Contents

Title: Quasar Graphics Library

The Quasar Graphics Library

This document gives a brief overview of the Quasar graphics library. The graphics library, unlike most other graphics programming libraries, tightly integrates CPU and GPU functionality. In particular the library wraps the OpenGL API (hardware rendering) and the Cairo drawing API (software rendering) in a uniform way. This allows the user to benefit from the advantages of both:

- **Cairo**: Highly precise graphics; export to .svg, .eps, .pdf
- **OpenGL**: fast hardware accelerated drawing (useful for visualization of large amounts of data).

But note that the rendering results may not be identical. This is similar to differences between software rendering and hardware accelerated rendering (e.g. Quake) in the old days. More specifically, the OpenGL rendering of fonts is implemented using textures, these textures may show bilinear interpolation artifacts when zooming in using geometric scaling transforms. On the other hand, Cairo uses high quality true type font rendering methods.

The graphics library can be used to:

- add layers on top of existing visualization objects (e.g. imshow)
- alter images (stored in variables), by adding lines, text etc to them.
- describe vector-graphics and render to a matrix object.
- implement custom visualization objects (e.g. plots) directly from within Quasar.
- improve the user interactivity (e.g. displaying contours drawn using the mouse).

In Quasar, there are two distinct drawing classes:

- qvectorlayer: implements a purely 2D drawing layer. 2D drawing layers can be added to qimshow objects, or they can be constructed out of the blue, without connection to any display object.
- qvectorlayer3d: implements a 3D drawing layer (OpenGL only). 3D drawing layers can only be attached to OpenGL display objects (qgldisplay class).

However, both drawing classes can seamlessly be combined, for example, suppose you want to render a 3D scene with a 2D head-up display (HUD) on top of it. Then you can use the qvectorlayer3d class for the 3D rendering and the qvectorlayer for the HUD.

Notes: when working over SSH, or with a remote desktop connection, OpenGL is disabled and the Cairo rendering method is selected. This is because of a limitation of the OpenGL driver.

Advantages of the Quasar graphics library:

- simplified but flexible object model (compared to the harder to use OpenGL functions, Direct3D classes etc.)
- most of the OpenGL functionality is exposed (exception: pixel shaders that are replaced by __kernel__ functions)
- seemless integration (transparent transfers between OpenGL memory and CUDA memory/system memory)
- support for accessing the hardware rasterization functionality (capturing rendering and depth buffers)
- support for multiple display windows

1. 2D rendering

The typical workflow of the 2D rendering in Quasar is as follows:

```
+-----------------+ +----------------------+ +---------+
|new(qvectorlayer)| --> |call drawing functions| --> |rasterize|
+-----------------+ +----------------------+ +---------+
```

It needs to be mentioned that the drawing functions actually have *no direct effect*. Instead, the vector layer keeps a list with all drawing commands. Once the drawing command list is finished, this list is used by the `rasterize` function to perform the actual drawing. This also means that, when an image (in the form of a matrix) is being drawn, the vector layer object keeps a *reference* to the image. When the image is no longer in use, the image will be destroyed, by reference counting. The drawing command queue has three primary advantages:

- Ability to draw multiple times using the same vector layer. Suppose that you use the vector layer to construct a HUD that is displayed on top of a 3D scene.
- Easy animation: when you use dynamically changing images (textures), there is no need to reinitialize the 2D vector layer.
- Optimization: Quasar will internally optimize the drawing commands whenever possible (e.g. by grouping similar commands and passing them at once to the hardware).

Only at the rasterization stage, the vector graphics are converted to a bitmap. The resulting bitmap is returned just as a regular M x N x 3 or M x N x 4 matrix, which can easily be used from Quasar (e.g. for future rendering commands). The vector graphics can also be saved to .svg, .eps, .ps or .pdf files.

