# luatikz

Axel Kittenberger

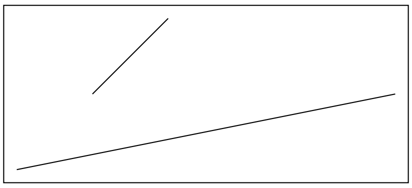
luatikz version 2.12

## 1 Introduction

Luatikz is a 2D graphics library to draw tikz graphics using the Lua programming language.

Following code draws two lines by specifying begin and end points. First using intermediatery variables and the second line within a single codeline:

```
tikz.within( '*' )

local p1 = p{ 1, 1 }
local p2 = p{ 6, 2 }
local l1 = line{ p1, p2 }

draw{ l1 }
draw{ line{ p{ 2, 2 }, p{ 3, 3 } } }
```



Following TEX wrapper can be used to compile the luatikz Lua code.

```
\documentclass[tikz]{standalone}
\usepackage{tikz}
\usepackage{luacode}

\begin{document}
\begin{tikzpicture}[scale=1]]
\directlua{dofile('line.lua')}
\end{tikzpicture}
\end{document}
```

Luatikz obviously needs LuaLATEX to compile.

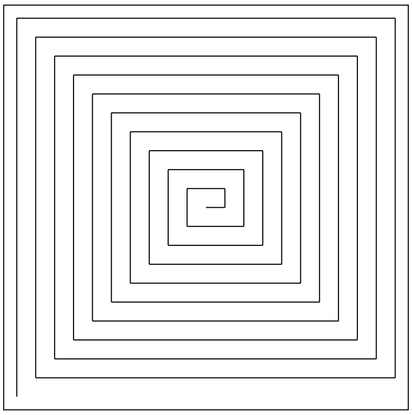
Note that 2D graphics is a vast topic, functionality to luatikz has been and will be added on a per need basis.

## 2 Pointer math

Mathematical operations are possible with points, adding them with each other or multiplying them with a scalar.

This example uses pointer math to create a spiral:

```
tikz.within( '*' )

local pi  = p{ 0,    0    }
local pdx = p{ 0.25, 0    }
local pdy = p{ 0,    0.25 }

for k = 1, 20
do
    local sign = 1
    if k % 2 == 0 then sign = -1 end
    local pn  = pi + sign * k * pdx
    local pn2 = pn + sign * k * pdy
    draw{ line{ pi, pn }, line{ pn, pn2 } }
    pi = pn2
end

```



## 3 Immutability

In luatikz pointers like all objects are immutable. That means a object once created in memory can no longer be changed. However the variable holding an object can be changed to another object with different attributes.

Thus following example is invalid:

```
tikz.within( '*' )

local p1 = p{ 1, 1 }
p1.x = p1.x + 2
```

This on the other hand are two valid methods to change the value of p1

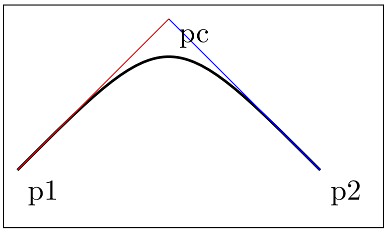
```
tikz.within( '*' )

local p1 = p{ 1, 1 }
p1 = p{ p1.x + 2, p1.y }
-- or
p1 = p1 + p{ 2, 0 }
```

# 4 luatikz objects

## 4.1 Bezier (quatratic)

A quadratic bezier curve is defined by it begin, end and a control point:

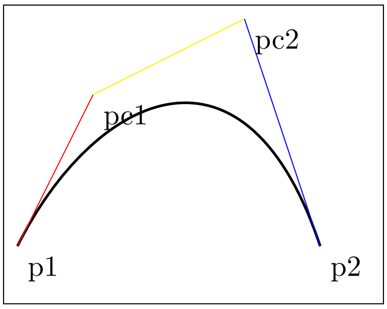
```
tikz.within( '*' )

local bz =
    bezier2{
        p1 = p{ 0, 0 },
        pc = p{ 2, 2 },
        p2 = p{ 4, 0 },
    }

draw{ line_width=1, bz }
bz.drawHelpers( 'p' )
```



