The lua-tikz3dtools Manual

Version 3.0.0

Jasper Nice

April 12, 2026

The lua-tikz3dtools Manual

Version 3.0.0

To mathematical visualization,
and to the pursuit of making high-quality
mathematical graphics easier to create.

"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 Figures

Preface

This manual is the working companion to the TUGboat article that introduced lua-tikz3dtools [1]. That article explains the motivation for the package and the two core algorithms: a transitive partial order occlusion comparator for affine simplices and a minimal partitioning procedure that resolves tiles capable of partitioning one another. The present document has a more practical aim: it is meant to help a user write scenes, render them, and reason about the resulting figures.

The package is built for a style of work in which the geometry remains visible to the author. One specifies vectors, matrices, parametric maps, filters, and lights. The scene is then partitioned, filtered, sorted, and drawn. This is not a black-box workflow, and it is not meant to be. The hope is that the user can understand not only what the package renders, but why it renders it in that way.

The chapters are arranged in the order in which a new user is most likely to need them. The introduction explains what sort of problem the package solves. The next chapters show how to prepare a document, how to build a scene, and how the individual commands behave. The later material collects practical advice on sampling, filtering, and figure design. An appendix reviews the linear-algebra conventions—homogeneous coordinates, row-vector multiplication, and composition order—that underlie the entire interface.

Acknowledgements

This work was typeset using TEX, the typesetting system created by Donald E. Knuth, together with LuaTEX and the many packages maintained by the TEX community. The TikZ and PGF system, created by Till Tantau and now maintained by Henri Menke and the PGF/TikZ team, provides the drawing layer on which this package rests.

The mathematical ideas behind the occlusion pipeline—a transitive partial order comparator for affine simplices and the minimal partitioning procedure—are my own, and they were first described in my TUGboat article [1]. Everything else benefited from assistance.

Use of artificial intelligence

I want to be straightforward about the role that AI tools played in this project. Large-language-model assistants—principally GitHub Copilot backed by models from OpenAI and Anthropic—were used extensively during development. Their contributions included:

- **Code.** Drafting, reviewing, and refactoring substantial portions of the Lua source code, the LATEX3 style file, and the test suite. The AI was an active pair-programming partner throughout.
- **Documentation.** Drafting and editing the prose of this manual, including chapter organization, wording, and stylistic consistency.
- **Debugging.** Diagnosing errors, suggesting fixes, and explaining unfamiliar corners of the LuaTEX and expl3 ecosystems.

The core algorithms and the mathematical point of view are mine. The implementation and exposition were shaped collaboratively with AI assistance. I see no reason to obscure that fact: these tools made the project substantially more practical to carry out as a single author, and acknowledging their role honestly is, I believe, more useful to readers than pretending otherwise.

Community

The broader TEX and LuaTEX communities deserve thanks for decades of freely shared knowledge. The CTAN maintainers, the authors of the `l3kernel` and `l3packages` bundles, and the many contributors to online forums and documentation all made this package possible in ways that are difficult to enumerate individually.

Chapter 1

Introduction

1.1 What this package is for

Many 3D mathematics illustrations are composed of tessellated parametric objects. Once those objects are projected onto the page, the immediate problem is no longer the 3D object itself but the 2D ordering of the projected tiles. If the tiles are drawn in a naive order, the resulting diagram is often visibly wrong: curves pass through surfaces they ought to go behind, front faces disappear behind back faces, and labels lose their geometric clarity.

The purpose of lua-tikz3dtools is to make that class of illustration practical in LuaLaTeX. The package extends TikZ with a small set of commands for appending points, line segments, triangles, labels, curves, surfaces, solids, and lights to a scene, and then rendering that scene with occlusion handled in a mathematically coherent way. In particular, the package works by reducing the visible geometry to points, line segments, and triangles, partitioning simplices when necessary, and then sorting them before they are drawn. When the resulting occlusion graph contains cycles, the package detects them through Tarjan's algorithm for strongly connected components and resolves them by further partitioning the offending triangles.

The algorithms behind this pipeline—a transitive partial order occlusion comparator for points, line segments, and triangles, and a minimal partitioning procedure that resolves tiles capable of partitioning one another—are described in the companion TUGboat article [1]. The present manual focuses on the user-facing interface rather than the internal mathematics.

