The lua-tikz3dtools Package, Version 1.0.0

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

Jasper

July 13, 2025

Contents

Chapter 1

Introduction

lua-tikz3dtools is a Lua-based 3D graphics package for Ti*k*Z.

1.1 What are the current capabilities of lua-tikz3dtools?

lua-tikz3dtools can render multiple non-intersecting triangulated mesh surfaces at once. It is also possible to draw a plane which has been clipped by a rectangular prism. There are commands for drawing and labelling geometric vectors too. Curves and surfaces can be defined using matrix transformations.

1.2 How does lua-tikz3dtools improve on existing packages?

There are two significant packages for 3D artwork in Ti*k*Z: pgfplots and tikz-3dplot. pgfplots can z-sort the faces of a single parametric surface, but not multiple surfaces at once. Both pgfplots and tikz-3dplot are capable of changing the camera view, but the result is necessarily orthographic, and the rotations are restricted to azimuth and elevation; that is, the z-axis is always vertical. tikz-3dplot offers useful pgfmath capabilities for doing vector operations. Neither pgfplots nor tikz-3dplot enable matrix transformations. lua-tikz3dtools is built to render multiple parametric objects at once, enable arbitrary object orientations in non-orthographic projections, and enable matrix transformations.

1.3 Where is the package at, and where it is heading?

The vision of the package lua-tikz3dtools is to one day be able to render arbitrary numbers of intersecting polygons, by clipping them with each other; this would enable arbitrary surface intersections, without visual artefacts. A more near term goal of lua-tikz3dtools is to also implement an ODE solving tool which approximates trajectories in vector fields. I also want to add capabilities for doing programming in Lua, instead of in pgf, essentially replacing the pgfmath capabilities of tikz-3dplot with faster Lua-based capabilities. Soon there will also be proper camera culling for projective scenes; that is, the action of not drawing segments which are behind the viewing plane.

There iss no way to store macros in Lua from the TEX end yet, so sometimes there is a problem with repetition. While the geometric vectors are nice, they are not yet rendered in 3D like the parametric objects—I want to fix this as well. I will also implement a way for the user to create their own reusable Lua functions.

Chapter 2

Geometric Vectors

2.1 How are geometric vectors defined in lua-tikz3dtools?

Geometric vectors are defined by two things: a start-coordinate, and a relative displacement. There are two classifications of geometric vectors; there are zero vectors, and non-zero vectors. A zero vector is represented by a small circle. A non-zero vector is represented by a line segment with an arrow. The endpoints of a non-zerovector may or may not have a circle centered on the endpoint. If there is no circle drawn, the arrow stops directly at the endpoint. If a circle is drawn, then the arrow tip is moved back by the circle's radius. Observe the tips of the vectors in Figure 2.1.1. Both tips lie on the same point, despite one arrow tip being offset by the circle's radius. The syntax for drawing Figure 2.1.1 is shown in the following code.

```
\begin{tikzpicture}
    \drawvector[points = both,yellow]{0,0}{2,0}
    \drawvector[blue]{4,0}{-2,0}
\end{tikzpicture}
```

There are four values for the key `points`: `neither`, `behind`, `front` and `both`. These tell the command `\drawvector` whether and where to put circles over the endpoints. See Figure 2.1.2, which uses the following code.

```
\begin{tikzpicture}
    \drawvector[points = neither]{0,0}{0,1} % a)
    \drawvector[points = behind]{1,0}{0,1} % b)
    \drawvector[points = front]{2,0}{0,1} % c)
    \drawvector[points = both]{3,0}{0,1} % d)
\end{tikzpicture}
```

The `\drawvector` syntax is mostly meant for diagrams which don't use z-sorting. Still, they can be used in some 3D still-diagrams; for instance, these vectors connect nicely. They will also eventually

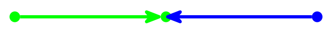
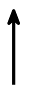
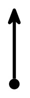
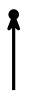


Figure 2.1.1:
The exact location of a geometric vector's tip.


(a) `points = neither`


(b) `points = behind`


(c) `points = front`

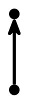

(d) `points = both`

Figure 2.1.2:
The values for the key `points`.

