Drawing with Mgraphics in Max

The Max Mgraphics package is a vector drawing system of the sort used in Adobe Illustrator and other modern drawing applications. The basic patch is similar to the jit.lcd patch. A jit.mgraphics object is used with the matrix dimensions as arguments (no need to specify type-- it's going to be a 4 char). Drawing occurs when commands are received, and the current state of the canvas is output when a bang comes in. The difference between mgraphics and lcd graphics is primarily in the underlying code. The lcd is a bitmapped world, with images specified pixel by pixel. When you send lcd a lineto command the program works out which pixels in the output memory need to be colored and colors them. Every other command does the same sort of thing, even when it means coloring some pixels twice. Drawing simple blocks of color is easy enough, but anything with complex curves requires some serious math on the part of the patch-- You will have seen this in other tutorials where we essentially calculate one pixel at a time. Sending one message per pixel really slows things down.

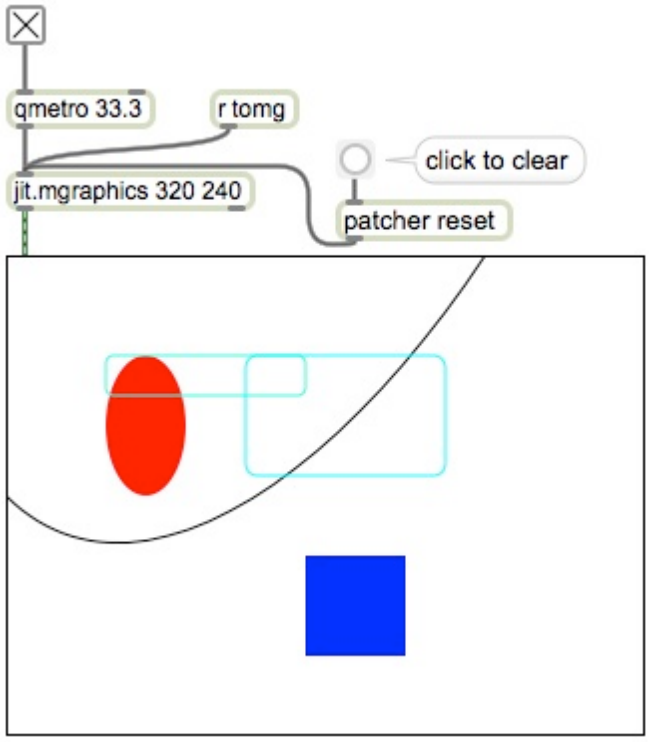
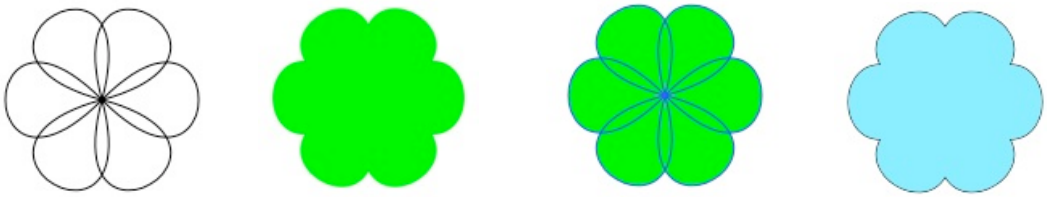


Figure 1.

Basic concepts

Vector drawings are constructed from "paths", in which we specify how far and in which direction a virtual pen should move. Once the path is specified, we can assign a color, then complete it with either a stroke or fill command. Of course the result will be the same in the end, because the output of jit.mgraphics is a matrix of pixels

just like jit.lcd. The advantages to the path approach are the efficiency of the drawing code (maybe 10x as fast as jit.lcd), and more drawing commands, such as smooth curves and the ability to draw at any angle.



Stroked Filled Fill-stroke Stroke-fill
Figure 2.

There are several metaphors to keep track of when working with mgraphics. The first is the notion of *source*, *mask*, and *destination*. The source may be a preexisting image, a solid color or a gradient. The destination is the canvas the results are painted on, in this case a jitter matrix. The mask controls which parts of the source are transferred to the destination. The combination of source, mask and destination are called the *context*.