Functions overview

qvectorlayer class	
Function name	Description
clear	Removes all rendering objects from this vector layer. Use this function to update the content of a vector layer.
setpencolor	Sets the current color of the pen for this vector layer. The pen affects all drawing operations that are strokes or curves.
setbrushcolor	Sets the current brush color for this vector layer. The brush affects all drawing operations that fill a certain area.
setpenwidth	Sets the current pen width for this vector layer. The pen affects all drawing operations that are strokes or curves. Use this function to draw thick lines.
setpendashstyle	Sets the current dash style for this vector layer. The pen affects all drawing operations that are strokes or curves.
drawrect	Draws a rectangle from point start to point end using the current pen color.
fillrect	Fills a rectangle from point start to point end using the current brush color.
drawellipse	Draws an ellipse using the current pen color.
fillellipse	Fills an ellipse using the current brush color.
drawline	Draws a line using the current pen color (see), the current pen width (see) and the current pen dash style.
drawpoly	Draws a polygon using the current pen color. The polygon consists of N vertices.
fillpoly	Fills a polygon using the current brush color. The polygon consists of N vertices.

qvectorlayer class	
drawcurve	Draws a curve using the current pen color. The curve is specified by its control points. The method used internally to draw the curve may be implementation-dependent (e.g. a bezier curve).
pushtransform	Pushes a copy of the current transform matrix for this vector layer to the stack. This function is useful when applying certain geometric transformations that need to be undone later.
poptransform	Pops one transform matrix from the stack and selects this matrix as the current transform matrix.
resettransform	Resets the current transform matrix (by replacing it with an identity matrix).
rotatetransform	Rotates the current coordinate axes by angle-degrees (clockwise).
scaletransform	Scales the current coordinate axes by a constant.
translatetransform	Shifts the current coordinate axes in the horizontal and vertical direction, by performing a translation.
setdefaultfont	Selects the default font for this vector layer
setfont	Changes the current font for this vector layer
drawstring	Draws the specified text using the current font (see).
pushclippingrect	Pushes the current clipping rectangle to the stack.
popclippingrect	Restores the clipping rectangle from the stack.
setclippingrect	Sets the current clipping rectangle. Clipping rectangles are useful for ensuring that all drawing is done inside a given rectangle (ignoring all operations that are outside the rectangle).
drawimage	Draws the specified image at the given location.
drawplot	Draws a plot at the specified location.
show	Displays the content of the vector layer in a new window.
save_eps (1)	Saves the plot as an encapsulated postscript (.EPS) file.
save_eps (2)	Saves the plot as an encapsulated postscript (.EPS) file (with size 8cm x 10cm).
save_pdf (1)	Saves the plot as a PDF file.
save_pdf (2)	Saves the plot as a PDF file (with size 8cm x 10cm).
save_ps (1)	Saves the plot as a postscript file.
save_ps (2)	Saves the plot as a postscript (with size 8cm x 10cm).
save_svg	Saves the plot as a scalable vector graphics (SVG) file.
save	Saves the plot as an image file (by automatically recognizing the file extension). Most image file formats are supported. The default dimensions are 8cm x 10cm, at 144 DPI.
rasterize	Rasterizes the specified surface (to obtain a matrix).

See the Documentation Browser in Redshift (topic User Interface / qvectorlayer) for more information.

Functionality that is currently not implemented yet, but this may be implemented in the future:

- gradient brushes (linear, gradient)
- pattern brushes
- regions
- paths

In practice, this functionality is available in Cairo, however, for OpenGL I would have to write some specific shaders for these operations.

Examples

** Note: for all 2D point coordinates, the x-component is placed first, then the y-component! **

1. Altering an image by adding text:

```
img = imread("image.png")
layer = new(qvectorlayer)
layer.drawimage([0,0], img)
layer.setbrushcolor("red")
layer.drawstring("Text", [10,10], [100,20])
img = layer.rasterize(size(img))
```

   Note that the image may be converted to 8-bit during this operation (which causes values outside the range [0,255] to be clipped and floating point values to be rounded).

2. Adding a vector layer to imshow

```
surf = new(qvectorlayer)
surf.setbrushcolor("green")
surf.drawstring("Marker", [170,170], [240,20])
surf.fillellipse([200,140],[220,160])
surf.setpencolor("red")
for k=0..2..8
    surf.drawellipse([190,130]+k,[230,170]-k)
endfor

h = imshow(imread("lena_big.tif"))
h.addlayer(surf)
```

3. Alpha blending

   In order to enable **alpha-blending**, it is sufficient to pass four-channel RGBA values to the functions setpencolor and setbrushcolor:

```
surf = new(qvectorlayer)
surf.setpencolor([0,0,1,0.2])
surf.drawrect([80,40],[112,72])
surf.setbrushcolor([0,0,1,0.2])
surf.fillrect([120,40],[152,72])
surf.show()
```

   The fourth color component (A) has a value between 0 and 1 where 0=transparent and 1=opaque.