1.2 How the package is meant to be used

The basic workflow is straightforward. A scene is built incrementally inside a TikZ picture. One may first define reusable objects—such as matrices or named vectors—with `\setobject`. One may then append visible geometry with commands such as `\appendpoint`, `\appendcurve`, `\appendsurface`,

`\appendtriangle`, and `\appendsolid`. Lights may be added with `\appendlight`, and labels with `\appendlabel`. Once the scene has been assembled, `\displaysimplices` partitions the geometry where needed, applies user filters, sorts the resulting simplices by occlusion, and emits the final TikZ paths.

This means that the order in which objects are appended is not, in general, the order in which they are drawn. The package follows a painter's-algorithm pipeline, but it does so only after placing the geometry into a state where that ordering is meaningful. The user should therefore think of the append commands as describing a scene rather than issuing immediate draw operations.

The expressions supplied to the package are Lua expressions evaluated in a restricted environment. Standard mathematical functions are available, together with the package's `Vector` and `Matrix` types and the constant `tau`. This makes it possible to write parametric objects in a direct algebraic style while keeping the evaluation environment narrow.

1.3 Intended audience

This manual is written for mathematics illustrators who want direct control over how 3D figures are constructed and displayed in TikZ. It is especially well suited to users who prefer to describe geometry algebraically: by giving vectors, matrices, parametric expressions, and predicates, rather than by delegating the whole rendering problem to a black-box graphics engine.

It is also written for readers who want the software to remain pedagogically transparent. A great deal of graphics software can produce acceptable output without showing the user why the output is correct. lua-tikz3dtools takes a different position. It exposes enough of the underlying structure that the user can reason about the geometry, the tessellation, the transformations, the clipping predicates, and the occlusion itself.

1.4 Scope of this manual

This manual documents version 3.0.0 of the package. It covers the command interface exposed by the style file, the expression environment used by the Lua backend, and the practical workflow for building occlusion-aware 3D illustrations in TikZ. Readers who want the algorithmic details should consult the TUGboat article [1]; the present document concentrates on how to use the package rather than how it works internally.

The package is centered on a simplicial pipeline. Parametric curves are sampled into line segments. Parametric surfaces are sampled into triangles. Parametric solids are currently represented through tessellated boundary surfaces. Directional lights contribute to a brightness value used during triangle rendering. Filters may be supplied as small Lua predicates which are evaluated on the resulting simplices.

As in the article, the conceptual frame here is the painter's algorithm rather

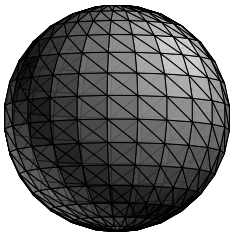

Figure 1.1: A sphere with a helix threading through it. The package handles occlusion between surfaces and curves: segments of the helix that pass behind the sphere are hidden, while those in front are drawn on top. This mixed-type occlusion is one of the package's core capabilities.

than a z-buffer. That choice is deliberate. The package is designed for document preparation, where the user benefits from explicit and deterministic geometry. The discussion that follows therefore concentrates on how to define scenes clearly and how to make the package produce diagrams whose occlusion is both correct and understandable.

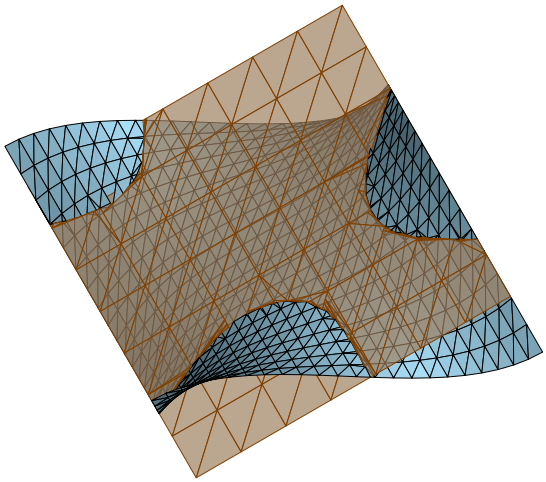

Figure 1.2: A monkey saddle $z = \frac{1}{2}(x^3 - 3xy^2)$ intersected by a tilted plane. The package partitions the simplices of both surfaces where they cross and draws each fragment in the correct depth order. Without partitioning, one surface would simply paint over the other at the intersection.

