

The `lua-tikz3dtools` package for 3D illustrations in \LaTeX

Jasper Nice

Abstract

The `lua-tikz3dtools` package (currently version 2.0.1) extends `TikZ` with support for occlusion handling of tessellated 3D parametric objects. It introduces two main algorithms: a transitive partial order comparator for points, line segments, and triangles, and a minimal partitioning procedure that resolves tiles which are capable of partitioning one another by subdividing them into fragments which are incapable of this. Together these provide a framework for drawing 3D mathematics illustrations in \LaTeX with clear and unambiguous occlusion. The remaining obstacle is the problem of resolving cyclic overlap, which is deferred to a later version.

1 Problem statement

1.1 The core problem being addressed

Many 3D mathematics illustrations are composed of tessellated parametric objects, which are themselves composed of points, line segments, and triangles. Of course, the final image is not the 3D shape itself, but rather a 2-dimensional representation of it after its tiles have been orthogonally projected onto the viewing plane. If we draw these projected tiles on the viewing plane in the order imposed by the traversal of the parametric equations of their parent objects, we will most often be left with an incorrectly occluded diagram.

`lua-tikz3dtools` provides two core contributions to the community. Firstly there is a transitive partial order occlusion comparator which is capable of occlusion-sorting points, line segments, and triangles; this comparator only functions on sets of these objects which are void of both objects which cyclically occlude each other and objects which can partition one another. The second algorithm is capable of taking a set of tiles which may contain some which are capable of partitioning one another, and it does exactly that; this has the effect of transforming the set of tiles into one which is void of tiles which are capable of partitioning one another. Figure 1 demonstrates how this tool improves the pedagogical coherence of a 3D parametric illustration.

The systematic elimination of cyclic overlaps through partitioning is planned for a future version of the software, and in combination with the other two would yield the complete algorithm which I've envisioned. Fortunately, most mathematics illustrations do not contain tiles which cyclically overlap,

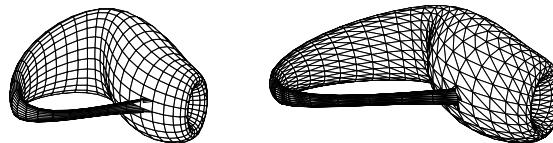


Figure 1: For purposes of showing the difference, the left figure is made with other software; compare to the right figure made with `lua-tikz3dtools`. The problem is pervasive among other graphics software.

hence the publication of the two current algorithms with the third only being planned.

1.2 Importance of the issue

Graphics software commonly includes some form of occlusion sorting for non-cyclically overlapping tiles which are incapable of partitioning one another. Some software uses incorrect tessellation methods, such as tessellating surfaces into quads instead of triangles — because quads are not always coplanar like triangles, some also support only the occlusion sorting of individual parametric objects and cannot sort the tiles of different objects relative to one another. Also, such software often relies on black-box methods or problematic techniques like sorting by centroid depth. `lua-tikz3dtools` possesses the capability to do this for an arbitrary set of points, line segments, and triangles under the specified conditions; the algorithm is a transitive partial order comparator for the systematic occlusion of tiles in 3D.

Sometimes when taking arithmetic sequences of samples along the dimensions of the domain, tessellating those points into tiles, and mapping those tiles into 3D, the resulting set of tiles will contain tiles which are capable of partitioning one another. Most graphics software is incapable of resolving this, though some do possess capabilities for partitioning tiles. `lua-tikz3dtools` uses an alternative approach to the elimination of tiles which are capable of partitioning one another through partitioning; in particular, this approach is designed to work for a set containing points, line segments, and triangles, containing members of the set which are capable of partitioning one another. After the procedure is performed, the problematic tiles are broken up into fragments which are incapable of partitioning one another.

1.3 Intended audience

This software is intended for mathematics illustrators who need to visualize parametric objects which have been tessellated into points, line segments, and triangles, including cases where those tiles are capable of partitioning one another. In future versions, when the cyclic overlap problem is resolved, that will expand the audience to those who encounter this issue.

In addition to being useful to mathematics illustrators, `lua-tikz3dtools` also serves as a framework for others who make 3D software to learn from and improve on.

2 Literature review

2.1 Attempts in practice and academia

`lua-tikz3dtools` uses an approach that is distinct from those in practice and academia. Hence, this discussion will mostly reflect my own learning journey making 3D math illustrations.

