lua-tikz3dtools Manual

Jasper Nice

Version 3.1.0
April 28, 2026

Abstract

lua-tikz3dtools is a LuaLaTeX package for building three-dimensional TikZ illustrations from points, line segments, triangles, tessellated parametric surfaces, and sampled solids. Its distinguishing feature is that it does not ask the user to manually sort visible pieces: the package partitions intersecting geometry where needed, applies user filters, performs occlusion sorting, and then emits ordinary TikZ paths. This manual documents the public interface, the Lua-expression model used by the keys, the transformation conventions, and the practical limits of the rendering pipeline.

Contents

1 Overview

lua-tikz3dtools is a LuaLaTeX package for constructing three-dimensional figures inside ordinary TikZ pictures. The package is aimed at mathematical and technical drawings where visibility matters: curves should disappear behind surfaces, intersecting triangles should be split when necessary, and the author should be able to describe geometry in coordinates instead of manually ordering every path.

The package sources and the evolving example collection are maintained at `https://github.com/Pseudonym321/TikZ-Animations/tree/master1/TikZ/lua-tikz3dtools`.

The public interface is intentionally small. In practice you define reusable Lua objects with `\setobject`, append geometry with commands such as `\appendtriangle` or `\appendsurface`, optionally add lights and filters, and finish the picture with `\displaysimplices`. Everything before that final command only builds an internal scene.

The checked-in gallery is intentionally larger than the minimal tutorial path. The second spread leans harder into showpiece scenes so the manual demonstrates both routine usage and the more decorative end of the package.

1.1 What the package does well

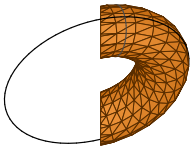
The package is strongest when the final picture can be reduced to triangles and line segments after sampling. This includes:

- individual triangles;
- sampled parametric curves;
- tessellated parametric surfaces;
- sampled solid boundaries;
- labels placed by three-dimensional coordinates;
- layered scenes where visibility must be computed automatically.

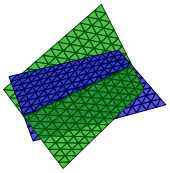
The package also provides a small but useful mathematical layer based on homogeneous vectors and matrices. Those objects are available directly from the Lua snippets embedded in the TikZ keys, so the same package can serve as both a scene description system and a lightweight geometry toolkit.

1.2 Public commands at a glance

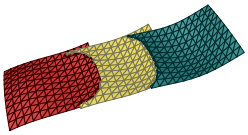
The checked-in style file currently exports the following commands:



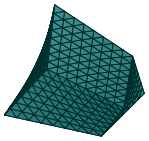
Filtered torus cutaway. An invisible slicing surface forces partitioning along the cut boundary before a screen-space filter removes one side of the torus, and two sampled highlight curves make the cut readable after the visibility pass.



Intersecting sheets. Two sampled surfaces intersect in space, so the renderer must partition and sort triangle pieces instead of relying on a single depth statistic.



A small surface family. Related graphs can be generated from the same parameter scaffold with different phase or amplitude choices, then displayed together in one scene.



Sampled solid boundary. A three-parameter map is rendered by tessellating the six faces of the parameter box, which is usually the right abstraction for illustrative solid boundaries.

Figure 1: Representative outputs. The gallery mixes shading, filter-driven cuts, intersecting surfaces, a small family of related surfaces, and a sampled solid boundary.

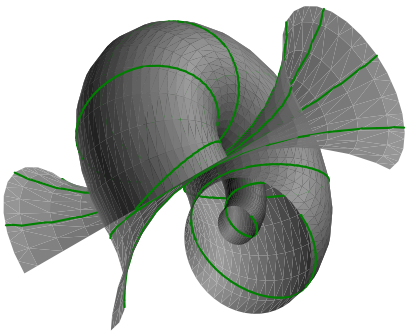


Figure 2: A still frame from a larger animated family. This manual version fixes the configuration at a 15° offset, keeps the shared view, and preserves the seven-branch green parameter-space curve family on the three linked surfaces.

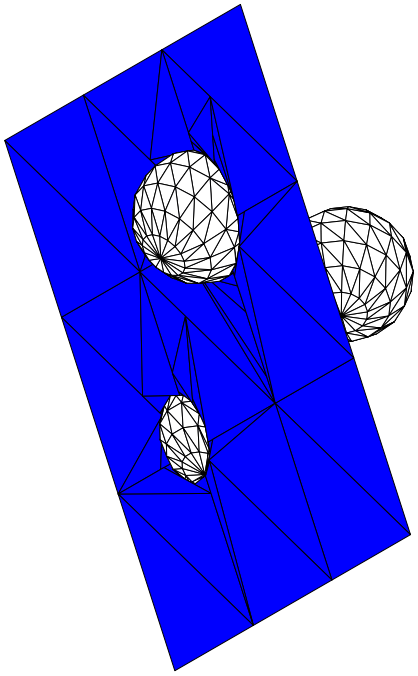


Figure 3: Three rotated spherical shells gather above a slanted plane, with a shared rigid transform and per-shell spin producing the interlocked stack.

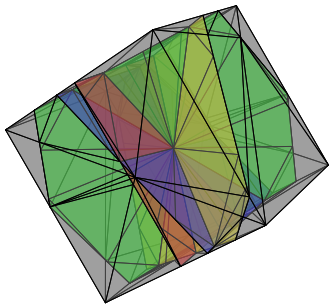


Figure 4: Four clipped planes rotate through a translucent cube, with the inverse transform used as a geometric filter to keep only the portions that pass through the shared box.

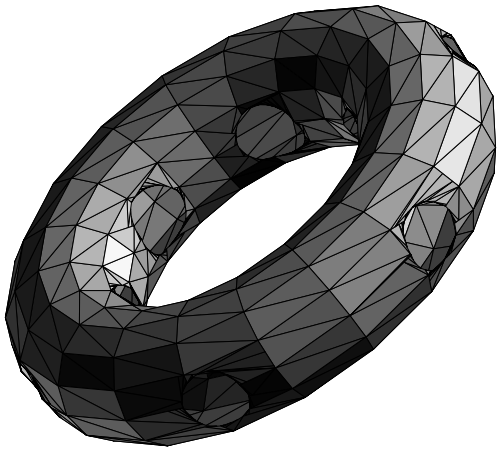


Figure 5: A torus threads through six hidden sleeves, with a single sampled view from a larger orbital animation revealing the ring of circular exclusions.

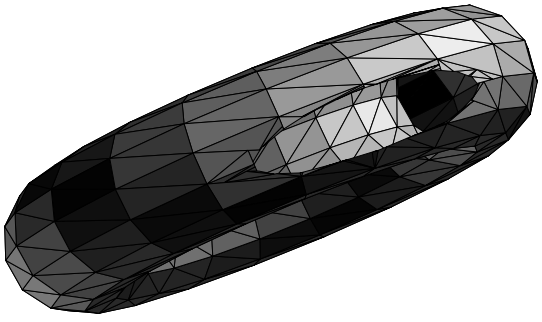


Figure 6: A torus is cut by a gray traced surface, while the visible mesh keeps only the region whose inverse-parameter image stays outside a circular hole in torus coordinates.