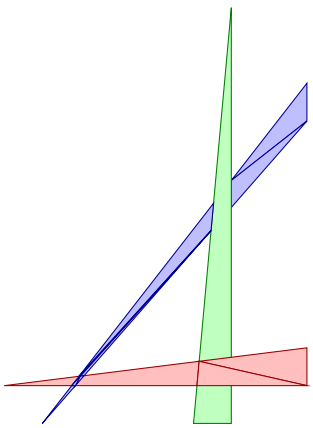

Figure 1.3: Three cyclically occluding triangles.

Chapter 2

Getting started

2.1 Requirements

The package is written for LuaLaTeX. In practical terms, this means that the document must be compiled with `lualatex` and that the package source must be visible to both the TeX and Lua search paths. If the package has been installed into a local or system `texmf` tree, that requirement is already satisfied. If one is working directly from the source tree, the usual approach is to point `TEXINPUTS` and `LUAINPUTS` at the `src` directory before compiling.

The package assumes that the user is comfortable with TikZ and with basic Lua expressions. It does not require one to write standalone Lua files, but it does ask the user to write algebraic expressions inside TikZ keys.

> **Note.** This package will not work under pdfLaTeX or XeLaTeX. The Lua backend is essential to the command interface.

2.2 A first document

The smallest useful way to meet the package is to draw a surface and a curve inside a `tikzpicture`, and then render the scene with `\displaysimplices`. The example below is deliberately modest. It is not meant to show everything; it is meant to establish the rhythm of use.

```
\documentclass{article}
\usepackage{tikz}
\usepackage{lua-tikz3dtools}

\begin{document}
\begin{tikzpicture}
 \appendlight[
  v={return Vector:new{0,0,1,1}}
 ]
```

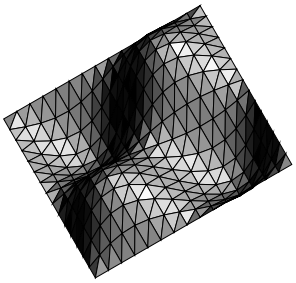

Figure 2.1: A wavy surface with a red curve floating above it. The scene is assembled by appending geometry and then rendered with a single call to `\displaysimplices`.

```
 \appendsurface[
  ustart=0, ustop=1, usamples=16,
  vstart=0, vstop=1, vsamples=16,
  v={return Vector:new{
   u-0.5,
   v-0.5,
   0.25*sin(u*tau)*sin(v*tau),
   1
  }},
  fill options={fill=ltdtbrightness, draw=black, very thin}
 ]
 \appendcurve[
  ustart=0, ustop=1, usamples=24,
  v={return Vector:new{u-0.5,0,0.35,1}},
  draw options={thick, red}
 ]
 \displaysimplices
\end{tikzpicture}
\end{document}
```

The workflow is worth stating plainly. Geometry is appended first. Nothing is drawn at that moment. Only when `\displaysimplices` is called does the package partition the scene where necessary, apply filters, sort the visible pieces, and emit the final TikZ paths.

2.3 How to read the key syntax

Each append command takes TikZ keys in its optional argument. Some of those keys hold plain numbers, such as `ustart=0`, while others hold Lua expressions, such as `v={return Vector:new{u,0,0,1}}`. Because TikZ also uses commas to separate keys, any Lua expression containing commas should be wrapped in

braces. In practice, it is simplest to place nearly all nontrivial Lua expressions inside braces from the start.

There are two recurring defaults throughout the interface. The `transformation` key defaults to `Matrix:identity3()`, and the `filter` key defaults to `return true`. This means that most first examples can ignore both. When the geometry becomes more elaborate, these two keys become central.

One point of notation is mildly unfortunate but harmless once learned. The key named `v` is overloaded. For points, labels, and lights, it means a vector-valued body. For curves, surfaces, and solids, it means the parametric map itself. In every case, the value of `v` is a Lua body that must contain an explicit `return` statement, just like `filter`. This allows multi-statement code when needed. The chapter on the command reference makes this precise for each command.

2.4 What to expect from the first compile

The first successful figure usually answers three immediate questions. First, whether the package is being found at all. Second, whether the Lua expressions are syntactically sound. Third, whether the tessellation density is already high enough to communicate the shape clearly. If the answer to the last question is no, the right response is usually to increase sample counts gradually rather than dramatically.

That final point matters because this package does real geometric work after tessellation. More samples mean more simplices to partition, filter, sort, and draw. The manual therefore encourages a coarse-to-fine workflow: begin with a figure that is merely recognizable, and refine it only once the geometry and occlusion are correct. Chapter 5 returns to this topic in more detail.

