lmt_scene

Hans Hagen & Mikael Sundqvist

Contents

1 Introduction

METAFUN and METAPOST are mainly two-dimensional drawing tools, but many figures also need a three-dimensional view. The `lmt_scene` commands give you one place to define the geometry, camera, lighting, and annotations for that view. In this document we collected what is currently available. Since the project started out as an AI experiment, we also let Chat GPT (Luna) generate parts of the content of this manual.

The interface is scene-based. With a scene you can:

- choose the camera, light, colors, output size, and other shared settings;
- draw several kinds of geometry: surfaces given by a function, $z = f(x, y)$, parametric surfaces given by $(x, y, z) = (f_1(u, v), f_2(u, v), f_3(u, v))$, implicit plots described by $F(x, y, z) = 0$, parametric curves $(x, y, z) = (f_1(t), f_2(t), f_3(t))$, finite plane patches given by a center point, a normal direction, and two sizes, and imported triangle meshes from external files.
- add reference geometry such as grids, axes, ticks, and annotations; and
- combine the objects with common camera and depth handling.

A figure starts as a description of its objects, camera, materials, and output settings. Surfaces are sampled over chosen domains, while imported or generated geometry is added as meshes. All of these objects share one camera and one depth-aware renderer, so they appear in the same picture. After the scene is rendered, projection helpers can place ordinary METAPOST labels and annotations.

Lighting is simple: a directional light works with the material color, ambient level, highlight, and transparency. These few controls are enough to show the shape of an object without making the setup hard to follow. Stippling is a raster postprocessor and is covered near the end of the manual.

The labels in the table of contents are hints about when to read a section: *optional* sections can wait, *advanced* sections assume the basic scene workflow, and *experimental* sections describe interfaces that may change. None of them is needed for a first scene.

Roadmap

This route gives a useful starting order:

- **Quick start** shows the smallest complete file and the three-step scene lifecycle.
- **Scene basics** explains the shared camera, output, lighting, and cropping settings; **Defaults** collects the principal defaults in one place.
- **Materials** explains how to control color, lighting, transparency, and line-like helpers.

- **Surface graphs** is the usual starting point when the geometry is given by $z = f(x, y)$.
- Continue with **parametric surfaces**, **implicit surfaces**, **curves**, and **intersections** as the geometry requires. The **Reference geometry** section covers **planes**, **segments**, **grids**, **axes**, and **ticks**.
- **Rendering pipeline** explains why mesh counts, raster resolution, and supersampling affect different stages of a render. The **Putting it all together** example then combines the main scene features in one file.
- Leave **caching**, **annotations**, **stippling**, **special overlays**, **low-level LUA helpers**, and **mesh inspection** until the basic scene workflow is familiar. External and internal meshes are introduced in the object table when those forms of geometry are needed.

Command sections explain the purpose of an interface, show one or more runnable examples, and collect the supported keywords nearby.

Quick start

To try a scene, save the complete file below as `hello3d.tex` and run `context hello3d.tex` in the same directory. You need a CONTEXT installation that includes the LUAMETATEX scene commands. The `offset` leaves a little space around the rendered page. Later examples use buffers so that their source and result can be shown together.

```
\starttext

\startMPpage[offset=1cm]
lmt_scene_start [
  width  = 7cm,
  height = 5cm,
  crop   = true,
] ;

lmt_scene_surface [
  code = "sin(x)*cos(y)",
  xmin = -pi,
  xmax =  pi,
  ymin = -pi,
  ymax =  pi,
] ;

lmt_scene_stop ;
\stopMPpage

\stoptext
```

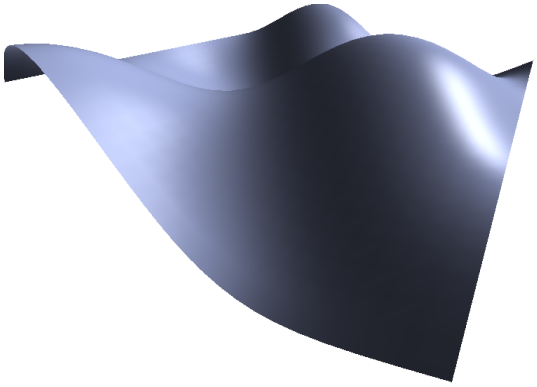


Figure 1 A first scene: the graph of $z = \sin(x)\cos(y)$.

This manual describes the scene module generated on 2026-05-01 or newer. Older installations may report unknown `lmt_scene_*` commands or keywords in the log; update CONTEXT when that happens.

The scene is built in three steps: start it, add one or more objects, and stop it. The `code` value is a quoted expression because it is evaluated while the surface is being generated. The names `x`, `y`, and `z` refer to coordinates; the available operators, functions, and constants are collected in the **Expression language** subsection below.

There are a few conventions worth remembering:

- A triplet such as `(1,0,0)` is a 3D point or direction. Its meaning depends on the keyword that receives it.
- Material and background colors are RGB triplets with components between `0` and `1`; for example, `(1,0,0)` is red. Line helpers use the same color values through their `color` keyword; a METAPOST color name such as `red` is accepted there as well. Even better, use colors that are shared with the TEX end, like `"red"`.
- Give an object an `id` when another object or a query must refer to it. An intersection can only refer to objects that have already been added.
- Start with the default camera and material. Add `eye`, `target`, and a named material only when the first picture needs adjustment.

Keep the first example small. Once it works, change one setting at a time. Add a named material when the default appearance is not enough, adjust `eye` or `target` when the view is wrong, and increase the mesh counts or raster resolution only when the result needs more detail. The later sections build on this same workflow.

Build on the first scene

Now give the surface a name and a material. The geometry stays the same; only the view and appearance change:

```
lmt_scene_start [
  width  = 7cm,
```

```
  height = 5cm,
  crop   = true,
  eye    = (3,-4,2.5),
  target = (0,0,0),
] ;

lmt_scene_material [
  name    = "wave",
  diffuse = (.15,.55,.90),
  ambient = .25,
] ;

lmt_scene_surface [
  id       = "wave",
  code     = "sin(x)*cos(y)",
  xmin     = -pi,
  xmax     =  pi,
  ymin     = -pi,
  ymax     =  pi,
  material = "wave",
] ;

lmt_scene_stop ;
```

The **Materials** section explains the appearance settings. The surface sections show how to replace this graph with a parametric or implicit object. The sections on planes, intersections, and reference geometry then show how to turn the picture into an annotated figure.

Common adjustments

When the picture needs tuning, change one group of settings at a time:

- If the surface looks faceted or misses a small feature, increase the object mesh counts such as `nx`, `ny`, `nu`, or `nv`. For an implicit surface, increase all three of `nx`, `ny`, and `nz`.
- If edges look jagged, increase `resolution` or try `supersample = 2` or `4`. These settings affect the raster, not the geometric mesh.
- If the object is seen from the wrong side, adjust `eye` and `target`. Use `crop = false` while comparing camera settings.
- If the shape looks flat, try another `light` direction, adjust the material's `ambient` value, or choose a smoother normal mode.
- If a plane should sit over the rest of the scene, give it a low `opacity`; if a line should remain visible, use a material with `ontop = true`.

Expression language

The quoted scene expressions (or course) use the LUA expression language. Arithmetic uses `+`, `-`, `*`, `/`, the remainder operator, and `^` for powers; comparisons use `==`, `~=`, `<`, `<=`, `>`, and `>=`; and the logical operators are `and`, `or`, and `not`. The remainder operator is written as follows in the expression itself:

```
x % y
```

Parentheses can be used wherever they make the order of evaluation clearer.

Angles are in radians. The constant `pi` is available, so a full turn is normally written `2*pi`; there is no degree conversion in the scene expression syntax.

The expression environment exposes LUAMETATEX's extended math table as `math` and also makes its names available at the top level. Useful names include `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `exp`, `log`, `log10`, `sqrt`, `cbrt`, `floor`, `ceil`, `round`, and `trunc`. For absolute values and extrema, use `abs`/`fabs`, `min`/`fmin`, and `max`/`fmax`; the `f...` spellings are the portable names in the extended `xmath` table. Related helpers include `hypot`, `pow`, `deg`, `rad`, `dotproduct`, `crossproduct`, and `normalize`.

When you use functions, be aware that the default environment is `xmath` but we also register `MP` (the use namespace), `documentdata`, `userdata`, `moduledata` and `globaldata`. By making `xmath` default performance remain good.

The variables depend on the object command. Surface graph expressions use `x` and `y`; parametric-surface coordinates and tangent expressions use `u` and `v`; implicit expressions and derivatives use `x`, `y`, and `z`; and curve coordinates use `t`. The LUA `pfunction` and `nfunction` forms receive `x`, `y`, and `z` as function arguments. The `z` value of a graph is produced by its expression and is not an independent input variable.

Expression loading and evaluation diagnostics are written to the job's `.log` file. A failed expression is reported with its expression name; inspect the log before changing the sampling grid or camera.

2 Scene basics

A scene contains one 3D picture. It holds the objects and the shared settings that control them: output size, camera, lighting, background, and quality. Surfaces, curves, grids, axes, and other helpers can all share the same scene.

Most scenes have this shape:

```
lmt_scene_start [ ... ] ; % open the scene
  % here we add our 3D graphics
lmt_scene_render ; % optional: render before annotations
  % here we add extra material
lmt_scene_stop ; % finish the scene
```

`lmt_scene_start` opens a scene and records its shared settings. The object commands between start and stop add geometry. Most figures need only `lmt_scene_stop`. Use `lmt_scene_render` when later METAPOST code needs the rendered size, projected positions, or visibility information before the scene is closed. Stopping the scene places the finished bytemap in the current picture.

Here is a small initialization:

```
lmt_scene_start [
  width       = 5cm,
  projection  = "perspective",
  eye         = (3.0,-4.2,2.2),
  target      = (0,0,0),
  crop        = true,
  light       = (-1,-1,-2),
] ;
```

This scene is five centimeters wide and uses perspective projection. The camera is at `(3.0,-4.2,2.2)`, and the light comes from `(-1,-1,-2)`. The other scene options fall into four groups: output size and quality, camera and view, lighting and background, and optional postprocessing.

Camera and output settings control different things. The camera decides what is visible; the output settings decide how the picture is sampled and placed in the document. The `eye` is the camera position, and `target` is the point it looks at. The `up` direction rotates the view. In a perspective scene, `fov` controls its width: a smaller value shows less and a larger value shows more.

`crop` controls the relationship between the camera frame and the objects. With `crop = true`, the result is fitted to the visible scene bounds. With `crop = false`, the camera frame stays fixed and objects outside it can be clipped. This makes it easier to compare settings such as `fov` and `target`.

At first, `width` and `height` are enough. They set the displayed document dimensions. `resolution` converts one dimension to raster pixels, and the other raster dimension

is derived from the scene bounds. Adding an object can therefore change the raster aspect ratio. If both document dimensions are given, they fix the displayed rectangle.

`bytewidth` and `byteheight` choose the raster dimensions directly. Give both byte dimensions and omit `width` and `height` when the raster shape must stay fixed:

```
lmt_scene_start [
  bytewidth  = 800,
  byteheight = 500,
  projection = "perspective",
  eye        = (0,-8,3),
  target     = (0,0,0),
  fov        = 45,
  crop       = false,
] ;
% add the scene objects here
lmt_scene_stop ;
```

This keeps the raster aspect ratio fixed. Use `width` or `height` when physical size is the main concern; use an explicit byte-size pair when the pixel dimensions must stay fixed.

Choosing an object

Choose an object command from the way the geometry is described:

Object	**Use it when**	**Command**
Graph surface	the height is given by $z = f(x, y)$	**`lmt_scene_surface`**
Parametric surface	the point (x, y, z) depends on (u, v)	**`lmt_scene_parametric`**
Implicit surface	an equation $F(x, y, z) = c$ describes the shape	**`lmt_scene_implicit`**
Parametric curve	the point (x, y, z) depends on t	**`lmt_scene_curve`**
Plane	a finite rectangular patch is enough	**`lmt_scene_plane`**
External mesh	the triangles are read from an external file	**`lmt_scene_external`**
Internal mesh	the geometry is generated by a LUA mesher	**`lmt_scene_internal`**

The first four are the usual starting points. Use a plane when a finite patch is enough. Use an external or internal mesh when the geometry already exists in another form or must be generated by LUA.

Defaults

The following table collects the principal defaults in one place. A value marked `unset` is derived from the scene bounds or from another dimension. Object sampling defaults are included here because they determine geometric detail, whereas `resolution` and `supersample` determine raster detail.

Group	Setting	Default and meaning
Scene	`width`, `height`	`unset`; the displayed size is derived unless a dimension is given.
Scene	`bytemap`	integer slot `1`; use another slot for a separate scene output, or set it to `false` to disable bitmap output.
Scene	`cache`	`false`; whole-scene rendered-bytemap caching is off.
Camera	`projection`	`"perspective"`.
Camera	`eye`	`(3,-4,2)`; `"auto"` fits a generated or imported scene.
Camera	`target`	`(0,0,0)`; `"auto"` aims at the fitted scene center.
Camera	`up`, `fov`	`(0,0,1)` and 45 degrees for perspective projection.
Camera	`crop`	automatic scene-bound fitting when not explicitly set.
Lighting	`light`	directional vector `(-1,-1,-1)`.
Lighting	`intensity`	`1`.
Lighting	`backgroundcolor`	white, `(1,1,1)`.
Material	`diffuse`	`(.70,.75,.95)`.
Material	`specular`, `shininess`	`(.50,.50,.50)` and `40`.
Material	`ambient`, `opacity`	`.18` and `1`.
Material	`twosided`, `ontop`, `dx`, `dy`	`false`, `false`, `0`, and `0`.
Raster	`resolution`	`300` raster pixels per inch when byte dimensions are not explicit.
Raster	`bytewidth`, `byteheight`	`unset`; derived from display dimensions and bounds.
Raster	`maxfragment`	`0`; use the normal single-depth storage.
Sampling	`supersample`	`1`; no temporary oversampling.
Sampling	surface `nx`, `ny`	`32`, `32` intervals.
Sampling	parametric `nu`, `nv`	`64`, `32` intervals.
Sampling	curve `nt`	`64` samples.
Sampling	implicit `nx`, `ny`, `nz`	required; there is no safe implicit-grid default.

Scene keywords

bytemap selects the integer slot used for the rendered METAPOST bitmap. The default is `1`; choose another slot when multiple scenes are rendered in one picture and their bitmap outputs must remain distinct. Set `bytemap = false` to disable bitmap output; this cannot be combined with `cache = true`.

name gives a cached scene its name. Use a distinct name for each scene definition that should have its own cache entry.

cache when true enables the whole-scene rendered-bytemap cache. The default is disabled; caching requires an enabled integer `bytemap`.

