Evas Programmers Guide

A guide to using Evas for Linux software development.

By Carsten Haitzler
raster@valinux.com
raster@rasterman.com

"When your current canvas just isn't good or fast enough for your software's needs..."

Prologue

The world is never perfect. There are always problems, and sometimes there are solutions, some of them bad, some of them good and some of them completely insane. Evas is a solution to a problem, that at the time had no other solution. The problem was that we needed a rendering abstraction layer for X11 that allows for alpha blending, anti−aliasing and image manipulation on a structured, rather than immediate−mode level. One that would also optimise the display of such structured objects and also allow for hardware on existing and future machines to accelerate this rendering without requiring the CPU to do all the work.

After looking around, it was found there were no other acceptable solutions that met these requirements at all under X11 that were available as Open Source, and would run on Linux. Thus was born Evas. Its goals are not lofty. Its implementation is not perfect. It is not miniscule nor all−encompassing. It does not do 3D. It does not solve every graphics problem ever invented. What it does do, is that it solves the problem it was intended to solve. It does it and does it well. If you find you need something like this to solve your problems, then Evas may be your answer. If it is not quite the perfect solution you may find it is a simple matter to add some code to make it the perfect solution.

Contents

Table of Contents

What is a Evas?

Evas is a canvas library, designed to enable the software developer using it produce windows that contain the rendered output of a canvas − specifically an Evas canvas. There are many fundamental differences between the drawing methods most programmers are used to (immediate mode drawing) and those used by a canvas (structured mode drawing). First an explanation of what constitutes each method of drawing is in order.

Immediate Mode Drawing

Immediate mode drawing is probably the most commonly found drawing mechanism. It is often the basic mechanism by which any display system and hardware work. It usually involves issuing drawing commands as they are needed and then having them processed by the display system immediately. After the draw is done it appears in the destination − but no information about it is actually ever retained. If that section of the screen is damaged (i.e. overlapping graphic is removed or something changes) the commands to draw that section of the screen need to be re−issued.

This kind of rendering system puts less onus on the system to know much about what is going on and leave that all up to the programmer writing for this system. It is good because what the software does is very close to what the hardware does, and thus allows for better control by the software, but the problem is, that this is only useful when the programmer knows the behaviour of the hardware, its abilities and how to optimise for it. Since hardware varies so much between machines these days, this is a hard job. This also requires the programmer to have a detailed knowledge of graphics programming, principles and the experience, the will, and the time, to deal with these issue to make a reasonably well written set of drawing routines for their application. Often this is not true. Sometimes the experienced graphics guru wishes simply to save time, or the programmer does not want to know much about the low level details and just wants to get the job done or get the effect they are after.

Structured Mode Drawing

Structured mode (as it's name implies) involves drawing graphics by describing its structure to the system, instead of drawing it immediately. The structured mode system handles the drawing of the objects however it sees is the best way to achieve that. The application developer describes the contents of the window or screen to the system using primitives just as they would in immediate mode, but the primitives are persistent and thus the drawing system will handle maintaining their integrity on the screen, rather than the programmer. Since it also knows where these primitives are ahead of time, and what their properties are, the system is able to perform complex optimisations and analysis on the current display to minimise the effort in redrawing all or any section of it since it was last updated. Objects remain in the system until destroyed, so the application still has to manage their creation and destruction, but the drawing is all handled by the structured mode system, and not the software the developer is working on, freeing them to think of higher level problems instead.

Evas is a structured mode rendering system and thus if you want to use it, you need to think in terms of a structured mode system instead of an immediate mode one. Once you get the hang of this, it all becomes easy from there on in and Evas will begin to show its benefits.

The Canvas

The canvas is an area − like a virtual piece of paper, that contains objects that are in that canvas. The actual size of that area is infinite, but the canvas only looks at a certain finite sized section of that area at any one time. This rectangular section of the canvas area being looked at is called a viewport. The only part of that canvas that is ever visible to the user is the area this viewport covers.

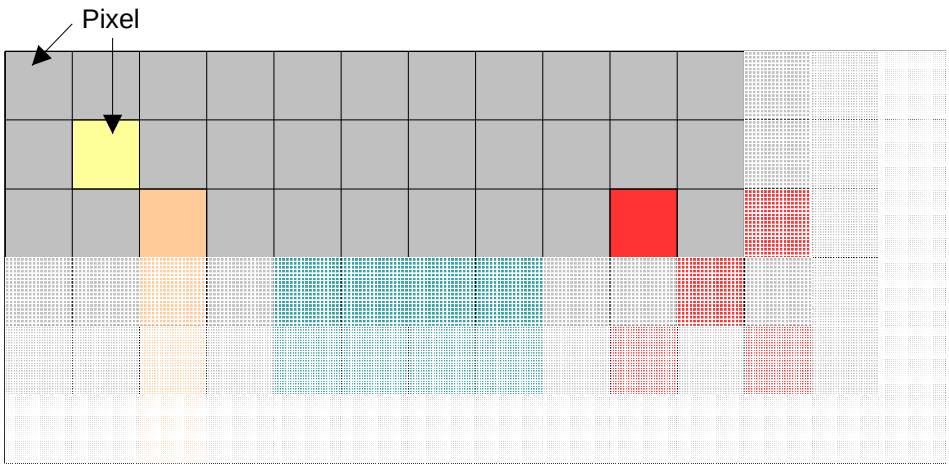
You can effectively scroll around the canvas simply by changing the viewport location, and you can zoom in and out by simply changing the viewport size (and not change the output window size).

You can create canvas objects anywhere in the canvas, such as images, rectangles, lines, polygons, text strings and more. These objects take up a certain amount of space in the canvas and can be hidden, visible, raised above or below other objects (within the layer that that object lives in, and there are approximately 4 billion layers to chose from). They can fade in and out, be resized, coloured, and much much more. The objects can have callback functions set on them that are called when certain events happen to them. Properties can be attached to the objects, and they can be named too. It is this flexibility that lets you do a lot with very little in the way of foot−work on the part of the programmer.

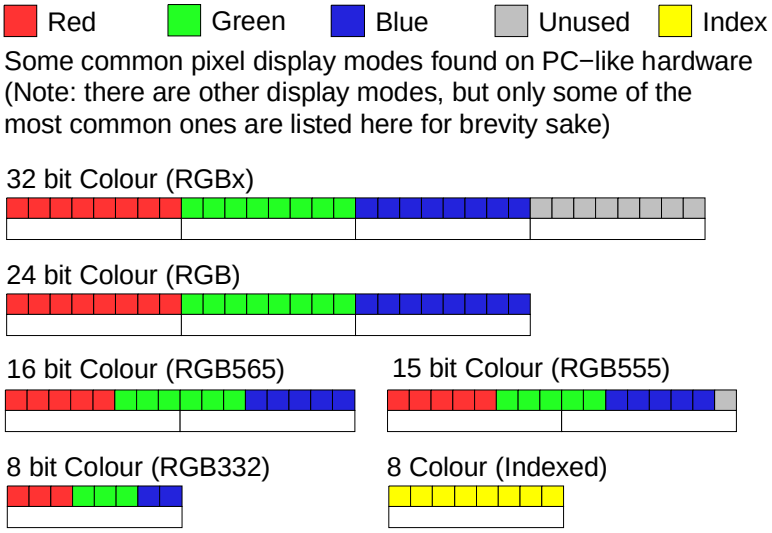
Colours and Rendering Output

One of the biggest problems with dealing with X is colours. X itself knows nothing about colour when drawing. It ONLY understands pixel numerical values. To properly explain this means we need to take a journey into framebuffers, history, and how X works.

A framebuffer is a large chunk of memory (generally memory residing on the graphics hardware itself to allow for more efficient access by the display output hardware) which is normally arranged in a linear format, from the top−left corner of the display, row by row from left top right until the bottom of the visible display is reached. Each pixel is represented by 1 or more bytes (in 256 colour mode, 8 bit or 8bpp it is 1 byte per pixel. In 16bpp, 65536 colours, it's 2 bytes per pixel, etc.). The value of this byte or set of bytes determines the colour to use for that pixel (a pixel being the smallest element of display − a single point in a fixed grid of these points that has a colour value).



Each pixel's colour is described by the pixel values in the framebuffer, in a fashion somewhat akin to the following diagram:



Some common pixel display modes found on PC−like hardware (Note: there are other display modes, but only some of the most common ones are listed here for brevity sake)

You will notice 32 and 15 bit displays modes have "Unused" bits. The reason for this is speed. PC hardware is fastest at accessing data on 4 byte (DWORD) boundaries − this is why 32bit colour has an extra byte of padding to make access to the framebuffer fast for both graphics display hardware and the CPU. It uses extra memory space, but does speed things up.

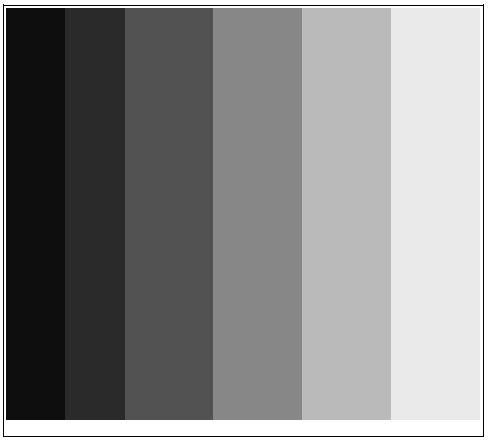
15 bit colour has 1 bit of extra padding to align the pixels to 2 byte (WORD) boundaries. This extra bit is thrown away − but it means you have the same number of bits for all colour channels, instead of the differing bits per channel that 16bit colour has. The quality for 16bit colour is slightly higher than 15bit (since it has double the number of colours), but it means "pure" gradient like grey for example, shows colouring since the red, green and blue values can NEVER be equal unless the colour is black or white. Shades in between get slightly coloured. Of course for real life images such as photos, and when dithering is used, this effect becomes unimportant and thus the extra gammut of colours becomes useful.

8bit colour can be treated 2 ways. We can allocate a static colour map where we have 3 bits for red, 3 for green and 2 for blue, but we cannot change the colours when using this to be a set other than this limited set of 256 colours. Being only 256 the quality degredation is quite noticeable. This is why there is another colour mode (which is much more commonly used), called pseudo colour. This is where applications can allocate a colour or a set of colours and so this set of 256 changes all the time and is dynamic, and the applications only allocate colours in this limited set of 256 that they actually need, thus conserving colours. If a colour cannot be allocated the closest match is given to the application and the application is told how close a match it is. It is the programmers job to deal with this.

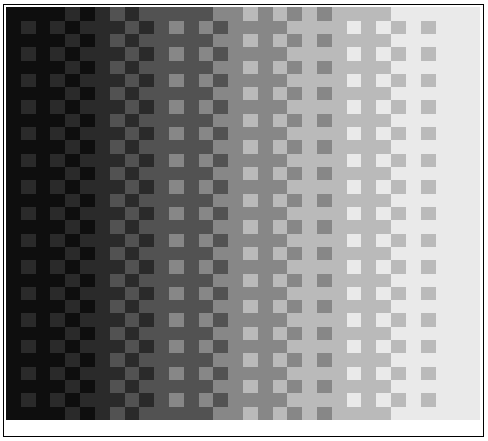
The problem with this is if it is not known ahead of time what colours are needed it can happen you allocate a lot of similar colours at the start because, lets say all your images are grey scale to start with, and then have all these greys and no colour entries left to allocate reds or blues for images that may be used later. Generally this is solved by pre−allocating a set of colours that spans the entire range of colours you could possibly want. This is often known as allocating a colour cube. Evas (or more correctly, Imlib2) uses this scheme of allocating colours.

Given that your display hardware, and thus X−Server, can be using any of these display modes (or possibly other ones not even discussed here) it starts to look like a bit of work handling the rendering and display of colour image data in X. Now let's discuss dithering. When you have a limited set of colours, lets say 256, you may want to draw "smooth" shades − lets take for example a gradient from black to white. Let's for now assume we use a 6x6x6 colour cube (that means 6 values for red, green and blue) which total 216 colours, and is the same set applications such as Netscape try to allocate for their display. At best we have 6 grey colours (including black and white) to use. This

leads to horrid "bands" on the screen thus completely destroying the desired effect of having a smooth gradient transition from black to white. Here is an example:



Notice the pronounced "banding" effect from left to right. There is a way of improving this by using a technique called dithering. This means you alternate between 2 or more colours within a small space to get the effect of having more colours than you really have. To the human eye at a bit of a distance the effect is quite convincing and might look something like this (using the same set of colours):



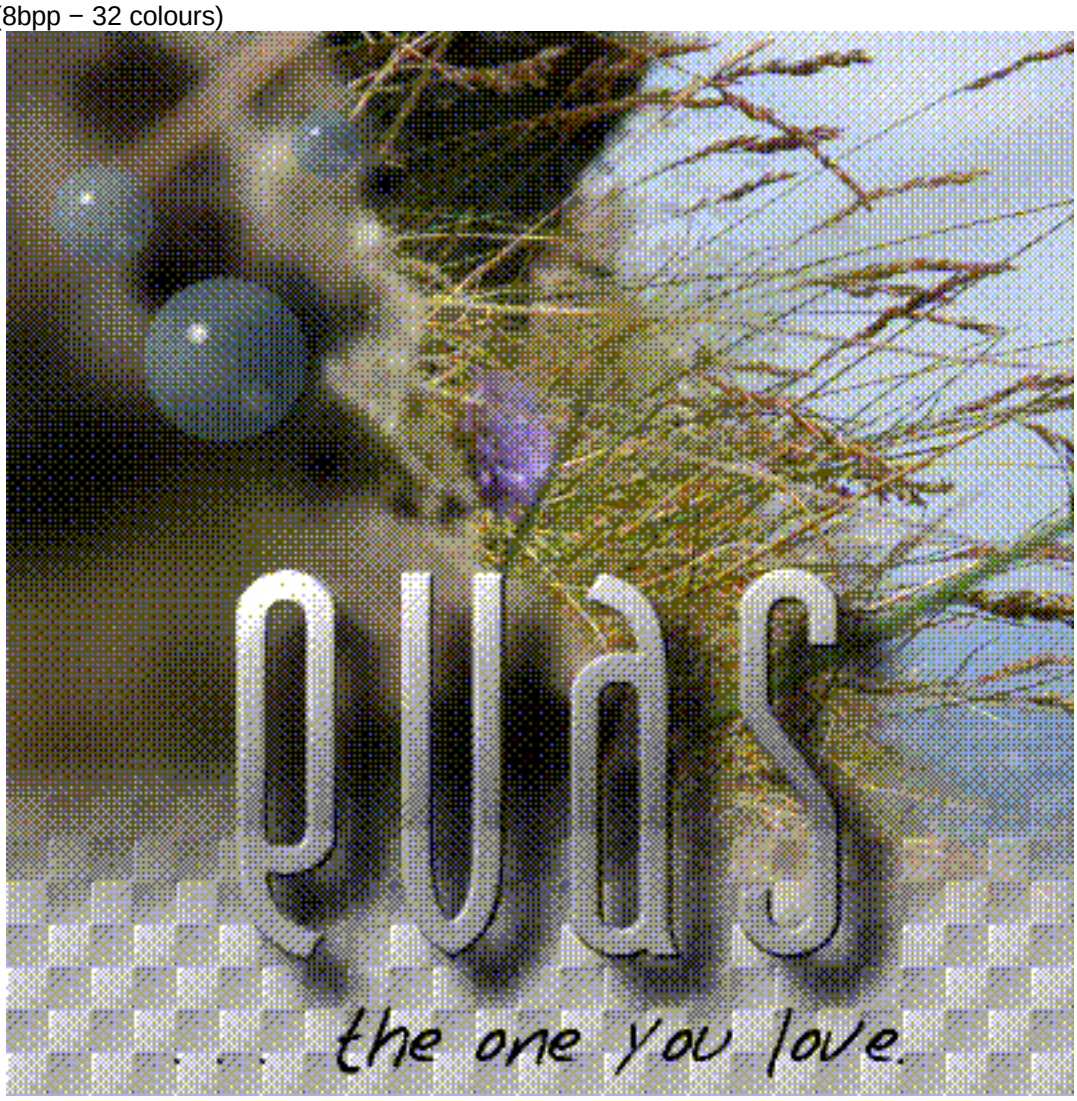
Doing dithering as well as handling colour allocation, mapping and more all at once and keeping the output looking good is quite a lot of work and requires a lot of graphics knowledge on the part of the programmer. The advantage of Evas is that it does this all for the developer, saving them a lot of hard work. Evas also does this with large amounts of optimisations, meaning the display not only looks good, but performs nicely.