Chapter 3

Building a scene

3.1 The scene model

lua-tikz3dtools is easiest to use when one thinks in terms of a scene rather than a sequence of immediate drawing commands. A point, curve, surface, or triangle appended to the scene is not drawn at once. Instead, it is stored as part of a temporary collection of simplices. When `\displaysimplices` is finally called, that collection is processed and emitted.

This distinction is important because it affects how one reasons about order. The order in which objects are appended is not generally the order in which they appear on the page. The package reserves the right to partition and sort the geometry before producing the final paths.

3.2 Named objects and reusable expressions

The command `\setobject` binds a Lua value into the expression environment. In practice, this is the natural way to give a name to a matrix, vector, or reusable geometric object. Once defined, that name becomes available to later expressions in the same document session.

```
\setobject[
 name=spin,
 object={Matrix.zrotation3(pi/6):multiply(
  Matrix.xrotation3(pi/8)
 )}
]

\appendtriangle[
 A={Vector:new{0,0,0,1}},
 B={Vector:new{1,0,0,1}},
 C={Vector:new{0,1,0,1}},
```

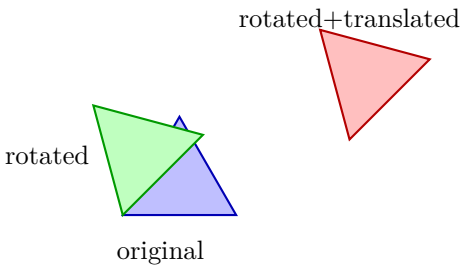

Figure 3.1: Transformation order for a simple object. The blue triangle is in its base position, the green one has been rotated by $\pi/4$ about the z-axis, and the red one has been rotated and then translated. Because the package uses row-vector convention, composed matrices read left to right.

```
 transformation={spin:multiply(Matrix.translate3(0,0,0.5))},
 fill options={fill=ltdtbrightness, draw=black}
]
```

One should not be shy about naming intermediate objects. A manual source file becomes significantly easier to read when rotations, translations, reference points, and clipping vectors are given names rather than repeated as raw expressions.

3.3 Transformations

Transformations are expressed with the package's `Matrix` type. The helpers currently exposed by the package are `Matrix.identity3()`, `Matrix.translate3(dx,dy,dz)`, `Matrix.scale3(sx,sy,sz)`, `Matrix.xrotation3(theta)`, `Matrix.yrotation3(theta)`, `Matrix.zrotation3(theta)`, and `Matrix.zyzrotation3(alpha,beta,gamma)`.

The package uses a row-vector convention. A point is multiplied on the left by its transformation matrix. Consequently, composed transformations read from left to right in the order in which they are applied to the point. If one writes `Matrix.scale3(...):multiply(Matrix.translate3(...))`, the point is first scaled and then translated.

> **Note.** This left-to-right composition rule is one of the first things to verify when a figure appears in the wrong place. Many users come to the package expecting the opposite convention from other graphics systems.

3.4 Filters

Filters are small Lua predicates supplied through the `filter` key. They are evaluated after tessellation but before the occlusion sort. This gives the user a convenient way to discard simplices that fall outside a desired region.

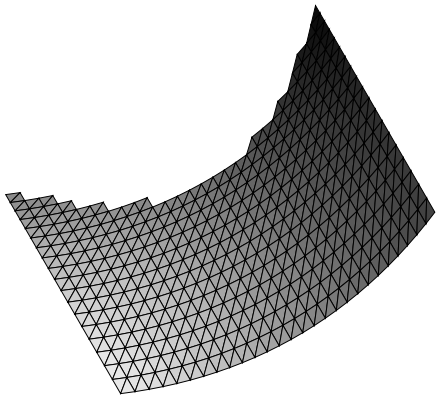

Figure 3.2: A saddle surface with the region below the xy-plane removed by a symbolic filter. The predicate `return A[3] >= 0 and B[3] >= 0 and C[3] >= 0` discards any triangle whose projected vertices have negative z-coordinates. Filtering acts on the tessellated simplices, not on the symbolic surface.