## 4.2 Bezier (cubic)

A cubic bezier curve is defined by it begin, end and two control points:

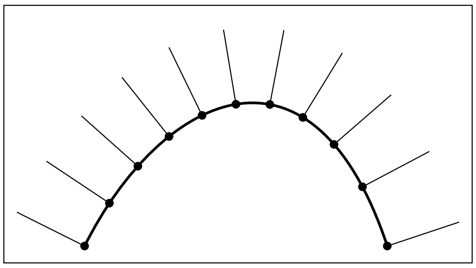
```
tikz.within( '*' )

local bz =
    bezier3{
        p1  = p{ 0, 0 },
        pc1 = p{ 1, 2 },
        pc2 = p{ 3, 3 },
        p2  = p{ 4, 0 },
    }

draw{ line_width=1, bz }
bz.drawHelpers( 'p' )
```



A cubic bezier has the functions pt and phit that return the point and angle ranging from 0..1 on the curve.

This example draw 10 normal lines onto the bezier curve:

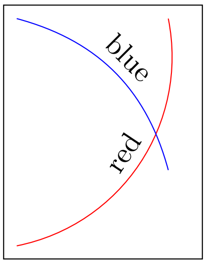
```
tikz.within( '*' )

local bz =
    bezier3{
        p1  = p{ 0, 0 },
        pc1 = p{ 1, 2 },
        pc2 = p{ 3, 3 },
        p2  = p{ 4, 0 },
    }

draw{ line_width = 1, bz }

for t = 0, 1, 0.1
do
    local pt = bz.pt( t )
    draw{
        fill = black,
        circle{
            at = pt,
            radius = 0.05,
        }
    }

    local phit = bz.phit( t ) + math.pi / 2
    draw{
        line{
            p1 = pt,
            phi = phit,
            length = 1,
        }
    }
end
```



### 4.3 BLine

A bline is defined by it's begin and it's end. Additionally to the standard line it is bend

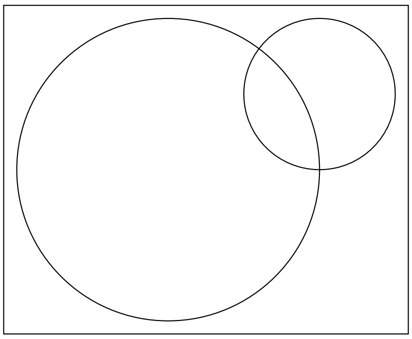
```
tikz.within( '*' )

local l1 = bline{ p{ 0, 0 }, p{ 2, 3 }, bend_right = 45 }
local l2 = bline{ p{ 0, 3 }, p{ 2, 1 }, bend_left  = 30 }

draw{ draw=red,  l1 }
draw{ draw=blue, l2 }
put{
    node{
        at = l1.pc,
        anchor = south,
        rotate = l1.line.phi * 180 / math.pi,
        text = 'red',
    }
}
put{
    node{
        at = l2.pc,
        anchor = south,
        rotate = l2.line.phi * 180 / math.pi,
        text = 'blue',
    }
}
```



## 4.4 Circle

A circle is defined by it's center ("at") and the circle "radius":

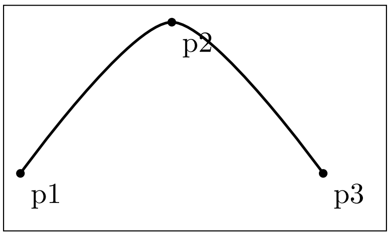
```
tikz.within( '*' )

draw{
    circle{
        at = p{ 0, 0 },
        radius = 2,
    },
    circle{
        at = p{ 2, 1 },
        radius = 1,
    }
}
```