As an example, here are some outputs from Evas at various display depths (colour depths). You should be able to easily tell the difference between the lower colour depths, and if you look closely you will notice that even in 16bit color dithering occurs to get smoother gradients and colour transitions.

(8bpp − 216 colours)

(8bpp − 128 colours)

(8bpp 64 colours)

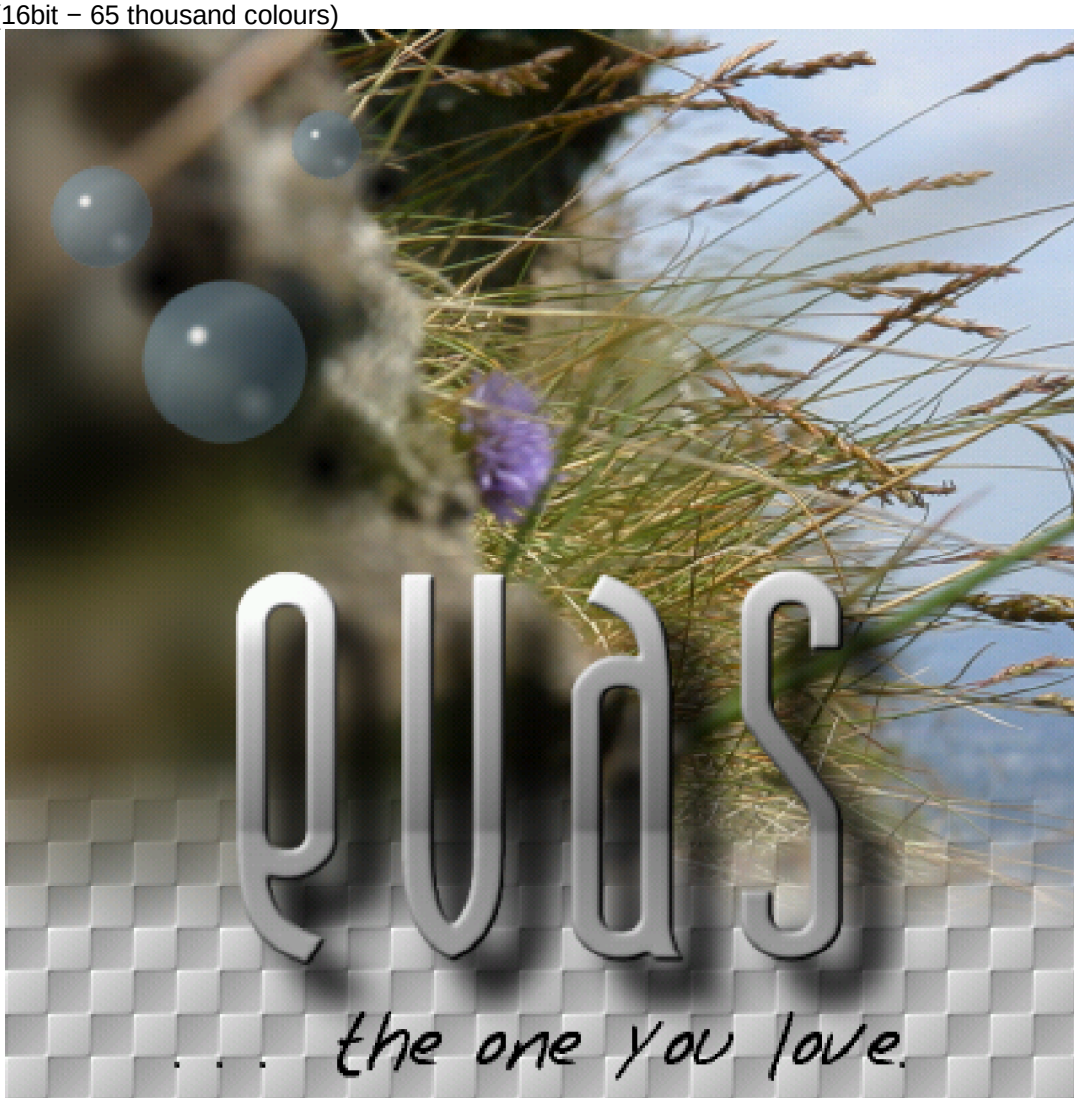


(8bpp − 32 colours)

(8bpp − 16 colours)

(8bpp − 8 colours)

(1bpp − 2 colour monochrome B&W)



(16bit − 65 thousand colours)

(24/32 bit − 16.7 million colors)

One thing you will have to note when using Evas is that the output quality for Evas will vary depending on the rendering engine being used. For example you may use the "3D Hardware (OpenGL)" rendering engine but Evas cannot guarantee the quality of the output of your OpenGL implementation, thus something the programmer should be aware of. It is a wise idea for the developer to ALWAYS provide a way for the user to select what rendering output to use for the application that uses Evas as stability of OpenGL and their X implementation may vary wildly, as well as quality and speed. There are also some cases where the software rendering engine outperforms the OpenGL engine by large amounts.

The developer should also realise that some rendering engines such as the X11 engine (Basic Hardware) do not do things such as alpha−blend for speed reasons. Basic X11 primitives cannot alpha blend and thus this engine cannot do it either (without going through hoops and thus degrading performance even below that of the software rendering engine) − but it should work at high speed on even the lowest end systems with slow CPU's and very basic graphics hardware.

For example here is the software rendering engine's output:

And here is what it looks like when you use the X11 rendering engine:

Notice that the soft shadows under the text and bubbles are now hard thresholds because the engine doesn't alpha−blend. The rendering engine will always try and output as accurately as it can given the constraints of the hardware set it is written to use. In general the software rendering engine will always be the most accurate, but possibly may not always be the fastest.

An Overview of Evas's Objects

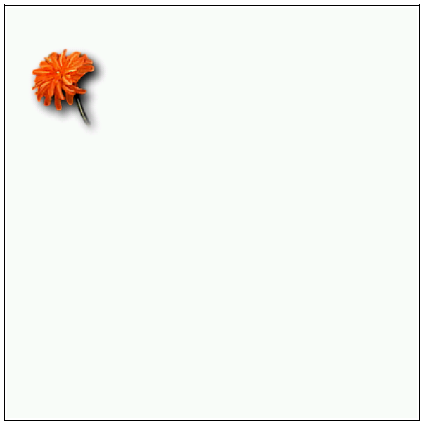
Evas provides a set of primitives for the programmer to work with. This is a quick overview of these primitives and how they work and how Evas helps you do amazing things with them that take you, the programmer no time flat to achieve, because Evas does all the hard work for you.

Images

One of the most versatile and appealing objects to use is the image object. You simply tell Evas where the image file is located, where to put the image what size the object is to be and how to fill the image object with the image data, and Evas handles the rest. This makes dealing with images in Evas child's play. Here is a quick example of how it would work:

```
object = evas_add_image_from_file(evas, "flower.png");
evas_move(evas, object, 10.0, 30.0);
evas_show(evas, object);
```

And it would look something like this:



By default, image objects are the size of the original image (in pixels) in units in the canvas. If we wish to make the object larger, and have the image scale up or down in size we could do:

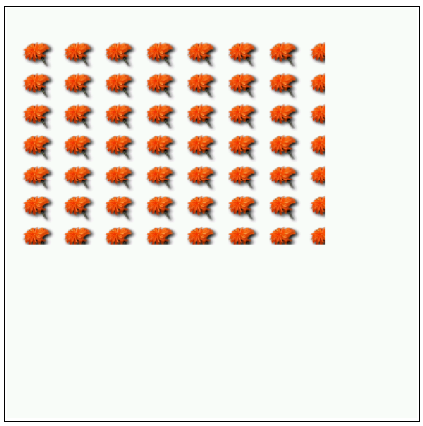
```
evas_resize(evas, object, 320.0, 240.0);
evas_set_image_fill(evas, object, 0.0, 0.0, 320.0, 240.0);
```

Now our canvas would look like this:

Easy, isn't it? Now let's say we want to have a 300x200 unit size rectangle filled with this image we loaded, but we want it to tile (repeat) starting the tile at co−ordinate 0,0 relative to the top−left corner of the image object area, but have the tiled image size be 20x50.

```
evas_resize(evas, object, 300.0, 200.0);
evas_set_image_fill(evas, object, 0.0, 0.0, 20.0, 50.0);
```

Our result would be something like this:



You may think "I can do this just as easily with normal X drawing primitives and a little extra sugar thrown in" but Evas does a whole lot more for you. You never need to know how to render the objects or handle re−rendering them. Evas does this, and optimises it for you. It handles layering and re−sizing, it optimises everything to minimise CPU effort with a dynamic set of objects. It handles the work to the point where ALL you need to do is create, destroy, show, hide, move, resize etc. objects and Evas does the rest of the nasty work for you.

As you can see − the program has to know and do very little. There are many powerful things you can do, like fade images in and out, re−colour them so they look tinted, have them rendered only within a certain area of the canvas, and much more.

Text

Text is a very important part of many applications. Evas of course provides a nice simple way of getting text into the Evas canvas you have up, without much fuss or bother AND to boot it will anti−

alias the text, so your character edges are smooth and not jagged, AND allow of the text to be semi−transparent and much more to boot.

Text allows you to display information in your Evas without needing image objects with pre−rendered text. But first you have to realise fonts are handled a little differently. First there is a font path. This is a list of directories where to find font files. Evas supports truetype fonts only − some may think this bad, but there are more truetype fonts than any other font format out there, and many are very good quality.

Evas looks at the fonts in a directory by their file name, so a font you reference in Evas as "myfont", Evas will look in the directories (in the order given) for a "myfont.ttf" file. It is case sensitive, but this system proves to be nice and simple. Here is a simple example.

```
object = evas_add_text(evas, "notepad", 18, "Here is a line of text!");
evas_move(evas, object, 120.0, 50.0);
evas_set_color(evas, object, 0, 0, 0, 255);
evas_show(evas, object);
```

This should create a text string object with its top−left starting at 120.0, 50.0 in the Evas canvas, with the "notepad" font at size 18, with the text "Here is a line of text!", and in solid black (Red = 0, Green = 0, Blue = 0, Alpha = 255). This is what it would look like:

As you can see − it is really easy to create some text in an Evas canvas, and it is just as easy to change its contents with calls such as the following

```
evas_set_text(evas, object, "Some new text here");
evas_set_font(evas, object, "arial", 30);
evas_set_color(evas, object, 255, 0, 0, 255);
```

Just modifying the properties of an object this way directly leads to it changing as you would expect. There are enough text query calls too, to implement almost anything you need using text objects, including text entry boxes, word processors, HTML and other text formatting engines and much, much more.

Gradients

Another useful primitive that can be used is the Gradient Box primitive. Basically this is a rectangular object that is filled with a linear range of colours at an arbitrary angle defined as part of the properties of that object.

Gradients are useful for when you want to fill a background or rectangle with something other than a

solid colour, but do not want to go to the expense of needing an image for it. They can be used to highlight areas with lighting and shadow effects and many other things.

Creating and managing these objects, like almost everything else, is also child's play. Take a look at this:

```
object = evas_add_gradient_box(evas);
evas_move(evas, object, 150.0, 100.0);
evas_resize(evas, object, 200.0, 120.0);
gradient = evas_gradient_new();
evas_gradient_add_color(gradient, 255, 255, 255, 255, 10);
evas_gradient_add_color(gradient, 255, 255, 0,   255, 10);
evas_gradient_add_color(gradient, 255, 0,   0,   255, 10);
evas_gradient_add_color(gradient, 0,   0,   128, 255, 10);
evas_gradient_add_color(gradient, 0,   0,   128, 0,   10);
evas_set_gradient(evas, object, gradient);
evas_gradient_free(gradient);
evas_set_angle(evas, object, 290.0);
evas_show(evas, object);
```

Now the program we are developing looks now something like this when displayed in the Evas canvas:

As you see, you create the gradient box object, place it somewhere in your Evas and set its size. Now you get to define the list of colours it is to use (from start of the gradient to end) and their relative "distance" apart. This is relative to the length of the gradient, not an absolute, so the total length is the sum of all the distances (except the first element in the gradient whose distance is ignored). We add in white, yellow, red, dark blue then "transparent" in the gradient in order, then set this gradient to be used by this gradient box. Once we are done setting up this gradient we can set it to be used by more than one gradient box, and when done with it finally, we can just free it. All we do now is determine the angle the gradient is at in the box (with the angle in degrees being the first element of the gradient and the angle starting at 12:00 o'clock (0 degrees), going all the way up to 360 degrees in clock−wise rotation), and then we show the object. All we need to do now is move and resize it, show it, change the angle etc. and Evas handles it for us.

This is not where it ends. There are other primitives that Evas supports. Polygons, Lines and Rectangles complete this list for now, but future plans at a later point may mean Video objects (e.g. MPEG streams as objects), curved objects (ellipses, circles, splines etc.) and more, but for now they are not supported. Some of these are rather complex to implement and thus may take some time before they come to fruition − especially the video objects, as most machines these days are already struggling to have enough power to play DVD MPEG2 streams.

Other Primitives

At your disposal you also have Rectangles, Lines and Polygons too. You can use these to fill in backgrounds, simulate circles (with polygons) and much more. They are versatile and simple and let you do basic drawing without requiring on−disk data to generate it from.

```
object = evas_add_rectangle(evas);
evas_move(evas, object, 20.0, 130.0);
evas_resize(evas, object, 50.0, 70.0);
evas_set_color(evas, object, 20.0, 50.0, 100.0, 130.0);
evas_show(evas, object);
```

Here we add a simple rectangle that's blue and partially transparent. It's easy to add − no problems. It may seem a fair bit of set−up code, but now we can just move, resize, show hide etc. the object whenever we want a change − Evas does all the rest for us − including layering, obscuring, figuring how to re−render it and optimising the rendering to only render what changed and much much much more.

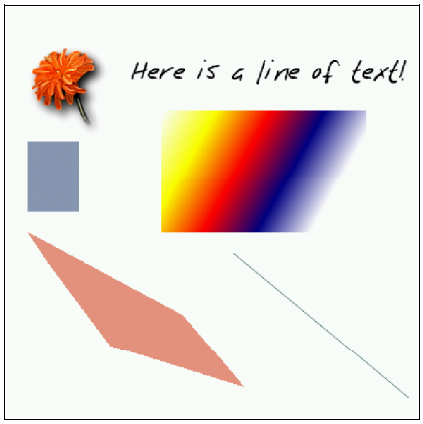
```
object = evas_add_poly(evas);
evas_add_point(evas, object, 0.0, 0.0);
evas_add_point(evas, object, 150.0, 80.0);
evas_add_point(evas, object, 210.0, 150.0);
evas_add_point(evas, object, 80.0, 110.0);
evas_add_point(evas, object, 20.0, 30.0);
evas_set_color(evas, object, 200.0, 40.0, 0.0, 130.0);
evas_move(evas, object, 20.0, 220.0);
evas_show(evas, object);
```

A polygon, reddish/orange, partially transparent with 5 points. We create the polygon relative to the top left of its bounding box then just move it where we want it later − this makes it easy to just move polygons around without having to reset their co−ordinates. Just like with polygons − Evas handles optimising all the other stuff surrounding these objects for us.

```
object = evas_add_line(evas);
evas_set_line_xy(evas, object, 220.0, 240.0, 390.0, 380.0);
evas_set_color(evas, object, 30.0, 80.0, 80.0, 200.0);
evas_show(evas, object);
```

And a line. Anti−aliased. Simple. Just change its colour and co−ordinates when you want. Evas does the rest.