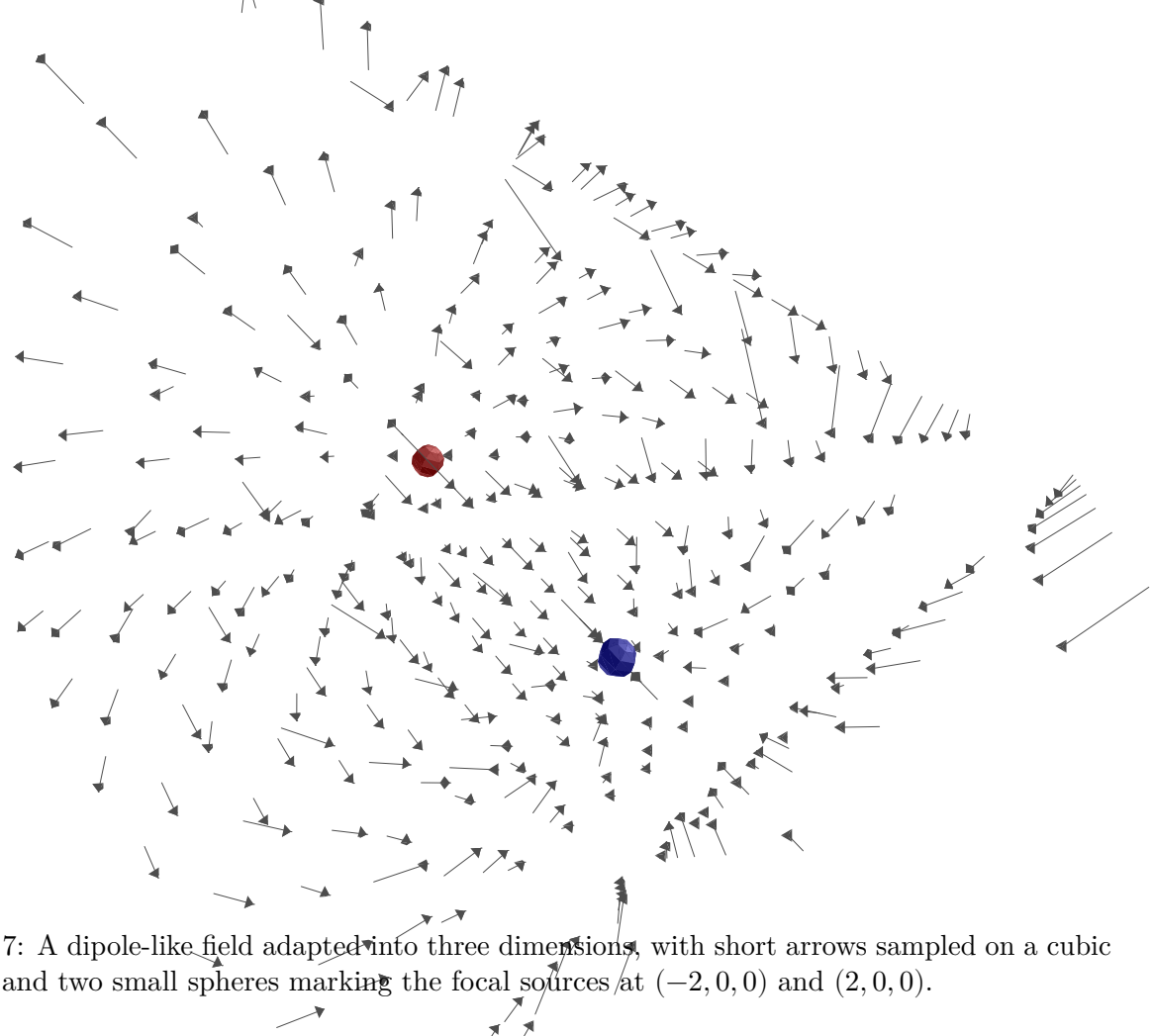


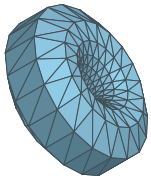
Figure 7: A dipole-like field adapted into three dimensions, with short arrows sampled on a cubic lattice and two small spheres marking the focal sources at $(-2, 0, 0)$ and $(2, 0, 0)$.

- `\luatikztdtoolsset`, a convenience wrapper around the package key family;
- `\setobject` for reusable Lua values;
- `\appendlabel` for labels placed by a three-dimensional point;
- `\appendlight` for directional lights;
- `\appendtriangle` for a single triangle;
- `\appendcurve` for a sampled parametric curve;
- `\appendsurface` for a sampled parametric surface;
- `\appendsolid` for a sampled solid boundary;
- `\displaysimplices` for partitioning, sorting, and rendering the scene.

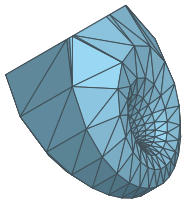
An internal point primitive exists on the Lua side, but the public `\appendpoint` macro is commented out in the present style file. For that reason this manual treats point-like markers as a design pattern rather than as an exported command; Section 6 discusses practical replacements.

1.3 How this manual is organized

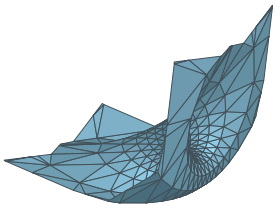
The first half of the manual is workflow-driven. It starts with a minimal document, explains the scene model, and then introduces the Lua-expression conventions and the transformation layer. The second half is reference-oriented: it records command keys, rendering behavior, limitations,



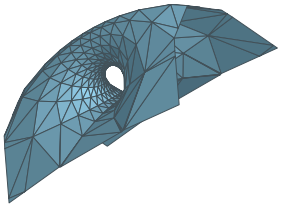
Frame 0.0



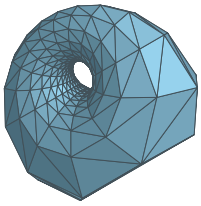
Frame 0.9



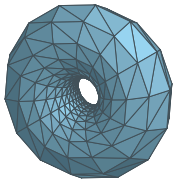
Frame 1.8



Frame 2.7



Frame 3.6



Frame 4.5

Figure 8: Six selected frames from a Clifford-surface sequence, arranged in reading order as a two-column page figure. Each panel keeps the same clipped view box while the phase parameter advances through the chosen samples.

and the low-level Lua API for readers who want to extend the package or reason about its internals.

If you are reading this package for the first time, the best path is Section 2, then Section 3, then Section 4, and only afterward the command reference in Section 9.