## 4.5 Curve

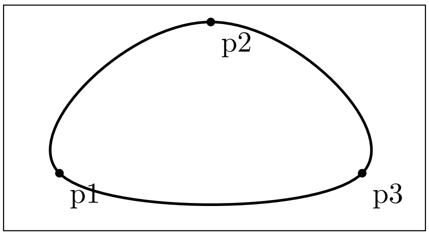
A curve is defined by a list of points:

```
tikz.within( '*' )

local c =
    curve{
        points =
        {
            p{ 0, 0 },
            p{ 2, 2 },
            p{ 4, 0 },
        },
    }

draw{ line_width=1, c }
c.drawHelpers( 'p' )
```



A curve can cycle and has a changeable tension:

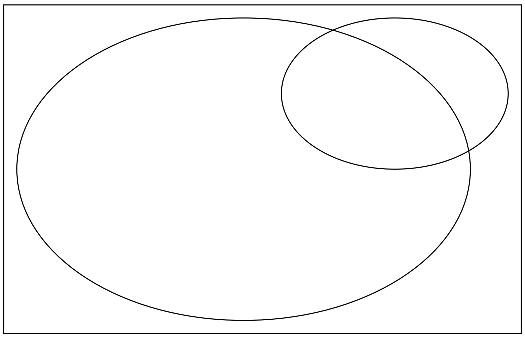
```
tikz.within( '*' )

local c =
    curve{
        points =
        {
            p{ 0, 0 },
            p{ 2, 2 },
            p{ 4, 0 },
        },
        cycle = true,
        tension = 1.0,
    }

draw{ line_width=1, c }
c.drawHelpers( 'p' )
```



## 4.6 Ellipse

An ellipse is difined by it’s center (“at”) and a x- and yradius:

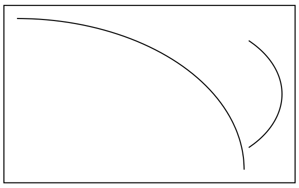
```
tikz.within( '*' )

draw{
    ellipse{
        at = p{ 0, 0 },
        xradius = 3,
        yradius = 2,
    },
    ellipse{
        at = p{ 2, 1 },
        xradius = 1.5,
        yradius = 1,
    }
}
```



### 4.7 EllipseArc

An ellipse arc is a an ellipse limited by it's "from" and "to" angle:

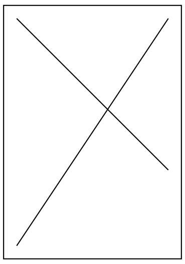
```
tikz.within( '*' )

draw{
    ellipseArc{
        at = p{ 0, 0 },
        from = 0,
        to = 90,
        xradius = 3,
        yradius = 2,
    },
    ellipseArc{
        at = p{ 2, 1 },
        xradius = 1.5,
        yradius = 1,
        from = -45,
        to = 45,
    }
}
```



## 4.8 Line

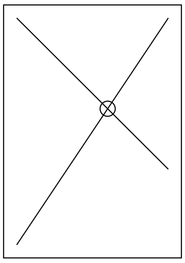
A line is defined by it's begin and it's end. There are two basic variants to define a line:

```
tikz.within( '*' )

local l1 = line{ p{ 0, 0 }, p{ 2, 3 } }
local l2 = line{ p1 = p{ 0, 3 }, p2 = p{ 2, 1 } }

draw{ l1 }
draw{ l2 }
```



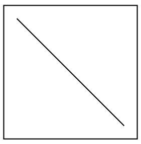
You can use intersectLine() to find the intersection of a line with another. If there is none "nil" will be returned:

```
tikz.within( '*' )

local l1 = line{ p{ 0, 0 }, p{ 2, 3 } }
local l2 = line{ p1 = p{ 0, 3 }, p2 = p{ 2, 1 } }
local pi = l1.intersectLine( l2 )

draw{ l1 }
draw{ l2 }
draw{ circle{ at = pi, radius = 0.1 } }
```