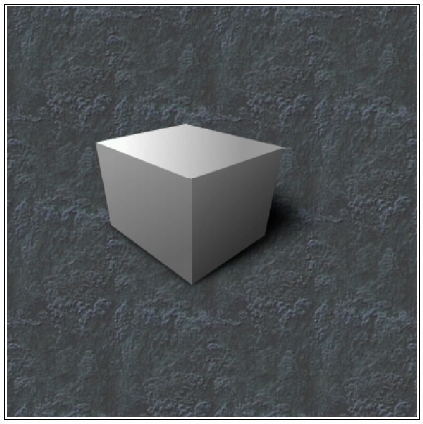
And now with these primitives added out Evas canvas will look like this:



There are many other important parts of using Evas to achieve the look and feel you want in your application. It is important you know what else is available so you don't do things "incorrectly" and as a result get slower performance or bad display quality.

Tinting and Fading

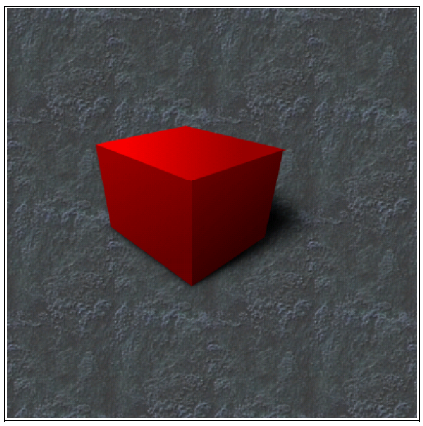
Doing effects in Evas is almost child's play like everything else. You can tint and fade image objects, for example, with very little effort. Let's start a new Evas with a single image object and a textured background.



Now lets "tint" this object red. That's easy:

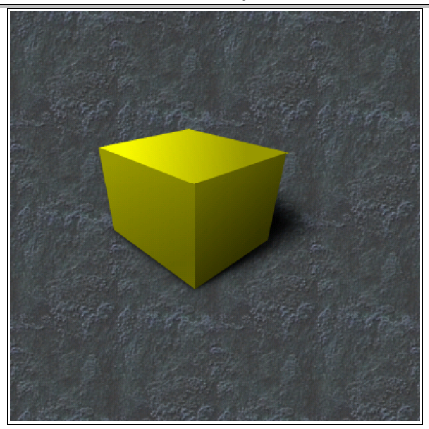
```
evas_set_color(evas, object, 255, 0, 0, 255);
```

When you set the colour of an image, every pixel of that image is multiplied by the colour value. So pseudo−code wise output.red = (pixel.red * color.red) / 255, and the same for the green, blue and alpha channels. Here is the result of multiplication by RGB 255,0,0,255 :



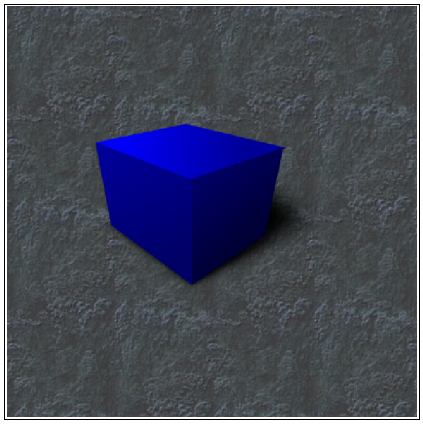
Now let's try tinting the cube yellow:

```
evas_set_color(evas, object, 255, 255, 0, 255);
```



How about blue?

```
evas_set_color(evas, object, 0, 0, 255, 255);
```

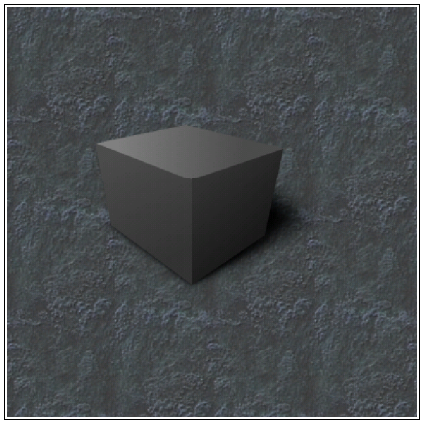


Let's try fading the cube half−out:

```
evas_set_color(evas, object, 255, 255, 255, 128);
```

Let's try making the cube just plain darker than it normally is:

```
evas_set_color(evas, object, 128, 128, 128, 255);
```



As you can see, fading images out, tinting images is almost no work at all. In addition to being able to scale them, raise and lower them above and below other objects, have them alpha blend and more, you have a pretty simple way of creating complex effects within an Evas' canvas.

Clipping

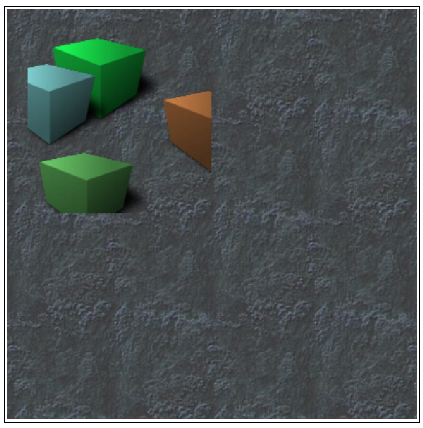
Now let's look at clipping in Evas. Clipping is the process by which you "clip off" bits of what you are drawing that lie outside the area you are clipping to. For example − you have an image and you clip it to a box that is smaller than the image placed in the middle of the images location. This means the image will now only be drawn in the area that clips it − i.e. this rectangle. The parts of the image lying outside the bounds of this rectangle will never be drawn. You can also think of this as windowing − allowing only what you can see through the "window" (or clip region) to show through. In theory you can use any object as a "window", or clip object, but due to back−end rendering engine limitations, Evas only supports using rectangle objects as clip objects for now. Using any other kind of object has undefined results. Clip objects can be used to implement things such as viewports, windows and much more − you will need to use your imagination to really use them fully.

Also just like a window that may be stained a certain colour, Evas can let you "Tint" or "Stain" the objects clipped by a clip object. Just like you set the colour of an object like and Image, and it gets "tinted", the same applies to clip objects. If the clip objects colour is opaque white (255, 255, 255, 255), it does not affect the colour of any object it clips − it merely limits the drawing region, but if the clip object has any other colour its RGBA values are multiplied by the colour of the objects it clips − thus being able to tint and fade objects that are clipped all in one go.

Let's say we already have an object we created and want it clipped to a rectangle (or we may want more objects clipped to this rectangle):

```
clip = evas_add_rectangle(evas);
evas_move(evas, clip, 20.0, 20.0);
evas_resize(evas, clip, 180.0, 180.0);
evas_set_color(evas, clip, 255, 255, 255, 255);
evas_show(evas, clip);
evas_set_clip(evas, object, clip);
```

So if we have lets say we have 8 cube image objects, and want them clipped to this rectangle, it will look something like this:



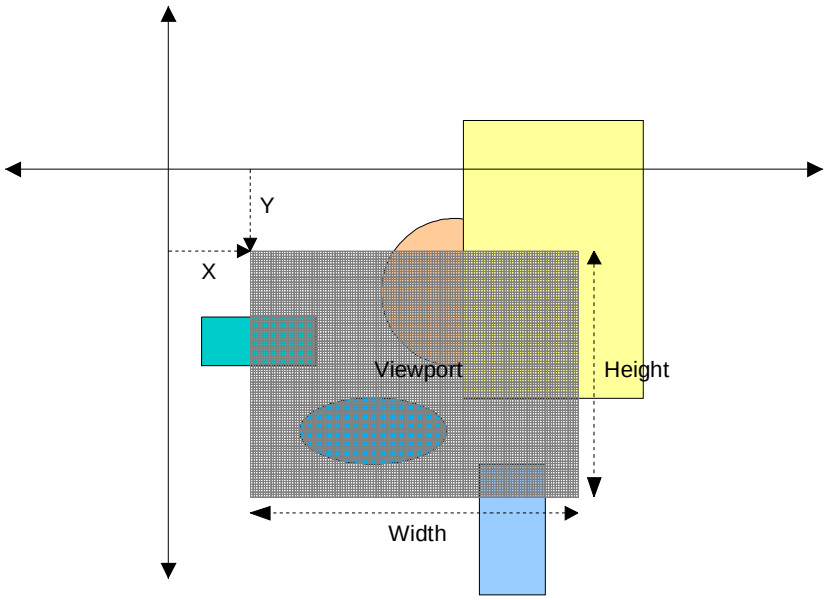
Notice that the cubes in the image here are "clipped" to an invisible rectangle that as per the code snippet above, has co−ordinates of 20.0, 20.0 and has a size of 180.0 x 180.0. You could easily fade the objects clipped by this rectangle out by changing the alpha value of the clip rectangle's colour. The same would apply to tinting. You can move the clip rectangle around and resize it and things will behave as expected (i.e. you have a virtual window moving around with the objects clipped by the clipping rectangle object only showing through where the clip object is), and you can also show and hide it to show and hide its clipped contents. Clip objects can also be clipped by other clip objects and so on recursively. You can use this specialised application of an object for doing things like building widget sets, viewports within an Evas (clip objects negate the need for being able to embed an Evas within an Evas). It is hoped one day that other objects will be able to be used as clip objects (polygons, text strings, gradients, image objects etc.).

The Viewport and Resizing

Evas operates on the concept of a viewport. A viewport is a rectangular region of a large area that defines what smaller part of that area we are looking at. In Evas you tell Evas what the location and size of the viewport is. An Evas canvas is almost infinite in size (as large a value as can be fit in a double precision number in both X and Y directions), and thus you need to tell Evas what limited area of the "world" (or canvas) you are looking at. This area is the viewport.

Evas also requires you tell it the size of the window it's outputting to. This is necessary for some of

the back−end rendering engines, but also is useful for being able to zoom and pan across the canvas. If you do not change the output size of your Evas, but change the viewport size, you will effectively zoom and and out of your Evas canvas. By changing the viewport location you can scroll around your Evas. If the size of your window changes you MUST tell Evas the new drawable output size before rendering again − or things will not display correctly.



The viewport looks like this when applied to an Evas canvas:

Handling Exposes

Evas itself does not hook to your event layer at all. It relies on you feeding events into it. One of these, required for consistent output, is telling Evas when a section of its output has been "exposed" (you get an expose event from X when this happens). This means that that area of the window has been damaged (either by the window being shown for the first time or a window that was on top being moved out of the way, or resized so more of your window shows through, or has been destroyed or unmapped) and needs to be redrawn. When you get an expose event you simply tell Evas about this event, and that's all you ever have to do. This is the only time you should need this function call at all since moving objects or changing things in the Evas canvas will internally trigger areas to be marked as dirty and be redrawn anyway.

To do this, all you need to do is call this function:

```
evas_update_rect(evas, x, y, width, height);
```

How Evas Renders

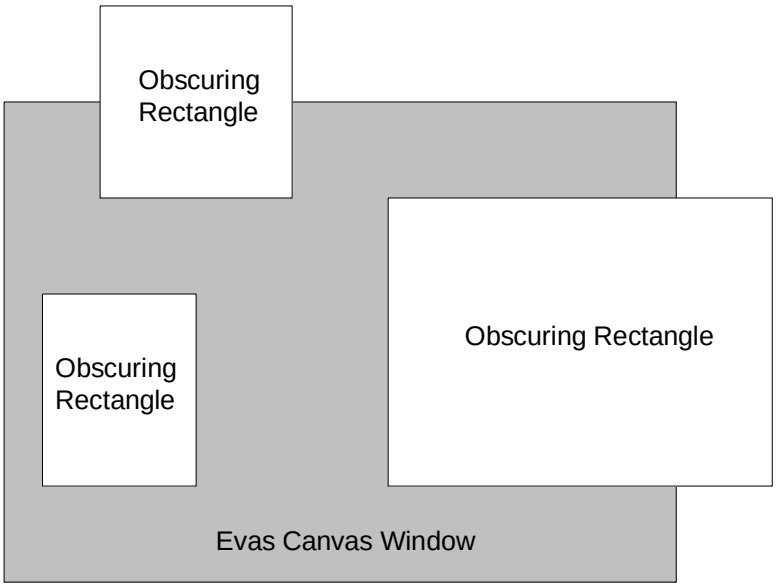
Evas minimises the amount to render whenever possible. As long as you don't force it to do rendering that it doesn't have to (i.e. uselessly call the evas_update_rect() call when you haven't gotten an expose, or call evas_render() or evas_render_updates() all the time whenever you make any changes at all to an Evas), Evas will keep rendering to a bare minimum for you, and still retain display quality and rendering speed. Remember to only call Evas's render calls when you program

goes IDLE, or you ABSOLUTELY NEED a rendering pass done − i.e. no more events to process and no state to change. Doing otherwise will impact performance badly.

Remember evas_render() does not return anything. If there is anything to draw it will draw what is needed, and then return. If there is nothing to be done it will return immediately. evas_render_updates() returns an Imlib_Updates data list that you can process with Imlib2 calls to find out what rectangles in the canvas Evas actually rendered to. Evas will return NULL if nothing was rendered. If you use this call it is your responsibility to free the updates when you are done processing them, otherwise they will keep being generated every render call, and thus memory will leak.

Also make use of obscure regions. Evas allows you to define regions of your Evas canvas that are COMPLETELY obscured by other windows (or obstacles) so the rendering output will never be seen.

You can clear the list of obscure rectangles by calling evas_clear_obscured_rects(). When these obscuring objects appear or go away or change it is a good idea to clear the list and then call evas_add_obscured_rect() to add in all the rectangles of objects that completely obscure Evas. Evas will use this information to reduce the amount of rendering it has to do and not render to the sections of the canvas completely obscured by obstacles. Remember that the list is persistent between renderings and thus you need to change it whenever the obstacles change.



The Rendering Engines

Evas provides you with several output targets for an Evas canvas. Once a rendering engine is set, it cannot be changed. You can select how Evas is to render to the display target by selecting the render method. Currently Evas supports a Software Rendering engine (the default) that uses Imlib2, an X11 primitives engine that uses X primitives for all the drawing − such as pixmaps, lines, rectangles etc., an OpenGL hardware accelerated engine and an Imlib2 image target engine, so you can render to an Imlib2 Image object in order to re−use as an image or save to disk etc.

Each rendering engine has a personality of its own, and at a later date more engines will be added to provide different targets for the canvas output (for example a postscript or pdf engine would certainly be feasible).

Here are the rendering engines available and some of their properties, limitations and benefits.

RENDER_METHOD_ALPHA_SOFTWARE:

This is the default rendering engine. It uses the CPU to do all its work and tries to minimise the amount the CPU has to do wherever it can. It can render to ANY X drawable and is ALWAYS available. It can use any colormap and any visual your X server has. It works locally and over a network connection to X and alpha blends, as well has having, in general, the highest quality output. Unlike software rendering in OpenGL (Mesa), software rendering in Evas is actually fast and usable, and thus should not be discounted as a possibility. If you do any development this should be the engine you use by default during development for testing and viewing of the output.

RENDER_METHOD_BASIC_HARDWARE:

This engine generates pixmaps and pixmap masks where needed for images and text, and otherwise uses raw X calls to do the rendering of objects in the canvas. This engine is dependant on how well your X Server's 2D calls are accelerated, and how good your graphics card's 2D hardware is. Pixmaps still have to be generated from the image data by the CPU, except short−cuts are taken to minimise the CPU usage even further for doing this, by disabling dithering for depths higher than 8 bit colour, and completely avoiding doing any alpha blending or dithering of alpha masks (due to massive slowdowns using complex masks in X). This engine gets you speed on really slow and old hardware & X Servers, but at the expense of quality. This engine can render to any X drawable using any colormap, and any visual, locally or across a network to X.