2 Getting Started

2.1 Requirements

The package is meant for LuaLaTeX. A typical document loads TikZ and then loads `lua-tikz3dtools` in the preamble. The runtime depends on the Lua module files as well as the style file, so both the TeX loader and the Lua module loader must be able to find the package installation.

For normal use, install the package into a TeX tree. For local development from the repository checkout, ensure that the `src/` directory is visible on both the TeX input path and the Lua input path before compiling the manual examples.

2.2 A minimal document

The following example shows the smallest complete workflow that still exercises the package properly: a light, a triangle, a label, and a final display call.

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

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

  \appendtriangle[
    m = {
      return Matrix:new{
        {0, 0, 0, 1},
        {2, 0, 0, 1},
        {0.5, 1.25, 1.0, 1}
      }
    },
    fill options = {fill=ltdtbrightness, draw=black}
  ]

  \appendlabel[
    v = {return Vector:new{0.8, 0.35, 0.35, 1}},
    text = {$T$}
  ]

  \displaysimplices
\end{tikzpicture}
\end{document}
```

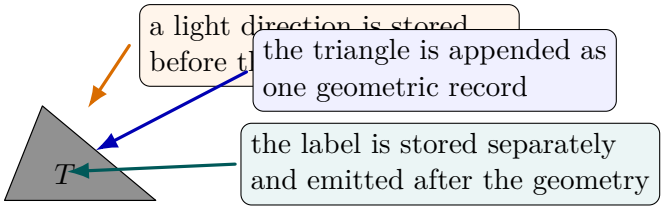


Figure 9: A first rendered scene. Even this tiny example already exhibits the package's central pattern: append geometry and labels in any order, then render everything at once with `\displaysimplices`.

There are three practical points hidden in this minimal example.

First, every geometric command appends data to the scene; nothing is drawn until `\displaysimplices` is executed. Second, lighting only affects the rendered triangle if the triangle's fill style actually refers to the dynamically defined color `ltdtbrightness`. Third, the coordinates are three-dimensional even though the final output is a two-dimensional TikZ path.

2.3 The basic workflow

Most figures built with lua-tikz3dtools follow the same order:

1. open a `tikzpicture`;
2. optionally define reusable Lua values with `\setobject`;
3. append geometry and labels in any convenient order;
4. optionally append lights;
5. call `\displaysimplices` exactly where the scene should be emitted.

The append order is not the final render order. The package recomputes that order from geometry. This is the main distinction between lua-tikz3dtools and the manual "draw the farthest object first" approach.

2.4 A slightly richer first scene

The next example shows two useful habits from the start: keep the transformation separate from the geometric definition, and use `\setobject` when the same value appears more than once.

```
\begin{tikzpicture}
  \setobject[
    name = tilt,
    object = {
      return Matrix.axis_angle(Vector:new{1, 0, 0, 1}, 0.9)
        :multiply(Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.7))
    }
  ]

  \appendlight[
    v = {return Vector:new{0.4, -0.3, 1, 1}}
  ]
```

```
  \appendtriangle[
    m = {
      return Matrix:new{
        {-1, -1, 0, 1},
        { 1, -1, 0, 1},
        { 0,  1, 0, 1}
      }
    },
    transformation = {return tilt},
    fill options = {fill=ltdtbrightness, draw=black}
  ]

  \appendlabel[
    v = {return Vector:new{0, 0.2, 0.1, 1}},
    transformation = {return tilt},
    text = {$\Sigma$}
  ]

  \displaysimplices
\end{tikzpicture}
```

This scene is already structured the same way as larger ones: a few named values, local command keys, and a single display call. Once that pattern is comfortable, the rest of the manual is mostly about richer sources of geometry.

3 The Scene Model

The package internally stores a scene as a list of simplices and labels. In this manual the word *simplex* is used in the package's broad internal sense: a point-like entry, a line segment, or a triangle. User-facing commands create those records indirectly.

3.1 What each append command contributes

- `\appendtriangle` contributes a single triangle;
- `\appendcurve` samples a function of one parameter and contributes line segments;
- `\appendsurface` samples a function of two parameters and contributes triangles;
- `\appendsolid` samples a function of three parameters and contributes triangles on the six boundary faces;
- `\appendlabel` contributes a label record that is rendered last;
- `\appendlight` contributes a light direction used when triangles are drawn.

All geometric data are stored in three dimensions before projection. The final TikZ paths are emitted only after the renderer has decided how overlapping pieces must be partitioned and in what order they should appear.

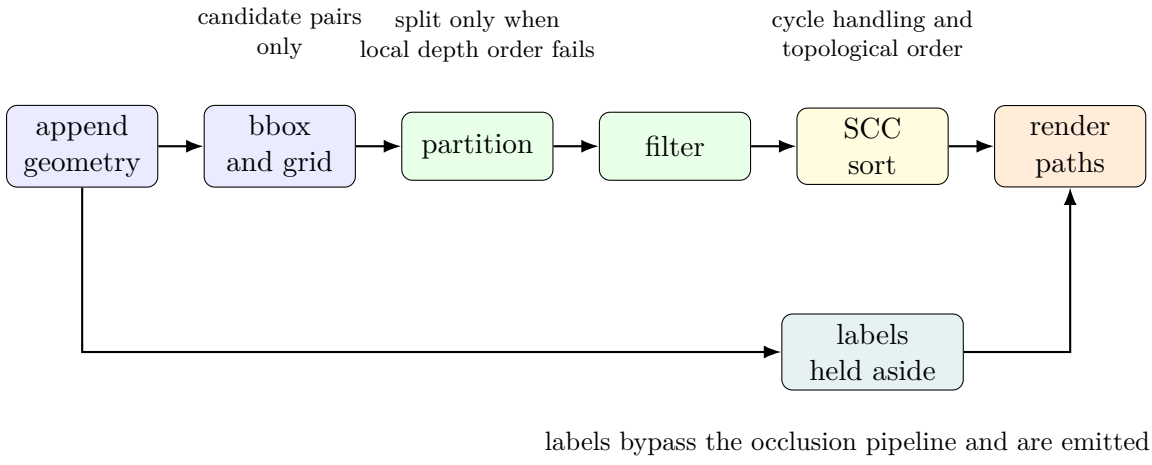


Figure 10: The scene model. Geometry is appended first and processed only at display time; labels are stored with the scene but bypass most of the visibility machinery before being emitted last.

3.2 Why `\displaysimplices` is the real render step

The command `\displaysimplices` performs more than one task.

1. It computes two-dimensional bounding boxes for non-label geometry.
2. It partitions geometry by overlapping parent pieces when an exact split is needed.
3. It applies user filters to the resulting pieces.
4. It computes an occlusion order.
5. It emits TikZ paths for segments and triangles and TikZ nodes for labels.
6. It clears the accumulated scene so the next display starts fresh.

Because the scene is cleared at the end of rendering, a typical document builds one scene, displays it, and then starts a new scene if another independent figure is needed later.

3.3 Labels are intentionally different

Labels are stored with the scene, but they are not treated like occluding surfaces. They are rendered after all geometric simplices have been emitted. This is a deliberate design choice: labels should remain legible, and the author typically wants to manage label visibility manually rather than through geometric clipping.

The consequence is simple but important. If a label should disappear when it is behind a surface, that behavior must be achieved through a filter or through a separate labeling strategy. It does not happen automatically.

3.4 The scene is geometric, not painterly

The package does not merely sort previously drawn objects by a single depth value. If two triangles intersect in projection, the renderer can partition one or both of them and sort the

resulting pieces separately. This is why figures made with lua-tikz3dtools remain usable even after the obvious "sort by centroid" or "draw in input order" methods fail.

At the same time, the package is still based on sampled geometry. Parametric curves, surfaces, and solids are discretized first. The visibility logic works on those discretized pieces, not on the continuous mathematical object that the user had in mind before sampling.

4 Lua Expressions and the Sandbox

Most package keys do not take plain numbers. They take Lua source code that is evaluated when the command runs. The source is evaluated in a read-only sandbox that exposes the mathematical objects needed for scene construction while blocking the usual dynamic-loading interfaces.

4.1 Names available in the sandbox

The following names are the ones most authors will use directly:

- `Vector` and `Matrix` for geometric construction;
- `math`, `string`, `table`, and related standard tables;
- `tau` as shorthand for 2π;
- any object previously defined with `\setobject`.

The sandbox is read-only. User snippets cannot rebind globals and cannot load new modules through `require`, `load`, `loadfile`, `dofile`, `package`, or `debug`. Those names are blocked explicitly.

In practice it is best to brace any nontrivial key value, especially if the Lua code spans multiple lines or contains commas that are not already protected by a nested table literal. The examples below follow that convention.

4.2 Expression keys versus block keys

The package uses three related conventions.

Single-expression keys. Some keys behave like expressions and may omit `return`. A transformation key such as `transformation = Matrix.identity()` is accepted because the package wraps the string in an implicit `return`.

Block keys. Some keys are evaluated as full Lua chunks. Those keys should usually begin with `return` when the intent is to produce a value. The `curve` key on `\appendsurface` is an example: it expects a chunk that returns a Lua table describing parameter-space segments.

Function-body keys. The key `v` on `\appendcurve`, `\appendsurface`, and `\appendsolid` is interpreted as the body of a Lua function in one, two, or three parameters, respectively. In other words, the package builds `function(u) ... end`, `function(u,v) ... end`, or `function(u,v,w) ... end` for you.

4.3 Typical return types

The most common return types are:

- a `Vector:new{...}` for a point, direction, light, or parameter triple;
- a `Matrix:new{...}` for a triangle or transformation matrix;
- a Boolean expression in a `filter` body;
- a Lua table returned by `curve` on a surface.

The package rejects invalid or non-finite values. If a projection produces a NaN, an infinity, or a degenerate simplex, the corresponding piece is not added to the scene.

4.4 Examples of each style