Figure 2.1.3:
Zero-vector

be able to be drawn flat on planes. There is also a zero-vector command available via `\drawpoint`. See Figure 2.1.3, with the following code.

```
\begin{tikzpicture}
    \drawpoint[red]{0,0}
\end{tikzpicture}
```

Chapter 3

The Math Module

When the user appends a parametric object to the segments list, they must specify several attributes for the object. In particular, when they specify the parametric equations for the object, this is done using Lua syntax. Specifically, the user writes Lua code into a tikz key, and that Lua code is later `load()`ed by Lua. A useful feature of the lua function `load()` is that it allows the user to input functions without their usual table prefix; this means that the user can write just `cos()`, instead of `math.cos()`. lua-tikz3dtools defines a growing handful of math operations, from elementary linear matrix transformations to stereographic projections. This list could always be expanded though as needs arise, so to make it robust, I will make a way for the user to make their own. See Chapter 1.

lua-tikz3dtools uses row-vector convention! This means that you put the transformation on the right of the matrix multiplication unlike with column vectors. Additionally, **lua-tikz3dtools uses matrix-vector convention!** This means that vectors are formatted like matrices—meaning that the vector table is nested within a matrix table. This makes it easier for the tools to communicate, and expains why we need to index our functions at depth 2 instead of depth 1.

3.1 The math functions.

In addition to those provided by the math module, these ones are also added.

- `matrix_multiply(A, B)`
- `matrix_scale(factor, A)`
- `reciprocate_by_homogenous(vector)`
- `matrix_add(A, B)`
- `matrix_subtract(A, B)`
- `transpose(A)`
- `inverse(matrix)`
- `det(matrix)`

- `yrotation(angle)`
- `translate(x, y, z)`
- `xscale(scale)`
- `yscale(scale)`
- `zscale(scale)`
- `scale(scale)`
- `xrotation(angle)`
- `zrotation(angle)`
- `euler(alpha, beta, gamma)`
- `sphere(longitude, latitude)`
- `dot_product(u, v)`
- `cross_product(u, v)`
- `norm(u)`
- `normalize(u)`
- `identity_matrix()`
- `stereographic_projection(point)`

Chapter 4

Parametric Objects

4.1 How is a parametric object defined?

Parametric objects are defined using a handful of different pieces of information. In particular, they use parametric functions, domain parameters and a samples parameter for each dimension in the domain. Of course, there are also parameters for color and such as well. Figure 4.1.1 is defined by the following code:

```
\begin{tikzpicture}
    \appendcurve[
        u min = 0
        ,u max = {tau}
        ,u samples = 200
        ,transformation = {euler(pi/2,pi/3,pi/2)}
        ,x = {2*cos(u)}
        ,y = {2*sin(u)}
        ,z = {u}
        ,draw options = {
            draw =purple
            ,ultra thick
            ,line cap = round
        }
    ]
    \appendsurface[
        u min = 0
        ,u max = {tau}
        ,u samples = 36
        ,v min = 0
        ,v max = {tau}
        ,v samples = 36*2/3
        ,transformation = {euler(pi/2,pi/3,pi/2)}
        ,x = {2*cos(u)+sphere(u,v)[1][1]}
        ,y = {2*sin(u)+sphere(u,v)[1][2]}
    ,z = {u+sphere(u,v)[1][3]}
    ,draw options = {
        draw = blue
        ,ultra thin
        ,line join = round
        ,line cap = round
    }
    ,fill options = {
        fill = green
        ,fill opacity = 0.7
    }
]
\foreach \u in {0,...,35} {
    \foreach \v in {0,...,35} {
        \pgfmathsetmacro{\uscale}{2*pi/(36-1)}
        \pgfmathsetmacro{\vscale}{2*pi/(36*2/3-1)}
        \pgfmathsetmacro{\u}{\u*\uscale}
        \pgfmathsetmacro{\v}{\v*\vscale}
        \appendcurve[
            u min = 0
            ,u max = 0.3
            ,u samples = 2
            ,transformation =
                {euler(pi/2,pi/3,pi/2)}
            ,x = {
                2 *
                cos(token.get_macro("u")) +
                (u + 1) *
                sphere(
                    token.get_macro("u")
                    ,token.get_macro("v")
                )[1][1]
            }
            ,y = {
                2 *
                sin(token.get_macro("u")) +
                (u + 1) *
                sphere(
                    token.get_macro("u")
                    ,token.get_macro("v")
                )[1][2]
            }
            ,z = {
                token.get_macro("u") +
                (u + 1) *
```