RENDER_METHOD_3D_HARDWARE:

This engine uses OpenGL (if Evas found OpenGL on compilation − otherwise Evas will fall back to RENDER_METHOD_ALPHA_SOFTWARE silently for you, to ensure it still renders something). If your OpenGL implementation is not fully accelerated, your 3D hardware is limited or such you may see strange effects or even slower performance than software rendering. It should be noted that the engine makes a lot of use of RGBA textures in OpenGL and thus if your OpenGL implementation or hardware don't support these well, you will have massive performance issues. It should also be noted that the vast majority of OpenGL drivers are incomplete, buggy or even completely unstable. Just because your favourite 3D game works under OpenGL does NOT mean Evas will. Evas uses features of OpenGL that games do not and uses them differently. You may even find your system hangs or completely crashes due to some of these bugs. This is system, hardware and driver dependant and may vary over time and with versions of the drivers and what else you are running at the time. This rendering engine is limited to what visuals and colormaps it can use. It will only work in true colour modes (i.e. 15 bit and up generally), and you MUST use the visual and colormap evas_get_optimal_visual() and evas_get_optimal_colormap() return (Note − these won't be valid until you set the rendering method with evas_set_output_method()). This engine can ONLY render to a window. It cannot render to pixmaps. It will not work over a network either. You may wish to stick to using evas_new_all() as a convenience function that creates the window for Evas with the right visual and colormap for you.

RENDER_METHOD_ALPHA_HARDWARE:

This engine currently is not implemented and does nothing. Do not use it.

RENDER_METHOD_IMAGE:

This rendering engine does not use any display, drawable, window, colormap, visual etc. information. You set the target output using evas_set_output_image(). When using this rendering engine Evas will render updates to the target image set. Remember that sections of the canvas that have transparency and no solid object at all in that area will be transparent during the merge operation onto the destination image. That means the destination will, if it is already transparent, inherit this transparency, or if the destination already contains solid image information, will have the

updated canvas areas blended on top of the destination. This engine is useful for when you want to save the output of the canvas to a file − such as a JPEG, PNG, TIFF or other such format file using Imlib2's routines. It is also useful if you need to generate pixmaps and pixmap masks from canvas contents for some other reason.

Event Handling within Evas

Evas provides a convenient event and callback system that keeps it separated from the output device, allowing the program using Evas to control what events are received by the Evas canvas and thus how Evas will react to these events.

The functions the program will have to use to pass events into the Evas canvas are evas_event_button_down(), evas_event_button_up(), evas_event_move(), evas_event_enter() and evas_event_leave(). These functions effectively input raw input device events into the canvas and let it process them. If you are passing events into a canvas at all (which you will need to do if you wish to use the callback system), you will need to use all of them, and not just a selection. IF you pass button down events in Evas expects button up events to come in at some stage, and if they don't, events won't be processed as expected.

Evas supports a single pointer device with up to 32 buttons (numbered 1 to 32). The semantics of how events are processed is similar to that of X. When the first mouse button is pressed on an object, and no buttons are currently down, the pointer is logically "grabbed" to that object. This means no enter or leave events will be processed until either the button is ungrabbed with evas_pointer_ungrab(), or until all mouse buttons are raised and none are down. At this point if the pointer has left the object a leave event is generated, and any new enter events on other objects are also generated. All motion events whilst the pointer is logically grabbed to that object are reported as callbacks on that object, even if the motion happens outside the object. All callbacks for events are handled synchronously. That means every time an event is inputted into Evas with one of the above event calls, the call does not return until all callbacks triggered as a result of this input event have been called and have returned.

The programmer can set callbacks on objects to be triggered by events. The functions evas_callback_add() and evas_callback_del() control these callbacks. When the callback is called the appropriate parameters to the callback are filed in. If you have a MOVE callback, the button parameter is unused, for example. The same applies with ENTER and LEAVE callbacks. The x and y parameters to the callback are filled in with the output pixel−space co−ordinates the pointer is currently at. The program can use evas_screen_x_to_world() and evas_screen_y_to_world() to translate the screen co−ordinates to world co−ordinates if it prefers to have the values in canvas unit space than output pixel space. The object and evas parameters are filled in with the evas an object handles of the object the event occured in and which Evas canvas that object is in.

In addition to the callbacks for pointer events on objects there is an added callback for when an object is freed (destroyed). This allows the program to free the memory of any data pointers it may have attached to that object when it is destroyed.

List Handling

Evas provides a convenient set of linked list handling functions. These functions all abstract a list (which is an optimised doubly−linked list), to allow easy manipulation of the list and its members. You may use these functions for anything you deem fit, and Evas provides these for its own internal use as well as for the use of being able to return lists of data to the program calling the Evas routines.

Some quick hints as to handling lists. To walk a list from beginning to end, a convenient way is to use a for loop:

```
Evas_List l, list;
for (l = list; l; l = l->next)
  {
    void *element;
    element = l->data;
    /* ... process the element here ... */
  }
```

You should remember that removing or adding elements to a list whilst walking it is dangerous since it modifies the data you are currently inspecting, and thus will probably lead to bugs, infinite loops or strangely truncated lists.

Caveats, Limitations etc.

There are some "Caveats" to use Evas. It has some limitations that are there mainly due to them being impractical to work around, others being the result of a design decision for one reason or another, and some being there because a better way wasn't known of at the time.

Size Limit:

Evas has an absolute maximum output size of 8192x8192 pixels for the output. Even this is impractical as to just have 1 buffer to render this in the first stage rendering for the software engine requires 256 Mb of RAM − that means 512 Mb of RAM would be needed (you will need another 256 Mb RAM for the second stage buffer) just to render this Evas once. You should stick to "sane" sized canvases, and if you wish to pan around a large canvas, use the viewport to do this. You will also find that image objects have 8192x8192 size limitations for original and post−scaling output size.

Performance and Demand Loading:

Evas demand−loads image data as needed. It will not load image data off disk until it actually needs it. Thus if an image is never displayed, and then later becomes visible, only then will the image be loaded off disk. If the object becomes invisible again the image data will be put back into the cache pool. Evas has an image (and font), cache just for this purpose. Keeping the cache at a reasonable size will use UP TO that amount of memory (in bytes) for cached image data (and for the font cache, glyph image data), in case Evas needs the image data again soon. This makes performance excellent even for naieve applications, but uses extra memory. Your program could, if it was smart, look at system resources and set the cache size accordingly as free memory becomes available or is used up. Setting the cache to 0 effectively causes the entire cache to be flushed and all tentatively held data to be freed, and then will mean Evas does not cache any more data until you set it to some value grater than 0. If you do not like this behaviour, set the cache to 0 at the start. Also note that this cache is shared between ALL Evas canvases in the application. Setting it to zero in any canvas sets it to zero for all canvases. Also all image data for objects, textures etc. is shared between canvases in the same application, so it is relatively inexpensive to create lots of canvases that have the same objects with the same image sources since the data is shared, which is the bulk of the resources used.

Background:

Evas has NO background. If you create objects and have nothing behind all your objects (such as a tiled image object or a large opaque rectangle or polygon), your objects will be rendered on top of undefined contents − likely random memory garble or old framebuffer contents. You need to create some object (be that a rectangle, image or anything) that will fill the canvas in the areas you expect transparency to exist in objects, otherwise you will see nasty trails and artefacts.

Silent Errors:

Evas is very robust. It will let you almost get away with murder. You can pass in NULL canvas and object handles and Evas will simply ignore them and march on and do nothing. If you tell Evas to make image objects from files that don't exist, Evas will gladly make the object but it will never render it. You need to check in on this kind of error using evas_get_image_load_error() if you care what happened to your image objects and their image data. As for other errors such as querying the string contents of an image object, Evas will try and return the most sensible values it can instead of just erroring out, in an attempt to make sure your application somehow marches on. Be careful of what properties you set on what objects, and what queries you call on which objects, as if they are returning what seems silly values (like a font size of 0 or a font name with a string pointer of NULL etc.), you may be querying the wrong object. The programmer needs to be careful whilst programming and be aware of this and "do the right thing" by not doing things to objects that cannot have those things done to them. Evas simply tries to be as robust as it can if such mistakes are made, and not fall over itself, but march on regardless.

Co−ordinates:

Remember Evas co−ordinate space and screen co−ordinate space may not always match. Make use of the evas_world_x_to_screen(), evas_world_y_to_screen(), evas_screen_x_to_world() and evas_screen_y_to_world() functions that will translate world and screen co−ordinates for you as needed depending if the canvas does or does not have a 1:1 world to screen co−ordinate ratio. Also note that if the output width to height ratio is not the same as the viewport width to height ratio you may get odd effects with text objects. Keep the ratios the same.

Layers:

Don't overuse layers for keeping objects above or below each other − you can always raise and lower objects and stack them above or below each other. Using too many layers just adds lots of overhead. Consider keeping the number fo layers to a few dozen at most, but then again − your mileage may vary.

Update Rectangles:

Don't call evas_update_rect() just because you change or move an object. Evas will figure this out itself. Call this only when you get an expose event for that area of the canvas window and/or that area of the canvas window is damaged and needs re−rendering. Also note that calling this simply queues a draw that Evas later optimises and uses ONLY when you call evas_render() or evas_render_updates().

API Reference

This section describes all the Evas functions, what they return and what they do. If you are new to Evas it is a good idea to read through these once to familiarise yourself with the available function calls and what they do, so you don't get stuck not knowing how to do something for lack of knowing the right function to call.

```
Evas evas_new_all(Display *display, Window parent_window,int x, int y,
                  int w, int h, Evas_Render_Method render_method,
                  int colors, int font_cache, int image_cache,
                  char *font_dir);
```

This function should be your first port of call when using Evas. It is a convenience function that wraps other Evas and X functions to create an Evas for you and a window with the right visual and colormap.

Return:

It returns an initialised Evas canvas handle ready for use and display (once you have mapped the window etc.)

Arguments:

display

This parameter is the display pointer you would get from calling X functions or wrappers such as XOpenDisplay(). You need to pass this in to indicate which display to use to create the window and query the colormap and visual etc.

parent_window

This is the parent window in which to create the window for Evas.

x, y

These are the x and y pixel co−ordinates in the parent window at which the top−left corner of the window that is created will be placed.

w, h

These parameters are the width and height parameters respectively (in pixels) for the window that Evas will create − they will also be the width and height in units of the Evas viewport, located at 0,0.

render_method

This specifies the rendering engine to be used for this Evas. This must be one of RENDER_METHOD_ALPHA_SOFTWARE, RENDER_METHOD_BASIC_HARDWARE, RENDER_METHOD_3D_HARDWARE, RENDER_METHOD_ALPHA_HARDWARE or RENDER_METHOD_IMAGE.

colors

This parameter specifies the maximum number of colours to try and allocate if the display mode is pseudo colour (8 bit colour). Any value from 2 to 256 is valid. Evas will attempt to allocate up to *colors* number of colours when the display type is like this, but not more. It may allocate less depending on how many available colours there are currently on the display

font_cache

This argument tells Evas how big to make its font cache (in bytes). Evas will use up to this number of bytes for font glyph pixel data in addition to the fonts required for display, in case

they are required again in the future.

image_cache

This specifies to Evas how many bytes of memory are to be used to cache image data that isn't currently required for rendering of the visible Evas area. This works just like the font cache, but for image pixel data.

font_dir

This specifies one directory in which Evas can find the true type font files it needs for fonts. You can add more directories later if you want.

```
Evas evas_new(void);
```

This function creates and returns a new Evas canvas that is not initialised and needs initialisation. You will need to call evas_set_output_viewport(), evas_set_output_colors(), evas_set_output_size(), evas_set_output_method() and possibly evas_set_output_image() or evas_set_output() in order to make use of this Evas canvas. You will need to supply an appropriate drawable, colormap and visual (or image) for Evas to draw to etc. You should only do things this way if you really know what you are doing.

Return:

A valid Evas canvas handle that needs initialising.

```
void evas_free(Evas e);
```

This function will free all objects in the Evas canvas passed to it, free the canvas and all memory used by that canvas. If Evas created the window for you for this canvas with evas_new_all(), this window will also be destroyed.

Arguments:

e

A valid Evas canvas handle.

```
Window evas_get_window(Evas e);
```

This function will return the output window id (or drawable as the case may be − since windows and pixmap id's are interchangeable for the purposes of rendering to them). If no output window id is set it returns 0.

Return:

The return value is a window (or drawable) id. If the return value is 0, no output window is set, or the Evas passed in is NULL.

Arguments:

e

A valid Evas canvas handle.

```
Display *evas_get_display(Evas e);
```

Returns the display pointer used for the Evas passed in.

Return:

The display pointer used for the Evas passed in. NULL is returned if no display output is set or the Evas handle passed in is NULL.

Arguments:

e

A valid Evas canvas handle.

```
Visual *evas_get_visual(Evas e);
```

This function returns the visual currently being used by the Evas passed in.

Return:

The return value is the currently used visual pointer by the Evas passed in. NULL is returned if no visual is being used or the Evas handle is NULL.

Arguments:

e

A valid Evas canvas handle.

```
Colormap evas_get_colormap(Evas e);
```

This function returns the currently used colormap that the Evas passed in is being used.

Return:

The currently used colormap by the Evas passed into the function. 0 is returned is no colormap is set or the Evas handle passed in is NULL.

Arguments:

e

A valid Evas canvas handle.

```
int evas_get_colors(Evas e);
```

This function returns the maximum colour allocation count for the Evas passed in as an argument. This does not mean it is the actual number of colours allocated for use, but the maximum number allowed to be allocated.

Return:

The return value is an integer in a range from 2 to 256 which is the maximum number of colours to be allocated when displaying on a display of depth less than or equal to 8 bits.

Arguments:

e

A valid Evas canvas handle.

```
Imlib_Image evas_get_image(Evas e);
```

This function returns the currently used Imlib2 image destination image when the RENDER_METHOD_IMAGE engine is being used.

Return:

The Imlib2 image handle that is currently being used for rendering is returned. If no image is being used or the Evas passed in is NULL, NULL is returned as the image handle.

Arguments:

e

A valid Evas canvas handle.

```
Evas_Render_Method evas_get_render_method(Evas e);
```

This function returns the currently used rendering method for the Evas passed in.

Return:

A valid rendering method. If the Evas passed in is NULL the return value is undetermined.

Arguments:

e

A valid Evas canvas handle.

```
void evas_update_rect(Evas e, int x, int y, int w, int h);
```

This function queues an update in the output for the Evas passed in for the rectangle defined by the x, y, w, and h co−ordinates. X and y are pixel positions from the top−left corner of the output window for the top−left corner of the rectangle. W and h are the width and height, respectively of the rectangle to update, in pixels.

This update is queued but not actually performed until evas_render() or evas_render_updates() is called. You should ONLY call this function when that rectangle for the output of the Evas has been damaged and needs re−rendering, such as an expose event. There should be no other reason to use this call otherwise.

Arguments:

e

A valid Evas canvas handle.

x

The x pixel co−ordinate of the top−left corner of the rectangle to update.

y

The y pixel co−ordinate of the top−left corner of the rectangle to update.

w

The width of the rectangle to update in pixels. Must be greater than 0.

h

The height of the rectangle to update in pixels. Must be greater than 0.

```
void evas_add_obscured_rect(Evas e, int x, int y, int w, int h);
```

This function adds a rectangle to the Evas canvas that is passed in that is FULLY obscured. The contents of the canvas in this rectangle will never be guaranteed to be rendered. Evas will attempt to optimise the rendering to avoid rendering in the obscured regions held in the Evas canvas's obscured rectangle list. Every time this function is called a new obscured rectangle is added to the list.