```
% Single-expression style.
transformation = {Matrix.identity()}

% Explicit block style.
object = {
  return Matrix.translate(Vector:new{0, 0, 2, 1})
    :multiply(Matrix.axis_angle(Vector:new{0, 1, 0, 1}, 0.5))
}

% Function-body style for a curve.
v = {return Vector:new{math.cos(u), math.sin(u), 0.3*u, 1}}

% Filter body style.
filter = {
  if A[3] < 0 then
    return false
  end
  return true
}
```

4.5 Reusable objects

The command `\setobject` evaluates one Lua chunk and stores the result under a chosen name. Every later snippet may refer to that name directly. This is the most convenient way to share a transformation matrix, a point, a scalar, or even a custom Lua function between multiple append commands.

```
\setobject[
  name = frame,
  object = {
    return Matrix.axis_angle(Vector:new{1, 0, 0, 1}, 0.8)
      :multiply(Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.4))
  }
]

\appendtriangle[
  m = {
```

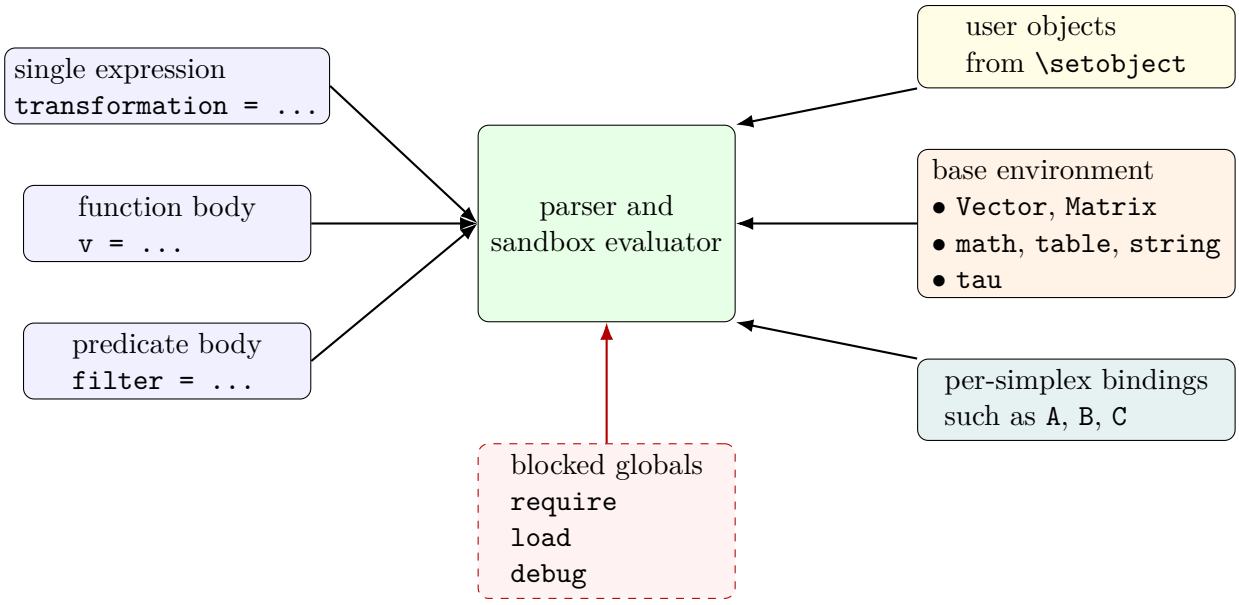


Figure 11: The sandboxed evaluation model. Each key body is interpreted against a read-only base environment, the user object table, and any local bindings created for filters. The blocked names highlight what the sandbox intentionally refuses to expose.