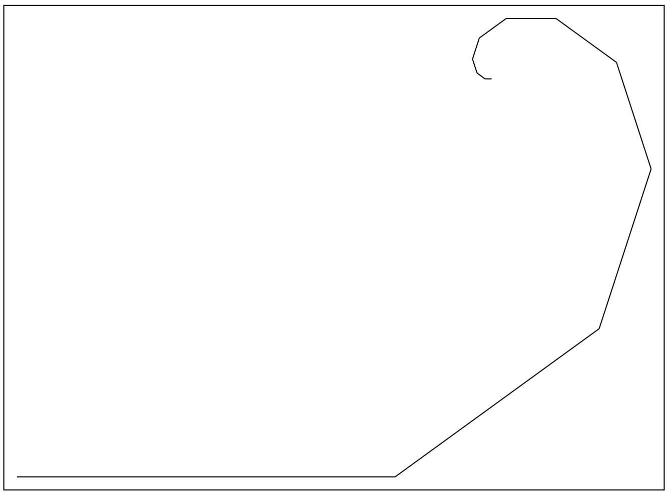
A line can also be defined by it's starting point, angle and length:

```
tikz.within( '*' )

draw{ line{
    p1  = p{ 0, 0 },
    phi = -math.pi / 4,
    length = 2,
} }
```



"length" and "phi" are attributes of a line:

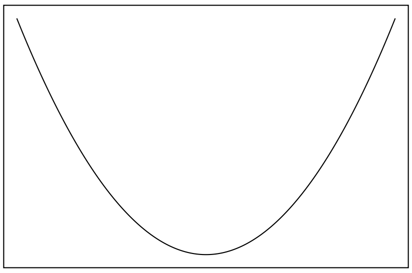
```
tikz.within( '*' )

local l = line{ p{ 0, 0 }, p{ 5, 0 } }
draw{ l }

for i = 1, 10
do
    l =
        line{
            p1     = l.p2,
            phi    = l.phi + math.pi / 5,
            length = l.length * 2 / 3,
        }
    draw{ l }
end
```



### 4.9 Plot

A plot uses a function to determine a series of points on a curve. It takes values going “from”–“to”. The function is to be a lua function that takes the input scalar and returns a point. The whole curve is offset “at” a point:

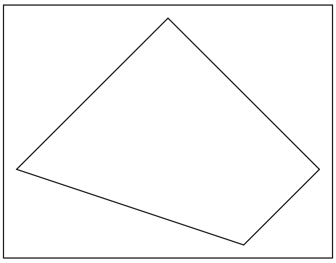
```
tikz.within( '*' )

draw{
    plot{
        at   = p{ 0, 0 },
        from = 0,
        to   = 5,
        step = 0.05,
        func =
            function( d )
                return p{ d, math.pow( d - 2.5, 2 ) / 2 }
            end
    }
}
```



### 4.10 Polyline

A polyline is a list of points to be connected via straight lines. If the string “cycle” is given at the end, it cycles:

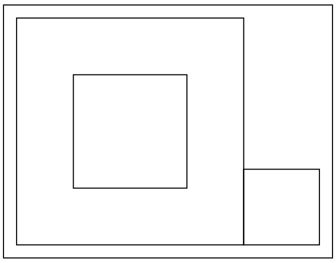
```
tikz.within( '*' )

draw{
    polyline{
        p{ 0, 0 },
        p{ 2, 2 },
        p{ 4, 0 },
        p{ 3, -1 },
        'cycle',
    }
}
```



### 4.11 Rect

Rectangles have various creation options. Also they provide a wide range of attributes of their points “pnw”, “pne”, “psw”, “pse”, “pn”, “pe”, “ps”, “pw”, “pc”, “height”, and “width”:

```
tikz.within( '*' )

local r1 =
    rect{
        pnw = p{ 0, 3 },
        pse = p{ 3, 0 },
    }

local r2 =
    rect{
        psw  = r1.pse,
        size = p{ 1, 1 },
    }

local r3 =
    rect{
        pc   = r1.pc,
        size = p{ 1.5, 1.5 },
    }

draw{ r1, r2, r3 }
```

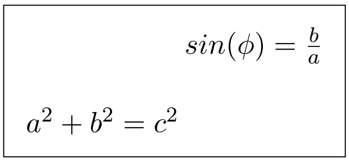


# 5 Labels a.k.a Nodes

Labels are created as nodes. Contrary to all other objects they do not need the “draw” command to be printed, but the “put” command. This is due any call to “draw” is turned into exactly one “draw” command to tikz and nodes in tikz are not using the “draw” command.