The rectangles are retained in the Evas canvas until cleared.

Arguments:

e

A valid Evas canvas handle.

x

The x pixel co−ordinate of the top−left corner of the rectangle to obscure.

y

The y pixel co−ordinate of the top−left corner of the rectangle to obscure.

w

The width of the rectangle to obscure in pixels. Must be greater than 0.

h

The height of the rectangle to obscure in pixels. Must be greater than 0.

```
void evas_clear_obscured_rects(Evas e);
```

This function clears the list of obscured rectangles in the Evas passed in so no obscured rectangles are left in the canvas.

Arguments:

e

A valid Evas canvas handle.

```
Imlib_Updates evas_render_updates(Evas e);
```

This function flushes all changes and updates that have been queued whilst objects have been created and destroyed in the Evas canvas, and properties of these objects have ben changed, as well as all update rectangles that have been queued. Evas will optimise the rendering process to minimise the work required to display the output.

Return:

An Imlib2 updates list that you can process with Imlib2 updates processing functions. If nothing is rendered, NULL is returned. You must remember to free this updates list when you are done processing it. NULL will also be returned if the Evas handle passed in is NULL.

Arguments:

e

A valid Evas canvas handle.

```
void evas_render(Evas e);
```

This function flushes all changes and updates that have been queued whilst objects have been created and destroyed in the Evas canvas, and properties of these objects have ben changed, as well as all update rectangles that have been queued. Evas will optimise the rendering process to minimise the work required to display the output.

Unlike evas_render_updates() this function does not return anything.

Arguments:

e

A valid Evas canvas handle.

```
Visual *evas_get_optimal_visual(Evas e, Display *disp);
```

This function returns the optimal visual to use for the destination window for Evas output if used on the specified display. It is NOT valid to call this before evas_set_render_method() has been called. If you use the RENDER_METHOD_ALPHA_SOFTWARE rendering engine, it can deal with any visual you give it, but it will have a preference for a visual of a higher depth for multiple depth displays. For example, your server has both 8 and 24 bit visuals (8 bit overlay) and the most common configuration for this kind of display is to have the 8 bit visual be the default to allow legacy software that is not able to cope with depths of greater than 8 bit to run. This means though that quality suffers and thus the optimal visual this function returns would be the 24 bit visual.

For some rendering engines (such as RENDER_METHOD_3D_HARDWARE), you can only use the visual this routine returns for the output window, as it will not work otherwise.

Return:

The visual the program should use for the output window if it is taking the option of creating the window itself, if it is to have a guarantee of Evas working at all or of having optimal display quality and speed.

Arguments:

e

A valid Evas canvas handle.

disp

The X display that the program intends to use this Evas on.

```
Colormap evas_get_optimal_colormap(Evas e, Display *disp);
```

This function returns the optimal colormap to use for the destination window for Evas output if used on the specified display. It is NOT valid to call this before evas_set_render_method() has been called. If you use the RENDER_METHOD_ALPHA_SOFTWARE rendering engine, it can deal with any colormap you give it, but it will have a preference for a visual of a higher depth for multiple depth displays. For example, your server has both 8 and 24 bit visuals (8 bit overlay) and the most common configuration for this kind of display is to have the 8 bit visual be the default to allow legacy software that is not able to cope with depths of greater than 8 bit to run. This means though that quality suffers and thus the optimal colormap this function returns would be the 24 bit colormap.

For some rendering engines (such as RENDER_METHOD_3D_HARDWARE), you can only use the colormap this routine returns for the output window, as it will not work otherwise.

Return:

The colormap the program should use for the output window if it is taking the option of creating the window itself, if it is to have a guarantee of Evas working at all or of having optimal display quality and speed.

Arguments:

e

A valid Evas canvas handle.

disp

The X display that the program intends to use this Evas on.

```
void evas_get_drawable_size(Evas e, int *w, int *h);
```

This function returns the currently set width and height in pixels of the output window, drawable or image into the integers pointed to by *w* and *h* respectively. If the Evas is NULL, the values written into the integers pointed to will be 0.

Arguments:

e

A valid Evas canvas handle.

w

A valid pointer to an integer to be written to to store the current Evas output width in pixels. If the pointer is NULL, no width will be output.

h

A valid pointer to an integer to be written to to store the current Evas output height in pixels. If the pointer is NULL, no height will be output.

```
void evas_get_viewport(Evas e, double *x, double *y,
                       double *w, double *h);
```

This function returns the location and size of the viewport within the Evas canvas co−ordinate space that Evas is currently looking at. If any parameter pointer is NULL that parameter is not recorded in any integer.

Arguments:

e

A valid Evas canvas handle.

x

A pointer to a double that after the call will contain the x co−ordinate of the top−left corner of the viewport in the Evas canvas space. If it is NULL this parameter will not be recorded.

y

A pointer to a double that after the call will contain the y co−ordinate of the top−left corner of the viewport in the Evas canvas space. If it is NULL this parameter will not be recorded.

w

A pointer to a double that after the call will contain the width of the viewport in the Evas canvas space. If it is NULL this parameter will not be recorded.

h

A pointer to a double that after the call will contain the height of the viewport in the Evas canvas space. If it is NULL this parameter will not be recorded.

```
void evas_set_output(Evas e, Display *disp, Drawable d,
                     Visual *v, Colormap c);
```

This function sets the X output characteristics of the Evas (if you didn't create it with evas_new_all() or if you change the destination drawable). You need to use this routine for all rendering engines except the RENDER_METHOD_IMAGE engine. Evas cannot sensibly render with any of the engines except this one, until this routine has been called. Note that evas_new_all() calls this routine for you.

Arguments:

e

A valid Evas canvas handle.

disp

A valid X display pointer.

d

A valid drawable (either window or pixmap − but see the caveats section as to the limitations dependant on the engine being used)

v

A valid visual pointer. It is suggested this should be the visual returned by evas_get_optimal_visual() that you call after you call evas_set_render_method().

c

A valid colormap. It is suggested this should be the colormap returned by evas_get_optimal_colormap() that you call after you call evas_set_render_method().

```
void evas_set_output_image(Evas e, Imlib_Image image);
```

This call is analogous to evas_set_output() in that it tells evas where to direct the rendering output, but in this case it is used only for the RENDER_METHOD_IMAGE rendering engine and defines the Imlib2 image to be used as the output for rendering. If you use this rendering engine you must call this routine to specify where evas is to render to, otherwise nothing will be rendered. You should call this after calling evas_set_render_method(). You should not call evas_render or evas_render_updates or create any objects in the Evas canvas until you can called this function, if the output engine is RENDER_METHOD_IMAGE.

Arguments:

e

A valid Evas canvas handle.

image

A valid Imlib2 image handle that will be used for rendering output.

```
void evas_set_output_colors(Evas e, int colors);
```

This function sets the maximum number of colours that will be allocated in a pseudo colour display (8bit or less). The default value for an Evas canvas is 216 colours (6 elements per red, green and blue channel). This will determine only the upper limit. Evas may not be able to allocate enough colours and will then fall back to smaller colour cubes automatically. You should set this before you call any rendering functions on this Evas if you want the number of colours allocated to be anything except the default.

Arguments:

e

A valid Evas canvas handle.

colors

An integer value between 2 and 256 (inclusive) that defines the upper limit on the number of colours that will be allocated if Evas needs to allocate colours.

```
void evas_set_output_size(Evas e, int w, int h);
```

This routine will tell Evas the size of its output drawable or image in pixels. You must call this before you call any rendering functions and the width and height (*w* and *h*) must be the actual size of the drawable as much rendering depends on this being correct.

Arguments:

e

A valid Evas canvas handle.

w

The width of the output drawable or image in pixels. Valid values are between 1 and 8192.

h

The height of the output drawable or image in pixels. Valid values are between 1 and 8192.

```
void evas_set_output_viewport(Evas e, double x, double y,
                              double w, double h);
```

This function is used to tell Evas what geometry the output viewport has. By default when you first create an Evas canvas, the viewport geometry is at 0,0 for its top−left corner and its width and height in units is the same as the output canvas size in pixels. Thus we have a 1 to 1 canvas co−ordinate space to pixel space ratio by default. You can change its size and location whenever you want. This will result in effectively being able to zoom into or scroll around the canvas.

Arguments:

e

A valid Evas canvas handle.

x

The top−left x co−ordinate in canvas units for the box that defines the viewport region.

y

The top−left y co−ordinate in canvas units for the box that defines the viewport region.

w

The width of the viewport in canvas units (must be greater than 0).

h

The height of the viewport in canvas units (must be greater than 0).

```
void evas_set_output_method(Evas e, Evas_Render_Method method);
```

This routine sets the rendering engine to be used by an Evas canvas. You can only call this once on an Evas canvas as once it's set, it cannot be changed. There are some technical reasons for this − some of which are constraints of rendering engines themselves, and some being easier canvas management in not allowing this.

Arguments:

e

A valid Evas canvas handle.

method

The rendering method to be used for this Evas canvas. It must only be one of RENDER_METHOD_BASIC_HARDWARE, RENDER_METHOD_3D_HARDWARE, RENDER_METHOD_ALPHA_HARDWARE and RENDER_METHOD_IMAGE.

```
void evas_set_scale_smoothness(Evas e, int smooth);
```

This function enables or disables "smooth scaling" when images get scaled in an Evas canvas. If "smooth scaling" is set to 1, image data us super sampled when scaling an image down, and interpolated when scaling up. This results in smoother images and better image quality when scaling images. By default smooth scaling is enabled. Not all rendering enignes may implement this so it is only a hint or suggestion. Currently the software, image and 3d hardware engines support it (with the 3d hardware engine only supporting it if the underlying hardware does). Setting this to 0 will mean worse image quality, but will mean faster rendering in software, if this is an issue.

Arguments:

e

A valid Evas canvas handle.

smooth

An integer with values of 0 or 1. To disable smooth scaling, use 0 as the value, otherwise use 1 to enable it (which is the default).

```
void evas_set_clip(Evas e, Evas_Object o, Evas_Object clip);
```

Calling this routine sets the "clip object" for another object. A clip object clips and modulates all objects that it clips. This means that the objects clipped by the clip object will only be rendered within the space the clip object occupies − just like a clip mask. In addition the colour of the clipped objects is modulated by the colour of the clip object. That means if the clip object is *CO* and the object being clipped is *O*, the output colour is:

output_red = CO.red x O.red / 255

output_green = CO.green x O.green / 255

output_blue = CO.blue x O.blue / 255

output_alpha = CO.alpha x O.alpha / 255

Currently the only objects allowed to be used as a clip object are rectangle objects. Using any other object as a clip object has undefined results at this stage.

Clipping also works recursively. That means you can clip a clip object and the output of the objects clipped by the clip object that is now clipped will be the intersection of the 2 clip objects. You can keep clipping infinitely this way, BUT do NOT create clip loops − that is a clip object clips another that clips another than clips the first object. This will have bad results, so avoid this.

You can set the clip object for the object even if it already has one and the new clip object will apply instead. You do not need to un set the clip first.

Arguments:

e

A valid Evas canvas handle.

o

The object that will be clipped and modulated.

clip

The object that will clip and modulate the object o as described above.

```
void evas_unset_clip(Evas e, Evas_Object o);
```

This routine disables clipping for the object specified. After this the object will no longer be clipped by any clip object and will be displayed as if it were never clipped in the first place.

Arguments:

e

A valid Evas canvas handle.

o

The object to remove the clipping from.

```
Evas_Object evas_get_clip_object(Evas e, Evas_Object o);
```

This routine will return the object that clips the specified object. If no object clips this object, NULL will be returned.

Return:

A valid Evas object handle that is the object that is currently clipping the object specified in the arguments. If no object is currently clipping the nominated object, NULL is returned

Arguments:

e

A valid Evas canvas handle.

o

The object you wish to query to find its clipping object.

```
Evas_List evas_get_clip_list(Evas e, Evas_Object o);
```

This function returns a list of objects clipped by the object specified in the arguments

Return:

An Evas list structure that contains the Evas object handles as data elements. This list is only valid as long as no objects that this object clips are deleted or removed from being clipped from this object or no objects are added to the set of objects clipped by this object. If no objects are clipped by this object NULL is returned.

Arguments:

e

A valid Evas canvas handle.

o

The object that you wish to get the list of objects it clips from (if it clips any).

```
void evas_del_object(Evas e, Evas_Object o);
```

This function deletes an object from the Evas canvas. This will mean the object dissapears, and any objects it clips will no longer be clipped by it and will this appear as if they were normally drawn.h

Arguments:

e

A valid Evas canvas handle.

o

The object to be deleted.

```
Evas_Object evas_add_image_from_file(Evas e, char *file);
```

This function adds an image object to the canvas. The image data is retrieved from the file specified. If the file does not exist, or the file is not readable, or it is in a format Evas cannot read and decode, then the image object is still created, but it is blank, and nothing is rendered. You can find out if there was an error in loading the image by using evas_get_image_load_error() to return an Imlib2 error type. The width and height of an image that failed to load will also be 0 x 0. The default size of the image object in Evas canvas units is the size of the image file in pixels, and the default location is at 0,0 and the default fill parameters are 0,0 and with a width and height of the original image size in pixels.

Return:

A valid Evas object handle of the newly created object. You will still need to show the object, move it, resize it etc. As desired to make sure it is usefully displayed.

Arguments:

e

A valid Evas canvas handle.

file

A pointer to an array of characters that is 0−byte terminated (a normal C string) that is the path to a file to be used to retrieve the image data from. You may also use this file path as a virtual path. For example − you wish to load an image from an Edb database file. You would use the same format Imlib2 does, that is "/path/to/file.db:/key_in.database" where the format is "filename:key", where the key is the string key in the Edb database where the image was stored (using Imlib2 to store it).

```
Evas_Object evas_add_image_from_data(Evas e, void *data,
                                     Evas_Image_Format format,
                                     int w, int h);
```

This function is currently not implemented.

```
Evas_Object evas_add_text(Evas e, char *font, int size, char *text);
```

This function adds a text string object to the Evas canvas specified. The object by default is at 0,0 with a default colour of 0,0,0,0. You still need to move the object where you want it and to show it etc.

Return:

A valid Evas object handle.

Arguments:

e

A valid Evas canvas handle.

font

A font name used to specify the font to use.

size

The font size in Evas canvas units (integer sizes only).

text

A pointer to C−string (0 byte terminated array of chars) array that is used to contain the text string to use.

```
Evas_Object evas_add_rectangle(Evas e);
```

This function creates a rectangle object in the specified Evas canvas. The default colour is 0,0,0,0 and default location is 0,0 with a default size of 1 x 1 units in size. The program still needs to show, move, resize, set the colour etc. of the object.

Return:

A valid Evas object handle.

Arguments:

e

A valid Evas canvas handle.

```
Evas_Object evas_add_line(Evas e);
```

This function adds a line object to the specified Evas canvas with a default set of co−ordinates 0,0 and 0,0 for the start and end points, and a default colour of 0,0,0,0. The application still needs to set the start and end point co−ordinates, show and set the colour of the object.

Return:

A valid Evas object handle.

Arguments:

e

A valid Evas canvas handle.

```
Evas_Object evas_add_gradient_box(Evas e);
```

This routine adds a gradient box object to an Evas canvas. By default there is no gradient set and it has an angle of 0 degrees (from 12 noon, clockwise), and at location 0,0 with a size of 1 x 1 units. You need to set a gradient on this object for it to render anything useful, and need to move, resize, show etc. the object.

Return:

A valid Evas object handle.

Arguments:

e

A valid Evas canvas handle.

```
Evas_Object evas_add_poly(Evas e);
```

This routine adds a polygon object to the Evas canvas specified. You still need to set the polygon points to have anything rendered, as well as the colour, as well as show, move etc. the object.

Return:

A valid Evas object handle.