The path is (metaphorically) applied to the mask. The path is built by a series of commands such as move_to, line_to, curve_to, ellipse and so on. Once the path is built, the source is transferred to the destination by either a stroke or fill command. Figure 2 shows the results of a path stroked, filled, filled then stroked, and finally stroked, then filled. You can see that stroke and fill define different areas, with the fill fitting neatly within the stroke. Actually, if you look carefully, the fill covers half of the stroke (you probably need a retina display to see it properly). Stroked lines are centered on the coordinates specified, and fills extyend to the coordinates[1].

The path building commands take image coordinates as arguments. There are two styles (chosen by an attribute): *absolute* pixel addresses with 0,0 in the upper left corner and *relative*, with 0,0 centered and the height ranging from -1.0 to 1.0. The left and right will vary according to the aspect ratio. I'll be using the absolute coordinates in most of these examples. Much of the power of mgraphics comes from the ability to play with the coordinates. For convenience, we can think of the coordinates used as arguments as in the *user coordinate system*. There is also a fixed *destination coordinate system*. All path arguments are processed by a transformation matrix to convert user to destination coordinates. This means we can use a set of commands with the same arguments to draw anywhere on the canvas!

Figure 3 shows these transformations in use. The basic path is a simple loop labeled 0. It is defined with some negative numbers, so only half of it shows. It is a closed loop that begins at and returns to 0,0. The path command used was *curve_to 80 -80 80 80 0 0, stroke*. This was followed by the command *translate 160 120, move-to 0 0*.

[1] In jit.lcd, pixels are drawn with their upper left corners at the coordinates.

Translate moves the drawing origin by the number of pixels specified. When the same curve_to command was repeated, it produced the loop labeled 1.
The next command was *rotate 1.047*. This turns the entire user coordinate system 1/6th of a circle. Point 0 0 remains where it is, but all other points are rotated. Now the path drawing command will produce loop 2. Repeating the rotate and path commands four more times will complete the image.

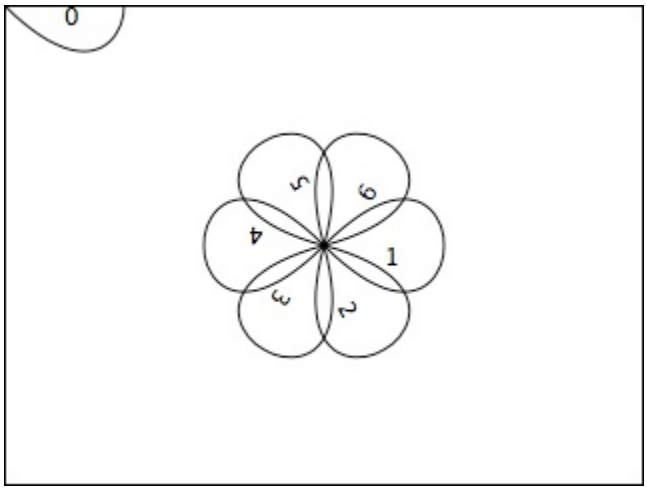


Figure 3.

The complete script for the central image (minus the numbers) is then:

```
Translate 160 120
Move_to 0 0
curve_to 80 -80 80 80 0 0
rotate 1.047
curve_to 80 -80 80 80 0 0
rotate 1.047
curve_to 80 -80 80 80 0 0
rotate 1.047
curve_to 80 -80 80 80 0 0
rotate 1.047
curve_to 80 -80 80 80 0 0
rotate 1.047
curve_to 80 -80 80 80 0 0
stroke
```

Figure 4 shows a patch to create this image. There are various ways to enter the messages, but the message box is probably the simplest. This lets us use an uzi to produce repeated steps.

Translation and rotation may seem an awkward approach to drawing, but it's really like turning your sketchpad to get a comfortable pen angle. The coordinates can be put back with the command *identity_matrix*.