   To draw images with an **alpha channel**, it suffices to create a four component RGBA image where the alpha channel has values between 0 (transparent) and 255 (opaque). Then you can use the function drawimage:

```
surf = new(qvectorlayer)
overlay = zeros(32,32,4)
overlay[:,:,1] = 255
overlay[:,:,3] = 20
surf.drawimage([40,40],overlay)
surf.show()
```

4. imshow overlay

An overlay for the function imshow can be created by simply catching the imshow object and calling the method addlayer. The coordinate system for the vector layer object is initialized automatically to correspond with the coordinate system of the image (i.e. the top-left corner of the image is [0, 0], the bottom-right corner is [size(img,1), size(img,0)]).
approach has the advantage that the vector layer is automatically rescaled when the user zooms in. The vector layer is rendered as an overlay at the best resolution for the selected zooming level.

```
layer = new(qvectorlayer)
layer.setbrushcolor("red")
layer.drawstring("Lighthouse", [10,10], [100,20])
layer.setpencolor("red")
layer.drawline([10,30],[200,30])

img = imread("lighthouse.png")
h = imshow(img)
h.addlayer(layer)
```

5. drawing text that is rotated 90°
6. user interactivity: drawing contours

2. Plot and scatter functions

The built-in functions plot and scatter also use the 2D drawing functionality internally. It is possible to select the rendering method using the popup menu of the plot, or using the function set_renderer:

```
h = scatter(x, y)
h.set_renderer("opengl")
```

This is particularly useful when large amounts of data are rendered. With the OpenGL renderer, displaying 100000 points should be very smooth.

Note that in case you want to make a plotting animation (by redrawing the plot over again), it is best to create a form with a display:

```
frm = form("Animated plot demo")
frm.width = 600
frm.height = 800
frm.center()

x = linspace(0,1,256)

disp = frm.add_display()

a = 0
while !frm.closed()
    a += 0.01
    y = sin(a) * sin(2*pi*x)
    disp.plot(x, y, x, -y).set_renderer("opengl")
    ylim([-1,1])
    xlabel("x")
```



```
        ylabel("sin(x)")
        title("Animated plot")
        pause(0.01)
    endwhile
```

It is possible to draw a plot onto a vector layer. This allows plots to be drawn on top of images, videos or any user-created content. This is done using the qvectorlayer.drawplot function.

```
import "imhist.q"

im = imread("lena_big.tif")

hist = imhist(im)
hist = hist / max(hist)

hold("invisible") % hides the plotting window
p = plot(hist)
print "old ticks: ", p.xtick
p.xtick = [0, 128, 256]
p.xticklabel = {"min", "medium", "max"}
print "new ticks: "
p.ytick = [0, 0.25, 0.50, 0.75, 1.0]
title("Histogram")

layer = new(qvectorlayer)
layer.drawplot([0.1,0.5],[0.8,1.0], p) % > DRAW PLOT

h = imshow(im)
h.addlayer(layer)
```

Note that it is necessary to make sure that the plot function does not create an additional output window (as it normally does). This can be obtained by calling the function hold(" invisible ") with parameter " invisible ". The function drawplot has the following signature:

```
function [] = qvectorlayer.drawplot(start : vec2, end : vec2, p : qplot)
```

where start is the left-top coordinate of the plotting rectangle, relative to the current viewport (for imshow, the viewport [0,0]−[1,1] corresponds to the displayed image). end is the right-bottom coordinate of the plotting rectangle. Note that by calling resettransform it is always possible to set a new coordinate system (e.g., relative to the display window rather than the displayed image).

3. 3D rendering

The 3D vector layer class (qvectorlayer3d) can be seen as an extension of the 2D vector layer class (qvectorlayer) to three dimensions. Where for qvectorlayer the drawing functions accept 2-D coordinates, qvectorlayer3d will accept 3-D coordinates. The 3D rendering itself is exclusively done via OpenGL (there is no option to choose Cairo, because Cairo only supports 2D drawings). To display a 3D vector layer object, a form must be created first, with a display containing an opengl_renderer. Next, the 3D vector layer object can be attached to the opengl_renderer.


```
frm = form("3D rendering demonstration")
frm.width = 1024
disp = frm.add_display()

layer = new(qvectorlayer3d)

... % drawing functions

renderer = disp.create_opengl_renderer()
renderer.background_color = "black"
renderer.enable_zbuffer = true
renderer.draw_backfaces = true
renderer.add(layer, "layer")
```