Arguments:

e

A valid Evas canvas handle.

```
void evas_set_image_file(Evas e, Evas_Object o, char *file);
```

This routine changed the image file used for an image object. If the object passed in is not an image object, nothing happens. In setting an image file successfully, the properties (size, image fill) are set as if it were a newly created image object. You can use evas_get_image_load_error() to see if the file exists and can be loaded.

Arguments:

e

A valid Evas canvas handle.

o

A valid Evas object handle that is the object you wish to change the image file of. It must be an image object otherwise nothing will happen.

file

The file path to the image file to be used (the same as would be used to create a new image object).

```
void evas_set_image_data(Evas e, Evas_Object o, void *data,
                         Evas_Image_Format format, int w, int h);
```

This function is currently not implemented.

```
void evas_set_image_fill(Evas e, Evas_Object o, double x, double y,
                         double w, double h);
```

This function sets how to fill an image object's space with the image data. The fill specifies at what unit co−ordinates within the image object, relative to its top left corner, the top−left corner of the image object is to start at, and what size this image data should be scaled to for output, in canvas units. If the object specified is not an image object, nothing happens.

The image object is tile filled with the image data so the whole area the object covers is filled with the image data in a tiled (repeated) fashion.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle to modify the image fill of.

x

The x co−ordinate of the top left origin of the image data relative to the top left corner of the image object where the tiled fill is to start.

y

The y co−ordinate of the top left origin of the image data relative to the top left corner of the image object where the tiled fill is to start.

w

The width of the output image data that is to be tiled, in canvas units.

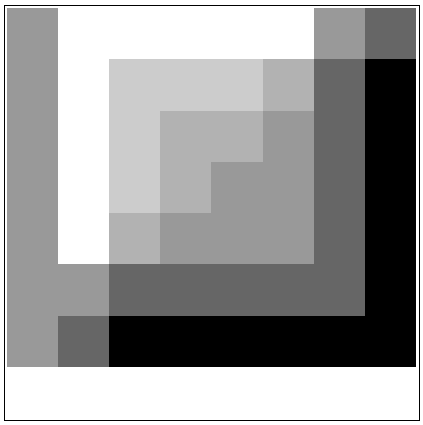
h

The height of the output image data that is to be tiled, in canvas units.

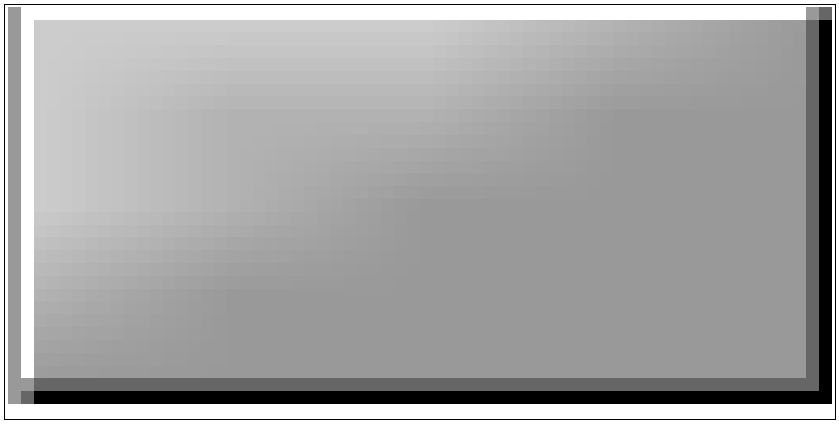
```
void evas_set_image_border(Evas e, Evas_Object o,
                           int l, int r, int t, int b);
```

This function sets the image border scaling parameters of an image object specified as an argument. These scaling parameters determine what original image pixels do not get scaled on output if the output image is scaled to a size that is not that of the original (in pixels). This is very useful for handling "bevels" in images so when scaled, only the middle scales and the edges do not. Here is an example of an image that is 8x8 pixels in size, where we set he left, right, top and bottom borders to 2 pixels.

This is the original image:



And here is the result of rendering when the above original image is scaled to an output of 64x32 pixels:



Notice that the middle scales (in this case smoothly because we have smooth scaling turned on), but the edges remain as in the original, allowing for applications to easily create button images and other complex images constructs without having to do as much themselves.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle to modify the image border scaling properties of.

l, r, t, b

The left, right, top and bottom border scaling parameters in pixel units. The default values for an image object are 0 for all of these parameters.

```
void evas_set_color(Evas e, Evas_Object o, int r, int g, int b, int a);
```

This function sets the colour value of an object. For Text objects it sets the red, green, blue and alpha values the text is rendered with. For line objects it sets the colour of the line. For rectangle objects it sets the colour of the rectangle. For polygon objects it sets the colour of the polygon. For image objects it sets the value each pixel in the image is multiplied by before composition (rendering) in the canvas.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle to modify the image fill of.

r, g, b, a

The red, green, blue and alpha values respectively to use for the object.

```
void evas_set_text(Evas e, Evas_Object o, char *text);
```

This function sets the text string contents of a text object to the text supplied in the arguments. This has no effect on objects other than text objects.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a text object to modify the textl of.

text

The C string (0 byte terminated array of chars) containing the ascii text to use for the text object.

```
void evas_set_font(Evas e, Evas_Object o, char *font, int size);
```

This function sets the font name and size (in integer canvas units) of a text object. It affects only text objects. The font name is the file name of a true type font in any directory in the font path set in Evas. It is case sensitive and does not need the extension of the file (often the file is names "font_name.ttf" and in this case the font name would be "font_n ame"). The true type font file that is found first in the list of directories in the path that matches the name of the font requested, is used.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a text object to modify the font and size of.

font

A C string name of the font to use.

size

An integer in the range from 1 upwards for the unit size of the font. (Warning some engines may have limitations on the largest glyph sizes internally. It is suggested to keep font sizes so that no character exceeds an output size of 255 x 255 pixels).

```
void evas_set_gradient(Evas e, Evas_Object o, Evas_Gradient grad);
```

This function sets the gradient to be used by an Evas gradient object. The gradient must be a valid gradient with 2 or more colours in it and a total spacing of 1 or more across the whole gradient. The object must be a gradient object, or nothing will happen.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a gradient object to set the gradient of.

grad

A valid gradient handle to be used as the gradient for the object.

```
void evas_set_angle(Evas e, Evas_Object o, double angle);
```

This function sets the angle in degrees from the vertical, in a clock−wise fashion, in which the first element of the gradient colour is to be found when drawn in a gradient object. This function only has an affect on gradient objects, and does not do anything if the object handle is not that of a gradient object. The angle can be any value at all.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a gradient object.

angle

The angle at which one would find the first colour element in a gradient, measured in degrees from the vertical, in a clock−wise fashion.

```
void evas_set_zoom_scale(Evas e, Evas_Object o, int scale);
```

This function currently has no effect.

```
void evas_set_line_xy(Evas e, Evas_Object o, double x1, double y1,
                      double x2, double y2);
```

This function sets the absolute co−ordinates of the end points of a line object. This function only has an effect on line objects.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a line object to modify the points of.

x1, y1

The co−ordinates, in canvas units of one end on the line.

x2, y2

The co−ordinates in canvas units of the opposite end of the line.

```
void evas_set_pass_events(Evas e, Evas_Object o, int pass_events);
```

This function enables events to pass through the object specified. If *pass_events* is 1, all events pass through the object as if it did not exist. The object becomes a purely visual element and has no bearing on the event capture and call back system as long as pass_events is set to 1 for this object. This is handy for using objects as mouse cursors or purely visual indicators in a canvas that have no actual function themselves.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of any object you wish to make impervious to events.

pass_events

An integer of value 0 or 1, where 0 indicates that events are to be captured by an object and their call backs called (which is the default state of an object), and 1, which indicates that the object is to ignore all events.

```
void evas_add_point(Evas e, Evas_Object o, double x, double y);
```

Calling this routine adds a point to a polygon object. The point is specified in absolute co−ordinates in canvas units.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a polygon object.

x, y

The co−ordinates of the point to be added in canvas units.

```
void evas_clear_points(Evas e, Evas_Object o);
```

This function clears the points list of a polygon object so there are no points, as is the default when a polygon object is created.

Arguments:

e

A valid Evas canvas handle.

o

The Evas object handle of a polygon object.

```
void evas_set_font_cache(Evas e, int size);
```

This function sets the size of the font cache for the engine the specified Evas canvas is using, in bytes. This cache is used to hold font glyph data when it is actually not currently used by any visible canvas object. This allows rapid changing between fonts and sizes without continual need for re−loading off disk and re−rasterisation of the font glyphs. If your application is sensitive to memory usage and can survive some performance degredation when it may keep changing text fonts, and sizes, then you are free to use a 0 sized cache, but it is suggested that in most circumstances it is a good idea to have a few hundred Kb of font cache − maybe 1 or 2 Mb if you use text heavily with various fonts and sizes. You may change the cache size at any time.

Arguments:

e

A valid Evas canvas handle.

size

The size of the memory pool in bytes to be used as cache for font glyphs that are currently not visible.

```
int evas_get_font_cache(Evas e);
```

This function returns the size of the font cache for the engine used by the specified canvas in bytes.

Return:

An integer equal or greater than 0, that is the number of bytes allowed to be used up as cache for non visible fonts.

Arguments:

e

A valid Evas canvas handle.

```
void evas_flush_font_cache(Evas e);
```

This function flushes all fonts out of the cache that are there, so nothing is held in cache anymore for the engine the specified canvas uses.

Arguments:

e

A valid Evas canvas handle.

```
void evas_set_image_cache(Evas e, int size);
```

This function sets the size of the image cache to be used for the rendering engine used by the canvas specified. The size of the cache is in bytes, and is used for the raw RGBA image data after it is decoded from disk. This means 1 pixel uses 4 bytes of cache. For a 100 x 100 image, that means 40,000 bytes (40kb) would be required in cache for that image to be retained in cache. This cache is only used for images that are not visible in the canvas, but will affect performance if images change visibility often.

Arguments:

e

A valid Evas canvas handle.

size

The size of the memory pool in bytes to be used as cache for image data that is currently not visible.

```
int evas_get_image_cache(Evas e);
```

This function returns the size of the image cache for the engine used by the specified canvas in bytes.

Return:

An integer equal or greater than 0, that is the number of bytes allowed to be used up as cache for non visible images.

Arguments:

e

A valid Evas canvas handle.

```
void evas_flush_image_cache(Evas e);
```

This function flushes all images out of the cache that are there, so nothing is held in cache anymore for the engine the specified canvas uses.

Arguments:

e

A valid Evas canvas handle.

```
void evas_font_add_path(Evas e, char *path);
```

This function adds the directory specified by the string *path* to the end of the list of directories scanned by the engine the canvas specified used to find fonts. If the path is already in the list for that engine it is removed and now added to the end.

Arguments:

e

A valid Evas canvas handle.

path

A C string (0 byte terminated) string that is a path to a directory where fonts can be found that will be appended to the end fo the current list for the engine the canvas uses.

```
void evas_font_del_path(Evas e, char *path);
```

This function deletes the specified path from the list of directories scanned for fonts by the engine used in the canvas specified.

Arguments:

e

A valid Evas canvas handle.

path

A C string (0 byte terminated) string that is a path to a directory where fonts can be found that will be appended to the end fo the current list for the engine the canvas uses. If this path is not in the list of directories used and is not an exact match, nothing is done.

```
void evas_set_layer(Evas e, Evas_Object o, int l);
```

This function sets the stacking layer used for the specified object to the numeric layer specified. The default layer for all objects is layer 0. Objects in layers with a greater number are stacked above objects in layers of lesser number − regardless if those other objects are raised. Objects can only be raised and lowered and otherwise stacked within their layer. If an object is intended to be stacked above objects with layer numbers greater than the object has, it must have a layer number higher than the objects it is to be stacked above. When an object's layer is changed it is always stacked at the top of the object stack in the layer it is in.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

l

The layer the object is to be stacked at the top of when this function is called. The object will remain in that layer until its layer number has changed. The default layer number is 0. Any integer can be used as a layer number.

```
int evas_get_layer(Evas e, Evas_Object o);
```

This function returns the current layer number of the object specified.

Return:

An integer that is the layer number of the object.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_set_layer_store(Evas e, int l, int store);
```

This function currently does nothing.

```
Evas_Gradient evas_gradient_new(void);
```

This function creates a new gradient, that is completely empty and returns a handle to it.

Return:

A valid gradient handle to an empty gradient.

```
void evas_gradient_free(Evas_Gradient grad);
```

This function frees a gradient and all colours in it.

Arguments:

grad

A valid gradient handle.

```
void evas_gradient_add_color(Evas_Gradient grad, int r, int g,
                             int b, int a, int dist);
```

This function adds a colour to the specified gradient at a specified "distance" from the colour that was previously added. The first colour added ignores the distance. The distances between colours in a gradient add up to a total distance (except for the first colour) that defines a length for the gradient. This gradient length is stretched across the space it is meant to fill.

Arguments:

grad

A valid gradient handle that is to have the colour added to.

r, g, b, a

The red, green, blue and alpha values of the colour to add (respectively) to the gradient.

dist

A distance that specifies how far away the colour is to be placed from the previously added colour.

```
void evas_raise(Evas e, Evas_Object o);
```

This function raises the specified object to the top of the stack of objects within its layer.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_lower(Evas e, Evas_Object o);
```

This function lowers the specified object to the bottom of the stack of objects within its layer.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_stack_above(Evas e, Evas_Object o, Evas_Object above);
```

This function raises the specified object above the *above* object specified object in the stack.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

above

A valid object handle in the canvas specified that the object *o* is to be stacked immediately above.

```
void evas_stack_below(Evas e, Evas_Object o, Evas_Object below);
```

This function lowers the specified object below the *below* object specified object in the stack.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

above

A valid object handle in the canvas specified that the object *o* is to be stacked immediately below.

```
void evas_move(Evas e, Evas_Object o, double x, double y);
```

This function moves a specified object in the canvas world so the co−ordinates of it's top−left corner are at the co−ordinates specified.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

x, y

The co−ordinates, in canvas units of the top−left corner of the object to be moved.

```
void evas_resize(Evas e, Evas_Object o, double w, double h);
```

This function resizes the specified object to the size in canvas units specified as arguments. This only works for rectangle, image and gradient objects.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

w, h

The width and height, respectively, of the new size for the object. These must be greater than 0 units in each direction to be valid.

```
void evas_get_geometry(Evas e, Evas_Object o, double *x, double *y,
                       double *w, double *h);
```

This function returns the current geometry of the specified object. The geometry is the bounding box that the object occupies, with the x and y co−ordinates defining the top left corner of the bounding box that the object occupies and the width and height being its size. If the pointer to any return parameter (x, y, w or h) is NULL that parameter is not filled in.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

x, y, w, h

Pointers to the variables that will be filled in with the object's geometry.

```
Evas_List evas_objects_in_rect(Evas e, double x, double y,
                               double w, double h);
```

This function returns a list of objects that intersect the rectangle in the canvas space defined by the box co−ordinates supplied. The list of objects is from bottom to top. If there are no objects then NULL is returned. The data member of each list element is an object handle. The calling program must free this list with evas_list_free() when it is done with it.

Return:

A list of objects, ordered from bottom to top, with the data member being a valid object handle of each list element, or NULL if none are found. The caller is responsible for freeing the list later.

Arguments:

e

A valid Evas canvas handle.

x, y, w, h

The co−ordinates of the box being queried for its contents of objects.

```
Evas_List evas_objects_at_position(Evas e, double x, double y);
```

This function returns a list of objects which intersect the co−ordinate specified, in the specified canvas. The list is ordered from bottom most object to top most object, with the data element being an object handle. If no objects intersect this point, NULL is returned. The caller needs to use evas_list_free() to free the list when they are done with it.

Return:

A list of objects, ordered from bottom to top, with the data member being a valid object handle of each list element, or NULL if none are found. The caller is responsible for freeing the list later.

Arguments:

e

A valid Evas canvas handle.

x, y

A point in canvas co−ordinate space that you wish to have a list of objects returned that intersect that point in the canvas.