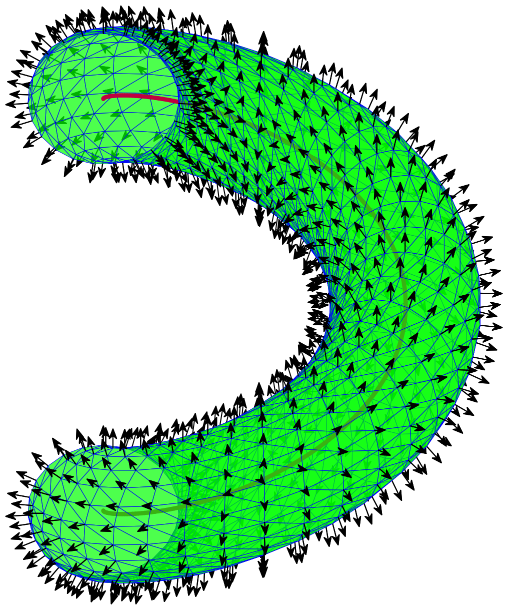


Figure 4.1.1:
A triangulated mesh surface and a curve.

```
                    sphere(
                        token.get_macro("u")
                        ,token.get_macro("v")
                    )[1][3]
                }
                ,draw options = {
                    -{Stealth[round]}
                    ,line cap = round
                    ,draw = black
                }
            ]
        }
    }
    \rendersegments
\end{tikzpicture}
```

lua-tikz3dtools is also capable of handling division by zero without erroring. This is because the math is handled in Lua, where division by zero doesn't error. We clip our results to be well within the scope of the TizZ canvas. The code for Figure 4.1.2 is as follows.

```
\begin{tikzpicture}
    \clip
        (-\textwidth/2,-5)
        rectangle
        (\textwidth/2,5)
    ;
```

```
\pgfmathsetmacro{\angle}{0}
\let\angle\angle
\foreach \longitude in {1,...,36}{
    \let\longitude\longitude
    \appendcurve[
        u min = 0
        ,u max = {pi}
        ,u samples = 4*45
        ,transformation = {euler(pi/2,pi/3,pi/6)}
        ,draw options = {
            line cap = round
            ,draw = black
        }
        ,x = {
            stereographic_projection(
                matrix_multiply(
                    sphere(
                        tau *
                        token.get_macro("longitude") /
                        36
                        ,u
                    )
                    ,euler(
                        0
                        ,pi/2
                        ,(2*pi/24) *
                        token.get_macro("angle") /
                        36
                    )
                )
            )[1][1]
        }
        ,y = {
            stereographic_projection(
                matrix_multiply(
                    sphere(
                        tau *
                        token.get_macro("longitude") /
                        36
                        ,u
                    )
                    ,euler(
                        0
                        ,pi/2
                        ,(2*pi/24) *
                    token.get_macro("angle") /
                    36
                )
            )
        )[1][2]
    }
    ,z = 0*u
]
\appendcurve[
    u min = 0
    ,u max = {pi}
    ,u samples = 23
    ,transformation = {
        matrix_multiply(
            euler(
                0
                ,pi/2
                ,(2*pi/24) *
                token.get_macro("angle") /
                36
            )
            ,euler(pi/2,pi/3,pi/6)
        )
    }
    ,draw options = {
        line cap = round
        ,draw = black
    }
    ,x = {
        sphere(
            tau *
            token.get_macro("longitude") /
            36
            ,u
        )[1][1]
    }
    ,y = {
        sphere(
            tau *
            token.get_macro("longitude") /
            36
            ,u
        )[1][2]
    }
    ,z = {
            sphere(
                tau *
                token.get_macro("longitude") /
                36
                ,u
            )[1][3]
        }
    ]
}
\foreach \latitude in {1,...,18}{
    \let\latitude\latitude
    \appendcurve[
        u min = 0
        ,u max = {2*pi}
        ,u samples = 4*45
        ,transformation = {euler(pi/2,pi/3,pi/6)}
        ,draw options = {
            line cap = round
            ,draw = black
        }
        ,x = {
            stereographic_projection(
                matrix_multiply(
                    sphere(
                        u
                        ,tau *
                        token.get_macro("latitude") /
                        36
                    )
                    ,euler(
                        0
                        ,pi/2
                        ,(tau/24) *
                        token.get_macro("angle") /
                        36
                    )
                )
            )[1][1]
        }
        ,y = {
            stereographic_projection(
                matrix_multiply(
                    sphere(
                        u
                        ,tau *
                        token.get_macro("latitude") /
                        36
                    )
                    ,euler(
                        0
                        ,pi/2
                        ,(tau/24) *
                        token.get_macro("angle") /
                        36
                    )
                )
            )[1][2]
        }
        ,z = 0*u
    ]
    \appendcurve[
        u min = 0
        ,u max = {2*pi}
        ,u samples = 45
        ,transformation = {
            matrix_multiply(
                euler(
                    0
                    ,pi/2
                    ,(2*pi/24) *
                    token.get_macro("angle") /
                    36
                )
                ,euler(pi/2,pi/3,pi/6)
            )
        }
        ,draw options = {
            line cap = round
            ,draw = black
        }
        ,x = {
            sphere(
                u
                ,tau*token.get_macro("latitude")/36
            )[1][1]
        }
        ,y = {
            sphere(
                u
                ,tau*token.get_macro("latitude")/36
```