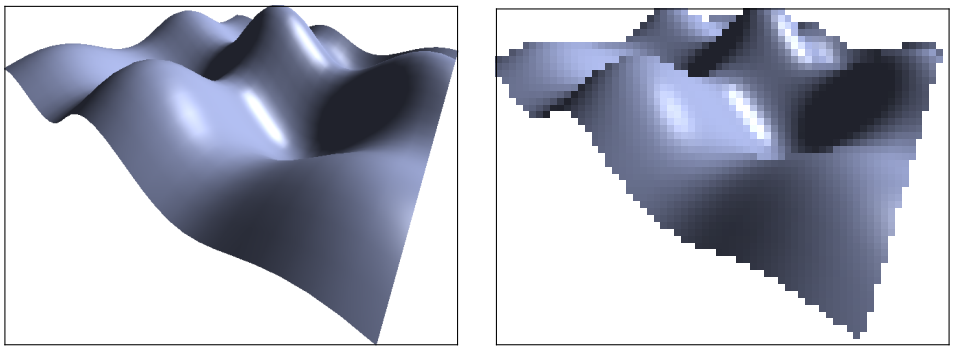
width/height sets the displayed width or height of the final result. These are document dimensions, not necessarily the number of pixels used while rendering. If only one is given, the other displayed dimension follows the raster aspect ratio; giving both fixes the displayed rectangle.
bytewidth sets the raster width before scaling to the displayed size. Use this together with `byteheight`, and without `width` or `height`, when the pixel size and raster aspect ratio should be chosen directly.
byteheight sets the raster height before scaling to the displayed size. Use this together with `bytewidth`, and without `width` or `height`, when the pixel size and raster aspect ratio should be chosen directly.
supersample sets the internal oversampling factor. It can make edges smoother at the cost of a larger temporary raster. The maximum is `4`.
maxfragment limits the number of transparent depth fragments retained per raster pixel. A larger value can preserve more overlapping translucent surfaces, at the cost of more memory and compositing work. Values below one are clamped internally; leave the default unless deep transparency is being lost.
resolution sets the conversion from a requested display size to raster pixels when no explicit byte-size pair is given.
backgroundcolor sets the raster background color.
projection selects the camera projection, normally `perspective` or `orthographic`. Perspective gives nearby parts more visual weight; orthographic keeps parallel directions parallel.
eye sets the camera position, in the same coordinate system as the objects in the scene. The special value `"auto"` derives a useful position from the bounds of generated or imported geometry.
target sets the point toward which the camera looks. The special value `"auto"` uses the center of the scene bounds.
up sets the upward camera direction, and therefore controls how the view is rotated around the line from `eye` to `target`.
fov sets the field of view for perspective projection. Smaller values act more like a long lens; larger values show more of the surroundings.
crop fits the rendered bitmap to the projected scene bounds when true. When false, the camera frame is kept and the bitmap is not trimmed to the objects.
light sets the direction of the directional light source. It is a direction, not a point where a lamp is placed; `(0,0,-1)` means that light travels down from above. The vector is normalized internally, so its length does not change the result.
intensity sets the strength of the directional light source. The default is `1`; lower values make the directional contribution weaker, while the material's `ambient` value still provides a base level.
stipple enables the raster stipple postprocessor and passes it a settings table. Its controls are described in the Stippling section.
process selects a named LUA raster-postprocessing hook. The hook is registered with `metapost.registermeshupper`; it runs on the rendered raster before the supersampled result is resolved. See Raster postprocessing hooks.

Scene settings examples

Each example changes one scene setting. The geometry stays the same, so the visible difference comes from the scene setup. The examples use `crop = true` except the

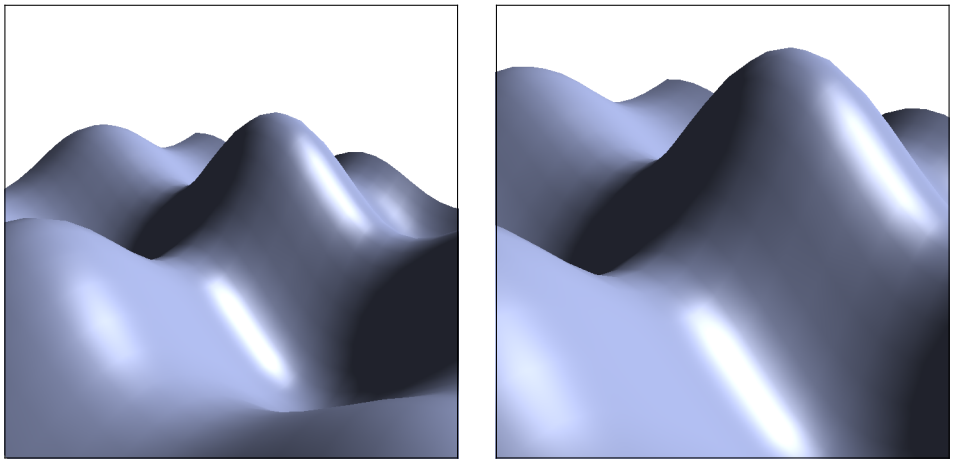
`fov` and `target` comparisons, which keep the camera frame fixed.

```
lmt_scene_start [
  width = 6cm,
  crop  = true,
] ;

lmt_scene_surface [
  code = ".55*exp(-.07*(x*x+y*y))*(cos(2.2*x)+sin(2.0*y))",
  xmin = -3,
  xmax =  3,
  ymin = -3,
  ymax =  3,
] ;

lmt_scene_stop ;

lmt_scene_start [
  width = 6cm,
  crop  = true,
  resolution = 30,
] ;

lmt_scene_surface [
  code = ".55*exp(-.07*(x*x+y*y))*(cos(2.2*x)+sin(2.0*y))",
  xmin = -3,
  xmax =  3,
  ymin = -3,
  ymax =  3,
] ;

lmt_scene_stop ;
```



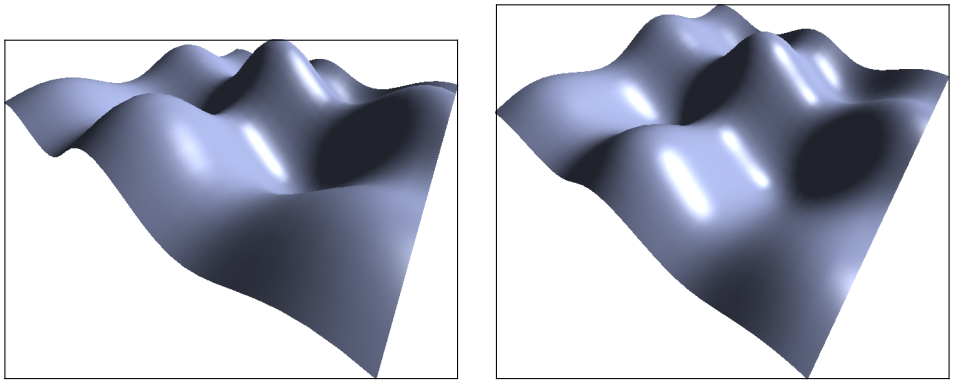
Default resolution 300dpi `resolution = 30`

Figure 2 Showing the influence of the `resolution` keyword.



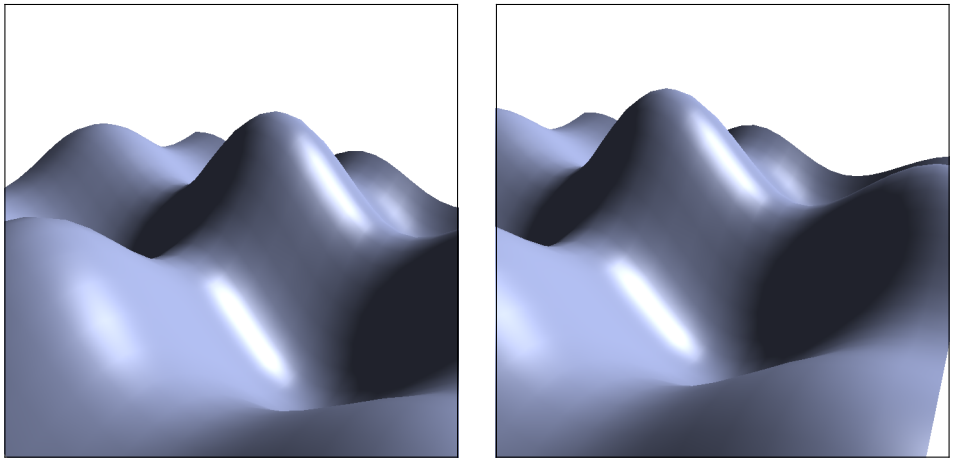
Default `fov = 45` `fov = 30`

Figure 3 Setting the field of view. Its effect is easiest to see when `crop = false`.



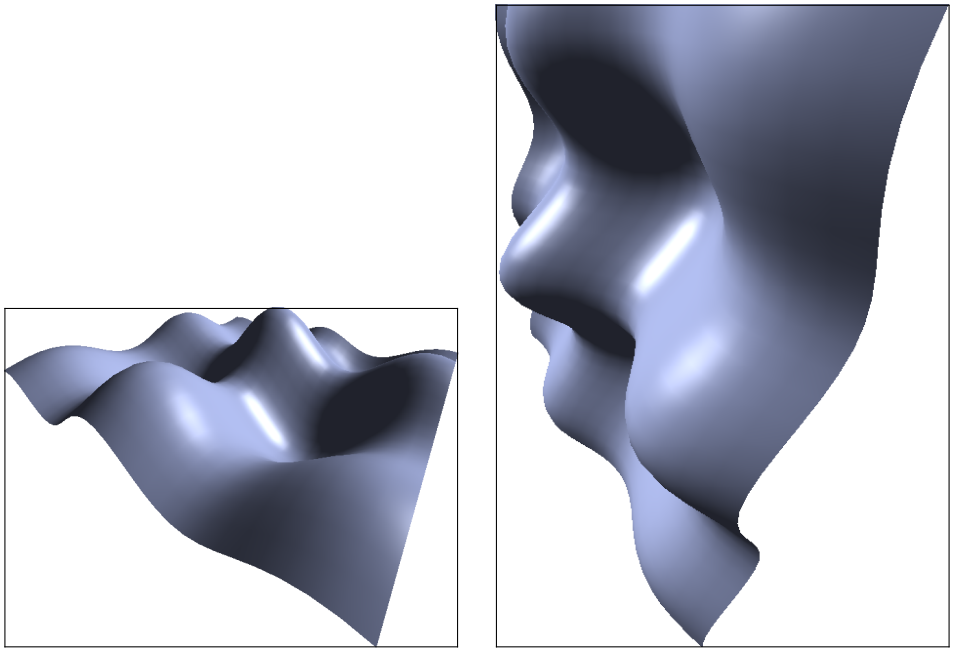
Default `eye = (3,-4,2)` `eye = (3,-4,4)`

Figure 4 Setting the position of the eye. Because the figure has a specified width, changing the eye can change its height substantially.



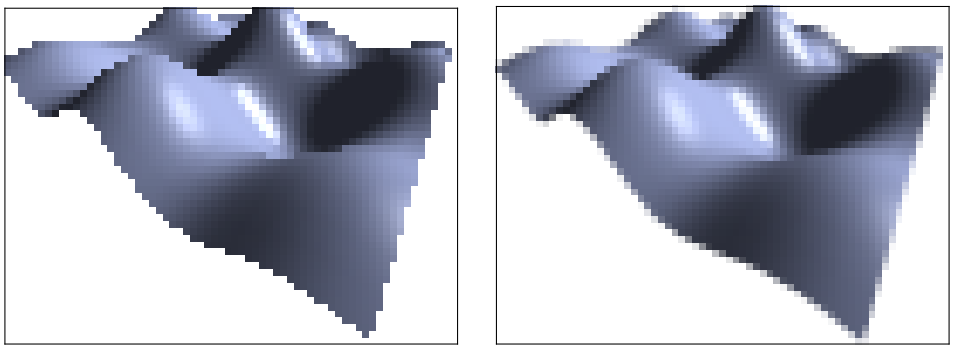
Default `target = (0,0,0)` `target = (1,0,0)`

Figure 5 Setting the target point. Its effect is easiest to see when `crop = false`.



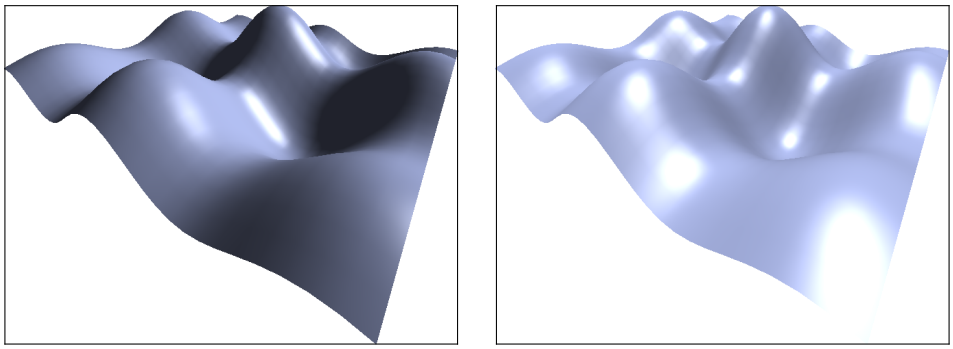
Default `up = (0,0,1)` `up = (1,0,0)`

Figure 6 Setting the `up` direction rolls the camera frame around the eye–target axis.



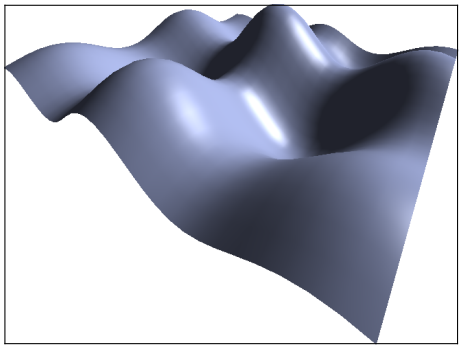
Default `supersample = 4`

Figure 7 Supersampling produces antialiasing. In this example, we also use a low resolution to make the difference clear.

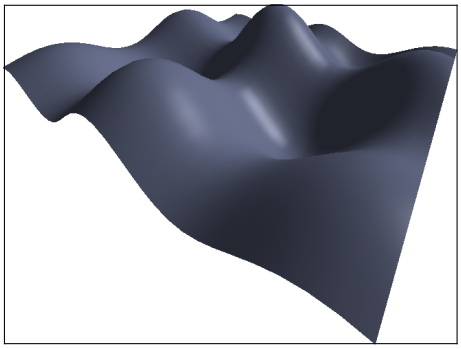


Default `light = (-1,-1,-1)` `light = (0,0,-1)`

Figure 8 Changing the direction of the light. The vector is a direction, not a position; `(0,0,-1)` points down from above.