```
        return Matrix:new{
          {-1, -1, 0, 1},
          { 1, -1, 0, 1},
          { 0,  1, 0, 1}
        }
      },
  transformation = {return frame},
  fill options = {fill=ltdtbrightness, draw=black}
]
```

Reserved base-environment names cannot be rebound through `\setobject`. In practice that means you should choose descriptive application names and avoid trying to redefine `Vector`, `Matrix`, `math`, and similar globals.

5 Coordinates and Transformations

lua-tikz3dtools uses homogeneous coordinates throughout. A point in space is typically represented as a row vector `Vector:new{x, y, z, 1}`. Transformations are represented by 4×4 matrices acting on the right.

5.1 Homogeneous points and directions

For day-to-day use, treat the final component as the homogeneous coordinate and write points with last component equal to 1. The package automatically performs the projective divide after applying a transformation to a point or to a sampled simplex.

The most common constructor forms are:

```
Vector:new{1.2, -0.4, 0.8, 1}
Matrix.identity()
Matrix.translate(Vector:new{0, 0, 2, 1})
```

5.2 Row-vector convention

This package uses a row-vector convention. If a point p is transformed by a matrix M, the package computes pM, not Mp. As a consequence, composition order is read from left to right:

$$p(AB) = (pA)B.$$

So if you want to rotate first and then translate, you should write the transformation as

```
Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.8)
  :multiply(Matrix.translate(Vector:new{2, 0, 0, 1}))
```

This is the opposite of the column-vector convention used in many graphics texts. It is worth keeping this straight early, because most apparent transformation bugs in user code are really order-of-composition mistakes.

5.3 Built-in transformation constructors

The matrix layer provides the following constructors for normal use:

- `Matrix.identity()`;
- `Matrix.translate(Vector:new{dx,dy,dz,1})`;
- `Matrix.scale_axis(Vector:new{sx,sy,sz,1})`;
- `Matrix.axis_angle(axis, theta)`;
- `Matrix.zyzrotation(Vector:new{alpha,beta,gamma})`;
- `Matrix.shear(kxy, kxz, kyx, kyz, kzx, kzy)`;
- `Matrix.reflect_axis(axis)`;
- `Matrix.perspective(Vector:new{px,py,pz,1})`;
- `Matrix.transform_about(point, transformation)`.

The older helper names `xrotation3`, `yrotation3`, `zrotation3`, `translate3`, `scale3`, and `zyzrotation3` are retained for compatibility, but the vector-valued constructors are clearer because they keep the entire transformation family in one style.

5.4 Transforming about a fixed point

It is often easier to build a local motion around a distinguished point than to write the full translation–rotation–translation sequence yourself. The helper `Matrix.transform_about(point, transformatio` does exactly that.

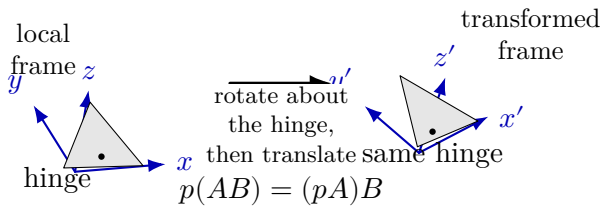


Figure 12: A row-vector view of transformation composition. The point is acted on from the left to the right, so the order in the code matches the order in which the geometric motion is applied.

```
\setobject[
  name = hinge,
  object = {return Vector:new{1, 0, 0, 1}}
]

\setobject[
  name = swing,
  object = {
    return Matrix.transform_about(
      hinge,
      Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.7)
    )
  }
]
```

5.5 Perspective

Perspective is encoded through a projective matrix and the automatic homogeneous divide that follows matrix application. Small values often suffice. Because the effect is projective rather than Euclidean, it is usually best to introduce perspective only after the underlying figure already renders correctly under an affine transform.

```
transformation = {
  return Matrix.axis_angle(Vector:new{1, 0, 0, 1}, 0.9)
    :multiply(Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.4))
    :multiply(Matrix.perspective(Vector:new{0, 0, 0.15, 1}))
}
```

When perspective is active, extremely coarse sampling can exaggerate visual artifacts. If a curve or surface looks jagged only after turning perspective on, the first adjustment to make is the sample count rather than the sorting logic.

6 Objects and Basic Primitives

6.1 Reusable objects with `\setobject`

The command `\setobject` takes two keys: `name` and `object`. The `object` key is a Lua chunk evaluated in the package sandbox, and the result is stored under the chosen name. The stored

object can then be reused in later expressions.

This mechanism is useful for:

- transformations that should be shared by several commands;
- named points or axes;
- scalar parameters such as radii or angles;
- helper functions that belong to one figure.

```
\setobject[
  name = spin,
  object = {return Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.5)}
]

\setobject[
  name = apex,
  object = {return Vector:new{0, 0, 1.8, 1}}
]
```

6.2 Triangles

The command **\appendtriangle** is the smallest direct route to a visible surface. It takes a single matrix-valued key, `m`. That matrix should have three rows, one per homogeneous vertex.

```
\appendtriangle[
  m = {
        return Matrix:new{
          {-1, -1, 0, 1},
          { 1, -1, 0, 1},
          { 0,  1, 0, 1}
        }
      },
  fill options = {fill=ltdtbrightness, draw=black}
]
```

If any two vertices coincide after evaluation, the triangle is discarded. The package does this intentionally so that degenerate faces do not pollute the partitioning and sorting stages.

6.3 Labels

Labels are appended by three-dimensional position but rendered as ordinary TikZ nodes in the projected plane.

```
\appendlabel[
  v = {return Vector:new{0.2, 0.6, 0.4, 1}},
  text = {$p$}
]
```

The `text` key is passed directly into the node body. Since labels are rendered after geometry, they are a good mechanism for annotations, captions, and callouts that should remain visible regardless of occlusion.

6.4 About isolated points

The Lua layer still understands point-like simplex records, but the public `\appendpoint` command is not exported by the current style file. If a figure needs a visible point marker, the easiest replacements are:

- a tiny surface patch oriented toward the viewer;
- a small triangle or short curve segment used as a marker;
- a label whose text is itself the marker, such as `$\bullet$`.

For example, a tiny disk-like patch can be approximated with a surface whose parameter domain is a small polar rectangle.

```
\appendsurface[
  uparams = {return Vector:new{0, tau, 16}},
  vparams = {return Vector:new{0, 0.08, 2}},
  v = {return Vector:new{0.6 + v*math.cos(u), 0.3 + v*math.sin(u), 0.2, 1}},
  fill options = {fill=black, draw=none}
]
```

6.5 Using `\luatikztdtoolsset`

The wrapper `\luatikztdtoolsset` is simply a convenience entry into the key family `/luatikz3dtools/.cd`. It is most useful when you want to establish defaults before several append commands. For example, a document may prefer to store a common surface style in the key tree and then override it only locally.

In most figures this is optional. Local keys attached to each append command are usually clearer than globally mutated defaults.

7 Parametric Curves, Surfaces, and Solids

The parametric commands are where lua-tikz3dtools becomes more than a triangle dispatcher. Each command samples a user-provided function over one, two, or three parameter directions and converts the result into simplices that the renderer can sort.

7.1 Parameter triples

The keys `uparams`, `vparams`, and `wparams` each expect a vector of the form