The filter environment depends on the simplex type. Points and labels expose `A`. Line segments expose `A` and `B`. Triangles expose `A`, `B`, and `C`. Each of these is a `Vector` object in homogeneous coordinates. A filter should return `true` for simplices that are to remain and `false` for simplices that are to be discarded.

```
\appendsurface[
 ustart=-1, ustop=1, usamples=24,
 vstart=-1, vstop=1, vsamples=24,
 v={return Vector:new{u,v,0.3*(u^2-v^2),1}},
 fill options={fill=ltdtbrightness, draw=black, very thin},
 filter={
  return A[3] >= 0 and B[3] >= 0 and C[3] >= 0
 }
]
```

This style of predicate is intentionally direct. It keeps the clipping rule in an algebraic form that can be inspected at a glance. In more elaborate scenes, it is often better to define the relevant vectors or matrices once with `\setobject` and then write the filter in terms of those named objects.

Filters become significantly more powerful when they operate in a coordinate system other than world space. A common technique is to define an inverse map that projects each simplex centroid back into the parameter domain of the surface, and then test membership in a region described in that domain. This lets one cut elaborate shapes from a surface even when the corresponding world-space boundary would be impractical to express analytically.

The following example demonstrates this approach on a torus. The object `torusinverse` maps a world-space point back to the torus parameters $(u, v) \in$

Figure 3.3: A torus with an excised region defined in parameter space. The named function `torusinverse` maps the centroid of each triangle back to the torus parameters (u, v); triangles whose image lies within a disc of radius 0.5 around the point $(2, 2)$ are discarded. A second, coarser surface fills the hole with a contrasting colour to make the boundary visible. This technique generalises to any surface for which an inverse parametrization can be written.

$[0, 2\pi)^2$, and the filter discards any triangle whose centroid is too close to a reference point in parameter space. A second surface fills the excised region with a contrasting colour, making the boundary clearly visible.

```
\setobject[
    name = {torusinverse},
    object = {
        function(p)
            local x, y, z = p[1], p[2], p[3]
            local theta = atan2(y, x)
            if theta < 0 then theta = theta + tau end
            local phi = atan2(sqrt(x^2+y^2) - 3, z)
            if phi < 0 then phi = phi + tau end
            return Vector:new{theta, phi, 0, 1}
        end
    }
]
\appendsurface[
    ...,
    filter = {
        torusinverse(
            A:hadd(B):hadd(C):hscale(1/3)
                :multiply(viewinverse)
        ):hdistance(Vector:new{2,2,0,1}) > 0.5
    }
]
```

3.5 Lighting and final rendering

Directional lights are appended with `\appendlight`. During rendering, each triangle's normal is compared to the light directions, and the package defines a TikZ color called `ltdtbrightness` from the resulting average intensity. The falloff is linear in the angle between the triangle normal and the light direction: a triangle facing the light is bright, and one perpendicular to it is dark.

In practical use, this means that surface fill options often take the form `fill=ltdtbrightness` together with a draw style. If no lights are present, the

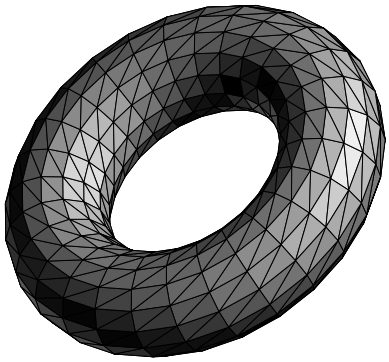

Figure 3.4: A torus illuminated by a single directional light. The colour `ltdtbrightness` is computed per-triangle from the angle between the triangle normal and the light direction. Regions facing the light appear white, while those perpendicular to it approach black.

brightness resolves to black. Lights are cleared after each call to `\displaysimplices`, just as the scene geometry itself is cleared.

Labels are handled differently from other scene elements. They pass through filtering but are emitted after the rest of the geometry. A label should therefore be thought of as an annotation layer rather than as an occluded object.

Chapter 4

Command reference

4.1 General conventions

All append commands take their input as TikZ keys. When a key contains a Lua expression with commas, braces should be used to protect that expression from TikZ's key parser. The keys `transformation` and `filter` recur throughout the interface. Unless stated otherwise, `transformation` defaults to `Matrix:identity3()` and `filter` defaults to `return true`.

The object types exposed to expressions are `Vector` and `Matrix`, together with standard mathematical functions and the constant `tau`. Expressions are evaluated in a restricted environment intended for geometry, not for general system access.

4.2 Scene management commands

`\setobject`

Use `\setobject` to bind a name to an arbitrary Lua value.