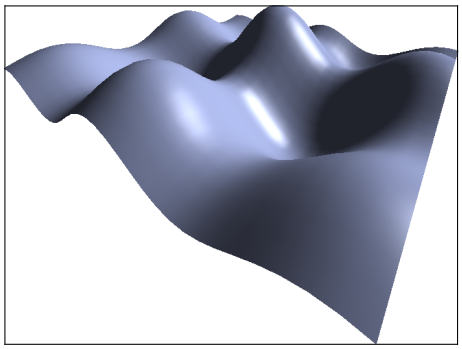


Default `intensity = 1`

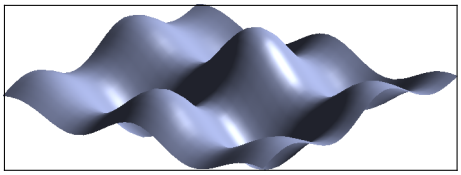


`intensity = 0.5`

Figure 9 The `intensity` key sets the strength of the light.

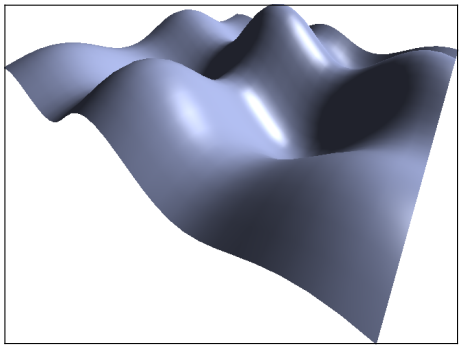


`projection = "perspective"`

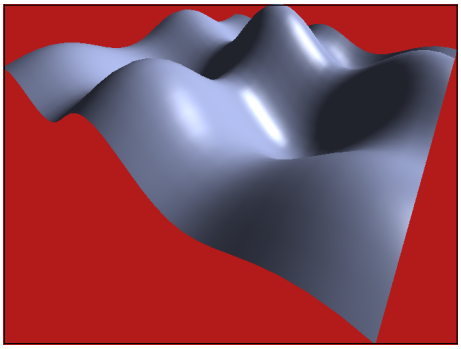


`projection = "orthographic"`

Figure 10 The `projection` keyword selects the projection type. The default is `projection = "perspective"`.



Default, not set



`backgroundcolor = (0.7,0.1,0.1)`

Figure 11 Changing the background color with `backgroundcolor`.

3 Materials

A material describes how a rendered object looks when the scene light reaches it. It is separate from the geometry itself: a surface, mesh, curve, or generated object can keep the same shape while getting a different color, highlight, ambient level, or transparency. When no material is selected, the scene uses the default material. Named materials are useful when several objects should share the same look.

We define a material with

```
lmt_scene_material [ ... ]
```

A material might look like this:

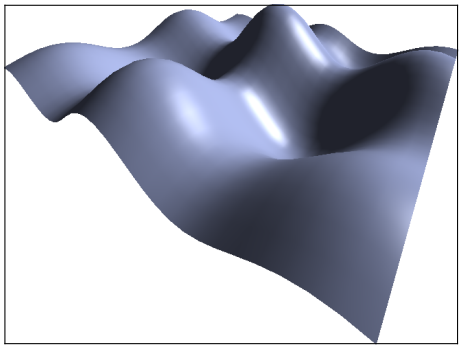
```
lmt_scene_material [
  name      = "softgreen",
  diffuse   = (.1,.8,.1),
  specular  = (.25,.25,.5),
  shininess = 10,
  ambient   = .28,
  opacity   = 1,
] ;
```

The renderer uses Lambertian diffuse shading with a Blinn–Phong specular term. The diffuse part follows the cosine rule for a matte surface. The optional highlight uses the halfway direction between the light and the eye. For the historical background, see the original **Phong paper** and **Blinn's refinement**. This is an illustration model, not a physically based renderer: the light is directional, there is no distance falloff or cast shadow, and final colors are clipped to the displayable range.

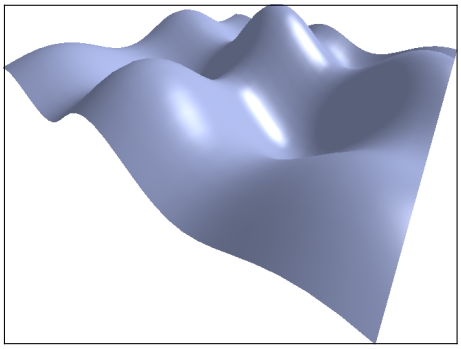
The public `light` vector gives the direction in which light travels, not the position of a lamp. Thus `(0,0,-1)` means that light comes from above and travels downwards. Its length does not matter. A surface is brightest when its normal points toward the light and darkest when it points away. The `ambient` value provides a floor, so the dark side need not be black. In rough terms, diffuse brightness is the ambient value plus an angle-dependent contribution, multiplied by `intensity` and limited to `0`–`1`. With `twosided = true`, the back side is lit as well.

The RGB `diffuse` triplet is the object's color. The renderer modulates it by the diffuse brightness and then adds the highlight from `specular`. Set `specular = (0,0,0)` for a matte object. Otherwise, `shininess` controls the size of the highlight: smaller values make it broad, while larger values make it small and sharp. The light `intensity` affects both the diffuse contribution and the highlight.

Start by choosing `diffuse`, keep `ambient` near its default, and adjust `light` if the shape is hard to read. Add a modest `specular` value for a glossy surface and increase `shininess` if the highlight is too broad.

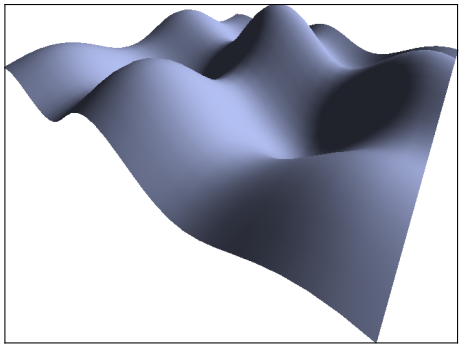


Default `ambient = .18`

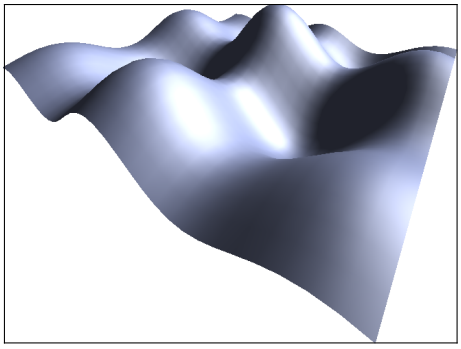


`ambient = .5`

Figure 12 The material's `ambient` value lifts the dark side without changing the direction of the light.



matte



glossy, `shininess = 8`

Figure 13 A `specular` color adds a highlight; `shininess` controls its size.

An object selects this material with `material = "softgreen"`. The remaining settings control reuse, lighting response, sidedness, transparency, and the raster size of point-like or line-like meshes.

If no material is selected, the scene uses the opaque blue-gray default material listed in the **Defaults** table. Defining a named material is therefore only necessary when the default appearance is not what you want or when several objects should share another appearance.

The material keywords are:

name gives the material a name so that surfaces and other objects can refer to it with their `material` keyword.

diffuse sets the main color that is affected by the directional light. Choose this color first.

specular sets the color of the specular highlight, the small bright part that can appear where the light reflects toward the eye.

shininess sets the sharpness of the specular highlight; larger values give a tighter highlight.

ambient sets the base light level that remains visible even where the directional light contributes little or nothing. Higher values make shaded parts less dark.

twosided also illuminates the back side

of an open surface. With `false`, a back-facing side receives only ambient light; with `true`, its normal is flipped for the lighting calculation. This helps with thin sheets or open surfaces viewed from either side.

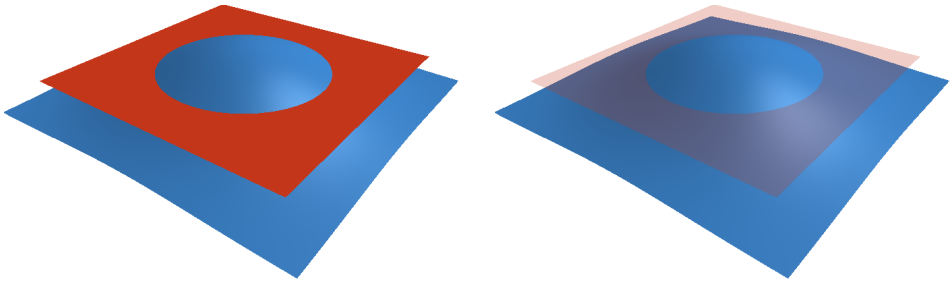
opacity sets the transparency of the material; `1` is fully opaque and `0` is fully transparent. Values in between are rendered in a separate transparent pass. A value around `.2` or `.3` is a useful starting point for a cutting plane.

ontop renders the material in the final on-top pass. It can be useful for helper geometry that should remain visible over the regular scene.

dx sets the horizontal thickness of curve, intersection, and other line-like or point-like material in raster pixels. For ordinary triangle surfaces this is normally left at the default.

dy sets the vertical thickness of curve, intersection, and other line-like or point-like material in raster pixels. For ordinary triangle surfaces this is normally left at the default.

Opacity is easiest to judge against geometry behind the material. The two scenes below keep the hill, plane, camera, and lighting fixed and change only the plane's `opacity`.



`opacity = 1` `opacity = .25`

Figure 14 The same cutting plane at two opacity values.

Two-sided materials

The effect of `twosided` is easiest to see on an open surface viewed from the back. The scene below shows two horizontal patches from underneath. The light is below them, so the patch with `twosided = false` receives no directional light, while the other patch is lit after its normal is flipped.

```
lmt_scene_start [
  width  = 10cm,
  crop   = true,
  eye    = (4,-6,-4),
  target = (0,0,0),
  up     = (0,0,1),
  light  = (0,0,1),
] ;

lmt_scene_material [
  name     = "backface",
  diffuse  = (.15,.45,.85),
```

```
  specular = (0,0,0),
  ambient  = 0,
  twosided = false,
] ;

lmt_scene_material [
  name     = "twoside",
  diffuse  = (.15,.45,.85),
  specular = (0,0,0),
  ambient  = 0,
  twosided = true,
] ;

lmt_scene_plane [
  center   = (-1.5,0,0),
  normal   = (0,0,1),
  usize    = 2.4,
  vsize    = 2.4,
  material = "backface",
] ;

lmt_scene_plane [
  center   = (1.5,0,0),
  normal   = (0,0,1),
  usize    = 2.4,
  vsize    = 2.4,
  material = "twoside",
] ;

lmt_scene_stop ;
```

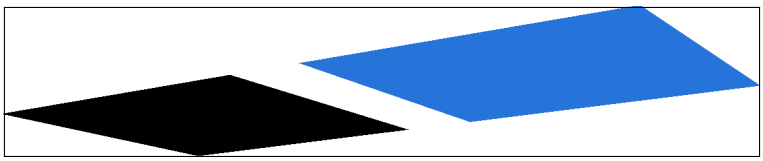


Figure 15 Viewing two horizontal patches from below: `twosided = false` receives no directional light, while `twosided = true` lights the back side.

The two patches have the same geometry and diffuse color. The difference comes only from the back-face lighting rule; setting `twosided` does not make an opaque surface transparent.

4 Surface graphs

A surface graph is the 3D version of a familiar function plot. For each point (x, y) in a chosen rectangle, the expression gives a height z. The renderer samples that rectangle, builds a triangle mesh from the sample points, and then shades the mesh like the other objects in the scene.

Add a surface graph $z = f(x, y)$ with

```
lmt_scene_surface [ ... ]
```

A graph surface might look like this:

```
lmt_scene_surface [
  code = "sin(x)*cos(y)",
  xmin = -pi,
  xmax =  pi,
  ymin = -pi,
  ymax =  pi,
  nx   = 32,
  ny   = 32,
] ;
```

The `code` value gives the height. The four bounds select the part of the (x, y) plane to sample, and `nx` and `ny` control the mesh density. These are the only object settings needed for a simple graph. The other settings handle naming, lighting normals, and material selection.

A normal is a direction perpendicular to the surface. It tells the lighting model which way each part of the mesh faces. Without derivative expressions, the renderer estimates normals from the sampled mesh. The `normal` setting chooses how to use them: `smooth` gives a continuous appearance, `flat` shows the triangle faces, and `up` uses the same upward direction everywhere.

The graph keywords are:

id gives the surface a name so that it can be referred to elsewhere, for example by an intersection.

code gives the function $z = f(x, y)$ as an expression in `x` and `y`. It is evaluated at each sampled point in the domain.

dfdx gives the slope in the `x` direction. It is optional; supply it together with `dfdy` when analytic slope information is available to improve the lighting normals.

dfdy gives the slope in the `y` direction. Together with `dfdx`, it can be used when calculating more accurate surface normals.

normal selects how normals are made for lighting. `smooth` interpolates normals across the mesh, `flat` exposes the triangle faces, and `up` uses `(0,0,1)` everywhere.

xmin sets the lower bound of the sampled rectangle in the `x` direction.

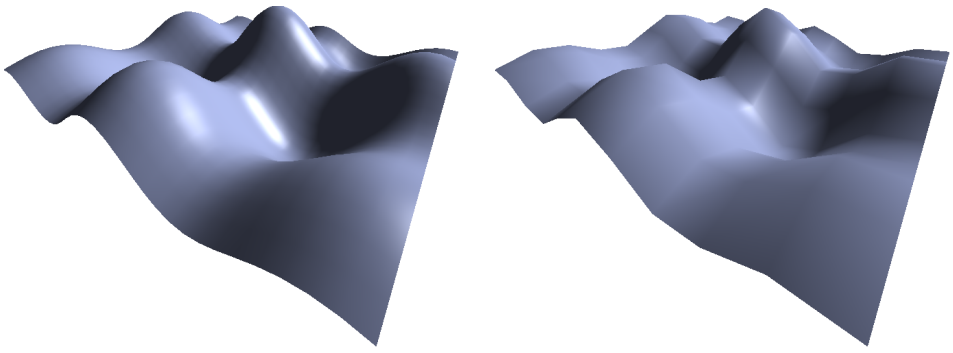
xmax sets the upper bound of the sampled rectangle in the `x` direction.

ymin sets the lower bound of the sam-

pled rectangle in the y direction.
ymax sets the upper bound of the sampled rectangle in the y direction.
nx sets the number of mesh intervals in the x direction. Higher values give more detail and more triangles.
ny sets the number of mesh intervals in the y direction. Higher values give more detail and more triangles.
material selects the named material used when rendering the surface.

