Segment

For testing the line-segment clipping routine, we set up a scenario where a red line segment clips a blue line segment. This checks that the algorithm correctly identifies the portion of the blue segment that lies in front of or behind the red segment based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {0}
            ,z = {0}
            ,draw options = {draw = red, line cap = round}
            ,transformation = {translate(0,0,-2)}
        ]
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {0}
            ,y = {u}
            ,z = {0}
            ,draw options = {draw = blue, line cap = round}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the blue line segment should be partially clipped by the red segment, confirming that the line-segment clipping routine handles segment-segment occlusion correctly.

Line Segment By Triangle

For testing the line-segment clipping routine against a triangular surface, we set up a scenario where a blue line segment intersects or passes behind a red triangle. This checks that the algorithm correctly identifies which portion of the line segment lies in front of or behind the triangle based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendcurve[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,x = {u}
            ,y = {u}
            ,z = {u}
            ,draw options = {draw = blue, line cap = round}
            ,transformation = {translate(0,0,-2)}
        ]
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {2}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {2}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,draw options = {draw = red, line cap = round}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the line segment should appear appropriately clipped or occluded by the triangle, confirming that the line-segment–triangle occlusion is working correctly.

Triangle By Triangle

For testing the triangle-triangle occlusion routine, we set up a scenario where a blue triangular surface intersects or passes behind a red triangular surface. This checks

that the algorithm correctly identifies which portions of each triangle lie in front of or behind the other based on depth.

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {4}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {4}
            ,x = {u+0.1}
            ,y = {v+0.2}
            ,z = {u+v+0.3}
            ,fill options = {fill = blue}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {4}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {4}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,fill options = {fill = red}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

Upon rendering, the blue triangle should appear appropriately occluded by the red triangle where depth dictates, confirming that the triangle-triangle occlusion routine is working correctly.

4.2.4 Performance Assessment

Each pair of simplices—or segments—must undergo comparison. For n segments, the total number of comparisons is

$$\sum_{i=1}^{n}(i-1).$$

Most comparisons are quickly resolved using a simple bounding-rectangle overlap check on the viewing plane. The computation becomes expensive primarily when many segments overlap, requiring numerous occlusion tests. This scenario typically dominates the algorithm's runtime.

4.2.5 Pedagogical Effectiveness of Visualizations Compared to Existing Methods

No other system is capable of illustrating intersecting three-dimensional tessellated parametric objects with the same precision. `lua-tikz3dtools` achieves this effectively. Consider the diagram produced by the following LaTeX document:

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \pgfmathsetmacro{\i}{1}
    \begin{tikzpicture}
        \setobject[
            name = {T}
            ,object = {
                matrix_multiply(
                    euler(pi/2,pi/3,7*pi/6)
                    ,translate(0,0,-6)
                )
            }
        ]
        \foreach \t in {1,2,3} {
        \pgfmathsetmacro{\rotation}{\t*pi/12}
        \appendsurface[
            ustart = {0}
            ,ustop = {1}
            ,usamples = {18}
            ,vstart = {0}
            ,vstop = {1}
            ,vsamples = {9}
            ,x = {2*cos(\t*tau/3) + sphere(u*tau, v*pi)[1][1]}
            ,y = {2*sin(\t*tau/3) + sphere(u*tau, v*pi)[1][2]}
```

```
            ,z = {\t/3 + sphere(u*tau, v*pi)[1][3]}
            ,transformation = {
                matrix_multiply(
                    euler(\rotation,\rotation,\rotation)
                    ,T
                )
            }
            ,fill options = {
                preaction = {
                    fill = white
                }
                ,postaction = {
                    draw
                    ,line cap = round
                    ,line join = round
                    ,ultra thin
                }
            }
        ]
        }
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {4}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {4}
            ,x = {4*u}
            ,y = {4*v}
            ,z = {\i + u}
            ,transformation = {T}
            ,fill options = {
                preaction = {
                    fill = gray!70!white
                }
                ,postaction = {
                    draw
                    ,line cap = round
                    ,line join = round
                    ,ultra thin
                }
            }
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