`name` The symbol to be introduced into the expression environment.

`object` A Lua expression whose value will be bound to that name.

This command is most useful for storing reusable matrices, vectors, and other objects that would otherwise be repeated across several append commands.

`\displaysimplices`

Use `\displaysimplices` to render the current scene. This command takes no keys. It partitions the geometry where necessary, applies filters, sorts the resulting simplices by occlusion, emits TikZ paths, and then clears both the scene and the current light list.

\luatikztdtoolsset

The helper `\luatikztdtoolsset` is a thin wrapper around `\tikzset` with the package family selected in advance. Most users will not need it in day-to-day work, but it remains available for defining package-local styles or experimenting with key defaults.

4.3 Point, label, and light commands

\appendpoint

Append a single point to the scene.

`v` Required. A Lua body producing a `Vector`.

`fill options` Optional. TikZ options used when drawing the point.

`transformation` Optional. A matrix applied to the point.

`filter` Optional. A predicate evaluated on the point simplex.

Internally the point is rendered as a small circle of fixed radius. If a scene demands points of visibly different sizes, a custom construction using curves or surfaces may be the better choice.

\appendlabel

Append a label anchored at a point.

`v` Required. A Lua body producing a `Vector`.

`text` Required. The label contents.

`transformation` Optional. A matrix applied to the anchor point.

`filter` Optional. A predicate evaluated on the label anchor.

Labels are emitted after the geometry rather than participating in occlusion. This makes them suitable for annotation, but not for physically hidden text.

\appendlight

Append a directional light vector.

`v` Required. A Lua body returning a `Vector` giving the light direction.

Lights affect triangles, including triangles generated from surfaces, solids, and curve arrowheads. The accumulated light state is cleared after each `\displaysimplices`.

4.4 Curve, surface, and solid commands

\appendcurve

Append a parametric curve, sampled into line segments.

ustart Optional. Default 0.

ustop Optional. Default 1.

usamples Optional. Default 10. Must be at least 2.

v Required. A Lua body in u returning a Vector.

transformation Optional. A matrix applied after evaluation.

draw options Optional. TikZ options for the line segments.

arrow tip Optional. Fill options for a generated arrowhead.

arrow tail Optional. Fill options for a generated tail marker.

filter Optional. A predicate evaluated on each segment.

The arrow tip and arrow tail keys do not invoke TikZ's native arrow-tip machinery. Instead, the package generates small parametric surfaces attached to the endpoints of the sampled curve.

\appendsurface

Append a parametric surface, sampled into triangles.

ustart Optional. Default 0.

ustop Optional. Default 1.

usamples Optional. Default 10. Must be at least 2.

vstart Optional. Default 0.

vstop Optional. Default 1.

vsamples Optional. Default 10. Must be at least 2.

v Required. A Lua body in u and v returning a Vector.

transformation Optional. A matrix applied after evaluation.

fill options Optional. TikZ options for the triangles.

filter Optional. A predicate evaluated on each triangle.

Each rectangular parameter cell is divided into two triangles. Degenerate cases are skipped when the sampled vertices collapse together.

\appendsolid

Append a parametric solid by tessellating the boundary of a three-parameter domain.

ustart, ustop, usamples Optional. Defaults `0`, `1`, `10`.

vstart, vstop, vsamples Optional. Defaults `0`, `1`, `10`.

wstart, wstop, wsamples Optional. Defaults `0`, `1`, `10`.

v Required. A Lua body in `u`, `v`, and `w` returning a `Vector`.

transformation Optional. A matrix applied after evaluation.

fill options Optional. TikZ options for the triangles.

filter Optional. A predicate evaluated on each triangle.

All three sample counts must be at least 2. At present the command tessellates the six outer faces of the parameter box rather than constructing a volumetric interior mesh.

4.5 Explicit simplex command

\appendtriangle

Append an explicit triangle.

A Required. A Lua expression producing the first vertex.

B Required. A Lua expression producing the second vertex.

C Required. A Lua expression producing the third vertex.

transformation Optional. A matrix applied to the triangle.

fill options Optional. TikZ options for the triangle.

filter Optional. A predicate evaluated on the triangle.

This command is useful when the desired object is already simplicial, or when one wants exact control over a small piece of visible geometry without passing through a parametric sampling stage.