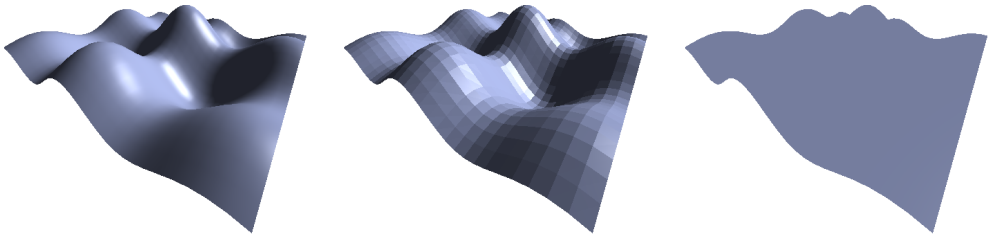
The examples below keep the function fixed and change one choice at a time: mesh resolution, normal appearance, slope information, material, and cropping. That makes the effect of each setting easier to see.

```
lmt_scene_start [
  width      = 6cm,
  crop      = true,
] ;

lmt_scene_surface [
  code = ".55*exp(-.07*(x*x+y*y))*(cos(2.2*x)+sin(2.0*y))",
  xmin = -3,
  xmax =  3,
  ymin = -3,
  ymax =  3,
] ;

lmt_scene_stop ;
```



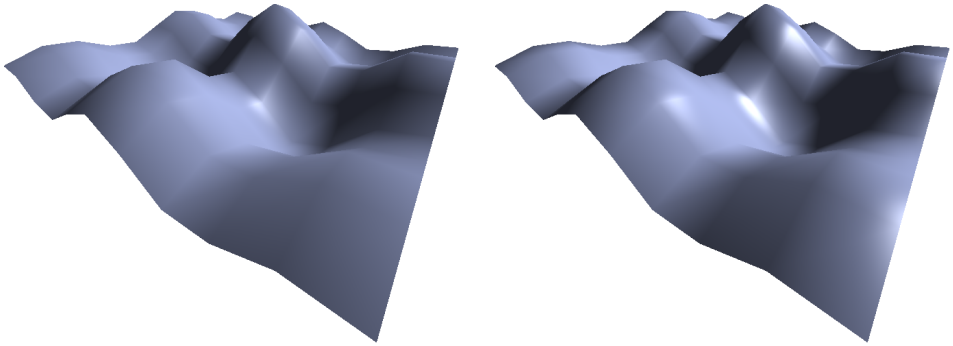
default nx = 32, ny = 32 nx = 10, ny = 10

Figure 16 Changing the graph mesh interval counts.



normal = "smooth" normal = "flat" normal = "up"

Figure 17 Different normal modes. normal = "smooth" is the default.

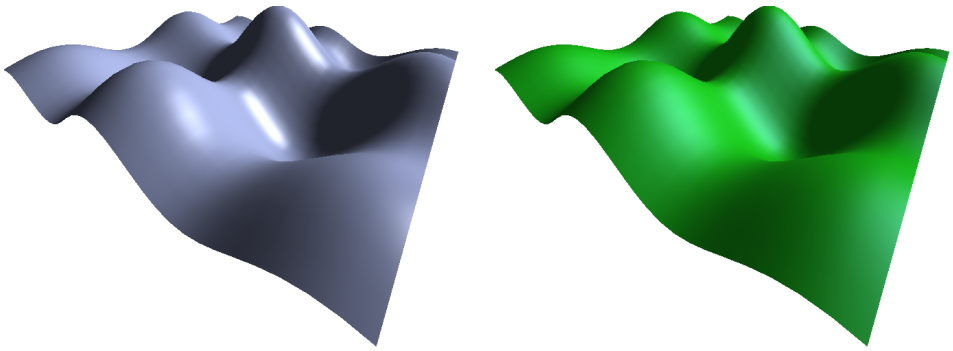


`nx = 10, ny = 10` also `dfdx = ..., dfdy = ...`

Figure 18 Comparing a surface with and without slope expressions.

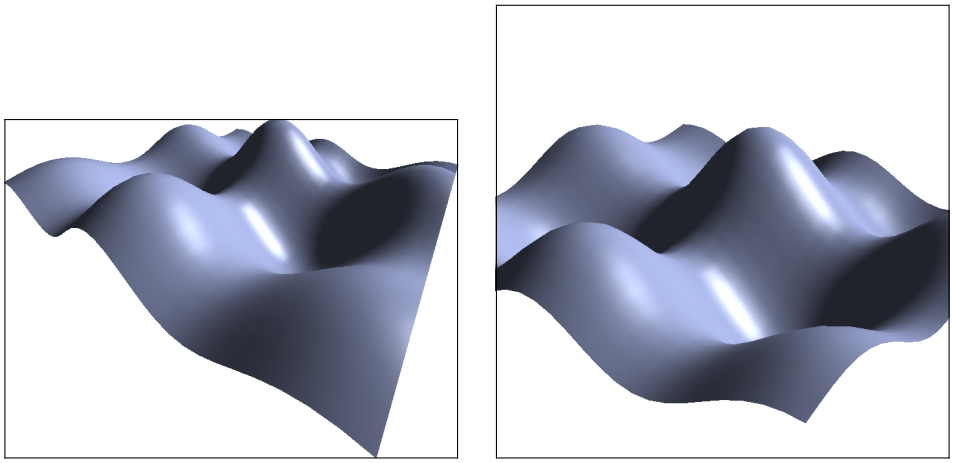
Change a surface's appearance by defining a material and selecting it with the `material` keyword.

```
lmt_scene_material [
    name      = "mymaterial",
    diffuse   = (.1,.8,.1),
    specular  = (.25,.25,.5),
    shininess = 10,
    ambient   = .28,
    opacity   = 1,
] ;
```



default `material = "mymaterial"`

Figure 19 Changing a surface's appearance with a material.



default `crop = false`

Figure 20 Cropping the rendered result.

5 Parametric surfaces

A parametric surface is built from two parameters rather than from a height function. For each sampled pair (u, v), three expressions give a point (x, y, z) in space. This covers shapes that are not naturally described as $z = f(x, y)$, such as tori, spheres, tubes, and twisted patches.

Add a parametric surface with

```
lmt_scene_parametric [ ... ]
```

A parametric surface might look like this:

```
lmt_scene_parametric [
  x    = "(1+.35*cos(v))*cos(u)",
  y    = "(1+.35*cos(v))*sin(u)",
  z    = ".35*sin(v)",
  umin = 0,
  umax = 2*pi,
  vmin = 0,
  vmax = 2*pi,
  nu   = 64,
  nv   = 32,
] ;
```

The coordinate expressions describe the shape. The `u` and `v` bounds choose the part of the parameter plane to sample, and `nu` and `nv` control the mesh resolution. The parameters `u` and `v` are just variables; they need not be spatial coordinates. The other settings handle naming, lighting normals, and material selection.

The parametric-surface keywords are:

id gives the surface a name so that it can be referred to elsewhere, for example by an intersection.

x gives the `x(u,v)` coordinate as an expression in `u` and `v`. It is evaluated at each sampled parameter pair.

y gives the `y(u,v)` coordinate as an expression in `u` and `v`. It is evaluated at each sampled parameter pair.

z gives the `z(u,v)` coordinate as an expression in `u` and `v`. It is evaluated at each sampled parameter pair.

xu/yu/zu describe the tangent direction when `u` changes. They are optional; together with the other three tangent components they provide analytic normals.

xv/yv/zv describe the tangent direction when `v` changes. Give all six derivative expressions together when analytic normals are desired. If they are omitted, normals are estimated from the sampled triangles.

normal selects how normals are made for lighting. `smooth` interpolates normals across the mesh, `flat` exposes the triangle faces, and `up` uses `(0,0,1)` everywhere.

normalstep sets the finite-difference step used when analytic tangent expressions are not supplied. A positive value can stabilize numerical normals for very

small or very large parameter domains; otherwise an automatic domain-scaled step is used.

umin sets the lower bound of the sampled rectangle in the u direction.

umax sets the upper bound of the sampled rectangle in the u direction.

vmin sets the lower bound of the sampled rectangle in the v direction.

vmax sets the upper bound of the sampled rectangle in the v direction.

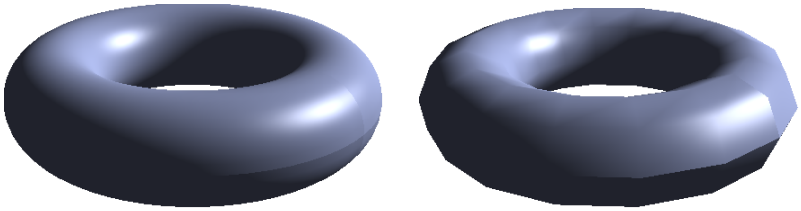
nu sets the number of mesh intervals in the u direction. Higher values give more detail and more triangles.

nv sets the number of mesh intervals in the v direction. Higher values give more detail and more triangles.

material selects the named material used when rendering the surface.

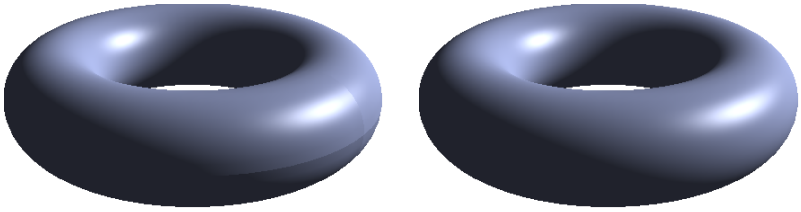
The examples below keep the torus formula fixed and change the mesh resolution, derivative information, or normal mode.

```
lmt_scene_start [
  width      = 5cm,
  crop       = true,
] ;

lmt_scene_parametric [
  x = "(1+.35*cos(v))*cos(u)",
  y = "(1+.35*cos(v))*sin(u)",
  z = ".35*sin(v)",
] ;

lmt_scene_stop ;
```



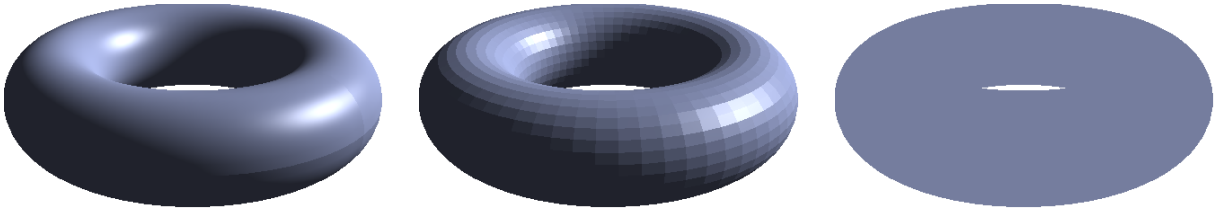
default nu = 16, nv = 8

Figure 21 Changing the parametric mesh resolution.



no derivatives with derivatives

Figure 22 Giving derivatives for the parametric normals.



default `normal = "flat"` `normal = "up"`

Figure 23 Changing the normal mode of a parametric surface.

6 Implicit surfaces

An implicit surface is described by an equation instead of explicit coordinates. The usual form is $F(x, y, z) = c$: all points where the function has the chosen value belong to the surface. This suits shapes that are more natural to describe as a condition in space, for example a sphere with $x^2 + y^2 + z^2 = 1$.

The renderer samples a finite 3D box and builds a triangle mesh where the function crosses the chosen iso value. This is a local approximation, not a global solution of the equation. The box and sampling resolution matter: parts outside the box are not seen, and a feature can be missed when the grid is too coarse to reveal it.

Add an implicit surface with

```
lmt_scene_implicit [ ... ]
```

An implicit surface might look like this:

```
lmt_scene_implicit [
  code = "x*x + y*y + z*z - 1",
  xmin = -1.2,
  xmax =  1.2,
  ymin = -1.2,
  ymax =  1.2,
  zmin = -1.2,
  zmax =  1.2,
  nx   = 40,
  ny   = 40,
  nz   = 40,
  iso  = 0,
] ;
```

Here the surface is where `code` evaluates to `iso`; in this example, that value is `0`. The same sphere could be written as `x*x + y*y + z*z` with `iso = 1`; subtracting `1` in the expression and keeping `iso = 0` is just a convenient alternative. The bounds define the box to search, and `nx`, `ny`, and `nz` control its sampling. All three counts are required and must be larger than one. The other settings handle naming, lighting normals, and material selection.

You can also supply the implicit function as a LUA function through `pfunction`. It receives `x`, `y`, and `z`, and returns the scalar value. Return `false` when a sampled point lies outside the part of the domain to draw. This lets you describe surfaces on specific, nonrectangular domains.

The function is defined in the extended `xmath` table; inside a scene expression the same table is reached as `math`, so the reference below uses `pfunction = "math.paraboloid_domain"`.

```
function xmath.paraboloid_domain(x,y,z)
```

```
    local r2 = x^2 + y^2
    if r2 < 0.25 or r2 > 1 then
        return false
    end
    return z - r2
end
```

This uses that function to draw a paraboloid with a circular hole:

```
lmt_scene_start [
  width       = 6cm,
  crop        = true,
  resolution  = 100,
  supersample = 2,
  projection  = "perspective",
  light       = (0,-1,-2),
  eye         = (3,-2,3),
  intensity   = 1.2,
] ;

lmt_scene_implicit [
  pfunction = "math.paraboloid_domain",
  xmin      = -1.25,
  xmax      =  1.25,
  ymin      = -1.25,
  ymax      =  1.25,
  zmin      = -1.25,
  zmax      =  1.25,
  nx        = 120,
  ny        = 120,
  nz        = 120,
] ;

lmt_scene_stop ;
```

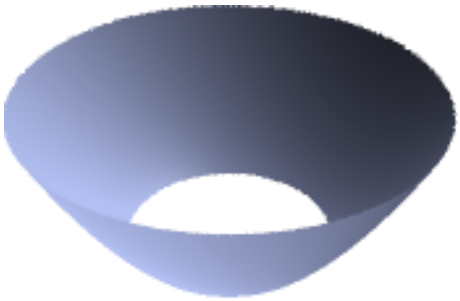


Figure 24 An implicit paraboloid restricted by a LUA-defined circular domain.

For an annular domain, add a second test such as `or r2 > 1` to the condition. Because the predicate is sampled on the implicit grid, increase `nx` and `ny` when a curved domain boundary needs more detail.