```
return Vector:new{start, stop, samples}
```

The sample count must be at least 2. Legacy keys such as `ustart`, `ustop`, and `usamples` are still recognized, but the triple form is better because it keeps each parameter direction self-contained.

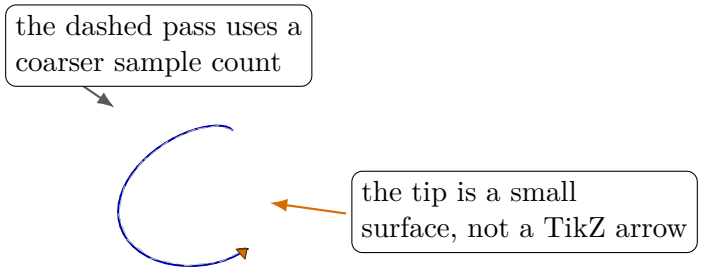


Figure 13: A sampled helix. The dashed pass makes the segment structure visible, while the highlighted tip shows that curve arrowheads are inserted as genuine geometry and therefore participate in the same visibility pipeline as the curve itself.

7.2 Curves

The command `\appendcurve` interprets its `v` key as the body of a Lua function in one variable `u`. Consecutive samples are connected by line segments. The visual style is controlled by `draw options`. Optional `arrow tip` and `arrow tail` values generate small surface geometry at the end points.

```
\appendcurve[
  uparams = {return Vector:new{0, 4*math.pi, 120}},
  v = {return Vector:new{math.cos(u), math.sin(u), 0.15*u, 1}},
  draw options = {draw=blue, very thick},
  arrow tip = {fill=blue, draw=none}
]
```

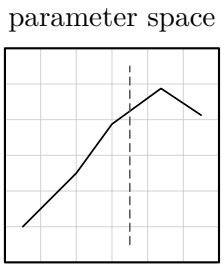
The arrowheads are not decorative TikZ path tips. They are small sampled surfaces inserted into the scene so that they participate in the same visibility logic as other geometric objects.

7.3 Surfaces

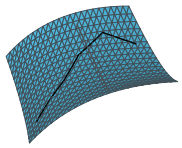
The command `\appendsurface` samples a function of two parameters and splits each parameter-space cell into two triangles. The function body should return a three-dimensional point as a homogeneous vector.

```
\appendsurface[
  uparams = {return Vector:new{-1.5, 1.5, 40}},
  vparams = {return Vector:new{-1.5, 1.5, 40}},
  v = {return Vector:new{u, v, 0.35*(u*u - v*v), 1}},
  fill options = {fill=ltdtbrightness, draw=black!20}
]
```

The package omits degenerate triangles automatically. This is useful when a parametric description pinches or when multiple parameter values map to the same three-dimensional point.



Parameter domain. The highlighted segments are straight in the (u, v) square, where they are easy to specify and easy to reason about.



Embedded on the surface. The same parameter-space segments are clipped against the surface triangles and carried barycentrically, so they remain attached even if the surface is later partitioned for visibility.

Figure 14: Embedded parameter-space curves. The left panel shows the curves where they are authored; the right panel shows the same segments after the surface map, the projection, and any later triangle partitioning.

7.4 Embedded parameter-space curves on surfaces

The key `curve` on `\appendsurface` is an advanced feature. It takes a Lua chunk that returns a table of parameter-space segments. Each segment is then clipped into the surface triangles and stored barycentrically so that it survives subsequent triangle partitioning.

Each segment table may use either numeric entries `{start, stop}` or named fields `start = ...` and `stop = ...`. The end points should be two-dimensional parameter points such as `Vector:new{u, v, 1}`. An optional Lua field `drawoptions` may be added per segment.

```
\appendsurface[
  uparams = {return Vector:new{0, 1, 32}},
  vparams = {return Vector:new{0, 1, 32}},
  v = {return Vector:new{2*u - 1, 2*v - 1, math.sin(math.pi*u)*math.sin(math.pi*v), 1}},
  fill options = {fill=ltdtbrightness, draw=none},
  curve = {
    return {
      {
        start = Vector:new{0.10, 0.10, 1},
        stop = Vector:new{0.90, 0.90, 1},
        drawoptions = "draw=black, thick"
      },
      {
        start = Vector:new{0.10, 0.90, 1},
        stop = Vector:new{0.90, 0.10, 1},
        drawoptions = "draw=black, dashed"
      }
    }
  }
]
```

This feature is especially useful for displaying coordinate curves, domain cuts, or trajectories that are naturally defined in the surface's own parameter domain.

7.5 Solids

The command `\appendsolid` samples a function of three parameters but does not try to fill the volume. Instead it tessellates the six boundary faces of the parameter box. This is exactly what most illustrative applications need: the visible boundary of a sampled solid.

```
\appendsolid[
  uparams = {return Vector:new{-1, 1, 16}},
  vparams = {return Vector:new{-1, 1, 16}},
  wparams = {return Vector:new{-1, 1, 16}},
  v = {return Vector:new{1.2*u, 0.9*v, 0.7*w, 1}},
  fill options = {fill=ltdtbrightness, draw=black!15}
]
```

The same caveat as for surfaces applies: the quality of the result depends on the sampling resolution. A boundary that bends sharply should be sampled more densely than a nearly planar one.

8 Lighting, Styling, and Filters

The package keeps geometry generation separate from visual styling. Geometry keys determine what simplices exist; style keys determine how those simplices are drawn once the visibility pipeline is complete.

8.1 TikZ styling keys

Triangles use `fill options`. Curves use `draw options`. Labels use the literal node body given by `text`. Because the package ultimately emits ordinary TikZ paths and nodes, these styling strings are plain TikZ option lists.

```
fill options = {fill=ltdtbrightness, draw=black, line join=round}
draw options = {draw=blue!70!black, ultra thick}
```

For triangles there is one package-specific convention worth remembering: if you want lights to affect a face, the fill style must actually reference `ltdtbrightness`. The renderer computes that color dynamically from the current lights and the face normal.

8.2 Directional lights

Lights are appended by `\appendlight` and supplied as vectors. The package normalizes those directions and computes a linear brightness factor from the angle between the light direction and the face normal. Multiple lights are averaged.