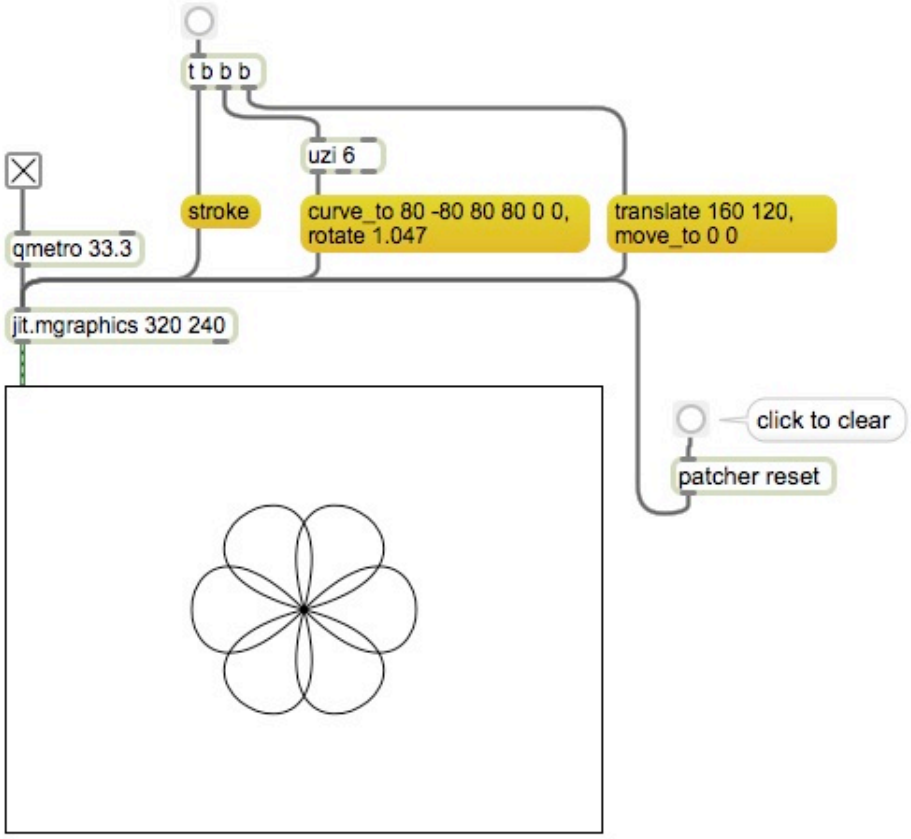


Figure 4.

The patch of figure 4 illustrates the three major phases of drawing any image:

- Set up the context (source and transforms)
- Create the path
- Stroke or fill

Here these phases are managed by the output of a trigger object. Note that I have stolen the reset subpatch from the help file. We'll explore what's in there a bit later.

Source Color Commands

The first phase of drawing is to set the source color. Here are the most pertinent commands: (Letters like r g b indicate arguments to the command)

set_source_rgb r g b	sets the entire source area to a single opaque color. Argument range is 0.0 to 1.0.
set_source_rgba r g b a	Sets source to single transparent color.

These are the most often used source commands. The color you select here will be used for following paint, stroke or fill commands.

Paint commands

You can transfer the entire source to the canvas with the paint command.

paint	Fills canvas with source color. If source is transparent existing image still shows through.
paint_with_alpha a	Fills canvas with source color using alpha supplied as argument. Any previous drawing will still show through. If source is transparent, the source alpha is multiplied by the argument.

Paint is the only way to set the entire canvas the same color. To get a black background do this:

```
set_source_rgb 0 0 0, paint
```

The arguments 1 1 1 will produce a white background. We'll visit more source commands later.

Path Commands

A path is a series of instructions. The instructions are built up in a list until a stroke or fill command is executed (or new_path, if you want to discard some instructions without drawing them). The first command is usually move_to.