```
Evas_Object evas_object_in_rect(Evas e, double x, double y,
                                double w, double h);
```

This function returns the top most object that intersects the rectangle described in the specified canvas. If no object intersects this rectangle, NULL is returned.

Return:

A valid object handle of the top most object that intersects this rectangle, if there is one, or NULL if none is found.

Arguments:

e

A valid Evas canvas handle.

x, y, w, h

Parameters describing a rectangle in the canvas with *x* and *y* describing its top−left corner, and *w* and *h* denoting its width and height (where these are greater than 0).

```
Evas_Object evas_object_at_position(Evas e, double x, double y);
```

This function returns the top most object that intersects the point described in the specified canvas. If no object intersects this point, NULL is returned.

Return:

A valid object handle of the top most object that intersects this point, if there is one, or NULL if none is found.

Arguments:

e

A valid Evas canvas handle.

x, y

A point in canvas co−ordinate space that you wish to have an object handle returned that intersects that point in the canvas.

```
Evas_Object evas_object_get_named(Evas e, char *name);
```

This function returns the object handle of an object whose name matches the specified name parameter passed in. There must be an object whose name matches the name specified to get back a valid object handle, otherwise NULL is returned. If there is more than one object with such a name it is undefined as to which object handle will be returned.

Return:

A valid object handle of the object that has the name requested, or NULL if none exists in the specified canvas.

Arguments:

e

A valid Evas canvas handle.

name

A C string (0 byte terminated) that contains the name being queried.

```
void evas_object_set_name(Evas e, Evas_Object o, char *name);
```

This function sets the name of a specified object in a canvas. It is suggested that the name be unique to that object in that canvas as it is assumed there will be a one to one, name to object mapping.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

name

A C string (0 byte terminated) that contains the name being set on the object. If it already has a name it will be replaced by this new name. If *name* is NULL, the name o the object will be cleared and it will have no name. Objects by default have no name. The name will be internally duplicated so the string passed in to this function can be used as the program sees fit.

```
char *evas_object_get_name(Evas e, Evas_Object o);
```

This function will return the name set on a specific object, if there is one, or NULL if there is not.

Return:

A pointer to a C string (0 byte terminated), that is the internal string of the object name. Or NULL if it has no name. This pointer is only valid as long as the program does not change the object name, or destroy the object.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
Evas_List evas_get_points(Evas e, Evas_Object o);
```

This function returns a list of points in a polygon object. If the polygon object has no points, or the object handle is not a polygon object, NULL is returned. Each data member in the list is an Evas_Point handle. A point structure has 2 members, x and y.

```
struct _Evas_Point
{
   double x, y;
};
typedef struct _Evas_Point * Evas_Point;
```

Return:

A list of point handles, or NULL if no points exist. This list is only valid as long as no points are added to the object queried, the points of the object are not cleared, and the object is not destroyed.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_show(Evas e, Evas_Object o);
```

This function makes a specified object visible.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_hide(Evas e, Evas_Object o);
```

This function hides a specified object so that it is no longer visible.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
int evas_get_image_alpha(Evas e, Evas_Object o);
```

This function returns if an image object has an alpha channel or not. If it does it returns 1, if it does not have an alpha channel, it returns 0. If the object is not an image object, 0 will be returned anyway.

Return:

An integer which is 0 or 1. 0 denotes that the object has no alpha channel, or is not an image object, and 1 denotes that the image object has an alpha channel.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
void evas_get_image_size(Evas e, Evas_Object o, int *w, int *h);
```

This function returns the size of an image object, in integer pixels, and places the result in the pointers to the *w* and *h* parameters passed in.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

w, h

Pointers to integers to be filled in with the image object's width and height respectively. If any one of these is NULL, the parameter is not filled in.

```
void evas_get_image_border(Evas e, Evas_Object o,
                           int *l, int *r, int *t, int *b);
```

This routine returns the border scaling settings of an image object and places the results in the integers pointed to. If the object is not an image object the results are undefined.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

l, r, t, b

Pointers to integers where the left, right, top and bottom (respectively) border scaling parameters for the image object will be written. If any parameter is NULL, it will not be written to.

```
Imlib_Load_Error evas_get_image_load_error(Evas e, Evas_Object o);
```

This function returns the error whilst trying to load an image object. If the object is not an image object the results are undefined.

Return:

A load status/ error value. It can be one of:

IMLIB_LOAD_ERROR_NONE, IMLIB_LOAD_ERROR_FILE_DOES_NOT_EXIST,
IMLIB_LOAD_ERROR_FILE_IS_DIRECTORY,
IMLIB_LOAD_ERROR_PERMISSION_DENIED_TO_READ,
IMLIB_LOAD_ERROR_NO_LOADER_FOR_FILE_FORMAT,
IMLIB_LOAD_ERROR_PATH_TOO_LONG,
IMLIB_LOAD_ERROR_PATH_COMPONENT_NON_EXISTANT,
IMLIB_LOAD_ERROR_PATH_COMPONENT_NOT_DIRECTORY,
IMLIB_LOAD_ERROR_PATH_POINTS_OUTSIDE_ADDRESS_SPACE,
IMLIB_LOAD_ERROR_TOO_MANY_SYMBOLIC_LINKS,
IMLIB_LOAD_ERROR_OUT_OF_MEMORY,
IMLIB_LOAD_ERROR_OUT_OF_FILE_DESCRIPTORS,
IMLIB_LOAD_ERROR_PERMISSION_DENIED_TO_WRITE,
IMLIB_LOAD_ERROR_OUT_OF_DISK_SPACE, IMLIB_LOAD_ERROR_UNKNOWN

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
int evas_world_x_to_screen(Evas e, double x);
```

This function transforms a horizontal point or distance in canvas space to output pixel units. The result depends on the ratio of the viewport size to the output pixel size.

Return:

An integer in pixel units.

Arguments:

e

A valid Evas canvas handle.

x

The value in canvas units to be transformed into output pixel units.

```
int evas_world_y_to_screen(Evas e, double y);
```

This function transforms a vertical point or distance in canvas space to output pixel units. The result depends on the ratio of the viewport size to the output pixel size.

Return:

An integer in pixel units.

Arguments:

e

A valid Evas canvas handle.

y

The value in canvas units to be transformed into output pixel units.

```
double evas_screen_x_to_world(Evas e, int x);
```

This function transforms a horizontal point or distance in pixel space to canvas units. The result depends on the ratio of the viewport size to the output pixel size.

Return:

An double precision value in canvas space units.

Arguments:

e

A valid Evas canvas handle.

x

The value in output pixel units to be transformed into canvas units.

```
double evas_screen_y_to_world(Evas e, int y);
```

This function transforms a vertical point or distance in pixel space to canvas units. The result depends on the ratio of the viewport size to the output pixel size.

Return:

An double precision value in canvas space units.

Arguments:

e

A valid Evas canvas handle.

y

The value in output pixel units to be transformed into canvas units.

```
char *evas_get_text_string(Evas e, Evas_Object o);
```

This function returns a pointer to the internal string of the specified object in the canvas specified. This is only valid as long as the object is not destroyed and the text of the string object has not been changed. NULL will be returned if the object has no text set or it is not a text object.

Return:

A pointer to a C string (0 byte terminated) that contains the text of the object, or NULL if no text string is set.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
char *evas_get_text_font(Evas e, Evas_Object o);
```

This function returns the font name used by a text object. If the object is not a text object, an empty string is returned. The font name returned is only valid so long as the object isn't freed or the font isn't changed.

Return:

A pointer to a C string (0 byte terminated) that contains the font name of the object.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
int evas_get_text_size(Evas e, Evas_Object o);
```

This function returns the size of the font used for the specified text object. If it is not a text object, 0 is returned.

Return:

An integer that is the size of the font used, or 0 if the object is not a text object.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
double evas_get_text_width(Evas e, Evas_Object o);
```

This function returns the total width of a text object in canvas units. If the object specified is not a text object, 0 is returned.

Return:

A double precision float value that is the width of the text object in canvas units.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
double evas_get_text_height(Evas e, Evas_Object o);
```

This function returns the total height of a text object in canvas units. If the object specified is not a text object, 0 is returned.

Return:

A double precision float value that is the height of the text object in canvas units.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

```
int evas_text_at_position(Evas e, Evas_Object o, double x, double y,
                          double *char_x, double *char_y,
                          double *char_w, double *char_h);
```

This function returns the character index of the character in the string object that the specified co−ordinates are over. The specified co−ordinates are relative to the top left corner of the text object (being 0,0) and are in canvas units. The index returned is the character in the string the co−ordinates are over, if any. If the co−ordinates are not within a character in the text object −1 is returned instead. If the co−ordinates do intersect a character, the co−ordinates of the bounding box of the glyph of that character, relative to the top left corner of the text object are returned to the values pointed to by the character metric pointers.

Return:

An integer from 0 to N − 1, where N is the number of characters in the text string. If the co−ordinates do not lie within a character of the string object, or the object is not a string object, −1 is returned.

Arguments:

e

A valid Evas canvas handle.

o

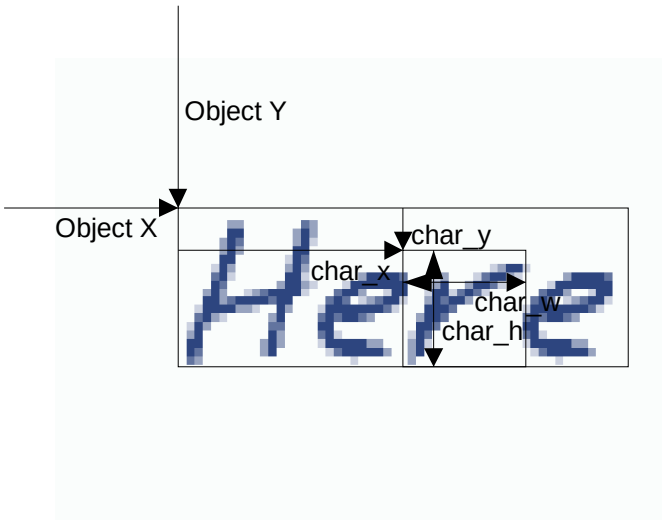
A valid object handle in the canvas specified.

x, y

The co−ordinates in canvas units relative to the top left corner of the text object that are to be queried for the contents of a character.

char_x, char_y, char_w, char_h

Pointers to double float values that will have the bounding box co−ordinates of the character queried filled in, if the pointers are non NULL. If the function returns −1 the contents of the variables pointed to will be unchanged.



```
void evas_text_at(Evas e, Evas_Object o, int index,
                  double *char_x, double *char_y,
                  double *char_w, double *char_h);
```

This function returns the character geometry (as returned by evas_text_at_position()) of the character index specified. If the index is invalid (with 0 being the first character, and valid indexes going from 0 to N − 1 where N is the number of characters in the text object string are valid values), the contents of the return geometry is unspecified. The contents of the return geometry is also unspecified if the object is not a text object.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

index

The character index in the string being queried. This must be greater or equal to 0 and less than the total number of characters in the text object string.

char_x, char_y, char_w, char_h

Pointers to double float values that will have the bounding box co−ordinates of the character queried filled in, if the pointers are non NULL.

```
void evas_text_get_ascent_descent(Evas e, Evas_Object o,
                                  double *ascent, double *descent);
```

This function returns the ascent and descent values in canvas units for the font used in the text object specified. If the object is not a text object the values written into *ascent* and *descent* are unspecified.

Arguments:

e

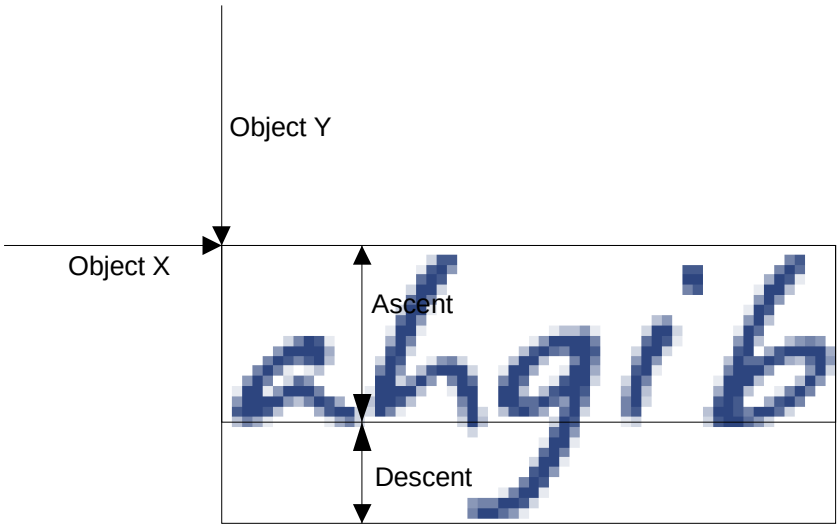
A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

ascent, descent

Pointers to double precision float values that will be filled in with the ascent and descent valued of the font used in the text object. If any of these pointers is NULL, it is not filled in.



```
void evas_text_get_max_ascent_descent(Evas e, Evas_Object o,
                                      double *ascent, double *descent);
```

This function operates the same way evas_text_get_ascent_descent() works, but returns the maximum extents of the ascent and descent values in the font, not just the values the font specifies (which may be the same or less than the ascent and descent values returned by evas_text_get_ascent_descent()). If the object is not a text object the values written into *ascent* and *descent* are unspecified.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

ascent, descent

Pointers to double precision float values that will be filled in with the ascent and descent valued of the font used in the text object. If any of these pointers is NULL, it is not filled in.

```
void evas_text_get_advance(Evas e, Evas_Object o,
                           double *h_advance, double *v_advance);
```

This function returns the number of canvas units to advance horizontally and vertically if more text objects are to be placed to the right or bottom of this object. If the object is not a text object the values written to the advance values is undefined.

Arguments:

e

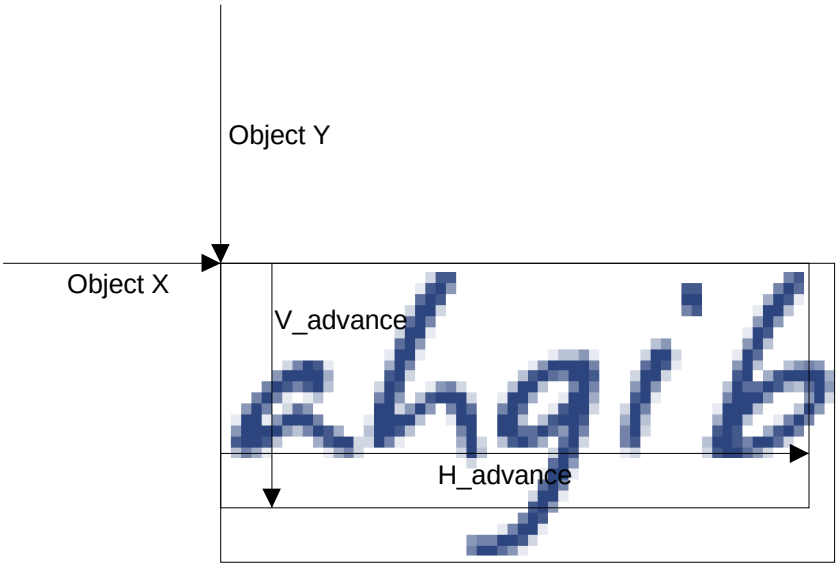
A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

h_advance, v_advance

Pointers to 2 double values that the horizontal and vertical advance values respectively will be written to. If any of them is NULL it will not be written to.