```
\appendlight[
  v = {return Vector:new{0.3, -0.2, 1, 1}}
]

\appendlight[
  v = {return Vector:new{-0.5, 0.4, 0.7, 1}}
]
```

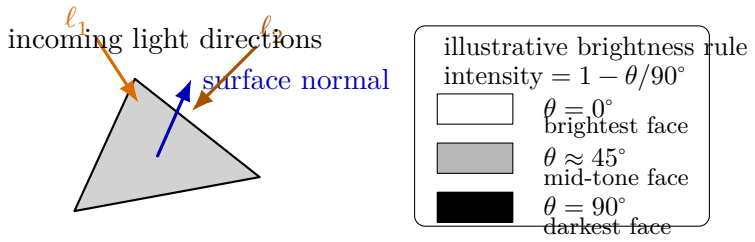


Figure 15: Directional lighting in the package. Triangle shading depends only on the angle between the face normal and each appended light direction, and the implemented falloff is linear in that angle rather than cosine-based.

The falloff is linear in the angle from 0° to 90°, not cosine-based radiometry. That choice is intentional: it gives predictable illustrative shading without demanding a physically based material model.

8.3 Filters

Every geometric append command accepts a `filter` key. The filter is evaluated after partitioning, so it sees the pieces that will actually participate in the sort. That makes the filter mechanism more powerful than a simple pre-check on the original source geometry.

The bound names depend on the simplex type:

- point-like and label entries bind `A`;
- line segments bind `A` and `B`;
- triangles bind `A`, `B`, and `C`.

Typical uses include removing back-facing slices, hiding geometry below a plane, or discarding pieces outside a region of interest.

```
filter = {
  local normal = (B:hsub(A)):hhypercross(C:hsub(A)):hnormalize()
  return normal[3] > 0
}
```

```
filter = {return A[3] >= 0 and B[3] >= 0}
```

Because the filter sees the partitioned pieces, it can be used as a clean visual cutting tool without forcing the user to pre-split the source geometry manually. When a cut should land exactly on a visible boundary, a practical trick is to append an auxiliary surface that lies on the intended slicing boundary, let the partitioner split the target geometry against that surface, and then give the auxiliary surface `filter = {return false}` so it disappears from the final render.

8.4 A style strategy that scales

For larger figures, it is usually best to settle on a small style vocabulary and reuse it consistently:

- triangles: `fill=ltdtbrightness, draw=...`;

- curves: `draw=...` through `draw options`;
- special overlays: embedded surface curves with their own `drawoptions`;
- annotations: labels without lighting.

This keeps the visual language clear and makes it easier to tune the lighting later without rewriting the geometry descriptions.

9 Command Reference

This section collects the public commands and their keys in reference form. Examples remain brief here; the larger workflow discussion is in the earlier sections.

9.1 `\luatikztdtoolsset`

Convenience wrapper for setting keys in the family `/lua-tikz3dtools/.cd`. Use it when document-wide or figure-wide defaults are worth centralizing.

9.2 `\setobject`

Key	Expected value	Meaning
`name`	nonempty string	Name stored in the sandbox object table.
`object`	Lua chunk	Evaluated object to store under `name`.

9.3 `\appendlabel`

Key	Expected value	Meaning
`v`	point expression	Three-dimensional label position.
`text`	TeX material	Node body emitted after the geometry.
`transformation`	matrix expression	Optional transform; default is identity.
`filter`	Lua block	Predicate on the projected point `A`; default returns true.

9.4 `\appendlight`

Key	Expected value	Meaning
`v`	vector expression	Directional light vector.

9.5 `\appendtriangle`

Key	Expected value	Meaning
`m`	matrix expression	Whole 3×4 triangle matrix, with one homogeneous vertex per row.
`transformation`	matrix expression	Optional transform; default is identity.
`fill options`	TikZ option list	Styling for the emitted triangle path.
`filter`	Lua block	Predicate on the triangle vertices `A`, `B`, `C`.

9.6 \appendcurve

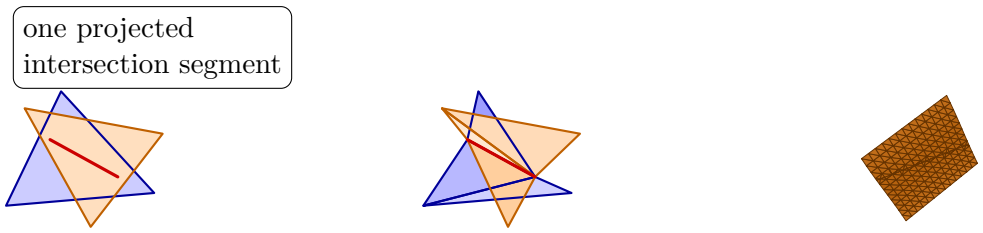
Key	Expected value	Meaning
uparams	parameter triple	Start, stop, and sample count for u.
ustart, ustop, usamples	legacy scalars	Older equivalent form for the u direction.
v	function body in u	Returns a point for each sample.
transformation	matrix expression	Optional transform; default is identity.
draw options	TikZ option list	Styling for emitted segments.
arrow tip	TikZ option list	Style for geometric arrowhead at the end.
arrow tail	TikZ option list	Style for geometric arrowhead at the start.
filter	Lua block	Predicate on segment endpoints A, B.

9.7 \appendsurface

Key	Expected value	Meaning
uparams, vparams	parameter triples	Start, stop, and sample counts for the two parameters.
ustart, ustop, usamples	legacy scalars	Older equivalent form for u.
vstart, vstop, vsamples	legacy scalars	Older equivalent form for v.
v	function body in u,v	Returns the sampled point on the surface.
transformation	matrix expression	Optional transform; default is identity.
fill options	TikZ option list	Styling for emitted triangles.
filter	Lua block	Predicate on triangle vertices A, B, C.
curve	Lua chunk	Returns a table of parameter-space line segments to embed on the surface.

9.8 \appendsolid

Key	Expected value	Meaning
uparams, vparams, wparams	parameter triples	Parameter ranges and sample counts for the three directions.
ustart, ustop, usamples	legacy scalars	Older equivalent form for u.
vstart, vstop, vsamples	legacy scalars	Older equivalent form for v.
wstart, wstop, wsamples	legacy scalars	Older equivalent form for w.
v	function body in u,v,w	Returns the sampled point in space.
transformation	matrix expression	Optional transform; default is identity.
fill options	TikZ option list	Styling for emitted boundary triangles.
filter	Lua block	Predicate on triangle vertices A, B, C.



Before partitioning. Each triangle changes depth order across the intersection segment, so neither face admits one global front-to-back placement.

After partitioning. The original faces are replaced by smaller pieces whose local depth order is now stable enough to sort.

Rendered case. Intersecting sampled sheets plus a simple filter create exactly the kind of local ambiguity that requires partitioning before occlusion sorting.

Figure 16: Partitioning makes a sortable scene out of a locally ambiguous one. The left and middle panels isolate the geometric idea; the right panel shows the same principle inside an actual surface rendering.

9.9 `\displaysimplices`

This command takes no keys. It partitions geometry if needed, applies filters, sorts by occlusion, emits TikZ paths, emits labels, and clears the accumulated scene and light list.

9.10 Defaults and common expectations

Unless a command says otherwise, the default transformation is the identity and the default filter accepts everything. Sample counts on parametric commands must be at least 2. The renderer silently ignores degenerate simplices and rejects non-finite projected values.

The current style file does not export `\appendpoint`. Authors who need a marker-like point should use one of the strategies discussed in Section 6.

10 Rendering Pipeline, Limitations, and Troubleshooting

10.1 How visibility is determined

The renderer combines several ideas.

1. It uses two-dimensional bounding boxes to avoid obviously irrelevant pairwise tests.
2. It partitions simplices where geometric intersection makes a single global order impossible.
3. It constructs an occlusion graph for the remaining candidate pairs.
4. It sorts strongly connected components and tries to break cycles by further partitioning along edge planes.
5. It emits the sorted simplices as ordinary TikZ paths.

This is considerably more robust than assigning a single depth statistic to each object. In particular, triangle–triangle interactions are handled in a way that survives many intersecting or partially overlapping configurations.

10.2 What the package does not promise

The renderer is strong, but it is not a full hidden-surface engine for arbitrary analytic geometry. The practical limitations are these.

- Curves, surfaces, and solids are sampled first. If the sample count is too low, the output approximates the wrong object accurately.
- Solids are boundary tessellations, not volumetric fills.
- Labels are rendered after geometry and do not participate in occlusion.
- The public point command is not currently exported by the style file.
- Some cyclic occlusion configurations may remain unresolved after recursive partitioning, in which case the package warns that the order may be unstable.

The last point is rare but conceptually important. Occlusion among sampled pieces can form cycles. The package tries to resolve those cycles by partitioning, but it eventually stops recursing rather than partitioning forever.

10.3 Reading the package behavior correctly

When a rendered scene looks wrong, the failure is often one of the following.

Too few samples. If a surface boundary or a curve appears faceted or visibly misplaced, increase the sample counts first.

Transformation order reversed. If geometry appears in the wrong place or orientation, re-check the row-vector composition order from Section 5.

Lighting style omitted. If triangles ignore lights, verify that `fill options` mentions `ltdtbrightness`.

Filter too aggressive. If geometry vanishes unexpectedly, temporarily remove the `filter` key and confirm that the geometry exists before debugging the predicate.

The wrong abstraction level. If the desired figure depends on exact analytic visibility of a continuous object, refine the discretization until the sampled geometry is an adequate model.

10.4 Suggestions for a troubleshooting chapter figure

If you plan to add one final didactic illustration to the manual, a good choice is an intentionally coarse surface rendered beside the same surface with a refined sample grid. That figure communicates, in one glance, the single most important practical limitation of the package: everything interesting happens after the continuous model has already been discretized.

A Vector and Matrix Cookbook

This appendix records the most useful Lua-side constructors and helpers for users who want to build scene data more explicitly.

A.1 General guidance

Use the validated constructors `Vector:new` and `Matrix:new` in document-level code. Internal unchecked constructors such as `Vector:\_new` and `Matrix:\_new` exist for performance inside the package implementation and are not the intended public interface.

A.2 Useful vector constructors and helpers

Function	Purpose
`Vector:new{x,y,z,1}`	General homogeneous point constructor.
`Vector.sphere(Vector:new{theta,phi,r})`	Point on a sphere from spherical coordinates.
`Vector.stereographic_projection(v)`	Stereographic projection of a three-dimensional point.
`Vector.inverse_stereographic_projection(v)`	Inverse stereographic projection from a planar point.
`v:hadd(w)`, `v:hsub(w)`	Homogeneous addition and subtraction.
`v:hinner(w)`	Homogeneous inner product ignoring the last coordinate.
`v:hhypercross(w)`	Cross-product-like construction used for normals and orthogonal vectors.
`v:hnormalize()`	Normalize the spatial part of a vector.
`v:orthogonal_vector()`	Return an arbitrary vector orthogonal to `v`.

A.3 Useful matrix constructors and helpers

Function	Purpose
`Matrix.identity()`	Identity transformation.
`Matrix.translate(delta)`	Translation by a vector `delta`.
`Matrix.scale_axis(scale)`	Axis-aligned scaling.
`Matrix.axis_angle(axis, theta)`	Rotation about an arbitrary axis through the origin.
`Matrix.zyzrotation(angles)`	ZYZ Euler rotation.
`Matrix.shear(...)`	Six-parameter shear matrix.
`Matrix.reflect_axis(axis)`	Reflection about an axis through the origin.
`Matrix.perspective(axis)`	Projective perspective transform.
`Matrix.transform_about(point, T)`	Conjugate a transform `T` by a fixed point.
`M:multiply(N)`	Matrix composition in row-vector order.
`M:get_bbox2()`, `M:get_bbox3()`	Axis-aligned bounding boxes.
`M:hcentroid()`	Homogeneous centroid of the rows.

A.4 Typical recipes

Rotate, then translate.