4.2.6 Robustness and Reliability Testing

`lua-tikz3dtools` is capable of handling most intersection cases; however, certain trivial cases are not yet resolved. Consider the following example that illustrates the bug:

```
\documentclass[
    tikz
    ,border = 1cm
]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
    \begin{tikzpicture}
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {4}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {4}
            ,x = {u}
            ,y = {v}
            ,z = {u+v}
            ,fill options = {fill = blue}
            ,transformation = {translate(0,0,-3)}
        ]
        \appendsurface[
            ustart = {-1}
            ,ustop = {1}
            ,usamples = {4}
            ,vstart = {-1}
            ,vstop = {1}
            ,vsamples = {4}
            ,x = {u}
            ,y = {v}
            ,z = {0}
            ,fill options = {fill = red}
            ,transformation = {translate(0,0,-3)}
        ]
        \displaysegments
    \end{tikzpicture}
\end{document}
```

4.3 Limitations of the Validation Process

The validation carried out in this chapter is intentionally narrow in scope. Tests were constructed around simple representative cases—points, line segments, and triangles—together with a small set of clipping and occlusion examples. While these confirm that the algorithms function correctly in the illustrated situations, they do not amount to an exhaustive survey of all possible geometric interactions. Performance was assessed in terms of comparison counts and overlap checks, but no large-scale benchmarks were attempted. Likewise, pedagogical evaluation was limited to a single illustrative example rather than a systematic study of learning outcomes.

Consequently, the validation should be understood as a demonstration of correctness in selected cases, not a comprehensive assessment of the system's behavior under all conditions. Broader testing, formal benchmarks, and structured user studies remain as important directions for future work.

4.4 Summary of Validation Results

The validation examples presented here show that the algorithms successfully resolve occlusion and clipping among points, line segments, and triangles, and that they produce visualizations faithful to the intended geometric relationships. The performance analysis confirmed that the dominant cost arises from overlapping segments, consistent with theoretical expectations. A demonstration of pedagogical effectiveness illustrated the system's ability to depict intersecting three-dimensional tessellated objects with a clarity not available in other tools. Although the validation was sparse and limited to selected cases, it provides evidence that the approach meets its stated goals of correctness, consistency, and interpretability, while also revealing directions for further refinement and expansion.

Chapter 5

Results and Analysis

5.1 Introduction to Results and Analysis

The purpose of this chapter is to present the results of the system and to analyze their significance in relation to the goals established in the methodology. Whereas the validation chapter focused on demonstrating correctness in selected test cases, here the emphasis is on synthesizing those outcomes into a broader assessment of the system's achievements and limitations. In particular, this chapter highlights the capacity of the algorithms to generate accurate and interpretable visualizations, their relative performance characteristics, and their pedagogical contributions.

The discussion is organized into sections on system achievements, validation outcomes, and synthesis across test domains, followed by consideration of the broader implications for both geometric computation and mathematical visualization. By connecting the technical development of the methods with their tested performance and educational value, this chapter provides a bridge between the detailed validation and the concluding arguments of the thesis.

5.2 System Achievements and Capabilities

The system developed in this work achieves its primary objectives by providing a structured pipeline for rendering three—dimensional geometric objects with high precision. Objects—including surfaces, curves, solids, and points—are first tessellated based on their parametric definitions. This tessellation forms the foundation for subsequent processing: the system detects intersections between tessellated objects and then performs depth-based occlusion sorting to ensure correct visual ordering. Validation examples confirm that this sequence produces accurate, interpretable visualizations in a variety of scenarios, including multi-surface, multi-solid, and mixed-object configurations.

A particularly notable capability is the syste's handling of intersecting tessellated surfaces and solids with mathematical precision. Unlike traditional static illustrations, the system preserves fine structural details of intersections and correctly orders overlapping objects in depth. Performance analysis shows that most comparisons are

quickly resolved using bounding checks, with only a fraction requiring full intersection and occlusion tests, demonstrating both efficiency and correctness.

Beyond correctness and performance, the system provides pedagogical value by enabling visualizations that were previously difficult or impossible to generate. The pipeline—from parametric tessellation to intersection and occlusion—ensures that even complex arrangements of objects remain faithful to their geometric definitions, supporting both interpretation and educational use.

5.2.1 Functional Capabilities

The system offers robust features for visualizing and manipulating three-dimensional surfaces, curves, solids, and points. Objects are first tessellated according to their parametric definitions, ensuring a faithful geometric representation. Tessellated objects are then checked for intersections, with overlapping regions accurately resolved, and finally sorted according to depth to produce correct occlusion in the rendered output. Validation examples across points, curves, surfaces, and solids confirm that the pipeline operates correctly in practice.