The renderer object has the type qgldisplay and has the following properties/methods:

qgldisplay class	
rasterize	Rasterizes the specified surface (to obtain a matrix).
clear	Removes all the displayed objects from this display
add	Adds a display object to this container.
set	Updates a display object in this container. [deprecated]
set_texture	Sets the current texture that is used for rendering
framerate	Gets or sets the display frame rate (in frames per second). The default is 50.
background_color	Gets or sets the background color for the OpenGL display control
lights	Gets the lights associated to this OpenGL display
zoom	Gets or sets the zoom factor of this OpenGL display (default: 1)
pitch	Gets or sets the camera pitch angle (in degrees) for this OpenGL display
yaw	Gets or sets the camera yaw angle (in degrees) for this OpenGL display
roll	Gets or sets the camera roll angle (in degrees) for this OpenGL display
show_coords	Gets or sets whether the current camera coordinates are shown
show_toolbar	Gets or sets the visibility of the toolbar
draw_backfaces	Gets or sets whether the back facing triangles are drawn (disables back-face culling).
enable_zbuffer	Enables/disables the z-buffer for rendering.
position	Gets or sets the camera position
projectionmtx	Gets the current projection transform matrix
worldtransformmtx	Gets the current world transform matrix
tooltip	Sets or gets the tooltip text for this control
htmltooltip	Sets or gets the tooltip text for this control
height	The control height (in pixels)
width	The control width (in pixels)

The function qgldisplay .add can be used to add a vector layer (either qvectorlayer or qvectorlayer3d) to the OpenGL display. Whenever the display is rendered (during a screen refresh, or during the rasterize function), the all layers of the OpenGL display are processed.

The typical workflow of the 3D rendering in Quasar is then as follows:

```
+-------------------+ +--------------+ +----------------------+ +---------+
|new(qvectorlayer3d)| --> |qgldisplay.add| --> |call drawing functions| --> |rendering|
+-------------------+ +--------------+ +----------------------+ +---------+
```


As in the 2D case, the drawing functions actually have *no direct effect*. Instead, the 3D vector layer keeps a list with all drawing commands. Once the drawing command list is finished, this list is processed during rendering. This also means that, when an image (in the form of a matrix) or a vertex buffer are being used, the vector layer object keeps a *reference* to the image/vertex buffer. When the image is no longer in use, the image will be destroyed, by reference counting.

Real-time animation then becomes really simple: it suffices to overwrite the content of the matrix/vertex buffers, and the changes will be reflected on screen. Internally, Quasar uses CUDA-OpenGL interoperability to transfer CUDA memory blocks to OpenGL.

Note: a 2D vector layer object `qvectorlayer` can also be added to the OpenGL display via `qgldisplay.add`. This allows you to add a 2D graphics overlay (e.g. HUD).

3D Vector layer class

The `qvectorlayer3d` class contains the following properties and method:

qvectorlayer3d class	
clear	Removes all rendering objects from this vector layer. Use this function to update the content of a vector layer.
settexture	Sets the current texture used for filling operations.
unsettexture	Unsets the current texture for filling operations (resulting in untextured primitives in subsequent filling operations).
setcolor	Sets the current color of the pen for this vector layer. The pen affects all drawing operations that are strokes or curves.
setpenwidth	Sets the current pen width for this vector layer. The pen affects all drawing operations that are strokes or curves. Use this function to draw thick lines.
setpointsize	Sets the current point size. The point size affects all point drawing operations.
setpendashstyle	Sets the current dash style for this vector layer.
drawline	Draws a line using the current color, the current pen width and the current pen dash style.
drawlines	Draws N lines using the current color, the current pen width and the current pen dash style.
drawlinestrip	Draws a line strip (a group of connected line segments) using the current color, the current pen width and the current pen dash style.
filltriangles	Fills N triangles using the current color. Each triplet of points is treated as an independent triangle. function [] = qvectorlayer3d.filltriangles(p : cube)
fillindexedtriangles	Fills N triangles using the current color and using the specified set of indices. Each triplet of indices corresponds to an independent triangle.function [] = qvectorlayer3d.fillindexedtriangles(points : cube, indices : cube[int])
filltrianglestrip	Fills N triangles using the current color. Each triangle is defined for each point presented after the two previous points. function [] = qvectorlayer3d.filltrianglestrip(p : cube)
filltrianglefan	Fills N triangles using the current color. Vertices 0, n-1 and n define a triangle.function [] = qvectorlayer3d.filltrianglefan(p : cube)
fillquads	Fills N quads using the current color. function [] = qvectorlayer3d.fillquads(p : cube)
drawpolygon	Draws a single polygon of N points using the current color, the current pen width and the current pen dash style.
fillpolygon	Fills a single convex polygon of N points using the current color.function [] = qvectorlayer3d.fillpolygon(p : cube)
drawpoint	Draws one single point using the current color.
drawpoints	Draws N points using the current color.
drawrect	Draws a rectangle using the current color.
fillrect	Fills a rectangle using the current color.
drawimage	Draws an image in 3D coordinates.
drawbox	Draws an axis-aligned 3D box the current color.
translatetransform	Shifts the current coordinate axes in the horizontal and vertical direction, by performing a translation.
scaletransform	Scales the current coordinate axes by a constant.
rotatetransform	Rotates the coordinate system by angle-degrees (clockwise) around a given vector.
multiplytransform	Applies the specified 4x4 matrix transform.