new_path	Clears all unexecuted path instructions. The drawing cursor is left at 0 0.
move_to x y	Moves the drawing cursor to the location x, y.
rel_move_to x y	Moves the drawing cursor x points right and y points down. (Use negative x and y to move opposite direction.)
line_to x y	Defines a line from the current cursor to x, y.
rel_line_to x y	Defines a line from the current cursor that extends x points right and y points down.
close_path	Adds a straight line from the current point to the location of the last move_to.
path_roundcorners cr	Rounds the corners in the current path with a radius of cr. A corner is any join between segments. Note that a line_to the start of another line does not create a join- the command close_path should be used to finish polygons if you want to round them.
arc xc yc r a1 a2	Defines an arc that is drawn clockwise. xc and yc define the center point, and r is the radius. These define the staring point. If the cursor is

	elsewhere, an unintended line will be added. (new_path will eliminate this.) a1 and a2 define the starting and ending angle. The angle is in radians, and 0 is at the rightmost point of the (complete) circle. To calculate radian values, divide 2pi by the fraction of a circle you want. Important points are 1.57 (bottom) 3.14 (left) and 4.71 (top). If you fill an incomplete arc, the fill ends on a chord between the endpoints.
arc_negative xc yc r a1 a2	Same as arc, but draws the other way (counterclockwise) around the circle.
ovalarc xc yc rx ry a1 a2	Similar to the arcs, but the radius for x and y can be different, resulting in an oval.
elipse x y w h	Defines an ellipse within the box with the upper left corner at x y, width of w and height of h.
rectangle x y w h	Defines a rectangle with the upper left corner at x y, and width of w and height of h.
rectangle_rounded x y w h ow oh	Defines a rectangle with rounded corners. The rounding is as if you took an ellipse of size ow by oh, cut it into four parts and used these for the corners.
curve_to x1 y1 x2 y2 x3 y3	Defines a Bezier curve from the current cursor location to x3,y3, using x1,y1 and x2,y2 as control points. The control points determine the direction of the ends of the curve as well as the general shape. (The curve does not go through the control points.)
rel_curve_to x1 y1 x2 y2 x3 y3	Defines a Bezier curve, with all x and y values relative to the current cursor location.
set_line_width w	Sets the line width to w pixels.
set_line_cap type	Sets the style for line endings. Type may be butt, round, square.
set_line_join type	Sets the style for line joints. Type may be miter, round or bevel.

Stroking and Filling

Stroking and filling are the final steps of drawing. As seen in figure 2, stroking produces lines and filling produces areas. Stroking will make all path segments defined since the last stroke (or new_path) command visible. The path is then discarded, unless the stroke_preserve command is used. Fill is the same way, the path is discarded unless fill_preserve is used. If you want to both stroke and fill a path (or vice-versa) use the preserve variant of the first command.

Strokes and fills copy the color from the source based on destination coordinates. The stroke_with_alpha variants allow you to specify transparency with an argument. The argument is multiplied by the alpha of the source.

If you fill an open path, the fill will be clipped to a line from the end to the beginning of the path. (Move_to defines endings and beginnings.) Figure 5 shows two filled open paths.

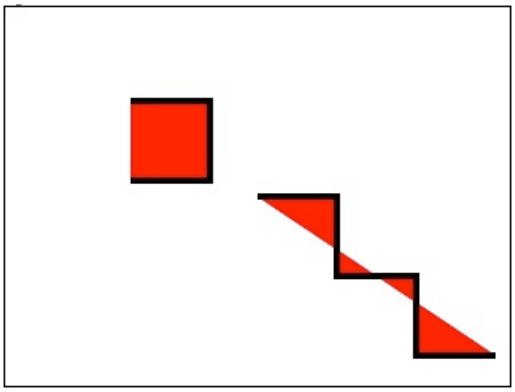


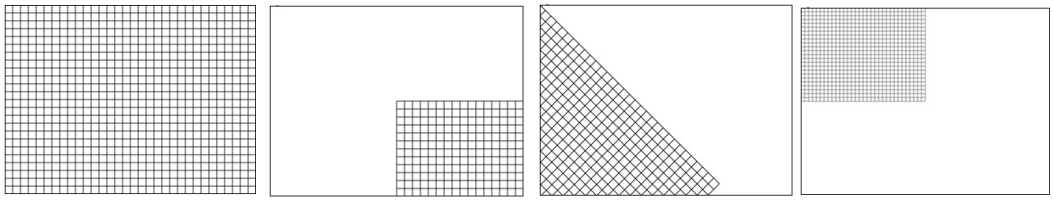
Figure 5

Stroke and fill commands:

stroke	Paint path with source colors, clearing the path afterwards.
stroke_preserve	Paint path with source colors and leave the path for further operations.
stroke_with_alpha a	Stroke the path, using the alpha provided.
stroke_preserve_with_alpha a	Stroke and preserve the path using the alpha in the argument.
fill	Fill current path with source colors, clearing the path afterwards.
fill_preserve	Fill current path with source colors and leave the path for further operations.
fill_with_alpha a	Fill the path using the alpha in the argument.
fill_preserve_with_alpha a	Fill the path using the alpha in the argument, preserving the path.