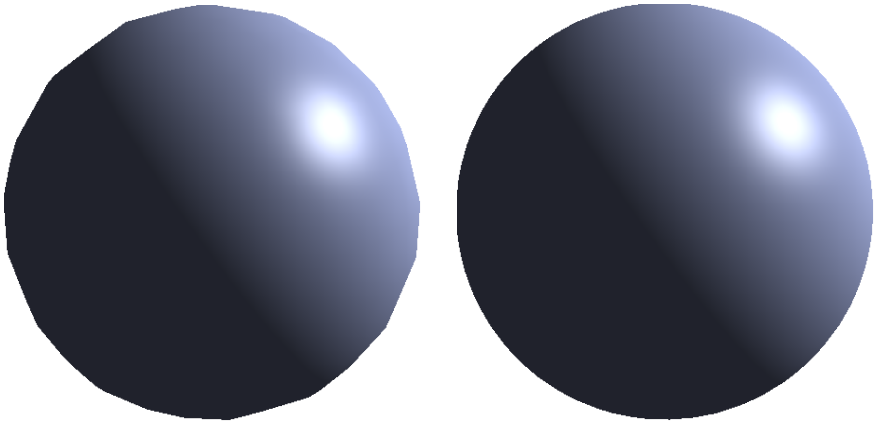
The example uses approximately `120^3` samples. This is a practical balance for a manual example: `200^3` gives a finer implicit grid but requires substantially more time and memory, especially when the document is compiled repeatedly (which is why we can cache).

Higher values of `nx`, `ny`, and `nz` add detail but can make rendering much slower. A grid cell larger than a thin feature can miss it, especially when the feature is small or disconnected. The finite bounds are a hard clipping box, so geometry outside them is not extracted. Start with a box that contains the part of the surface you need, then increase the counts until the visible structure stops changing. Lower counts are useful for draft renders. Unlike the graph and parametric commands, these counts describe sampling points, not mesh intervals.

When analytic derivatives are available, give all three of `dfdx`, `dfdy`, and `dfdz`. They are used to make smoother and more accurate lighting normals. They improve the normals on the sampled mesh, but do not replace the mesh extraction or recover a component that the grid has missed.

The comparison below keeps the equation, bounds, and derivatives fixed and changes only the grid resolution:

```
lmt_scene_start [
  width = 5.5cm,
  crop  = true,
] ;

lmt_scene_implicit [
  code = "x*x + y*y + z*z - 1",
  dfdx = "2*x",
  dfdy = "2*y",
  dfdz = "2*z",
  xmin = -1.2,
  xmax =  1.2,
  ymin = -1.2,
  ymax =  1.2,
  zmin = -1.2,
  zmax =  1.2,
  nx   = 10,
  ny   = 10,
  nz   = 10,
  iso  = 0,
] ;

lmt_scene_stop ;
```



coarse `nx = ny = nz = 10` fine `nx = ny = nz = 32`

Figure 25 The same implicit sphere sampled on a coarse and a fine grid.

The coarse mesh shows the faceting caused by its larger cells. The fine mesh follows the curved surface more closely, but costs more to sample.

The implicit-surface keywords are:

id gives the surface a name so that it can be referred to elsewhere, for example by an intersection.

code gives $F(x, y, z)$ as an expression in `x`, `y`, and `z`. The extracted surface is where this function reaches the chosen `iso` value.

pfunction gives a LUA expression that evaluates to a function of `x`, `y`, and `z`. It takes precedence over `code`; the function can return `false` to omit a sampled point and restrict the surface to a nonrectangular domain.

splitdepth sets the maximum number of times a cube is subdivided when its corners contain a mixture of numeric and nonnumeric results from the point function. This provides local refinement at a domain boundary made with `pfunction`; an integer value of `1` subdivides a mixed cube once, while larger values allow deeper refinement. The default is `2`, and values outside the range `1`–`5` use that default. Cubes whose corners are all numeric are polygonized immediately, while cubes with no numeric corners are discarded, so this setting does not refine an ordinary implicit function that returns a number everywhere.

nfunction gives a LUA expression that evaluates to a function returning the normal components `(nx,ny,nz)` for `x`, `y`, and `z`. It takes precedence over the three derivative expressions when normals are calculated, while `code` or `pfunction` still defines the surface.

dfdx gives the partial derivative with respect to `x`. When all three derivatives are given, they are used for smoother and more accurate normals.

dfdy gives the partial derivative with respect to `y`. When all three derivatives are given, they are used for smoother and more accurate normals.

dfdz gives the partial derivative with respect to `z`. When all three derivatives are given, they are used for smoother and more accurate normals.

normal selects how normals are made for lighting. `smooth` interpolates normals across the mesh, `flat` exposes the triangle faces, `up` uses `(0,0,1)` everywhere,

and `auto` estimates the gradient of the implicit function from nearby samples.

normalstep sets the finite-difference step used for numerical gradient normals and for normal queries. A positive value overrides the automatic step; otherwise the step is derived from the implicit bounding box.

xmin sets the lower bound of the sampled box in the `x` direction.

xmax sets the upper bound of the sampled box in the `x` direction.

ymin sets the lower bound of the sampled box in the `y` direction.

ymax sets the upper bound of the sampled box in the `y` direction.

zmin sets the lower bound of the sampled box in the `z` direction.

zmax sets the upper bound of the sampled box in the `z` direction.

nx sets the number of sampling points in the `x` direction. It is required and has to be larger than one. Higher values can capture more detail but take more work.

ny sets the number of sampling points in the `y` direction. It is required and has to be larger than one. Higher values can capture more detail but take more work.

nz sets the number of sampling points in the `z` direction. It is required and has to be larger than one. Higher values can capture more detail but take more work.

iso sets the function value that will be extracted as the surface. The default sphere example uses `iso = 0`.

material selects the named material used when rendering the surface.

7 Parametric curves

A parametric curve is a one-parameter version of a parametric surface. For each sampled value of t, three expressions give a point (x, y, z) in space. Use curves for paths, space curves, helper lines, and any curve that should use the same depth handling as the surrounding surfaces.

The curve is sampled into points and rendered as a connected line in the scene raster, so surfaces can hide parts behind them. Curves obtain their color from their selected material, not from a separate curve-color setting; the material's `dx` and `dy` values control the line-like thickness in raster pixels.

Add a parametric curve with

```
lmt_scene_curve [ ... ]
```

A curve might look like this:

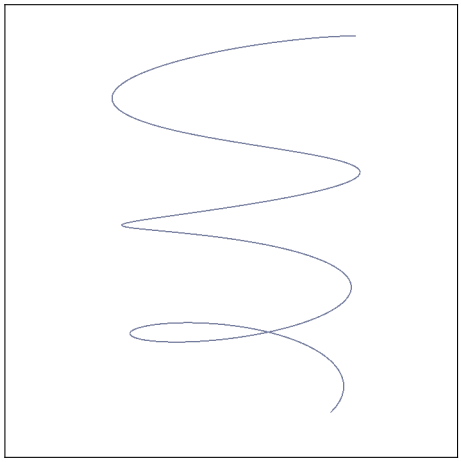
```
lmt_scene_curve [
  id   = "helix",
  x    = "cos(t)",
  y    = "sin(t)",
  z    = ".15*t",
  tmin = 0,
  tmax = 6*pi,
  nt   = 160,
] ;
```

The coordinate expressions describe the path. The `t` bounds choose the part of the parameter line to sample, and `nt` or `tstep` controls the number of samples. More samples make a curved path smoother, but also add more segments to the rendered scene.

The curve keywords are:

id gives the curve a name in the scene.
x gives the `x(t)` coordinate as an expression in `t`. It is evaluated at each sampled parameter value.
y gives the `y(t)` coordinate as an expression in `t`. It is evaluated at each sampled parameter value.
z gives the `z(t)` coordinate as an expression in `t`. It is evaluated at each sampled parameter value.
tmin sets the lower bound of the sampled parameter interval.
tmax sets the upper bound of the sampled parameter interval.
nt sets the number of sampling points along the curve. Both endpoints are included, so the visible step size is based on `nt-1` intervals.
tstep requests an approximate sampling step in the parameter direction. When it is positive, the point count is derived from the interval length and the actual step is adjusted so that the curve still ends at `tmax`.
material selects the material used when rendering the curve. For curves, the

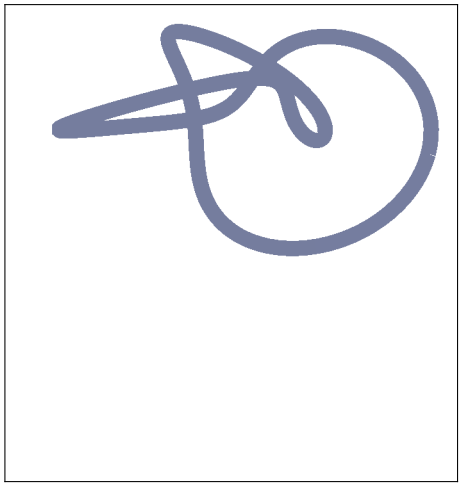
material controls the visible color, opacity, on-top behavior, and the raster size settings used for point-like or line-like meshes.



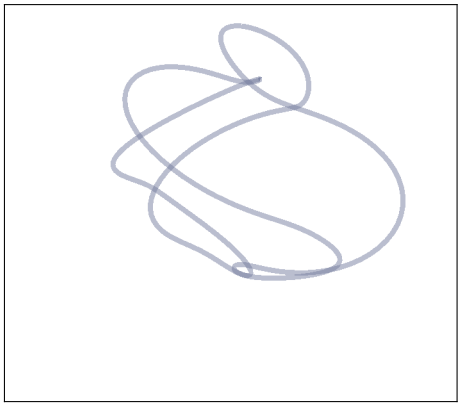
default

custom material

Figure 26 A space curve with the default and a custom line material.



knot



woven curve

Figure 27 Two sampled space curves.

8 Intersections

An intersection finds where two objects already in the scene meet. The result is added as point-like or segment-like geometry and can be rendered with the source objects. Use intersections for level curves, cutting planes, and the places where two sampled surfaces pass through each other.

Intersections are computed from sampled geometry, so the result is a sampled visualization rather than an exact geometric operation. Its appearance depends on the source resolution and the selected method; finer source meshes usually give a cleaner result. The `tolerance` is a small numerical threshold used when crossings are classified and nearby results are joined; the useful value depends on the scale of the objects. The most dependable case is a generated surface or mesh together with a finite plane. Intersections between two generated surfaces are still experimental and may produce no result for some object types.

Request an intersection with

```
lmt_scene_intersection [ ... ]
```

An intersection might look like this:

```
lmt_scene_intersection [
  id        = "slice",
  first     = "surface",
  second    = "cutplane",
  tolerance = .001,
  material  = "default",
] ;
```

Here `surface` and `cutplane` are the `id` values of objects that have already been added to the same scene. A graph or mesh can be intersected with a finite plane patch. Mesh–mesh requests are experimental; for a cutting curve, start with the plane form shown here.

The intersection keywords are:

id gives the intersection request a name. The important names for this lookup are the source object names given by `first` and `second`.

first selects the first object in the intersection. It should match the `id` of an earlier object in the same scene.

second selects the second object in the intersection. It should match the `id` of another earlier object in the same scene.

tolerance sets the numerical threshold used by the selected intersection method. It should normally remain small relative to the coordinate scale of the two objects.

material selects the material used when rendering the generated intersection result. For these line-like results, the material controls the visible color, opacity, on-top behavior, and raster size settings.

The complete example below uses a parametric torus and a finite horizontal plane. By

the time the intersection is calculated, the torus is a triangle mesh, so this exercises the mesh–plane path. The red loops are the generated line-like intersection. The plane is translucent so that both the mesh and the result remain visible.

```
lmt_scene_start [
  width      = 6cm,
  crop       = true,
  eye        = (3,-4,2.6),
  target     = (0,0,0),
  light      = (-1,-1,-2),
] ;

lmt_scene_material [
  name      = "torus",
  diffuse   = (.20,.48,.82),
  ambient   = .25,
  opacity   = 1,
] ;

lmt_scene_material [
  name      = "slice",
  diffuse   = (.72,.78,.68),
  opacity   = .20,
  twosided  = true,
] ;

lmt_scene_material [
  name      = "section",
  diffuse   = (.95,.05,.03),
  dx        = 6,
  dy        = 6,
  ontop     = true,
] ;

lmt_scene_parametric [
  id       = "torus",
  x        = "(1+.35*cos(v))*cos(u)",
  y        = "(1+.35*cos(v))*sin(u)",
  z        = ".35*sin(v)",
  umin     = 0,
  umax     = 2*pi,
  vmin     = 0,
  vmax     = 2*pi,
  nu       = 48,
  nv       = 24,
  material = "torus",
] ;
```

```
lmt_scene_plane [
  id       = "slice",
  normal   = (0,0,1),
  center   = (0,0,0),
  usize    = 3,
  vsize    = 3,
  material = "slice",
] ;

lmt_scene_intersection [
  id        = "section",
  first     = "torus",
  second    = "slice",
  tolerance = .001,
  material  = "section",
] ;

lmt_scene_stop ;
```

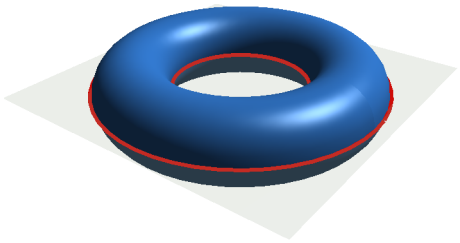


Figure 28 A tested mesh-plane intersection of a torus and a finite plane.

9 Reference geometry

Reference geometry provides finite planes, segments, grids, axes, and ticks that can be mixed with sampled surfaces and meshes. These helpers use raster-pixel widths where they draw lines, so their visual weight is independent of the document's physical dimensions. The line-like helpers are depth-tested with the other scene geometry; grids, axes, and ticks additionally describe an origin, direction vectors, bounds or positions, color, and width.

Planes

A plane object is a flat rectangular patch in 3D space. Use it as a visible reference sheet, a transparent cutting plane, or a simple surface for comparison. The plane is positioned by a point and oriented by a normal direction; its two sizes set the finite visible patch.

The patch uses the same camera, light, material, and depth handling as the other scene objects. Its local u and v directions are chosen automatically from the normal.

Add a plane with

```
lmt_scene_plane [ ... ]
```

A plane might look like this:

```
lmt_scene_plane [
  id       = "slice",
  normal   = (0,0,1),
  center   = (0,0,.35),
  usize    = 3,
  vsize    = 3,
  material = "default",
] ;
```

The patch is centered at `(0,0,.35)` and has the normal `(0,0,1)`. The two sizes set the visible rectangle in directions perpendicular to that normal. For a helper or cutting surface, a transparent or on-top material often makes the plane easier to read with the rest of the scene.