Advanced parametric and tessellated objects can be combined, rotated, translated, and visualized while maintaining precise intersection and occlusion relationships. The system's ability to handle intersecting tessellated surfaces and solids with clarity and accuracy demonstrates its alignment with the original design goals of producing reliable, correct, and pedagogically effective three-dimensional visualizations.

5.2.2 Comparative Advantages Over Existing Methods

No existing system can accurately visualize intersecting tessellated surfaces, solids, curves, and points in three dimensions with the same level of precision. Traditional 3D rendering tools either do not support parametric tessellation directly in LaTeX, or they fail to correctly handle intersections and depth-based occlusion, resulting in misleading or incomplete visualizations. In contrast, this system explicitly tessellates parametric objects, computes intersections, and then sorts occlusions, ensuring that rendered diagrams faithfully represent the geometric relationships.

The advantages are both technical and practical. Complex intersections that would require manual adjustments in other tools are resolved automatically, saving time and reducing errors. Visual clarity is improved because overlapping objects are properly ordered, and subtle intersections are preserved rather than approximated. While performance was not exhaustively benchmarked, preliminary tests show that the pipeline scales efficiently for small to moderately complex scenes, resolving most non-overlapping objects quickly. Overall, the system offers a capability not available in existing LaTeX-based or general-purpose 3D visualization tools, making it both novel and directly useful for research and educational purposes.

5.2.3 Limitations Observed in Practice

While the system achieves its primary objectives, several practical limitations were observed. First, the occlusion can become slow for scenes with many overlapping

objects. Although bounding checks help resolve most comparisons efficiently, complex overlaps still incur significant computational cost, limiting scalability. Second, certain trivial intersection cases are not correctly handled due to a known bug, which occasionally results in visual inaccuracies. These issues do not undermine the correctness of the majority of tested examples, but they highlight areas for refinement.

It is also important to distinguish between technical limitations and inherent scope constraints. The system was designed for accuracy and pedagogical clarity rather than raw speed or exhaustive large-scale benchmarking. Its focus on parametric tessellation and precise intersection means that performance will naturally degrade as complexity grows, which is an expected trade-off. Recognizing these limitations provides clear directions for future improvement, including optimization of the pipeline, more robust handling of edge cases, and broader performance testing.

5.3 Validation Results and Interpretation

The validation tests demonstrate that the system reliably performs tessellation, intersection detection, and occlusion sorting for points, curves, surfaces, and solids in the cases examined. Point–point, point–curve, point–surface, curve–curve, curve–surface, and surface–surface scenarios all produced the expected visual outcomes, confirming that the processing pipeline functions correctly in practice. These results align closely with the validation strategy described earlier, which emphasized representative test cases to ensure correctness across object types and interactions.

Interpretation of these results highlights both strengths and practical implications. The system's ability to accurately resolve intersections and maintain proper occlusion ensures that visualizations are faithful to the underlying geometric definitions, which is particularly important for educational and research applications. While the tests were sparse and focused on small-to-medium-scale scenes, they indicate that the core algorithms operate as intended and can serve as a reliable foundation for more complex visualizations. Additionally, the pedagogical demonstration showed that previously difficult or ambiguous geometric interactions can now be rendered clearly, supporting improved comprehension.

However, the results also reveal the system's current limitations. Processing can be slow for highly overlapping or densely tessellated objects, and certain trivial intersection cases are not fully handled due to a known bug. These observations do not negate the correctness of validated scenarios but emphasize that further optimization and robustness checks are necessary. Overall, the validation provides evidence that the system meets its primary functional goals while also identifying specific areas for future refinement and expansion.

5.3.1 Internal Evaluation Outcomes

The controlled tests conducted on points, curves, surfaces, and solids show that the system reliably produces correct intersections and maintains proper occlusion ordering in all tested scenarios. Accuracy was consistently high in these representative cases, and the rendered visualizations faithfully reflected the underlying geometric

definitions. Efficiency was generally acceptable for small to moderately complex scenes, although performance decreased noticeably for configurations with many overlapping tessellated objects. Robustness was also strong in typical use cases, but a small number of trivial intersection cases remain unresolved due to a known bug.