Transforms

As I discussed above, a lot of the power of mgraphics comes from the ability to change the coordinate system prior to defining part of a path. Figure 6 shows the possibilities-- a translation moves the origin horizontally and vertically, and a rotation rotates the coordinates around the origin. Scale changes the spacing of the coordinates. The command *identity_matrix* puts the origin back. In absolute mode the origin is at the top left of the matrix. In relative mode, the origin is in the center.



Normal Translated 160 120 Rotated 0.78 Scaled 0.5 0.5
Figure 6.

There are four transformations available. These are cumulative, so they affect further transformations as well as path commands.

identity_matrix	Restores the coordinates to default.
scale sx sy	Changes the spacing of the coordinates. X values are multiplied by sx, Y values are multiplied by sy.
translate tx ty	Adds tx to all X values and ty to all Y values.
rotate n	Rotates the coordinates by n radians.

Advanced Transformations

If you understand the math behind these transformations, you can create your own with the *set_matrix* command. This does not refer to jitter matrices, but rather the math matrix used in the underlying affine transformation used on the path coordinates. (That is a tutorial for another day.) You can easily save the matrix created by a series of transform commands however. The command *get_matrix* will report the current matrix from the right outlet. I capture that in a message box, change the preamble to set_matrix and I have a command ready to use.

set_matrix xx yy yx yy xO yO	Set up the current transform matrix in the form: $\begin{bmatrix} xx & xy & xO \\ yx & yy & yO \\ 0 & 0 & 1 \end{bmatrix}$
transform xx yy yx yy xO yO	Multiplies the current transformation matrix by the values given. This has the same effect as repeated commands.
get_matrix	Returns the current transform matrix.

Even if you don't understand the math, you can discover some interesting transforms by trial and error. For instance, the identity matrix[2] is [1 0 0 1 0 0]. Change that to [1 0.3 0 1 0 0] and you will get a transform that steps x right by 0.3 for every step in the y direction (shear transform).

[2] i.e. a transform that makes no change.

Text

Text can be added at any point in the drawing. There are two basic commands: *show_text* and *text_path*. Both are followed by the desired text in double quotes. (If you need to create the text on the fly, the sprintf object with the symout argument may be the best tool--see my tutorial Max&ASCII.) *Show_text* will post the text immediately at the current cursor position. This will not clear the path, but it will move the cursor. *text_path* adds the text to the current path to be included in the next stroke or fill. Stroke produces the outline of the path text, as shown in figure 7. Current colors and transforms apply to both styles of text entry.

Figure 7.

When working with text, you often need to determine where the cursor is after the text is shown (this will vary with font), as well as where it is safe to draw. The *text_measure* command will report the width and height of any text included as an argument. The *font_extents* command will show how much clearance to allow above and below the text baseline. Note that these numbers may be slightly inaccurate for some of the fancier faces.

Text commands

show_text "theText"	Immediately posts the text at the current cursor position. The text is stroked and filled.
text_path "theText"	The text is added to the current path.
select_font_face name slant weight	Chooses the font to use. Names include all installed fonts. Slant may be normal or italic, and weight may be normal or bold.
font_extents	Reports ascent, descent and height from the right outlet with the prefix font_extends.
text_measure	Reports the width and height of a rectangle enclosing the text with current settings.
getfontlist	Lists all installed fonts at the right outlet, with the prefix getfontlist. This is really awkward to use. The fontlist object with a umenu (as illustrated in the fontlist help patcher) is much more convenient.