My first real exposure to surface tessellation and occlusion sorting was when I tried implementing surface generation software for drawing a sphere in `TikZ`. Back then, my surfaces were tessellated into quads, and those quads were ordered by the depths of their centroids [1, 3]. Soon after, I started tessellating my surfaces into triangles because they are always coplanar, unlike quads. Initially, I attempted a simple centroid depth sort on the tessellations of various surfaces into triangles, only to come to the realization that this often produces significant occlusion errors [4].

I had the strong conviction that a set of triangles in 3D which do not cyclically overlap in the direction of the viewer, and which are incapable of partitioning one another, should be capable of being ordered with respect to the direction of the viewer such that they would appear with essentially the same occlusive relationships as they would under a z-buffer, except by using a transitive partial order occlusion relation instead of taking vast numbers of samples.

I later approached the question of implementing an occlusion sorter for just triangles, and received some very insightful advice from a user: have a definitive relationship between two triangles only if their orthogonal projections on the viewing plane overlap [2]. This was a foundational breakthrough in the way I thought about occlusion of tiles in 3D.

After a few unsuccessful attempts at implementing this idea, alongside a concurrent study of basic linear algebra, I had the insight to express the entire problem in terms of what I call affine linear algebra. I started with the occlusion comparator even though that is meant to follow the partitioning step; this was because partitioning would be useless without it, whereas occlusion ordering is useful on its own.

There are three types of tiles which 0–2-dimensional parametric objects are tessellated into: points, line segments, and triangles. Each type of tile can be compared to any other type or even its own; this means that there are six total cases. I worked through each case methodically, starting from the lowest-dimensional cases and working my way up

to the higher-dimensional cases. In fact, the higher-dimensional cases are greatly simplified by being written in terms of the solutions to lower-dimensional cases.

In a similar fashion, the partitioning algorithm was conceived by analyzing each tile relationship where partitioning made coherent sense, in an order that started with low-dimensional tiles and worked its way up. Both algorithms are described in technical detail in the methodology section below.

2.2 Gaps in existing tools

Some support for occlusion ordering of tiles is a common feature of 3D graphics software. For example, spline surfaces are supported in some, while some others use a z-buffer. Spline surfaces do not correctly handle surfaces which are capable of partitioning each other, whereas a z-buffer is capable of resolving this problem to a desired resolution.

None of these algorithms even recognize the existence of the transitive partial order occlusion relation that exists for any set of 3D points, line segments, and triangles, which do not cyclically overlap in the direction of the viewer, and which are incapable of partitioning one another.

Some algorithms use implementations of the BSB tree to resolve sets of tiles which are capable of partitioning one another, but this method commonly performs unnecessary partitions. The method in `lua-tikz3dtools` partitions a tile by another tile if and only if the first tile is capable of being partitioned by the other tile.

3 Methodology

3.1 How the approach works in technical detail

3.1.1 Parametric objects and their tessellations

3D parametric objects are mappings from a 0–3-dimensional domain to a 3-dimensional codomain. We tessellate the domain prior to its mapping to the codomain, and, depending on the domain’s dimension, the constituent tiles will be points, line segments, or triangles. Normally a 3-dimensional domain is composed of tetrahedrons, but `lua-tikz3dtools` just tessellates the surfaces of the outer faces. Currently, the tessellation is accomplished by breaking up each dimension of the domain into an arithmetic sequence, and tiling them into affine simplices.

Affine simplices These shapes — points, line segments, and triangles — are technically called the 0–2-dimensional affine simplices. An affine simplex is a set of linearly independent vectors which originate

from a common point, where the point does not necessarily coincide with the global origin.

For instance, a 0-dimensional affine simplex is a point with zero linearly independent basis vectors attached — a point. In contrast, a line segment is a 1-dimensional affine simplex: a point of origination with exactly one linearly independent vector protruding from it — spanning a line through the affine origin. A 2-dimensional affine simplex is a triangle, and its basis vectors span a plane through the affine origin.

The affine simplex representation is useful because it lets us navigate the 1–2-dimensional hyperplanes which are spanned by 1–2-dimensional affine simplices.

Parametric equations A 3D parametric equation is a set of three functions which determines the coordinates for each dimension of the codomain. These functions accept a 0–3-dimensional domain, corresponding to points, curves, surfaces, and solids respectively. For example, a common parametric equation for the sphere is

$$f\left(\begin{bmatrix} u \\ v \end{bmatrix}\right) = \begin{bmatrix} \sin(v) \cos(u) \\ \sin(v) \sin(u) \\ \cos(v) \end{bmatrix},$$

where $u \in [0, \tau]$ and $v \in [0, \pi]$.