Chapter 5

Practical guidance

5.1 Choosing sample counts

The most common practical adjustment in this package is the choice of sample counts. A curve with `usamples=8` may be sufficient to establish the shape of an arc, while a surface that is meant to carry lighting cleanly may need a denser tessellation. There is no universal rule, but there is a useful discipline: start with the coarsest tessellation that communicates the shape, then increase counts only where the figure demands it.

That advice matters because more samples mean more simplices, and more simplices mean more work for partitioning, filtering, and occlusion sorting. Solids are especially expensive because three sample directions affect the boundary tessellation.

5.2 Writing source that remains readable

The package rewards a style of source code in which repeated objects are named, transformations are factored out, and filters are short enough to be read as mathematical statements. In practical terms, this usually means using `\setobject` for recurring matrices or vectors, keeping draw and fill styles consistent within a figure, and separating logically distinct scene elements into their own append commands.

It is also wise to keep the final call to `\displaysimplices` visibly separate from the scene construction above it. That single line is the moment at which the scene is committed to the page.

5.3 Troubleshooting a figure

When a figure behaves unexpectedly, four checks usually resolve the issue.

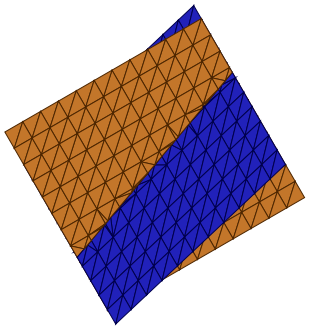

Figure 5.1: Two intersecting tilted planes rendered with occlusion-aware sorting. The package partitions the simplices where the surfaces cross and draws them in the correct depth order. Without the pipeline, one surface would simply paint over the other.

Bracing Make sure that Lua expressions containing commas are wrapped in braces.

Sample counts Check that `usamples`, `vsamples`, and `wsamples` are at least 2 where required.

Transformation order Re-read composed matrices from left to right in the order applied to points. See Appendix A for the composition convention.

Filter semantics Remember that filters act on tessellated simplices, not on symbolic objects.

Two further behaviors are worth remembering because they can initially look like errors. Labels are drawn after the geometry, and lights are cleared after each display call. Both behaviors are intentional.

5.4 Illustration program for the manual

Since this document is intended to become an illustrated manual, the figures should do more than decorate the text. Each figure ought to settle a specific question in the reader's mind. The most useful illustrations are therefore not merely attractive scenes, but comparisons and decompositions: before-and-after occlusion, coarse versus fine tessellation, filtered versus unfiltered geometry, and the effect of changing the transformation order.

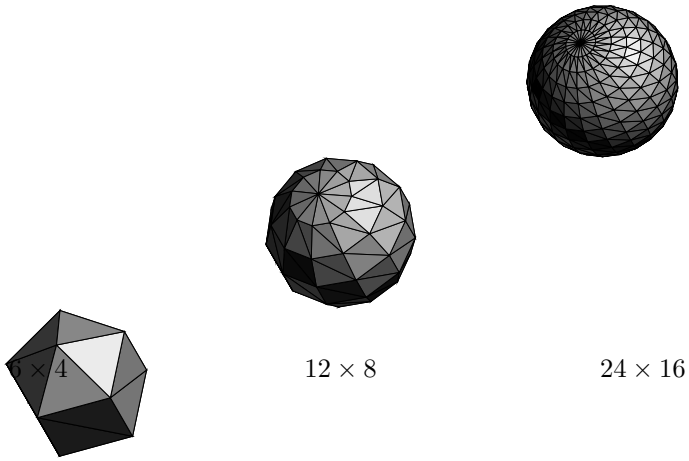

Figure 5.2: The same sphere at three sampling densities. At 6×4 the shape is barely recognizable; at 12×8 the sphere is clear but faceted; at 24×16 the surface appears smooth. Higher sample counts produce more simplices for partitioning and sorting, so the tradeoff is visual fidelity against compile time.

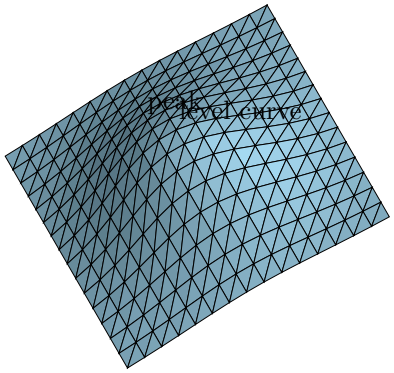

Figure 5.3: A Gaussian bump with a labelled peak and a dashed level curve. Labels are emitted after the geometry and therefore appear as an annotation layer on top of the scene.