qvectorlayer3d class	
pushtransform	Pushes a copy of the current transform matrix for this vector layer to the stack. This function is useful when applying certain geometric transformations that need to be undone later (using the function qvectorlayer3d.poptransform.
poptransform	Pops one transform matrix from the stack (pushed using qvectorlayer3d.pushtransform) and selects this matrix as the current transform matrix.
resettransform	Resets the current transform matrix (by replacing it with an identity matrix).
print	Prints text in 3D at the specified position, using the current color.
rasterize	Rasterizes the specified surface (to obtain a matrix).
enable_-zbuffer	Enables/disables the z-buffer for rendering.
enable_-lighting	Enables or disables OpenGL lighting. Note: you can turn on/off lighting at any time in between different rendering commands

Alpha blending

To enable alpha blending, it suffices to pass colors as vectors of 4 components vec4 (RGBA) to the drawing functions. The last component will then be used as an alpha channel (values between 0 and 1), where 0 is transparent and 1 is opaque.

To draw (semi-)transparent images, you can use a similar approach by allocating a matrix with four components. In this case, the R,G,B,A values are between 0 and 255. A=0 corresponds to transparent and A=255 to opaque.

3D vertex formats

In Quasar, it is very easy to define your own vertex formats. The vertices can contain all the properties you want (e.g. position, normals, texture coordinates, color, ...). By defining a vector of vertices, you obtain a vertex buffer that can then transparently be passed to the qvectorlayer3d class for rendering.

```
type vertex : class
    position : vec3 % vector of length 3
    normal : vec3 % vector of length 3
    color : vec4 % RGBA
    texcoord : vec2 % texture coordinates
endtype

type vertex_buffer : vec[vertex]
```

The vertex buffer contains the vertices of the primitives in consecutive order. For example, for triangles, the length of the vertex buffer needs to be a multiple of 3. This way, you can draw point lists, line lists, triangle lists, quad lists etc. In case you are not rendering primitives with textures, you can omit the texture coordinates, as in the following example.

```
type vertex : class
    position : vec3 % vector of length 3
    normal : vec3 % vector of length 3
    color : vec4 % RGBA
endtype
```


Drawing indexed primitives

It is possible to identify the primitives using both an index array and a vertex buffer. Suppose that several primitives (faces) reuse the same vertices multiple times, by storing each vertex only once and by letting the indices point to the correct vertices, often a lot of memory can be saved. For example, a cube can be drawn as follows:

```
type vertex : class
    position : vec3
endtype
indices = `0,1,2,0,2,3, % Front
          5,4,7,5,7,6, % Back
          4,0,3,4,3,7, % Left
          1,5,6,1,6,2, % Right
          3,2,6,3,6,7, % Top
          4,5,1,4,1,0' % Bottom
layer.translatetransform([-0.5,-0.5,0])
p = vec[vertex](8)
p[0] = vertex(position:=[0,0,1])
p[1] = vertex(position:=[1,0,1])
p[2] = vertex(position:=[1,1,1])
p[3] = vertex(position:=[0,1,1])
p[4] = vertex(position:=[0,0,0])
p[5] = vertex(position:=[1,0,0])
p[6] = vertex(position:=[1,1,0])
p[7] = vertex(position:=[0,1,0])
layer.fillindexedtriangles(p, indices)
layer.translatetransform([0.5,0.5,0])
```

There are 36 indices and 8 vertices. Without an index array, we would have to use 36 vertices to store the cube data. Note that OpenGL restricts the data formats for the index buffer to vec[int], vec[uint32] and vec[uint16].