3.1.2 Projective transformations

In `lua-tikz3dtools`, the `TikZ` canvas is our window onto the world. The observer does not move or change its orientation; instead, we transform the objects in the world around us so they make a nice picture in the window we are looking through. This gives us a useful standard affine basis for our 3D coordinate system, where the x -axis points to the right, the y -axis points up, the z -axis points towards us, and the origin is at the point $(0, 0, 0)$ on the `TikZ` canvas.

`lua-tikz3dtools` is designed to use projective transformations, which collectively include the linear, affine, and perspective transformations. Affine and perspective transformations are achieved using homogeneous transformation matrices. According to Rogers and Adams [5], “The 4×4 homogeneous transformation matrix can be partitioned into four separate sections:

$$\left[\begin{array}{cc|cc} 3 \times 3 & 3 \times 1 \\ \hline 1 \times 3 & 1 \times 1 \end{array} \right]$$

The 3×3 matrix produces a linear transformation in the form of scaling, shearing and rotation. The 1×3 row matrix produces translation, and the 3×1 column matrix produces perspective transformation. The final single element produces overall scaling.”

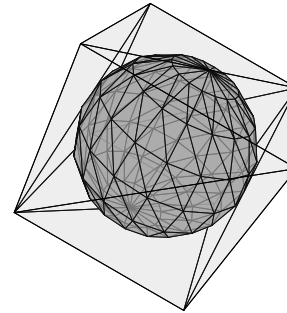


Figure 2: A parametric sphere in perspective.

3.1.3 User interface

The user is provided with commands for generating tessellations of projectively transformed parametric objects, as well as a command which also automatically handles partitioning and occlusion. After these automatic procedures, the command also outputs each tile to `TikZ`. In the future, this command will also endeavor to eliminate cyclic overlap through partitioning. For example, if a user wanted to draw a sphere in perspective, they could Lua^LA^TE^X the following code, illustrated in Figure 2.

```
\documentclass[tikz,border=1cm]{standalone}
\usepackage{lua-tikz3dtools}
\begin{document}
\begin{tikzpicture}
  \setobject[
    name = {T}
    ,object = {
      matrix_multiply(
        matrix_multiply(
          euler(pi/3,pi/3,-3*pi/6)
          ,matrix_multiply(
            translate(0,0,-2)
            ,{
              {1,0,0,0}
              ,{0,1,0,0}
              ,{0,0,1,-1/3}
              ,{0,0,0,1}
            }
          )
        )
      )
      ,matrix_multiply(
        xscale(2)
        ,yscale(2)
      )
    }
  ]
  \appendsurface[
    ustart = {0}
    ,ustop = {1}
    ,usamples = {18}
    ,vstart = {0}
    ,vstop = {1}
    ,vsamples = {9}
```

```

,transformation = {T}
,x = {sphere(u*tau, v*pi)[1][1]}
,y = {sphere(u*tau, v*pi)[1][2]}
,z = {sphere(u*tau, v*pi)[1][3]}
,fill options = {
  preaction = {
    fill = gray!70!white
    ,fill opacity = 0.7
  }
,postaction = {
  draw = black
  ,ultra thin
  ,line cap = round
  ,line join = round
}
}
]
\appendsolid[
  ustart = {-1}
  ,ustop = {1}
  ,usamples = {2}
  ,vstart = {-1}
  ,vstop = {1}
  ,vsamples = {2}
  ,wstart = {-1}
  ,wstop = {1}
  ,wsamples = {2}
  ,transformation = {T}
  ,x = {u}
  ,y = {v}
  ,z = {w}
  ,fill options = {
    preaction = {
      fill = gray!70!white
      ,fill opacity = 0.1
    }
  ,postaction = {
    draw = black
    ,ultra thin
    ,line cap = round
    ,line join = round
  }
}
]
\displaysegments
\end{tikzpicture}
\end{document}

```

3.1.4 Partitioning of tiles

Tiles which are capable of partitioning one another 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. In the scope of this work, two tiles intersect if and only if one is capable of partitioning the other. Intersection of two tiles is not taken in the set theoretic sense, while

intersections of two 0–2-dimensional affine subspaces are taken in the set theoretic sense.

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.

Line segment by 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 segment’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.

Line segment by 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.

Line segment by 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 a common cross-product method. 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.