```
return Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.6)
  :multiply(Matrix.translate(Vector:new{2, 0, 0, 1}))
```

Build a tilted sphere point.

```
return Vector.sphere(Vector:new{u, v, 1.4})
  :multiply(Matrix.axis_angle(Vector:new{1, 0, 0, 1}, 0.8))
```

Apply a local motion about a non-origin point.

```
return Matrix.transform_about(
  Vector:new{1, 2, 0, 1},
  Matrix.axis_angle(Vector:new{0, 0, 1, 1}, 0.4)
)
```

B Low-Level Geometry API

The geometry module exists mainly to support the renderer, but its methods are also installed onto the `Vector` and `Matrix` metatables for backward compatibility. This appendix records the main categories for readers who need to reason about the implementation or write custom Lua-side utilities.

B.1 Point-level queries

The vector metatable receives methods for:

- point–point coincidence tests;
- point-in-triangular-prism tests;
- point–triangle intersection tests;
- point versus point, segment, and triangle occlusion comparisons;
- collinearity and opposite-direction tests.

Representative names include `hpoint_point_intersecting`, `hpoint_in_triangular_prism`, `hpoint_triangle_intersecting`, `hpoint_line_segment_occlusion_sort`, and `hpoint_triangle_occlusion_`

B.2 Segment- and triangle-level queries

The matrix metatable receives methods for:

- segment–point, segment–segment, and segment–triangle intersection tests;
- line–line and line–plane intersections in basis form;
- partitioning a segment or triangle by another geometric object;
- segment–segment, segment–triangle, and triangle–triangle occlusion comparisons.

Representative names include `hline_segment_point_intersecting`, `hline_segment_line_segment_intersect` `hline_plane_intersection`, `hpartition_triangle_by_triangle`, and `htriangle_triangle_occlusion_sort`

B.3 Spatial indexing and sorting

At the module level, the geometry layer also provides the functions that drive the renderer itself:

- `build_grid` and its candidate-query helpers for two-dimensional spatial indexing;
- `partition_simplices_by_parents` for recursive splitting based on overlapping parents;
- `scc` for strongly connected component sorting with cycle handling;

- `occlusion_sort_simplices` as the main type-dispatch entry point.

These functions are useful if you are extending the package internals or if you need to experiment with the renderer outside TeX. They are not required for normal document use.

B.4 A practical boundary

As a rule, if a task can be expressed through `\setobject` plus the public append commands, stay at that level. Reach for the low-level geometry API only when you are deliberately building new package behavior or debugging geometric edge cases.