In it’s simplest form a node is specified by it’s position and text. The double square brackets are Lua’s way to make string constants that may contain simple blackslashes:

```
tikz.within( '*' )

put{
    node{
        at = p{ 0, 0 },
        text = [[$a^2 + b^2 = c^2$]],
    },
    node{
        at = p{ 2, 1 },
        text = [[$sin(\phi) = \frac{b}{a}$]],
    }
}
```

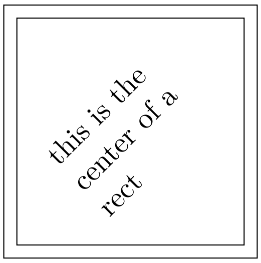


All constructors options of node are as follows; they correspond to the standard tikz options:

- above
- anchor
- align
- at
- below
- color
- draw
- left
- minimum_height
- node_distance
- name
- right
- rotate
- text
- text_width

Another node example

```
tikz.within( '*' )

local r1 =
    rect{
        pc   = p{ 0, 0 },
        size = p{ 3, 3 },
    }

draw{ r1 }

put{
    node{
        at = r1.pc,
        text = [[this is the center of a rect]],
        text_width = '2cm',
        rotate = 45,
    },
}
```

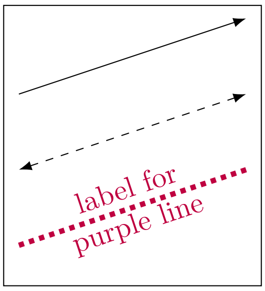


## 6 Styling

Styling is applied to draw commands by specifying style options, luatikz autodetects the difference between objects and styles.

An example using arrows and dashes

```
tikz.within( '*' )

local l1 = line{ p{ 0,  0 }, p{ 3,  1 } }
local l2 = line{ p{ 0, -1 }, p{ 3,  0 } }
local l3 = line{ p{ 0, -2 }, p{ 3, -1 } }

draw{ arrow, l1 }
draw{ double_arrow, dashed, l2 }
draw{ color = purple, line_width = 2, dotted, l3 }

put{ node{
    at = l3.pc,
    anchor = center,
    align = center,
    color = purple,
    rotate = l3.phi * 180 / math.pi,
    text = 'label for purple line',
    text_width = '2cm',
} }
```

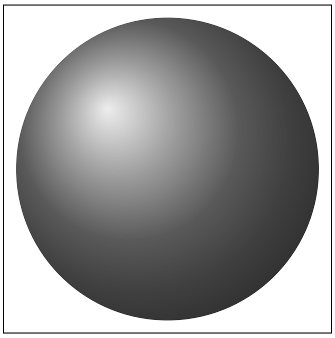


# 7 Shades

The “shade” command works analogous to classical tikz.

For example a ball:

```
tikz.within( '*' )

shade{
    ball_color = gray,
    circle{
        at = p0,
        radius = 2,
    },
}
```



Recognized shade options are:

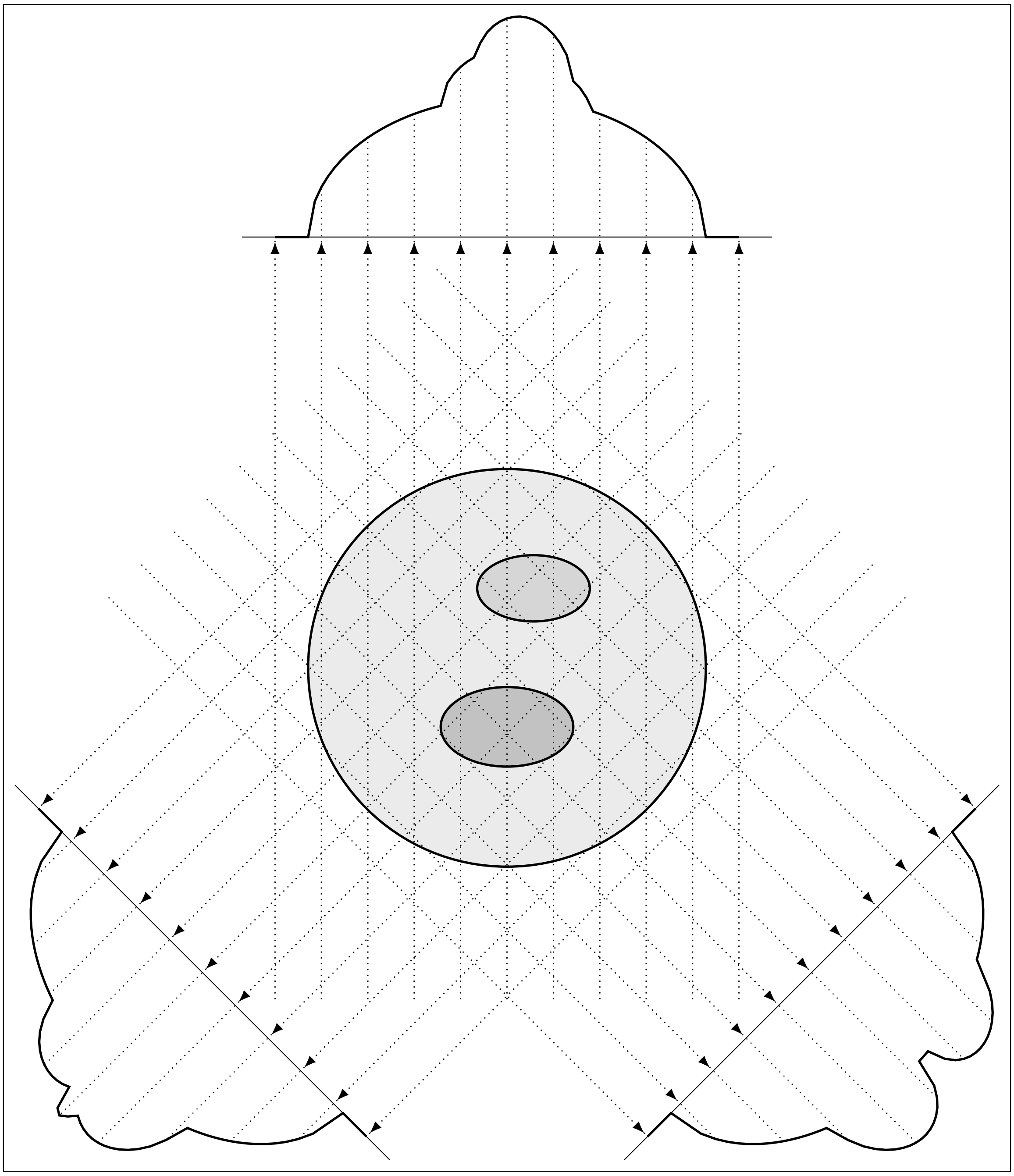
- ball_color
- left_color
- lower_left_color
- lower_right_color
- opacity
- right_color
- shading
- upper_left_color
- upper_right_color