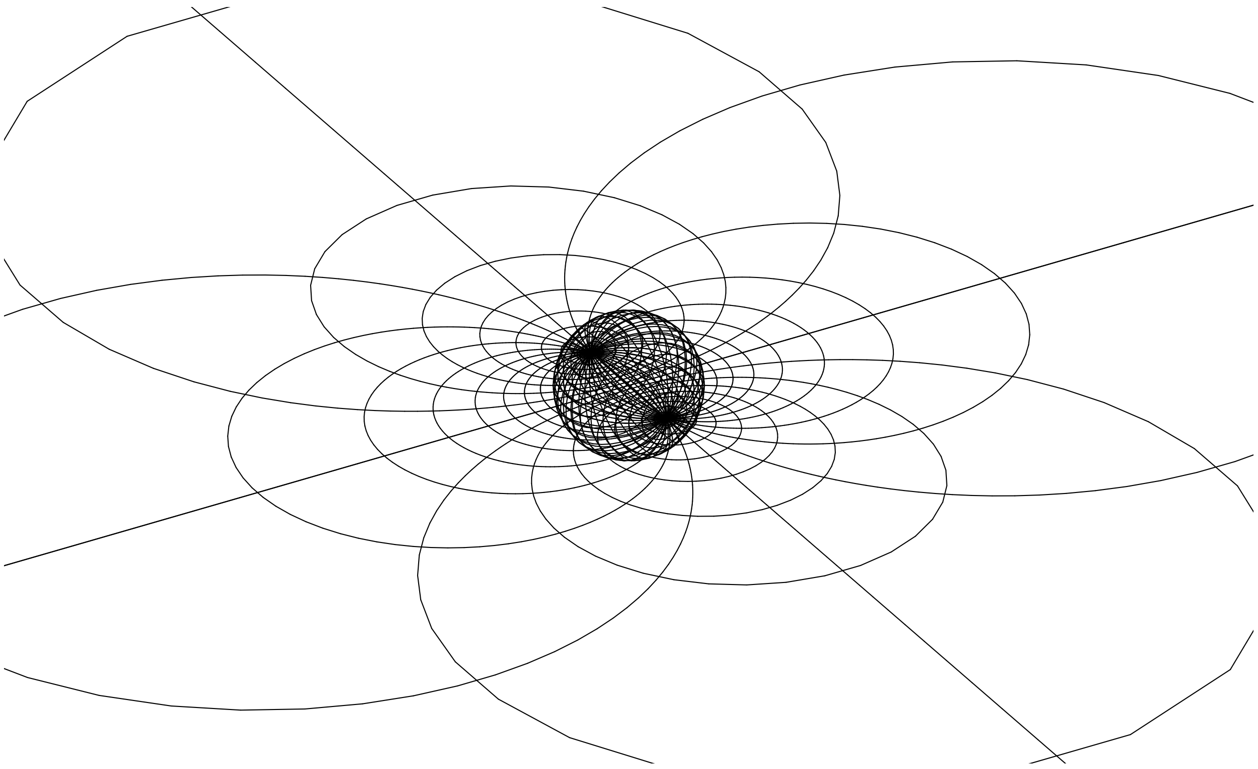


Figure 4.1.2:
Automatic division by zero handling.

```
                )[1][2]
            }
            ,z = {
                sphere(
                    u
                    ,tau*token.get_macro("latitude")/36
                )[1][3]
            }
        ]
    }
    \rendersegments
\end{tikzpicture}
```

lua-tikz3dtools is meant to be compatible with Ti*k*Z, and they are meant to be used together—at least for now. Maybe one day lua-tikz3dtools will have enough capabilities to do things like for loops and conditionals—*at home*. The code for Figure 4.1.3, which illustrates the use of a conditional, is given.

```
\begin{tikzpicture}
    \clip
        (-\textwidth/2,-5)
        rectangle
    (\textwidth/2,5)
;
\foreach[count = \c from 1] \angle in {20,40,...,80} {
    \pgfmathsetmacro{\angle}{\angle*pi/180}
    \let\angle\angle
    \appendsurface[
        u min = pi/6
        ,u max = 0.75*tau
        ,u samples = 36
        ,v min = 0
        ,v max = tau
        ,v samples = 36
        ,transformation = {euler(pi/2,pi/3,pi/6+pi)}
        ,draw options = {
            draw = black
            ,line cap = round
            ,line join = round
        }
        ,fill options = {
            \ifnum\c=1
                fill = red
            \fi
            \ifnum\c=2
                fill = yellow
            \fi
            \ifnum\c=3
                fill = green
            \fi
            \ifnum\c=4
                fill = blue
            \fi
            ,fill opacity = 0.6
        }
        ,x = {
            (1 / cos(token.get_macro("angle"))) *
            cos(u) +
            sqrt(
                (
                    1 /
                    cos(token.get_macro("angle"))
                )^2 - cos(token.get_macro("angle"))
            ) *
            sphere(u,v)[1][1]
        }
        ,y = {
                (1 / cos(token.get_macro("angle"))) *
                sin(u) +
                sqrt(
                    (
                        1 /
                        cos(token.get_macro("angle"))
                    )^2 - cos(token.get_macro("angle"))
                ) *
                sphere(u,v)[1][2]
            }
            ,z = {
                sqrt(
                    (
                        1 /
                        cos(token.get_macro("angle"))
                    )^2 - cos(token.get_macro("angle"))
                ) *
                sphere(u,v)[1][3]
            }
        ]
    }
    \rendersegments
\end{tikzpicture}
```