Source Patterns

The source can be more complex than a simple color. Gradient patterns can be specified and images can be loaded from files. To generate a pattern, you first specify a vector for the gradient. That's a line from X_0,Y_0 to X_1,Y_1. This defines a direction and length for the color change. You then add "color stops" along the vector to specify the colors to use. Color stop positions are specified by proportion of the vector. A stop at 0 is at the beginning of the vector, a stop at 0.5 is the middle, and a stop at 1 is the end. Figure 8 illustrates the effects of color stop position. The gradient vector is from 60, 40 to 260, 200. The stops on the left image are at 0 and 1, the stops for the right image are at 0.25 and 0.75.

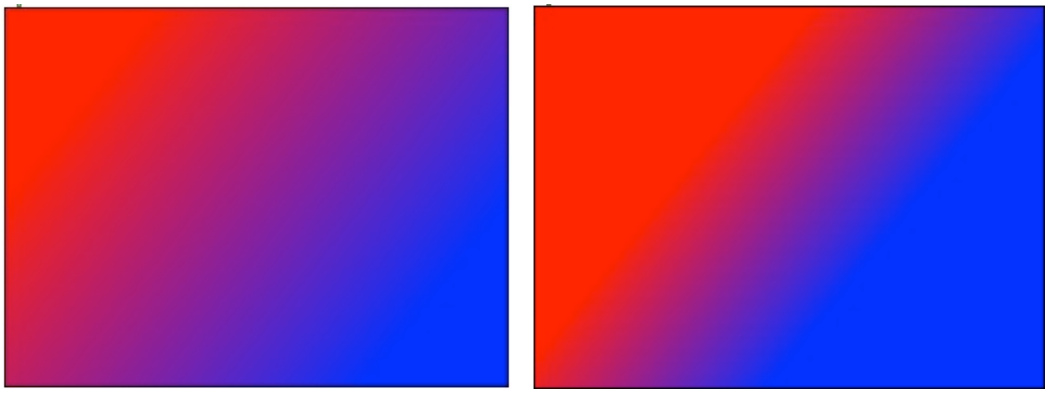


Figure 8.

Actually, in order to produce the right image, I had to establish a red color stop at 0 in addition to one at 0.25. I don't know if this is a bug or undocumented behavior, but it's probably always a good idea to clearly define the ends of the vector. Figure 9 shows how the gradient affects fill and stroke.

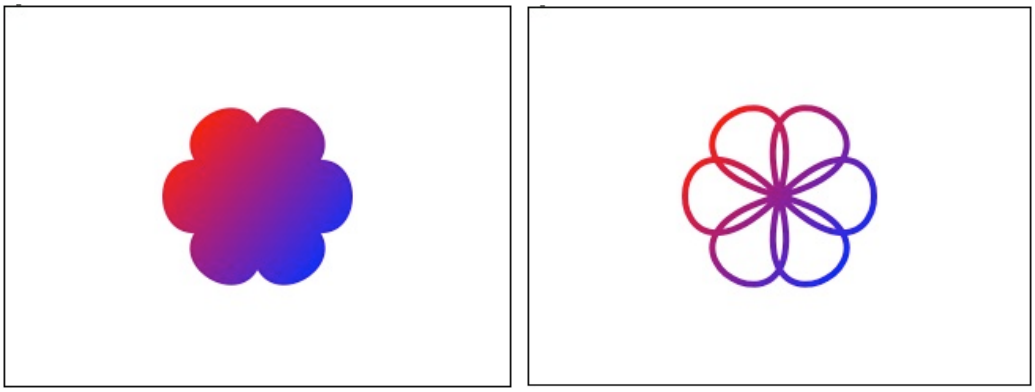


Figure 9.

When you create vectors, you give them a name, and they are saved in the current drawing context. To use them, use the command *set_source* with the pattern name.

Gradients can also be radial, as in figure 10.