4. 3D rendering examples

Point-cloud rendering example

As an example, consider the rendering of a point cloud. To ease working with point coordinates and colors, we define our own vector3 and rgb_color classes. In this case, we only want to specify position coordinates and RGB colors to the vertices. Therefore, the vertex definition only consists of a position and a color-field.

```
% Source: Samples/opengl_pointcloud.q
import "Quasar.UI.dll"

type vector3 : class
    x : scalar
    y : scalar
    z : scalar
endtype

type rgb_color : class
    x : scalar
    y : scalar
    z : scalar
```



```
endtype

type vertex : class
    position : vector3
    color : rgb_color
endtype

function [] = update_pointcloud(v : vec[vertex], A : mat, t : scalar)
    % Calculate the coordinates based on the current time-index
    coord = 12*abs(A).^(3+2*sin(t)).*sign(A)

    % Fill the vertex buffer with the vertices
    #pragma force_parallel
    for k = 0..numel(v)-1
        r = sum(abs(coord[k,0..2]))
        v[k] = vertex(position:=vector3(x:=coord[k,0], y:=coord[k,1], z:=coord[k,2]),
                      color:=rgb_color(x:=r, y:=0.0, z:=1-r))
    endfor
endfunction

function [] = main()
    frm = form("OpenGL point-cloud visualization demo")
    frm.width = 800 % Sets the width of the form
    disp = frm.add_display() % Attaches a display object
    disp.sync_framerate(40) % Render at 40 FPS
    renderer = disp.create_opengl_renderer() % Creates the OpenGL 3D renderer
    layer = new(qvectorlayer3d) % Creates a 3D vector layer
    renderer.add(layer, "layer") % Adds the vector layer to the renderer
    frm.show() % Displays the form (otherwise the form remains hidden)

    % Generate a set of random coordinates
    A = rand(1000000,3)-0.5
    v = vec[vertex](size(A,0)) % Initializes the vertex buffer
    layer.drawpoints(v) % Draws the vertices onto the 3D vector layer
    % Note that the vertices have not been initialized at this stage.
    t = 0.0
    while !frm.closed()
        t += 0.01 % Time step
        update_pointcloud(v, A, t) % Updates the point cloud with new vertex data
        pause(10)
    endwhile
endfunction
```

Lines, images, alpha blending, rasterization

This example shows the use of images and alpha blending for 3D rendering. Once the 3D vector layer is ready, the complete layer is rasterized, resulting in a RGB color image and a depth image.

```
import "Quasar.UI.dll"

frm = form("Surface plot")
frm.width = 800
disp = frm.add_display()
disp.width = 1024
disp.height = 768
```



```
% Loads a test image - assigns an alpha channel
im = 128*ones(512,512,4) % Set alpha channel to 128
im[:,:,0..2] = imread("lena_big.tif")

layer = new(qvectorlayer3d)
layer.scaletransform([0.5,0.5,0.5])
layer.drawbox([-1,-1,-1],[1,1,1])
layer.enable_zbuffer = false % Disable z-buffer for transparent rendering
layer.setcolor("red") % The colors act as a color filter. It is also possible to specify an alpha
                      % here. This alpha channel would be used for a second multiplication.
layer.drawimage([0.5,0,0],[0,1,0],[0,0,-1],im)
layer.setcolor("green")
layer.drawimage([0,0.5,0],[1,0,0],[0,0,-1],im)
layer.setcolor("blue")
layer.drawimage([0,0,0.5],[1,0,0],[0,1,0],im)
layer.setcolor("yellow")
layer.drawimage([0,0,-0.5],[1,0,0],[0,1,0],im)
layer.setcolor("cyan")
layer.drawimage([0,-0.5,0],[1,0,0],[0,0,-1],im)
layer.setcolor([1,0,1,0.5]) % magenta
layer.fillrect([-0.5,0,0],[0,1,0],[0,0,-1])
layer.enable_zbuffer = true % Enables the Z-buffer again

renderer = disp.create_opengl_renderer()
renderer.background_color = "white"
renderer.enable_zbuffer = true
renderer.draw_backfaces = true
renderer.add(layer, "layer")
renderer.pitch = 340

colorimg = zeros(480,640,4)
depthimg = zeros(480,640)
layer.rasterize(colorimg, depthimg)
imshow(colorimg)
imshow(depthimg)

frm.show()
frm.wait()
```