The plane keywords are:

id gives the plane a name so that it can be referred to elsewhere, for example by an intersection.

normal sets the plane normal. It gives the orientation of the plane and is normalized internally, so it does not have to be a unit vector.

center sets the center point of the finite plane patch.

usize sets the size of the plane patch in the local u direction. This direction lies in the plane and is chosen automatically from the normal.

vsize sets the size of the plane patch in

the local v direction. Together with `usize` it determines the visible rectangle.

material selects the material used when rendering the plane. This is where color, transparency, and on-top behavior are normally controlled.

Here is a standalone plane scene:

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
  target     = (0,0,0),
] ;

lmt_scene_material [
  name      = "plane",
  diffuse   = (.25,.55,.85),
  twosided  = true,
  opacity   = .75,
] ;

lmt_scene_plane [
  id       = "plane",
  normal   = (0,0,1),
  center   = (0,0,0),
  usize    = 3,
  vsize    = 2,
  material = "plane",
] ;

lmt_scene_stop ;
```

Figure 29 A standalone finite plane patch.

Segments

A segment is a straight line between two 3D points. Use it for explicit helper lines: marking a distance, showing a direction, or highlighting an edge that is not part of a surface.

Segments are rendered as line-like scene geometry, so they take part in the same depth handling as curves and intersections. A segment can therefore disappear behind an opaque surface unless it is styled through the usual on-top mechanisms.

Add a visible 3D segment with

```
lmt_scene_segment [ ... ]
```

A segment might look like this:

```
lmt_scene_segment [
  id     = "baseline",
  first  = (-1,0,0),
  second = ( 1,0,0),
  color  = red,
  width  = 1,
] ;
```

The segment keywords are:

id gives the segment helper a scene name.
first sets the first endpoint as a 3D point.
second sets the second endpoint as a 3D point.
color sets the segment color.
width sets the rendered segment width in raster pixels. This helper width controls the raster thickness directly, rather than the document width.

Here is a standalone segment scene:

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
  target     = (0,0,.3),
] ;

lmt_scene_segment [
  id     = "segment",
  first  = (-1.4,-.5,0),
  second = ( 1.4,.8,1),
  color  = red,
  width  = 4,
] ;

lmt_scene_stop ;
```

Figure 30 A standalone 3D segment with a four-pixel raster width.

Grids

A grid is a set of straight helper lines in a plane. It is defined by an origin and two direction vectors, `uaxis` and `vaxis`. The bounds and step sizes then say where the grid lines are placed in those two local directions.

They work well as reference planes under or through a surface, or as a lightweight coordinate frame when a full axis setup would be too much.

Add a grid with

```
lmt_scene_grid [ ... ]
```

A grid might look like this:

```
lmt_scene_grid [
  origin = (0,0,0),
  uaxis  = (1,0,0),
  vaxis  = (0,1,0),
  umin   = -2,
  umax   =  2,
  ustep  = .5,
  vmin   = -2,
  vmax   =  2,
  vstep  = .5,
  color  = (.62,.62,.62),
  width  = 1,
] ;
```

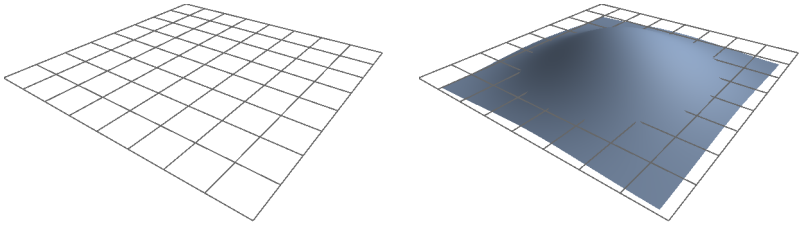
The grid keywords are:

id gives the grid helper a scene name.
origin sets the point where the local u and v coordinates start from.
uaxis sets the first direction vector of the grid. Its length also sets the scale of one unit in the local u direction.
vaxis sets the second direction vector of the grid. Its length also sets the scale of one unit in the local v direction.
umin sets the lower bound in the local u direction.
umax sets the upper bound in the local u direction.
ustep sets the spacing between grid lines in the local u direction.
vmin sets the lower bound in the local v direction.
vmax sets the upper bound in the local v direction.
vstep sets the spacing between grid lines in the local v direction.
color sets the color of the visible grid.
width sets the line width of the visible grid in raster pixels.

The examples show the grid by itself and with an opaque surface. In the second case it is ordinary scene geometry, so the surface hides the parts behind it. Both examples use the grid from the code above.

Axes

An axis overlay draws the three coordinate axes as straight 3D segments. It is a compact way to show orientation and scale without drawing a full grid. The axes share one origin, but each direction vector can be changed, so the overlay can also be used for a local coordinate frame.



grid only with surface

Figure 31 A grid on its own and depth-tested with a surface.

Add an axis with

```
lmt_scene_axis [ ... ]
```

An axis overlay might look like this:

```
lmt_scene_axis [
  origin = (0,0,0),
  xaxis  = (1,0,0),
  yaxis  = (0,1,0),
  zaxis  = (0,0,1),
  xmin   = -2,
  xmax   =  2,
  ymin   = -2,
  ymax   =  2,
  zmin   =  0,
  zmax   =  1,
  color  = black,
  width  = 1,
] ;
```

The axis keywords are:

id gives the axis helper a scene name.
origin sets the common origin point of the three axes.
xaxis sets the direction vector of the local x axis. Its length sets the scale of one local x unit.
yaxis sets the direction vector of the local y axis. Its length sets the scale of one local y unit.
zaxis sets the direction vector of the local z axis. Its length sets the scale of one local z unit.
xmin sets the lower bound of the visible local x axis.
xmax sets the upper bound of the visible local x axis.
ymin sets the lower bound of the visible local y axis.
ymax sets the upper bound of the visible local y axis.
zmin sets the lower bound of the visible local z axis.
zmax sets the upper bound of the visible local z axis.
color sets the color of the visible axes.
width sets the line width of the visible axes in raster pixels.

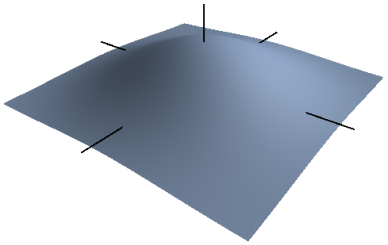


Figure 32 Axes depth-tested together with a surface.

Ticks

Ticks are short helper marks placed along one direction. They are normally used together with an axis: `axis` says where the tick positions lie, and `tickaxis` says which way the small marks extend.

The tick command only draws the marks themselves; labels, when needed, are added separately.

Add ticks with

```
lmt_scene_ticks [ ... ]
```

A tick setup might look like this:

```
lmt_scene_ticks [
  origin   = (0,0,0),
  axis     = (1,0,0),
  tickaxis = (0,1,0),
  first    = -2,
  last     =  2,
  step     =  1,
  size     = .15,
  color    = red,
  width    = 1,
] ;
```

The tick keywords are:

id gives the tick helper a scene name.
origin sets the point from which tick positions are measured.
axis sets the direction vector along which the ticks are placed. Its length sets the scale of one tick-position unit.
tickaxis sets the direction vector along which each tick mark extends.
first sets the first tick position on the axis.
last sets the last tick position on the axis.
step sets the spacing between consecutive tick positions.
size sets the length of each tick mark, measured along `tickaxis`.
color sets the color of the visible ticks.
width sets the line width of the visible ticks in raster pixels.

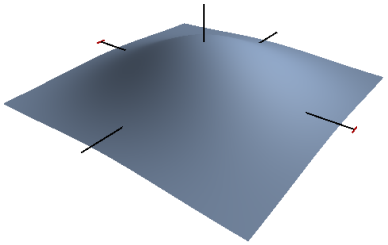


Figure 33 Axes and ticks depth-tested together with a surface.

10 Putting it all together

The complete example below brings the basic workflow together. It combines a named surface, three named materials, a finite plane, their sampled intersection, and coordinate axes. The intersection refers to the two objects by their `id` values.

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3.2,-4.4,3.0),
  target     = (0,0,.1),
] ;

lmt_scene_material [
  name      = "hill",
  diffuse   = (.25,.55,.85),
  specular  = (.12,.12,.12),
  shininess = 12,
  ambient   = .22,
] ;

lmt_scene_material [
  name      = "cut",
  diffuse   = (1,.65,.08),
  opacity   = .28,
  twosided  = true,
] ;

lmt_scene_material [
  name      = "section",
  diffuse   = (.9,.08,.06),
  dx        = 5,
  dy        = 5,
  ontop     = true,
] ;

lmt_scene_surface [
  id       = "hill",
  code     = ".8*exp(-.35*(x*x+y*y)) - .25",
  dfdx     = "-.56*x*exp(-.35*(x*x+y*y))",
  dfdy     = "-.56*y*exp(-.35*(x*x+y*y))",
  xmin     = -2,
  xmax     =  2,
  ymin     = -2,
  ymax     =  2,
  nx       = 56,
  ny       = 56,
```

```
  material = "hill",
] ;

lmt_scene_plane [
  id       = "cutplane",
  normal   = (0,0,1),
  center   = (0,0,.15),
  usize    = 3.6,
  vsize    = 3.6,
  material = "cut",
] ;

lmt_scene_intersection [
  id        = "section",
  first     = "hill",
  second    = "cutplane",
  tolerance = .01,
  material  = "section",
] ;

lmt_scene_axis [
  origin = (0,0,0),
  xaxis  = (1,0,0),
  yaxis  = (0,1,0),
  zaxis  = (0,0,1),
  xmin   = -2,
  xmax   =  2,
  ymin   = -2,
  ymax   =  2,
  zmin   = -.4,
  zmax   =  .8,
  color  = black,
  width  = 1,
] ;

lmt_scene_stop ;
```

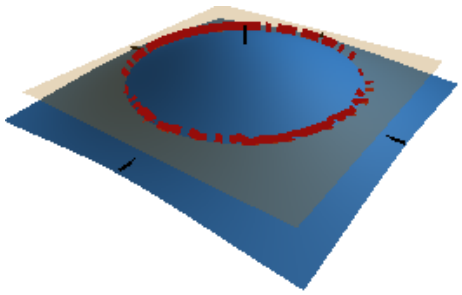


Figure 34 A complete scene: a surface, a cutting plane, their intersection, and coordinate axes.

Add the surface and plane before the intersection: `first` and `second` refer to objects that already exist in the scene. The axis has no such dependency, so it comes after the intersection to make the construction order clear. All four objects use the same camera, lighting, materials, and depth handling.

The intersection is a sampled graph–plane result for visual cutting curves. Its numerical and tolerance-related limits are described in the **Intersections** section. Use this scene as a workflow example; the individual sections remain the command reference.

The `width` above is the displayed document size. Since no explicit `bytewidth` or `byteheight` is given, `resolution` determines the raster size used before the result is placed at that width. With `crop = true`, the displayed bitmap is fitted to the projected bounds of the complete scene, including the plane, intersection, and axes.

`lmt_scene_stop` renders the scene when no earlier render was requested. If ordinary METAPOST labels or markers need the rendered bounds or projected positions, call `lmt_scene_render` after all scene objects and before `lmt_scene_stop`; the query helpers are described in the **advanced annotation section**.

11 Rendering and annotation helpers (advanced)

These helpers work on a rendered scene. Use them when ordinary METAPOST material must be positioned relative to the result. Otherwise, `lmt_scene_stop` renders and finishes the scene for you.

`lmt_scene_render` renders the scene once. After rendering, `lmt_scene_width` and `lmt_scene_height` give the pixel dimensions of the final raster. They need not equal the document dimensions requested with `width` and `height`. The `lmt_scene_bytemap` and `lmt_scene_bounds` helpers give access to the selected bytemap and its displayed bounds.

The projection helper `lmt_scene_project_point` maps a 3D point to the picture. The visibility helper `lmt_scene_project_visible` reports whether that point is visible at its projected position. Their shorter aliases are `lmt_scene_point` and `lmt_scene_visible`.

For an object whose `id` is known, `lmt_scene_calculate_normal` returns a normal at a coordinate or parameter value; its shorter alias is `lmt_scene_normal`. Make these queries after `lmt_scene_render` and before `lmt_scene_stop`.

Here is a complete query-and-annotation example. The scene is rendered first; the rest is ordinary METAPOST code using the returned bounds, projected point, visibility result, raster dimensions, and normal.

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
] ;

lmt_scene_parametric [
  id       = "surface",
  x        = "u",
  y        = "v",
  z        = ".7*exp(-.6*(u*u+v*v)) - .2",
  xu       = "1",
  yu       = "0",
  zu       = "-.84*u*exp(-.6*(u*u+v*v))",
  xv       = "0",
  yv       = "1",
  zv       = "-.84*v*exp(-.6*(u*u+v*v))",
  umin     = -1.8,
  umax     =  1.8,
  vmin     = -1.8,
  vmax     =  1.8,
  nu       = 48,
```

```
  nv       = 48,
] ;

lmt_scene_render ;

path     helper_bounds ;
pair     helper_probe ;
pair     helper_label ;
triplet  helper_normal ;

helper_bounds  := lmt_scene_bounds ;
helper_probe   := lmt_scene_point((0,0,-.5)) ;
helper_normal  := lmt_scene_normal("surface",(0,0)) ;
helper_label   := llcorner helper_bounds + (4,6) ;

draw helper_bounds withpen pencircle scaled .5 withcolor (.7,.7,.7) ;
draw fullcircle scaled 6 shifted helper_probe withcolor red ;
draw scenepoint "surface" (0,0,-.5)
  withpen pencircle scaled 2
  withcolor blue ;
draw scenenormal "surface" (0,0)
  withpen pencircle scaled .5
  withcolor green ;
draw textext("projected hidden point") shifted helper_probe shifted (4,
  16) ;

if not lmt_scene_visible((0,0,-.5)):
    draw textext("depth: hidden") shifted helper_probe shifted (4,-4) ;
fi ;

draw textext("raster " & decimal lmt_scene_width & " x " &
    decimal lmt_scene_height) shifted (helper_label shifted (0,14)) ;

draw textext("normal " & decimal redpart helper_normal & ", " &
    decimal greenpart helper_normal & ", " &
    decimal bluepart helper_normal) shifted (helper_label shifted (0,28
      )) ;

lmt_scene_stop ;
```

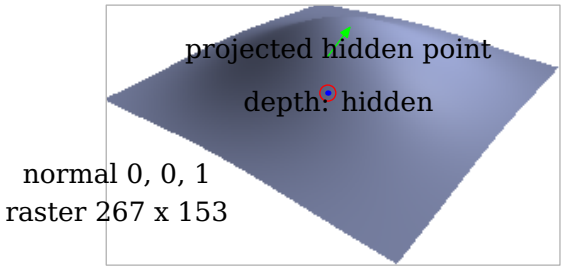


Figure 35 Using the rendered scene as a basis for ordinary METAPOST annotations.