# 8 Some larger examples

## 8.1 A radon projection

A radon projection, calculating correct projection curves:

```
tikz.within( '*' )

local cos = math.cos
local sin = math.sin
local pi  = math.pi

-- granularity of calculation
local fine = 10

-- factor the projection length is reduced
local fproj = 0.35

-- strength of the ellipse mediums
local s1 = 1.00
local s2 = 1.50
local s3 = 1.75

local e1 = ellipse{ at=p{ 0.00,  0.00 }, xradius=3.00, yradius=3.00 }
local e2 = ellipse{ at=p{ 0.40,  1.20 }, xradius=0.85, yradius=0.50 }
local e3 = ellipse{ at=p{ 0.00, -0.89 }, xradius=1.00, yradius=0.60 }

draw{ fill='black!08!white', line_width=1, e1 }
draw{ fill='black!16!white', line_width=1, e2 }
draw{ fill='black!24!white', line_width=1, e3 }

-- list of projection angles
local listphi =
{
    -1/4 * pi,
     2/4 * pi,
     5/4 * pi,
}

-- length of projection lines
local lenmaintop = 5.0
local lenmainbot = 6.5

for proji, phimain in ipairs( listphi )
do
    local lmain =
        line{
            p{ cos( phimain ), sin( phimain ) } * -lenmaintop,
            p{ cos( phimain ), sin( phimain ) } *  lenmainbot,
        }
    local phinorm = phimain - pi/2

    -- gets a point on the projection.
    -- dp: distance from projection center
    -- dv: projection value
    function getpproj( dp, dv )
        return(
            lmain.p2
            + p{ cos( phinorm ) * dp,         sin( phinorm ) * dp         }
            + p{ cos( phimain ) * dv * fproj, sin( phimain ) * dv * fproj }
        )
    end

    -- distance of projection line
    local ddis = 1 / fine

    -- list of all projection points
    local listp = { }

    for i = -3.5 * fine, 3.5 * fine
    do
        local p1i =
            lmain.p1
            + p{ cos( phinorm ) * ddis * i, sin( phinorm ) * ddis * i }
        local li = line{ p1i, p1i + ( lmain.p2 - lmain.p1 ) }
        local lv = 0
        local is1 = e1.intersectLine( li )

        if is1 and #is1 > 1
        then
            local lis = line{ is1[ 1 ], is1[ 2 ] }
            lv = lv + lis.length * s1
        end

        local is2 = e2.intersectLine( li )
        if is2 and #is2 > 1
        then
            local lis = line{ is2[ 1 ], is2[ 2 ] }
            lv = lv + lis.length * s2
        end

        local is3 = e3.intersectLine( li )
        if is3 and #is3 > 1
        then
            local lis = line{ is3[ 1 ], is3[ 2 ] }
            lv = lv + lis.length * s3
        end

        local pproj = getpproj( ddis * i, lv    )
        if i % ( 0.7 * fine ) == 0
        then
            draw{
                dotted, arrow, line_width=0.5,
                line{ li.p1, getpproj( ddis * i, -0.20 ) },
            }
            draw{
                dotted, line_width=0.5,
                line{ li.p1, pproj },
            }
        end
        table.insert( listp, pproj )
    end

    -- draws the projection screen
    local lenlproj = 8
    draw{
        line{
            getpproj(  lenlproj / 2, 0 ),
            getpproj( -lenlproj / 2, 0 ),
        }
    }

    -- draws the projection curve
    draw{ line_width=1, polyline{ table.unpack( listp ) } }
end
```



## 8.2 A sierpiński fractal

A sierpiński fractal. This one creates own temporary lua objects:

```
tikz.within( '*' )

local s60 = math.sin( 60 / 180 * math.pi )
local c60 = math.cos( 60 / 180 * math.pi )

-- creates a table object having ptop, pleft and pright as points
-- and a "draw yourself" function
function equilateralTriangle( ptop, len )
    return {
        ptop   = ptop,
        pleft  = ptop + p{ -len/2, -len*s60 },
        pright = ptop + p{  len/2, -len*s60 },
        draw =
        function( self )
            draw{
                fill = black,
                draw = none,
                polyline{ self.ptop, self.pleft, self.pright, 'cycle' }
            }
        end
    }
end

-- one step into the fractal
-- ptop:  top point of triangle
-- len:   current length
-- level: current fractal level
function drawFractal( ptop, len, level )
    if level == 1
    then
        local t = equilateralTriangle( ptop, len )
        t:draw( )
    else
        local ttop   = equilateralTriangle( ptop, len / 2 )
        drawFractal( ttop.ptop,   len / 2, level - 1 )
        drawFractal( ttop.pleft,  len / 2, level - 1 )
        drawFractal( ttop.pright, len / 2, level - 1 )
    end
end

drawFractal( p{ 0, 0 }, 8, 6 )
```