These outcomes indicate that the system is reliable for its intended scope: generating precise and interpretable visualizations for parametric geometric objects in controlled or moderately complex scenes. While not fully optimized for large-scale or highly dense configurations, the results demonstrate that the core pipeline—tessellation followed by intersection and then occlusion—functions as designed, providing a solid foundation for both practical use and further development.

Pedagogical Effectiveness of Visualizations

The visualizations produced by the system clearly show intersections and occlusion among surfaces, solids, curves, and points. In the examples tested, it was immediately apparent which objects overlapped or were in front, making the geometric relationships easier to interpret than with static diagrams.

No formal study or quantitative measurements were conducted. All conclusions about clarity and usefulness are based solely on direct observation of the rendered outputs. This demonstrates that the system can generate interpretable visualizations.

5.4 Synthesis and Interpretation of Results

The system demonstrates consistent accuracy across all tested object types—points, curves, surfaces, and solids. The pipeline, consisting of tessellation from parametric definitions, followed by intersection detection and depth-based occlusion sorting, reliably produces correct visual outputs in the scenarios examined. Functional testing confirms that these core capabilities work as intended, while performance assessment shows that most non-overlapping objects are handled efficiently, with slower processing occurring only in complex overlapping configurations.

Validation results indicate that the rendered visualizations faithfully represent the underlying geometry. The system correctly resolves intersections and occlusion relationships in all tested examples, providing clear and interpretable outputs. Observed limitations include slow processing for dense or highly overlapping scenes and a small number of trivial intersection cases that fail due to a known bug. These results are consistent with the system's design focus on accuracy over speed and do not contradict functional expectations.

Overall, these findings show that the system meets its primary objectives of correctness, reliability, and interpretability in the tested scenarios. The results demonstrate that the approach provides precise three-dimensional visualization and a solid f oundation for practical use and further development.

5.5 Implications of Findings

The validation results show that the algorithm reliably handles tessellation of parametric 3D objects, computes intersections between tessellated objects, and performs depth-based occlusion sorting. This represents a significant advancement in computational geometry, as it enables the accurate ordering and visualization of complex intersecting surfaces, solids, curves, and points—a problem not fully addressed by existing methods.

From a computational perspective, the algorithm demonstrates that carefully structured processing—tessellation first, intersection second, occlusion last—can systematically resolve overlapping objects with mathematical precision. While performance slows for dense or highly overlapping scenes, the correctness and generality of the approach provide a foundation for further optimization and scaling.

The broader impact extends beyond any specific rendering environment. The algorithm can be integrated into simulation, modeling, CAD, or visualization systems to handle complex parametric geometries accurately. It establishes a new method for processing intersections nd occlusions in 3D, which could influence both practical applications and future research in computational geometry, geometric modeling, and computer graphics.

5.6 Summary of Results and Analysis

The system successfully implements a pipeline for tessellating parametric 3D objects, detecting intersections, and performing depth-based occlusion sorting, producing accurate and interpretable outputs for points, curves, surfaces, and solids. It demonstrates capabilities beyond existing methods by reliably resolving complex intersections that other tools cannot handle, though performance slows for dense or highly overlapping scenes and a small number of trivial intersection cases remain unresolved. These results confirm the algorithm's correctness and generality, highlight its practical and computational significance.

Bibliography

[htta] Jasper (https://math.stackexchange.com/users/1499599/jasper). *Exact Triangle Sorting for Orthographic Rendering of a Triangulated Surface.* Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/5063772 (version: 2025-05-17). URL: https://math.stackexchange.com/q/5063772.

[httb] P123 (https://math.stackexchange.com/users/1499599/p123). *How does the dot product give correct depth ordering in orthographic 3D projections?* Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/5062592 (version: 2025-05-06). URL: https://math.stackexchange.com/q/5062592.

[httc] M.B. (https://math.stackexchange.com/users/2900/m-b). *Determining if an arbitrary point lies inside a triangle defined by three points?* Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/51328 (version: 2011-07-14). eprint: https://math.stackexchange.com/q/51328. URL: https://math.stackexchange.com/q/51328.

[Jas] Jasper. *TeX StackExchange user profile: Jasper.* https://tex.stackexchange.com/users/319072/jasper. Accessed: 2025-08-12.

[Ski25] Skillmon. *Answer to "3d point sorting in tikz".* Accepted answer with LaTeX3-based segment list and sorting macros. 2025. URL: https://tex.stackexchange.com/a/734098.