The point returned by `lmt_scene_point` is already in the displayed picture's coordinate system, so it can be used directly for a marker or label. The example uses an analytic parametric surface, so the pair passed to `lmt_scene_normal` is its `(u,v)` parameter pair. For an implicit surface, pass a 3D coordinate instead. The normal itself is a `triplet`; the example formats its three components into a label. The projected hidden point demonstrates that a point can still have a position even when `lmt_scene_visible` reports that the depth test hides it.

`scenepoint` is a convenience form of `lmt_scene_point`: the object tag comes first and the point follows it, and the result is an ordinary projected METAPOST pair. `scenenormal` uses the same tag-and-parameter syntax but returns a small arrow picture. Both forms require `lmt_scene_render` first; they are useful when the result is drawn directly rather than stored for further calculations.

12 Scene cache (optional)

You can cache a complete scene as a rendered bytemap. This helps when repeated runs spend time generating the same geometry. Give the scene a stable `name` and set `cache = true`:

```
lmt_scene_start [
  name       = "cached-plane",
  cache      = true,
  width      = 4cm,
  resolution = 120,
] ;

lmt_scene_plane [
  id     = "ground",
  normal = (0,0,1),
  center = (0,0,0),
  usize  = 3,
  vsize  = 3,
] ;

lmt_scene_stop ;
```

The first run renders the scene and writes a cache file in the job directory. A later run with the same job name, scene name, and scene description reuses the cached bytemap instead of generating the 3D scene again. Changes to the documented scene or object settings invalidate the entry and cause a fresh render. Set `cache = false`, or omit `cache`, to bypass caching.

The cache stores the complete rendered scene, not individual meshes. Use a stable, distinct `name` for each scene definition that should be cached. Cache files are not removed automatically, so obsolete entries can be deleted from the job directory. Caching requires an enabled `bytemap`, which is the default; `bytemap = false` is not supported with `cache = true`. You can use `context --purge` to get rid of cache files.

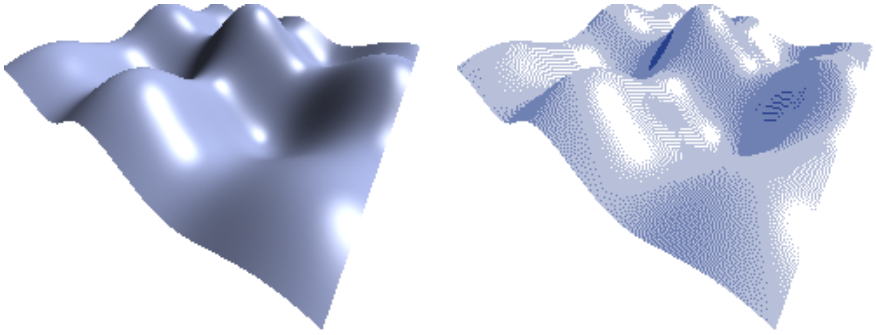
Scenes with a `process` or `stipple` postprocessor are rendered again when needed; those postprocessing passes are not reused from the scene cache.

13 Stippling (advanced)

Stippling converts the rendered scene into a field of colored dots. It is a raster postprocessing step: geometry, camera, lighting, and materials are resolved first, then the raster colors are converted to ink coverage. Depth and surface normals help preserve silhouettes, creases, and separation between visible surfaces.

Enable stippling with a table on `lmt_scene_start`. The example below renders the same graph as a continuous raster and as a four-level stippled image:

```
lmt_scene_start [
  width      = 5.5cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
  target     = (0,0,0),
  light      = (-1,-1,-2),
  stipple    = [
    color           = (.15,.25,.55),
    backgroundcolor = (1,1,1),
    edgeboost       = .20,
    coveragegamma   = 1.25,
    levels          = 4,
    tonedots        = true,
    outline         = true,
  ],
] ;

lmt_scene_surface [
  code = ".55*exp(-.07*(x*x+y*y))*(cos(2.2*x)+sin(2.0*y))",
  dfdx = ".55*exp(-.07*(x*x+y*y))*(-.14*x*(cos(2.2*x)+sin(2.0*y))-2.2*sin(2.2*x))
    ",
  dfdy = ".55*exp(-.07*(x*x+y*y))*(-.14*y*(cos(2.2*x)+sin(2.0*y))+2.0*cos(2.0*y))
    ",
  xmin = -3,
  xmax =  3,
  ymin = -3,
  ymax =  3,
  nx   = 64,
  ny   = 64,
] ;

lmt_scene_stop ;
```



continuous raster four-level stipple

Figure 36 The same graph rendered as a continuous raster and as a stippled image.

The left image is the ordinary rendered raster. The right applies stippling after lighting and visibility have been resolved. Stippling changes the raster, not the scene geometry. Its appearance depends on raster resolution; use `resolution` or explicit `bytewidth` and `byteheight` to control the dot spacing. The default `dotradius = 0` uses one-pixel dots. Larger radii can overlap and may need to be adjusted with the raster size.

The stipple settings are in the nested table passed to `stipple`:

color sets the ink color used for dots.

backgroundcolor sets the paper or background color. It is also used for the intermediate colors produced by `tonedots`.

nobackground keeps the existing raster background instead of clearing it to the background color.

levels selects evenly spaced tone levels. Give an integer from `3` through `16`; omit it for binary stippling.

tonedots uses colors between the background and ink colors for intermediate tone levels. The default is true.

errorthreshold sets the threshold used by binary stippling. The default is `.5`; it has no effect when `levels` is active.

coveragegamma reshapes the coverage curve before quantization. The default is `1.25`; `1` leaves the curve unchanged.

minimumcoverage/maximumcoverage clamp coverage after edge boosting and before the gamma adjustment. Their defaults are `0` and `1`.

edgeboost adds ink coverage near detected depth or normal edges. The default is `.20`.

usenormal includes surface-normal continuity in edge detection and error diffusion. Set it to false to use depth information only.

creaseangle sets the normal-angle threshold for crease edges. The default is `25` degrees.

sameangle sets the normal-angle threshold used to decide whether diffusion can cross neighboring pixels. The default is `15` degrees.

dzsamemultiplier/dzedgemultiplier scale the automatically estimated depth differences for same-surface and edge tests. Their defaults are `2` and `8`.

mindepthstep supplies the minimum absolute depth step used by those tests. The default is `1e-12`.

serpentine alternates the scan direction on successive rows. The default is true.

dotradius sets the radius of filled dots in raster pixels. Zero produces single-pixel dots.

clipdots prevents larger dots from being

painted onto background pixels. The default is true.

outline adds a final edge-outline pass. The default is false.

outlinethreshold sets the edge strength required for the outline. The default is `.85`.

outlineradius sets the radius of outline dots in raster pixels. The default is zero.

luminance sets the RGB weights used to convert rendered colors to intensity. The default is `(.2126,.7152,.0722)`.

The numeric form of `levels` is convenient for ordinary use. A custom form can instead provide a table of levels, each with a `value`, an optional `radius`, and optional `red`, `green`, and `blue` values. Use it when you need custom coverage values, dot sizes, or colors; use the numeric form when evenly spaced levels are enough.

Stippling is applied to the final raster, so raster size and visible detail both matter. Inspect the result at its intended output size and adjust the resolution, dot radii, tone levels, and edge settings together. A stippled scene is rendered again when needed instead of being reused from the scene cache.

14 External meshes (advanced)

An external mesh supplies its triangles from a file instead of sampling a function in the scene description. Add one with

```
lmt_scene_external [ ... ]
```

The external loader reads standard binary STL files and the `lmtmesh` representation used by CONTEXT. It does not support ASCII STL. The file name is passed through CONTEXT's file resolver, so you can give a file in the job directory or in a searched CONTEXT tree by name. A path also works.

The example uses `luametafun-threed-sphere.stl`, a binary STL file shipped with the CONTEXT distribution. The automatic camera settings fit the imported mesh after its bounds have been read.

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  projection = "perspective",
  eye        = "auto",
  target     = "auto",
  fov        = 60,
  light      = (-1,-1,-2),
  intensity  = 1.1,
] ;

lmt_scene_material [
  name      = "sphere",
  diffuse   = (.72,.82,1),
  specular  = (.7,.7,.7),
  shininess = 30,
] ;

lmt_scene_external [
  filename = "luametafun-threed-sphere.stl",
  id       = "sphere",
  material = "sphere",
] ;

lmt_scene_stop ;
```

The external-mesh keywords are:

filename names the external mesh file. The loader accepts standard binary STL and `lmtmesh` files; use a path or a file name that CONTEXT can resolve.

id gives the imported object a scene name. Use a distinct name when the

object is queried later or inspected with `lmt_scene_mesh`.

material selects the named material used when the mesh is rendered.

normal selects how imported normals are handled. The default smooth mode uses supplied mesh normals when they are available; `flat` discards them and shades each triangle as an independent face.

flipnormals reverses supplied packed normals before shading. Use it when the file's orientation is opposite to the intended outside of the object.

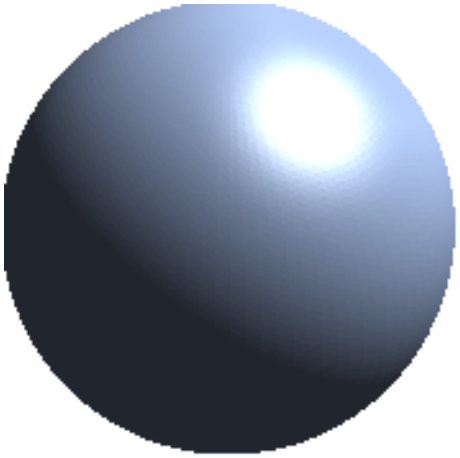


Figure 37 A binary STL mesh rendered as an ordinary shaded scene.

The coordinates in the file are used as supplied. The preset still contains a `scale` key for compatibility, but the implementation does not apply it; scale the mesh before importing if needed. Binary STL facet normals are preserved when the file supplies nonzero normals, so their quality and orientation affect the shading. Use `normal = "flat"` when those normals are missing or unreliable, or `flipnormals = true` when their direction is reversed. The imported triangles otherwise use the same camera, lighting, material, depth handling, and cropping as other scene objects.

Large external meshes can contain many more triangles than the sampled examples in this manual. Start with a modest output `resolution`, and increase it only when the final figure needs more raster detail; raster resolution does not reduce the cost of loading or transforming the mesh.

15 Internal meshes (advanced)

An internal mesh is an extension point for geometry that already exists in LUA or is easier to generate there than with one of the mathematical surface commands. A named generator returns vertices and either triangle indices or a compact mesh type; it can also return one normal per triangle. The mesh can then be placed, shaded, and combined with the other scene objects.

You can skip this section for ordinary graphs, parametric surfaces, and implicit surfaces. It requires LUA code and a registered mesh generator.

Here is a small pyramid generator:

```
metapost.registermesher("manual-pyramid", function(specification)
    return {
        vertices = {
            { -1, -1, 0 },
            {  1, -1, 0 },
            {  1,  1, 0 },
            { -1,  1, 0 },
            {  0,  0, 1.5 },
        },
        triangles = {
            { 1, 2, 3 }, { 1, 3, 4 },
            { 1, 5, 2 }, { 2, 5, 3 },
            { 3, 5, 4 }, { 4, 5, 1 },
        },
    }
end)
```

The indices in `triangles` are one-based LUA indices into `vertices`. The `triangles` field can be omitted when the generator returns `meshtype`: type `5` treats every three consecutive vertices as one triangle, while type `7` treats every four consecutive vertices as a quadrilateral split into two triangles. Explicit indices are usually the clearest choice for irregular meshes. An optional `normals` field supplies one normal per triangle, either as a table of triplets or as a packed `vector.normals` container.

After registering the generator, select it with `name`:

The internal-mesh keywords are:

id gives the generated mesh a name inside the scene, so that an intersection can refer to it.

name selects the LUA generator registered with `metapost.registermesher`.

material selects the named material used to render the mesh.

normal selects the normal mode for the generated mesh. In particular, `flat` expands the triangles so that each face is shaded independently; otherwise supplied triangle normals can be used for smooth shading.

flipnormals reverses a packed normal

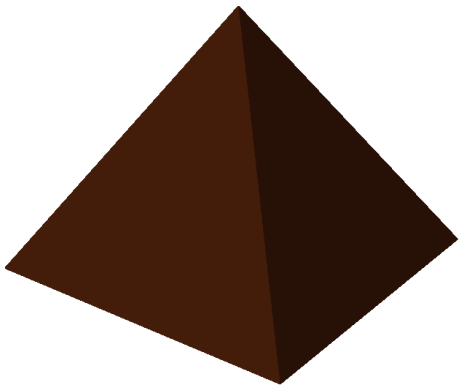


Figure 38 A

container returned by the generator before it is used for shading.

The generator receives the complete parameter specification, so it can define settings for a particular mesh. Those settings belong to the generator, not to the general `lmt_scene_internal` command.

16 Low-level LUA helpers (advanced)

The scene commands are normally enough. LUA code that generates meshes or postprocesses a rendered image can also use two lower-level userdata libraries. The `vector` library stores points, normals, and indexed meshes; the separate `zbuffer` library stores a rendered pixel buffer. This section covers the small subset needed to connect a LUA mesher or image postprocessor to the scene code. It is not a complete reference to every operation exposed by the engine.

A registered internal mesher can return packed containers directly. This is the packed equivalent of the table-based pyramid in the previous section:

```
local vertices = vector.points.new(5,1) -- default: xyz+n
vertices(-1,-1,0)
vertices( 1,-1,0)
vertices( 1, 1,0)
vertices(-1, 1,0)
vertices( 0, 0,1.5)

local triangles = vector.mesh.new(6) -- indexed triangles
triangles(1,2,3)
triangles(1,3,4)
triangles(1,5,2)
triangles(2,5,3)
triangles(3,5,4)
triangles(4,5,1)

metapost.registermesher("packed-pyramid", function(specification)
    return {
        vertices = vertices,
        triangles = triangles,
    }
end)
```

Use `name = "packed-pyramid"` with `lmt_scene_internal` as in the previous section. The default point type is `0`, with the layout `xyz+n`; it is the layout expected by the low-level triangle rasterizer. Types `1` (`xyz`) and `2` (`xy`) are useful for coordinate-only calculations, but should not be passed directly as the vertex container for rasterized 3D triangles. When the optional normal components are not supplied, they remain zero and the selected scene normal mode determines how the mesh is handled.

The main container operations are:

vector.point creates one point from coordinates or a small LUA table. Points have one-based components, support arithmetic, and provide `dotproduct`, `crossproduct`, `normalize`, `distance`, and `length`.

vector.points.new takes `rows`, `columns`, an optional point type, and an optional growth step. Its initial capacity is `rows*columns`; call the container with coordinates to fill entries in order. `get`, `getxyz`, `getnormal`, `set`, `totable`, and `getbounds` provide ordinary access. Indices are one-based.

vector.mesh.new takes the number of entries and an optional mesh type. The default type is 3, an indexed triangle. Call the container with three one-based vertex indices, or use `get`, `set`, and `totable`.

vector.normals creates a separate normal container with `new`; its entries can be filled or changed with `set`, inspected with `get`, and reversed with `flip`.

For postprocessing, allocate and configure a z-buffer directly:

```
local buffer = zbuffer.new()

zbuffer.setup(buffer, {
    width       = 5,
    height      = 4,
    perspective = false,
    eye         = { 0, 0, 5 },
    target      = { 0, 0, 0 },
    up          = { 0, 1, 0 },
    viewport    = { width = 5, height = 4 },
    camera      = { near = 0, far = 10, scale = 1 },
})

local visible, behind, vx, vy, vz, px, py =
    zbuffer.project(buffer, { 0, 0, 0 })
```

The top-level `width` and `height` allocate the buffer. `viewport` controls the projected pixel area, while `eye`, `target`, and `up` define the camera basis. With an orthographic setup, `camera.scale` controls the view scale; a perspective setup uses `camera.tanhalf` instead. `project` returns camera-space coordinates `vx`, `vy`, and `vz`, followed by projected coordinates `px` and `py`. A point behind the camera returns only `false, true`.

The pixel accessors `get`, `set`, `getcolor`, `setcolor`, `getdepth`, `setdepth`, `getnormal`, and `setnormal` use one-based `row,column` coordinates. `getsize` returns the number of rows and columns. `transform` converts a point to camera-space coordinates, and `tobytes` returns the RGB byte string together with its height, width, and channel count. The separate global name `zbuffer` is intentional; it is not a field of `vector`.

These calls are enough for a controlled LUA integration or a diagnostic postprocessor. Matrix operations, contour builders, direct triangle rasterization, and stipple stages remain implementation-facing and are not listed as a complete low-level API.

17 Raster postprocessing hooks (advanced)

The scene `process` setting selects a named LUA hook that runs after the scene has been rasterized and before supersampling is resolved. It is useful for color effects or diagnostics that need access to the rendered colors, normals, or texture coordinates. Register the hook before using the scene:

Select it by name in the scene:

```
lmt_scene_start [
  process = "dim",
  ...
] ;
```

The following rendered example uses that hook; the scene is otherwise an ordinary surface scene. The dimmed result makes the data flow through `process = "dim"` visible in the document rather than only in source.

```
lmt_scene_start [
  width      = 5.5cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
  process    = "dim",
] ;

lmt_scene_surface [
  code = ".7*exp(-.6*(x*x+y*y)) - .2",
  xmin = -1.8,
  xmax =  1.8,
  ymin = -1.8,
  ymax =  1.8,
  nx   = 48,
  ny   = 48,
] ;

lmt_scene_stop ;
```

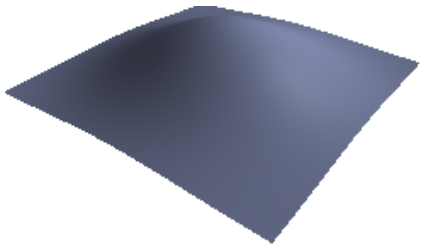


Figure 39 A scene rendered through the dim postprocessing hook.

The registration table accepts `name` and a required `process` callback. Set `color`, `normal`, `texture`, or `slot` to request the corresponding inputs. The callback arguments

are concatenated in that order: `slot` supplies zero-based `(row,column)`, `color` supplies `(r,g,b)`, `normal` supplies `(nx,ny,nz)`, and `texture` supplies `(x,y,z,t)`. For example, a callback with `color = true` and `normal = true` receives `(r,g,b,nx,ny,nz)`. It returns up to three RGB values; returned numeric values are clamped to the `0`–`1` range.

By default, background pixels are skipped. Set `skip = false` when the background should also be passed to the callback. Optional `initialize` and `finalize` functions receive the registration table before and after the pixel pass.

For direct access to the z-buffer, set `direct = true` or `method = "direct"`. In that form the callback receives the z-buffer and the registration table instead of per-pixel values; use the z-buffer operations described in the low-level LUA helpers section.

The hook runs before stippling and is rerun when the scene must be rendered again. Keep it limited to the raster work that belongs in this stage.

18 Special overlays (advanced)

`lmt_scene_special` adds small, depth-tested overlays to a scene. The supported kinds are `point`, `segment`, `normal`, and `arrow`. Add an overlay after the object it refers to and before `lmt_scene_stop`. The `normal` form needs a named parametric object and its derivatives.

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
] ;

lmt_scene_parametric [
  id  = "surface",
  x   = "u",
  y   = "v",
  z   = ".7*exp(-.6*(u*u+v*v)) - .2",
  xu  = "1",
  yu  = "0",
  zu  = "-.84*u*exp(-.6*(u*u+v*v))",
  xv  = "0",
  yv  = "1",
  zv  = "-.84*v*exp(-.6*(u*u+v*v))",
  umin = -1.8,
  umax =  1.8,
  vmin = -1.8,
  vmax =  1.8,
  nu   = 48,
  nv   = 48,
] ;

lmt_scene_special [
  kind  = "point",
  id    = "probe",
  at    = (0,0,.5),
  color = red,
  width = 2,
] ;

lmt_scene_special [
  kind  = "segment",
  id    = "baseline",
  from  = (-1.2,-1,0),
  to    = ( 1.2,-1,0),
  color = blue,
```

```
  width = 1,
] ;

lmt_scene_special [
  kind   = "normal",
  id     = "normal",
  name   = "surface",
  u      = 0,
  v      = 0,
  length = .2,
  span   = .12,
  color  = green,
  width  = 1,
] ;

lmt_scene_special [
  kind   = "arrow",
  id     = "arrow",
  from   = (0,0,0),
  to     = (0,0,1),
  length = .15,
  span   = .12,
  color  = (.9,.4,0),
  width  = 1,
] ;

lmt_scene_stop ;
```

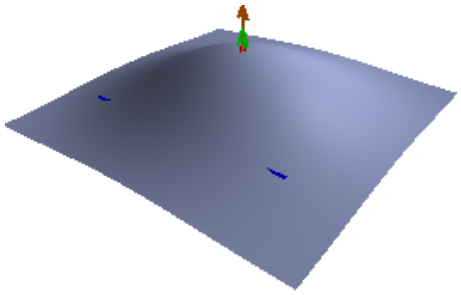


Figure 40 Point, segment, normal, and arrow overlays in one parametric scene.

The special-overlay keywords are:

kind selects the overlay form: `point`, `segment`, `normal`, or `arrow`.
id gives the overlay a name.
at gives the literal 3D point where a `point` overlay is drawn. A point overlay draws a small cross there.
point gives a spatial coordinate when a named object is resolved. The same coordinate can be given with `x`, `y`, and `z`.
from gives the first endpoint of a `segment` or `arrow`.
to gives the second endpoint of a `segment` or `arrow`.
name selects a named scene object for

point or normal lookup. It is required by the `normal` form.

u gives the first parameter when a named parametric object is resolved.

v gives the second parameter when a named parametric object is resolved.

t gives the parameter when a named curve is resolved.

length controls the size of a point cross or the length of an arrowhead.

span controls the width of an arrowhead.

rotation rotates an arrowhead around its direction.

color selects the overlay line color.

width sets the overlay line width in raster pixels.

The `normal` form resolves the named object using the same coordinate or parameter lookup and evaluates its normal; supplying analytic derivatives makes a parametric normal reliable. All four forms are depth-tested like other scene geometry.

The named-normal example uses a parametric surface. For a graph or implicit surface, use the direct `lmt_scene_normal` query after rendering.

19 Mesh inspection (experimental)

This final example is diagnostic only. It shows the triangle paths captured for a scene object, which helps you check sampling, winding, and mesh coverage. It is not a normal rendering interface. Set the object's experimental `vector` flag, render the scene, and draw `lmt_scene_mesh` using its `id`:

```
lmt_scene_start [
  width      = 6cm,
  resolution = 120,
  crop       = true,
  eye        = (3,-4,2.6),
] ;

lmt_scene_surface [
  id       = "surface",
  vector   = true,
  code     = ".7*exp(-.6*(x*x+y*y)) - .2",
  dfdx     = "-.84*x*exp(-.6*(x*x+y*y))",
  dfdy     = "-.84*y*exp(-.6*(x*x+y*y))",
  xmin     = -1.8,
  xmax     =  1.8,
  ymin     = -1.8,
  ymax     =  1.8,
  nx       = 24,
  ny       = 24,
] ;

lmt_scene_render ;

draw lmt_scene_mesh "surface"
  withpen pencircle scaled .2pt
  withcolor red ;

lmt_scene_stop ;
```

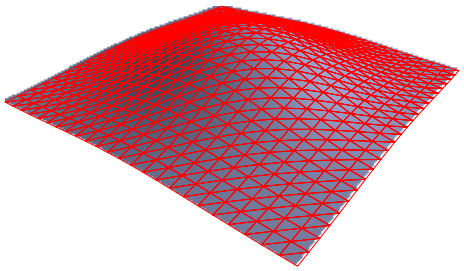


Figure 41 Diagnostic view of the sampled triangle mesh.

`lmt_scene_mesh` returns a path made from the captured projected triangles. The path can include triangles hidden in the shaded rendering, so use it to inspect the mesh rather than to produce a clean visible outline. Call it after `lmt_scene_render`;

the object's `vector = true` flag captures the paths. This flag and helper depend on experimental vector-point support and may change, so use them while debugging a scene.

20 Rendering pipeline (advanced)

The scene commands are declarative, but a rendered figure passes through several distinct stages. This distinction is useful when deciding whether to increase an object's mesh counts, the raster resolution, or the supersampling factor. It also explains why rendering must happen before projection and normal queries can be used.

The main sources of render time are dense implicit grids (`nx*ny*nz`), large sampled surface grids, high `supersample` values which enlarge the temporary raster area roughly with the square of the factor, large external meshes, and mesh–mesh intersections. When a figure is slow, inspect these sources separately and increase one of them at a time.

For an ordinary render without a cache hit, the overall path is:

```
scene description
    -> object generation and triangle meshes
    -> camera, crop, and raster setup
    -> projection and z-buffer rasterization
    -> resolution and raster postprocessing
    -> final crop and bytemap conversion
    -> MetaPost placement and scene queries
```

The stages have the following roles:

- The scene description records the shared camera, lighting, materials, output settings, and the objects that belong to the scene. Object commands do not immediately draw pixels; they add specifications to the scene.

- Object generation turns those specifications into geometry. A graph or parametric surface is sampled over its parameter grid, an implicit surface is extracted from samples in a 3D box, and external or internal mesh objects provide their triangles directly. Normals are generated or attached at this stage so that they are available during shading. The object mesh counts control this geometric stage.

- Camera and raster setup chooses the internal pixel dimensions, allocates the z-buffer, and derives the camera transform. `width`, `height`, `bytewidth`, `byteheight`, and `resolution` determine the displayed and raster dimensions; `supersample` enlarges the temporary raster used while drawing. With automatic cropping, the projected object bounds also help fit the camera transform to the scene.

- Projection transforms mesh vertices into camera and screen coordinates. Triangle rasterization then writes color, depth, and normal information to the z-buffer. Depth tests decide which surfaces are visible, while materials, lights, opacity, and the selected normals determine the shaded result. Transparent and on-top objects are handled in their corresponding rendering passes.

- After the internal drawing pass, an optional `process` hook (described in the **Raster postprocessing hooks** section) can inspect or modify the rendered buffer. The supersampled buffer is then reduced to the display raster. Stippling, if enabled, is applied later to that resolved raster; neither operation changes the generated triangle mesh.

- Cropping then trims the resolved raster to the occupied scene bounds when requested. Thus `crop` affects both the camera fitting used before rasterization and, where appropriate, the dimensions of the final displayed buffer. The resulting pixel data is converted to a METAPOST bytemap.

- `lmt_scene_render` makes the rendered result available to projection, visibility, bounds, bytemap, and normal helpers. `lmt_scene_stop` performs the render when needed and places the finished bytemap in the current picture. Queries and ordinary METAPOST annotations therefore come after rendering, while the scene is still open.

This separation also explains several common adjustments. Increasing `nx`, `ny`, `nz`, `nu`, or `nv` changes the geometric approximation and can reveal features that a coarse mesh misses. Increasing `resolution` or `supersample` changes the raster quality of an already generated scene; it does not recover geometric detail that was never sampled. Stippling similarly changes the final raster rather than the surface or its projection.

When caching is enabled and a valid cached result is available, the saved rendered scene can be restored instead of repeating the ordinary generation and rasterization stages. The cache is scene-level: it is not a reusable cache of individual object meshes.

21 About the cover page

The cover captures the rendered scene in a METAPOST picture with `picture q := image(...)`. The scene is rendered while the argument of `image` is evaluated; afterwards, shifting by `-center q` moves the picture's center to the origin, and the final shift places that center on the paper. This is a useful technique when a scene must be captured, scaled, and positioned independently of the surrounding METAPOST drawing.

```
\startMPcode
StartPage ;

picture q ; q := image (

    lmt_scene_start [
        width           = 18cm,
        crop = true,
        projection      = "perspective",
        eye             = (2.7,-4.0,2.4),
        light           = (0,0,-1),
        intensity       = 1,
    ] ;

    lmt_scene_material [
        name    = "sphere",
        diffuse = (1,1,1),
    ] ;

    lmt_scene_parametric [
        id       = "sphere",
        x        = "sin(u)*cos(v)",
        y        = "sin(u)*sin(v)",
        z        = "cos(u)",
        umin     = 0,
        umax     = 2pi,
        vmin     = 0,
        vmax     = 2*pi,
        nu       = 5,
        nv       = 5,
        material = "sphere",
        normal   = "flat",
    ] ;

    lmt_scene_stop ;
) ;

q := q shifted - center q
       shifted (0.5PaperWidth,0.5PaperHeight) ;
```

```
draw q ;

picture t[] ;
t[0] := textext("\sansbold lmt_scene") ;
t[1] := textext("\sansbold Hans Hagen & Mikael Sundqvist") ;

t[0] := t[0] xsized .6 PaperWidth ;
t[1] := t[1] xsized .6 PaperWidth ;

t[0] := t[0]
    shifted - ulcorner t[0]
    shifted (1cm,-1cm)
    shifted (0*PaperWidth,1*PaperHeight) ;

t[1] := t[1]
    shifted - lrcorner t[1]
    shifted (-1cm,1cm)
    shifted (1*PaperWidth,0*PaperHeight) ;

draw t[0] withcolor red ;
draw t[1] withcolor red ;

StopPage ;
\stopMPcode
```

Index