Triangle by 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.

3.1.5 The transitive partial order occlusion relation

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 is 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 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 is 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 line segments is found, and the occlusion is reduced to a point-point comparison of the inverse orthogonal projection of that point onto both original line segments.

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

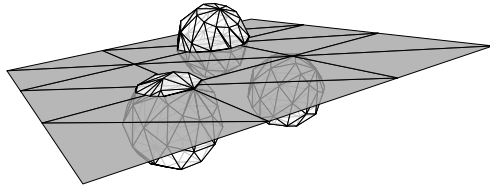


Figure 3: A tessellated plane seamlessly passing through some tessellated spheres.

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.

3.2 How the approach was arrived at

I wanted to illustrate properly occluded tessellations of 3D parametric objects, including those which were capable of partitioning one another, without occlusion artifacts and without a z-buffer. I spent a long time reasoning through every case and making many failed attempts along the way before I eventually had the insight to express everything in terms of affine linear algebra. That insight led to a few more failed attempts before I eventually got it right.

4 Validation of the visualizations in pedagogical terms

`lua-tikz3dtools` produces illustrations with unambiguous occlusion of tessellated 3D parametric objects, including when the tiles are capable of partitioning one another. For example, Figure 3 demonstrates the partitioning algorithm for triangles.

`lua-tikz3dtools` is also capable of camera positioning, though that is just an application of inverse affine transformations. For example, in Figure 4 we view a torus from the inside, with the tiles behind the camera being omitted. Currently we clip the canvas to our desired frame, but in the future this will be handled by keys. Due to the camera being the xy -plane through the origin, any tile which exceeds $z = 0$ will be culled. Finally, Figure 5 illustrates the stereographic projection of a mesh sphere under a perspective projection.

5 Results and analysis

These illustrations can be computationally intensive. Externalization through the `external TikZ` library or other means may be useful.

The ability to occlusion-order 0–2-dimensional affine simplices in 3D is a feat on its own, even if those simplices do not cyclically overlap and are incapable of partitioning one another. `lua-tikz3dtools`

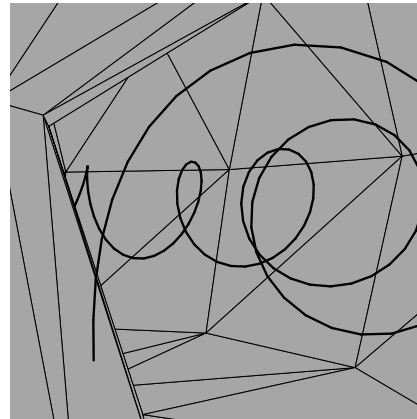


Figure 4: Viewing torus with a nested toroidal helix from within the torus.

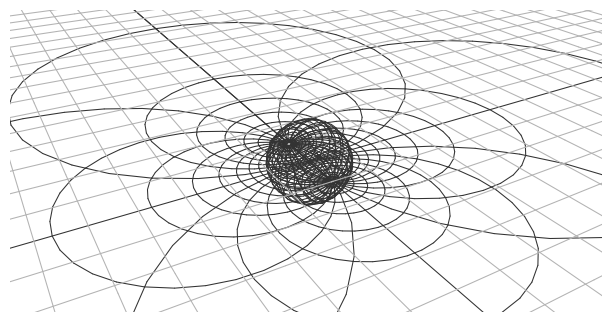


Figure 5: Stereographic projection of a sphere under perspective projection.

accomplishes this alongside a companion algorithm which is capable of partitioning simplices to eliminate ambiguity. My hope is that this software will inspire other 3D illustrators to more deeply examine their own approaches to occlusion, and hopefully help them pull out a few weeds.

References

- [1] J. Nice. 3d point sorting in tikz. TeX.SX. tex.stackexchange.com/q/734097/319072
- [2] J. Nice. Exact triangle sorting for orthographic rendering of a triangulated surface. Math.SX. math.stackexchange.com/q/5063772/1499599
- [3] J. Nice. How does the dot product give correct depth ordering in orthographic 3d projections? Math.SX. math.stackexchange.com/q/5062592
- [4] J. Nice. Tikz parametric surface debugging request. TeX.SX. tex.stackexchange.com/q/734691/319072
- [5] D.F. Rogers, J.A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, 1st ed., 1976.

◇ Jasper Nice
 animatetikz (at) gmail dot com
<https://tex.stackexchange.com/users/319072/jasper>