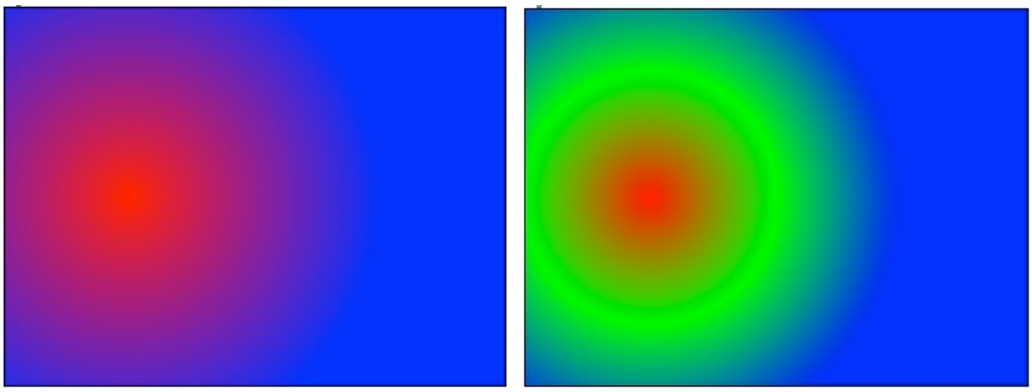


Figure 10.

Radial gradients are also created with a vector. The beginning of the vector will be the center and the end of the vector will be the circumference of the gradient. As with the linear gradient, color stops define the color at proportional points along the radius. The right image was defined with a vector from 80, 120 to 240, 120 and color stops at 0 (red) 0.5 (green) and 1 (blue). [Note: there is a third type of gradient, in which the colors are mapped from the circumference of one circle to another. The *pattern_create_radial* command actually requires enough arguments to perform this function, but it is not currently working.]

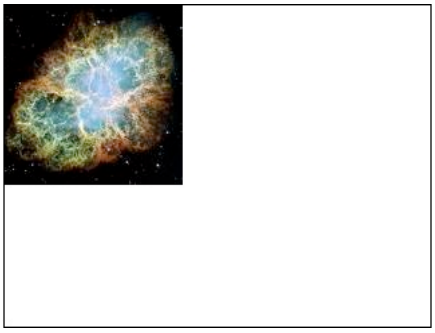
Gradient Commands

pattern_create_rgba name	Creates a solid color with the name given.
pattern_create_linear name x0 y0 x1 y2	Creates an uncolored gradient with the name given along the vector x0, y0 x1,y1.
pattern_create_radial name x0 y0 r0 x1 y1 r1	Creates an uncolored radial gradient with a radius defined by x0,y0 x1,y1. r0 and r1 are non functional but required.
pattern_add_color_stop_rgba name p r g b a	Adds a color stop to the named gradient at proportional position p.
set_source name	Sets the named pattern as the drawing source.
pattern_destroy name	Disposes of the named pattern. You can add as many color stops to a gradient as you wish, but the only way to get rid of them is destroy the pattern and start over.
pattern_set_extend name type	Sets what happens in the area beyond the patter's defining vector. Types may be none, repeat, reflect, pad. [This feature is not currently implemented.]
pattern_get_extend name	Reports the type use in the command above.

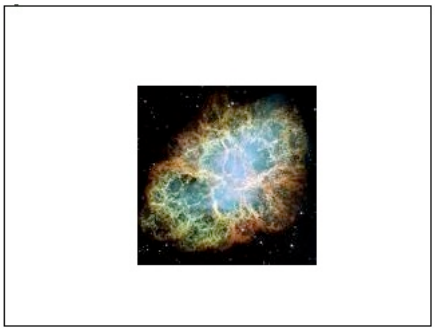
Images as Source

Images can be drawn directly into the source or destination. Images can come directly from files or from a named matrix. I prefer the latter as it provides a convenient way to modify the image size. There are several ways to paint images:

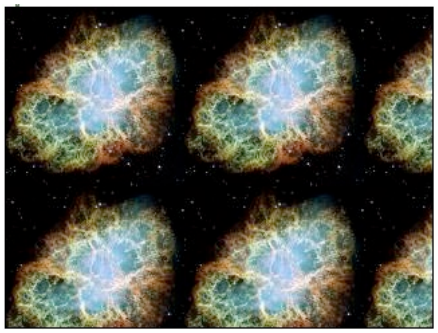
image_surface_draw_fast image	This will paint the image directly to the output. There is no rescaling if the sizes don't match.
image_surface_draw image	This will paint the image to the output using the transformation matrix for offset and scale. However, if the transformation matrix includes a rotation, nothing is drawn.
image_surface_create name image w h	Creates an image surface from image with the supplied name and rescaled to size. The name can be used in drawing or source commands.
set_source_surface image	Draws the named image to the source, ready to use with paint, stroke, and fill. If the image is small, it will be repeated.
pattern_create_for_surface name wh	Acquires an image from location wh and loads it into a source pattern with the supplied name. It can be used for drawing with the surface_draw commands or set as the source for paint, stroke and fill.



image_surface_draw_fast



image_surface_draw (with translate)



set_source_surface

pattern_create_for_surface

Figure 11.