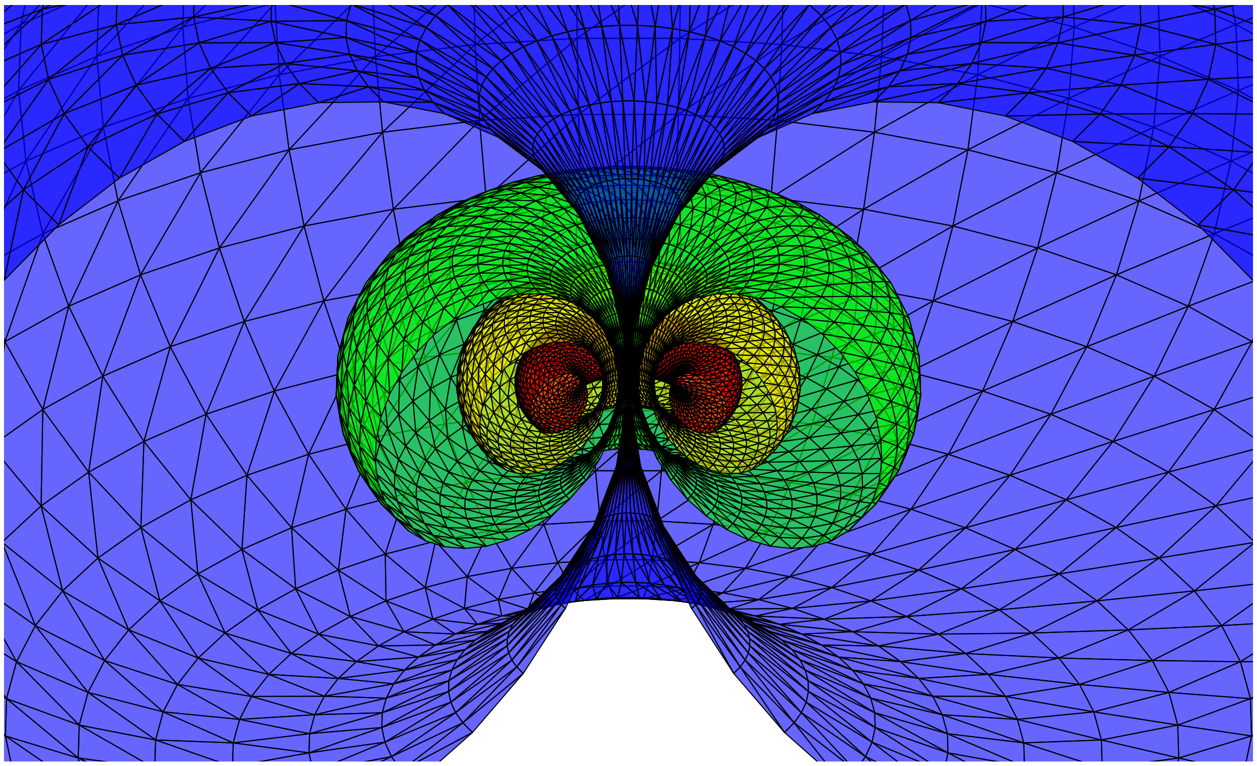


Figure 4.1.3:
Stereographically nested tori.

Chapter 5

Clipped Subspaces

A big focus for the future of this package is triangle-triangle clipping, which generalizes to polygon clipping. This dream is twofold; firstly it will enable parametric surfaces to intersect, and secondly it will enable intersecting plane diagrams which haven't been achieved in full generality in tikz yet.

5.1 Clipping individual planes.

This is where the technology is at currently. We can clip a single plane by a rectangular prism, and draw them. Eventually, when the polygon clipping gets going, this will evolve into a tool for graphing intersecting planes without using approximations. The code for Figure 5.1.1 is given.

```
\begin{tikzpicture}
    \appendprism[%
        a = 1
        ,b = 2
        ,c = 3
        ,d = 4
        ,x min = -1
        ,x max = 1
        ,y min = -1
        ,y max = 1
        ,z min = -1
        ,z max = 1
        ,transformation = {euler(pi/2,pi/3,pi/3)}
    ]
    \appendplane[%
        a = 1
        ,b = 2
        ,c = 3
        ,d = 1
        ,x min = -1
        ,x max = 1
        ,y min = -1
```

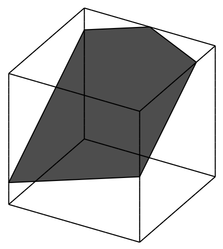


Figure 5.1.1:
A plane which has been clipped by a rectangular prism.

```
        ,y max = 1
        ,z min = -1
        ,z max = 1
        ,transformation = {euler(pi/2,pi/3,pi/3)}
    ]
    \rendersegments
\end{tikzpicture}
```