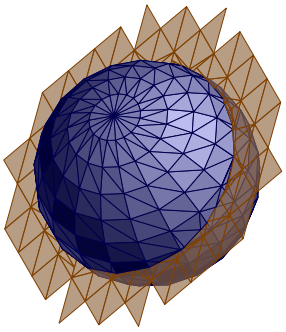

Figure 5.4: A sphere intersected by a tilted disc. The package partitions both the sphere and the disc where they cross, producing fragments that are drawn in the correct depth order. The disc's filter clips it to a circular region, demonstrating how filtering and partitioning cooperate in a single scene.

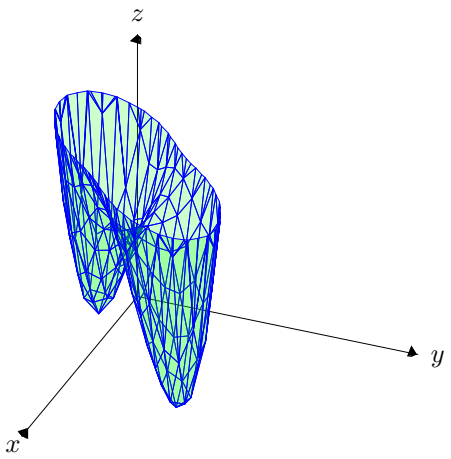

Figure 5.5: The surface $z = u^4 + v^4 - 4uv + 1$ rendered inside a bounding box with labelled coordinate axes. The filter clips the surface to the box by transforming each triangle's vertices back to the local frame and testing all three coordinates against the box limits. Arrow-tipped curves serve as axes, and labels provide annotation. This style of figure—a surface, a bounding volume, axes, and labels—is a common pattern for mathematics illustrations.

Appendix A

Linear algebra reference

A.1 Homogeneous coordinates

The package works with homogeneous 4-vectors for 3D affine geometry. In the examples throughout the manual, a point in ordinary 3-space therefore appears as a vector such as `Vector:new{x,y,z,1}`. This convention allows translations, rotations, and scalings to be handled uniformly by matrix multiplication.

For most user-facing work, the final coordinate should simply be taken as `1`. The manual does not require one to manipulate the homogeneous coordinate directly unless a specialized construction calls for it.

A.2 The `Vector` type

User expressions have access to the validated constructor `Vector:new{...}`. The type supports the operations needed by the package internally and by user-written filters and helper expressions. Among the most useful are homogeneous subtraction `:hsub(...)`, homogeneous inner product `:hinner(...)`, homogeneous normalization `:hnormalize()`, and ordinary matrix multiplication through `:multiply(...)`.

The unchecked constructor `Vector:_new{...}` also exists, but it is meant for internal hot paths. In manual examples, the validated constructor is the better default.

A.3 The `Matrix` type

User expressions likewise have access to `Matrix:new{...}` for explicit matrices and to a number of 3D helper constructors:

`Matrix.identity3()` The identity transformation.

`Matrix.translate3(dx,dy,dz)` Translation in 3D.

`Matrix.scale3(sx,sy,sz)` Axis-aligned scaling.

`Matrix.xrotation3(theta)` Rotation about the x-axis.

`Matrix.yrotation3(theta)` Rotation about the y-axis.

`Matrix.zrotation3(theta)` Rotation about the z-axis.

`Matrix.zyzrotation3(alpha,beta,gamma)` A ZYZ Euler rotation.

These helpers cover most document-scale figures. When a custom matrix is more natural, `Matrix:new{...}` remains available.

A.4 Composition order

The transformation pipeline uses row vectors. A point is multiplied on the left by its transformation matrix. This means that if a user writes `A:multiply(B)` as a transformation, the effect on a point is first the transformation `A` and then the transformation `B`.

This convention is simple once internalized, but it differs from the column-vector convention used by many other texts and libraries. When a figure seems to move in the wrong direction or to rotate about the wrong place, the first question should usually be whether the composed matrices have been written in the intended order.

References

[1] Jasper Nice. "The lua-tikz3dtools package for 3D illustrations in LaTeX". In: *TUGboat* 46.3 (2025), pp. 358–364. DOI: 10.47397/tb/46-3/tb144nice-lua3dtools.