Figure 11 shows the effects of the draw and create functions (the latter after a paint command). Figure 12 shows the result of stroke and fill after the pattern_create_for_surface operation. This is the most flexible, because you can load a variety of images with their own names.

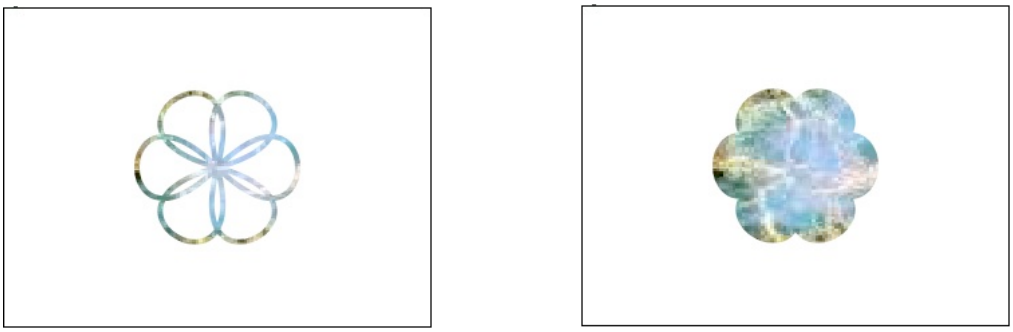


Figure 12.

Source transforms

You can also set transforms to use with patterns or images that have been created for surface. These transforms are similar to the path transforms, offering offset, scale and rotation functions. However, it is a quirk of the underlying graphics package that the arguments for pattern transforms are opposite those for paths. The signs for offset and rotate are negative and the scale is inverted.

- To offset the pattern origin to 160 120, use -160 and -120.
- To rotate the pattern clockwise, use a negative angle.
- To scale the pattern by 0.5 use 2 as the argument.

This is all because the transforms are applied after the pattern is created, not before, as with paths. Once a pattern has been set as source, no more transforms may be made. Figure 13 shows these effects. They may be combined.

Original Translate -160 -120 Scale 2 2 Rotate - 1.047

Figure 13.

Commands

pattern_identity_matrix name	Removes all transforms from the named pattern.
pattern_translate x y	Moves pattern origin to -x -y.
pattern_scale sx sy	Scales pattern size by 1/sx 1/sy.
pattern_rotate a	Rotates pattern image by a radians **counter-clockwise**.

pattern_get_matrix	Reports the current pattern transform matrix.
pattern_set_matrix xx xy yx yy xO yO	Sets the pattern matrix to $\begin{bmatrix} xx & xy & xO \\ yx & yy & yO \\ 0 & 0 & 1 \end{bmatrix}$

Miscellany

There are several commands that are still under development at the time of this writing. These are things that are not functional, or require some not-yet-implemented functions to be of any use.

scale colors

scale_source_rgba r g b a will affect the next set_source_rgba command, scaling all of the values provided. For instance if you send scale_source_rgba 0.5 0.5 0.5 1, set_source_rgba 1 1 1 1 will produce a gray. The scale commands are cumulative, so if scale by 0.5 is followed by a scale by 2, you will get back to the original color. However, if you ever scale by 0, set_source_rgba will not work properly until the jit.mgraphics object is reset. (You can reset any object by editing it-- just type in a space.)

push_group

groups are a powerful concept that will let us make drawings from earlier drawings or use our drawing as a source. However, this set of functions is now incomplete, so push_group just stops all drawing.

Save context

Save and restore are related functions. If you save a drawing context, it can be restored later. However, if you restore before saving, or restore more times than you have saved, the object just stops working.

Mgraphics is still a work in progress. Watch for updates.