```
double evas_text_get_inset(Evas e, Evas_Object o);
```

This function returns the inset from the left side of the text object and where the horizontal font specified starting co−ordinate is. If the object is not a text object, 0 is returned.

Return:

A value in canvas units that is the inset between the left edge of the font object and the start of the text.

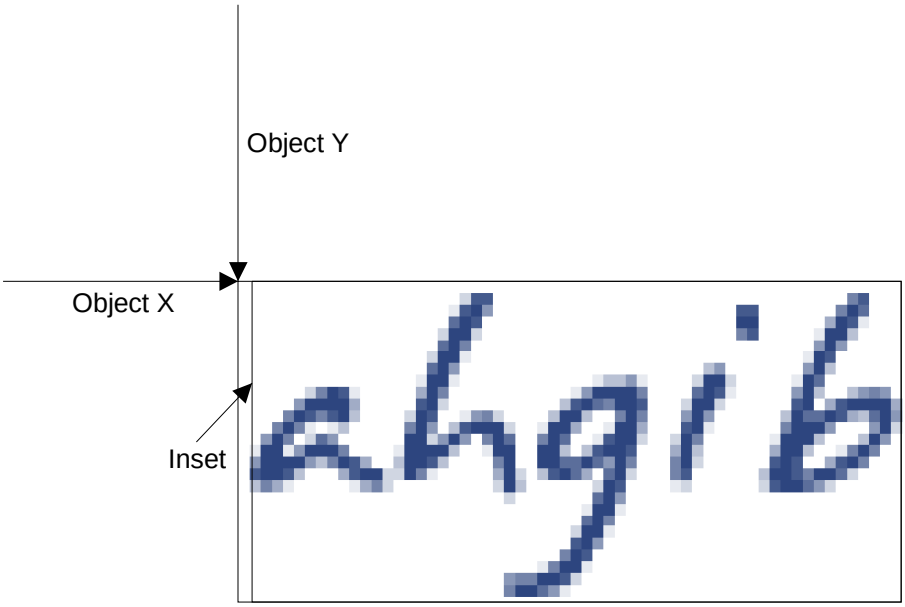
Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.



```
void evas_get_color(Evas e, Evas_Object o,
                    int *r, int *g, int *b, int *a);
```

This function returns the colour set on an object and writes the colour values into the pointers passed into the function.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

r, g, b, a

Pointers to integers where the values for red, green, blue and alpha respectively will be written.

```
Evas_Object evas_get_object_under_mouse(Evas e);
```

This function returns a handle to the object that is currently under the mouse pointer position that is currently held in the Evas canvas. If there is no object under the cursor position, NULL is returned.

Return:

A valid object handle of an object under the pointer, or NULL if there is none.

Arguments:

e

A valid Evas canvas handle.

```
void evas_put_data(Evas e, Evas_Object o, char *key, void *data);
```

This function attaches a pointer to specified data with a string name to an object in an Evas canvas. If a pointer is already attached to that object with that string key, it is replaced.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

key

A C string (0 byte terminated) that contains the key string to attach the data pointer to.

data

A pointer to any data the program wishes to attach to the object.

```
void *evas_get_data(Evas e, Evas_Object o, char *key);
```

This function returns the data pointer attached to an object under the specified key. If no data is attached, NULL is returned.

Return:

A pointer to data attached under the key to the specified object, otherwise NULL if no data is attached.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

key

A C string (0 byte terminated) that contains the key string to be queried for the data pointer attached.

```
void *evas_remove_data(Evas e, Evas_Object o, char *key);
```

This function removes the data pointer attached to the specified object and returned the pointer. If no data was attached NULL is returned.

Return:

A pointer to data attached under the key to the specified object, otherwise NULL if no data is attached.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

key

A C string (0 byte terminated) that contains the key string to be removed for the data pointer attached.

```
void evas_event_button_down(Evas e, int x, int y, int b);
```

This function feeds a mouse button down event into the specified canvas. The button number can be a value from 1 to 32, and the co−ordinates are locations on the output canvas in pixel co−ordinate space. If you are passing in button down events you MUST pass in button up events or the event system will not work properly. Use evas_event_button_up() to pass in button raise events into the canvas. As with all the event functions, callbacks are synchronous and will be called if they are triggered the moment this function is called, and it will not return until all callbacks triggered have returned.

Arguments:

e

A valid Evas canvas handle.

x, y

Pixel space co−ordinates where the mouse button was pressed down.

b

The button number that was pressed down. Can be a value from 1 to 32 only.

```
void evas_event_button_up(Evas e, int x, int y, int b);
```

This function passes a mouse button raise event to the Evas canvas specified. As with all the event functions, callbacks are synchronous and will be called if they are triggered the moment this function is called, and it will not return until all callbacks triggered have returned.

Arguments:

e

A valid Evas canvas handle.

x, y

Pixel space co−ordinates where the mouse button was raised.

b

The button number that was raised. Can be a value from 1 to 32 only.

```
void evas_event_move(Evas e, int x, int y);
```

This function passes a mouse motion event to the Evas canvas specified, in output pixel co−ordinates. This should be used if the mouse buttons are pressed or not. As with all the event functions, callbacks are synchronous and will be called if they are triggered the moment this function is called, and it will not return until all callbacks triggered have returned.

Arguments:

e

A valid Evas canvas handle.

x, y

Pixel space co−ordinates where the mouse button has moved to.

```
void evas_event_enter(Evas e);
```

This function hints to the Evas canvas that the mouse cursor has entered the canvas output space. As with all the event functions, callbacks are synchronous and will be called if they are triggered the moment this function is called, and it will not return until all callbacks triggered have returned.

Arguments:

e

A valid Evas canvas handle.

```
void evas_event_leave(Evas e);
```

This function hints to the Evas canvas that he mouse cursor has actually left the canvas output space. As with all the event functions, callbacks are synchronous and will be called if they are triggered the moment this function is called, and it will not return until all callbacks triggered have returned.

Arguments:

e

A valid Evas canvas handle.

```
int evas_pointer_in(Evas e);
```

This function returns if the mouse cursor is currently considered by the canvas to be inside it. IT returns 1 if it is, and 0 if it is not.

Return:

It returns 1 if the pointer is considered to be in the canvas, and 0 if it is not.

Arguments:

e

A valid Evas canvas handle.

```
void evas_pointer_pos(Evas e, int *x, int *y);
```

This function returns the pointer position that the canvas considers the pointer to currently be at. The output pixel positions are written into the pointers to the co−ordinates provided.

Arguments:

e

A valid Evas canvas handle.

x, y

Pointers to 2 integers that will be filled in with the co−ordinates of the current cursor position in the specified Evas canvas. If any pointer is NULL it will not be filled in.

```
int evas_pointer_buttons(Evas e);
```

This function returns a mask as an integer with the bits that are set to 1 corresponding to the buttons that are currently pressed down. Bit 0 (the least significant bit) corresponding to button 1 being pressed, bit 1 corresponding to button 2 and so on.

Return:

An integer mask containing a bit mask of the buttons currently pressed in the specified canvas.

Arguments:

e

A valid Evas canvas handle.

```
void evas_pointer_ungrab(Evas e);
```

This function removes the implicit grab of the mouse button once it has become active. This allows leave and enter events to proceed as per normal which they would not until the mouse button that was presses to initiate the grab is released.

Arguments:

e

A valid Evas canvas handle.

```
void evas_callback_add(Evas e, Evas_Object o, Evas_Callback_Type callback,
                       void (*func) (void *_data, Evas _e,
                                     Evas_Object _o, int _b,
                                     int _x, int _y),
                       void *data);
```

This function adds a callback on an object. The *func* parameter is the function pointer to be called when the event triggering the *callback* type callback is triggered on the specified object. The *data* pointer parameter is passed to the callback added to the object if it is called.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

callback

The callback type. This can be one of:

CALLBACK_MOUSE_IN, CALLBACK_MOUSE_OUT, CALLBACK_MOUSE_DOWN, CALLBACK_MOUSE_UP, CALLBACK_MOUSE_MOVE, CALLBACK_FREE.

func

The pointer to the function to be called when the callback type event is triggered on the specified object.

data

A pointer to data or NULL that is passed to the calback when it is triggered.

```
void evas_callback_del(Evas e, Evas_Object o,
                       Evas_Callback_Type callback);
```

This function removes all callbacks set on the specified object that is of the type *callback*, if there are any.

Arguments:

e

A valid Evas canvas handle.

o

A valid object handle in the canvas specified.

callback

The callback type. This can be one of:

CALLBACK_MOUSE_IN, CALLBACK_MOUSE_OUT, CALLBACK_MOUSE_DOWN, CALLBACK_MOUSE_UP, CALLBACK_MOUSE_MOVE, CALLBACK_FREE.

```
Evas_List evas_list_append(Evas_List list, void *data);
```

This is one of the list convenience functions in Evas. This function appends a pointer to data to an existing list at the end − be it empty or not. Example usage:

```
Evas_List list = NULL;
void *data;

list = evas_list_append(list, data);
```

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data to be appended.

```
Evas_List evas_list_prepend(Evas_List list, void *data);
```

This is one of the list convenience functions in Evas. This function pepends a pointer to data to an existing list at its start − be it empty or not. Example usage:

```
Evas_List list = NULL;
void *data;

list = evas_list_prepend(list, data);
```

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data to be prepended.

```
Evas_List evas_list_append_relative(Evas_List list, void *data,
                                    void *relative);
```

This is one of the list convenience functions in Evas. This function appends a pointer to data to an existing list after the element with the data pointer specified by *relative*. If the element does not exist in the list the item is appended to the end of the list. Example usage:

```
Evas_List list = NULL;
void *data, *relative;

list = evas_list_append_relative(list, data, relative);
```

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data to be appended.

relative

A pointer to the data to be appended immediately after.

```
Evas_List evas_list_prepend_relative(Evas_List list, void *data,
                                     void *relative);
```

This is one of the list convenience functions in Evas. This function prepends a pointer to data to an existing list before the element with the data pointer specified by *relative*. If the element does not exist in the list the item is prepended to the start of the list. Example usage:

```
Evas_List list = NULL;
void *data, *relative;

list = evas_list_prepend_relative(list, data, relative);
```

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data to be prepended.

relative

A pointer to the data to be prepended immediately before.

```
Evas_List evas_list_remove(Evas_List list, void *data);
```

This function removes the element of the list with the data pointer specified by *data* if it is in the list. If it isn't it does nothing. Example usage:

```
Evas_List list = NULL;
void *data;

list = evas_list_remove(list, data);
```

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data of the element to be removed from the list.

```
Evas_List evas_list_remove_list(Evas_List list, Evas_List remove_list);
```

This function removes The tail of a list starting at the list pointer *remove_list* from the *list* list passed in, if it exists, and returns the modified original list.

Return:

A modified list handle.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

remove_list

The list element pointer to remove from the list as the head of the new list. All elements after this element in the list are now at the tail of the newly removed list.

```
void *evas_list_find(Evas_List list, void *data);
```

This function finds an element in the list with the data pointer specified. If it exists the data pointer specified is returned, if not NULL is returned.

Return:

The data pointer to the list element found, or NULL if none is found.

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

data

A pointer to the data of the element to be found in the list.

```
Evas_List evas_list_free(Evas_List list);
```

This function frees the entire list pointed to and returns NULL.

Return:

NULL

Arguments:

list

A filled or empty list handle (an empty list being a NULL pointer).

Code Examples

The following snippets of code are examples of how to use Evas, and to demonstrate how easy it really is. Note that these examples require the Evas library AND the Ecore library that wraps and handles X Events for us so we don't have to do as much X coding as we would normally have to to get these programs to run. You will notice the Ecore routines start with an e_ notation.

Example 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
void setup(void);

/* callbacks for evas handling */
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);

/* globals */
Evas_Object o_flower;
Evas_Object o_bg;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;

static void
e_idle(void *data)
{
   evas_render(evas);
}

static void
e_window_expose(Eevent * ev)
{
   Ev_Window_Expose      *e;

   e = (Ev_Window_Expose *)ev->event;
   evas_update_rect(evas, e->x, e->y, e->w, e->h);
}

/* meat */
void
setup(void)
{
   Window win, ewin;
   int i;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);

   /* create a 400x300 evas rendering in software - conveience function
that */
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
                       MAX_EVAS_COLORS, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                       FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
```

```
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_rectangle(evas);
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);
   evas_set_color(evas, o_bg, 255, 255, 255, 255);
   evas_show(evas, o_bg);

   o_flower = evas_add_image_from_file(evas, "flower.png");
   evas_move(evas, o_flower, 10.0, 30.0);
   evas_show(evas, o_flower);
   evas_resize(evas, o_flower, 300.0, 200.0);
   evas_set_image_fill(evas, o_flower, 0.0, 0.0, 40.0, 30.0);
}

int
main(int argc, char **argv)
{
   /* command line parsing */
     {
      int i;

      for (i = 1; i < argc; i++)
        {
           if (!strcmp(argv[i], "soft"))
             render_method = RENDER_METHOD_ALPHA_SOFTWARE;
           if (!strcmp(argv[i], "x11"))
             render_method = RENDER_METHOD_BASIC_HARDWARE;
           if (!strcmp(argv[i], "hard"))
             render_method = RENDER_METHOD_3D_HARDWARE;
        }
     }
   /* init X */
   e_display_init(NULL);
   /* setup handlers for system signals */
   e_ev_signal_init();
   /* setup the event filter */
   e_event_filter_init();
   /* setup the X event internals */
   e_ev_x_init();

   /* program does its data setup here */
   setup();
   /* and now loop forever handling events */
   e_event_loop();
}
```

Example 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
void setup(void);

/* callbacks for evas handling */
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);

/* globals */
Evas_Object o_flower;
Evas_Object o_bg;
Evas_Object o_text;
Evas_Object o_grad;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;

static void
e_idle(void *data)
{
   evas_render(evas);
}

static void
e_window_expose(Eevent * ev)
{
   Ev_Window_Expose      *e;

   e = (Ev_Window_Expose *)ev->event;
   evas_update_rect(evas, e->x, e->y, e->w, e->h);
}

/* meat */
void
setup(void)
{
   Window win, ewin;
   int i;
   Evas_Gradient grad;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);

   /* create a 400x300 evas rendering in software - conveience function
that */
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
```

```
                        MAX_EVAS_COLORS, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                        FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_rectangle(evas);
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);
   evas_set_color(evas, o_bg, 255, 255, 255, 255);
   evas_show(evas, o_bg);

   o_flower = evas_add_image_from_file(evas, "flower.png");
   evas_move(evas, o_flower, 10.0, 30.0);
   evas_show(evas, o_flower);

   o_text = evas_add_text(evas, "notepad", 18, "Here is a line of text!");
   evas_move(evas, o_text, 120.0, 50.0);
   evas_set_color(evas, o_text, 0, 0, 0, 255);
   evas_show(evas, o_text);

   o_grad = evas_add_gradient_box(evas);
   evas_move(evas, o_grad, 150.0, 100.0);
   evas_resize(evas, o_grad, 200.0, 120.0);
   grad = evas_gradient_new();
   evas_gradient_add_color(grad, 255, 255, 255, 255, 10);
   evas_gradient_add_color(grad, 255, 255, 0,   255, 10);
   evas_gradient_add_color(grad, 255, 0,   0,   255, 10);
   evas_gradient_add_color(grad, 0,   0,   128, 255, 10);
   evas_gradient_add_color(grad, 0,   0,   128, 0,   10);
   evas_set_gradient(evas, o_grad, grad);
   evas_gradient_free(grad);
   evas_set_angle(evas, o_grad, 290.0);
   evas_show(evas, o_grad);
}

int
main(int argc, char **argv)
{
   /* command line parsing */
     {
      int i;

      for (i = 1; i < argc; i++)
        {
           if (!strcmp(argv[i], "soft"))
             render_method = RENDER_METHOD_ALPHA_SOFTWARE;
           if (!strcmp(argv[i], "x11"))
             render_method = RENDER_METHOD_BASIC_HARDWARE;
           if (!strcmp(argv[i], "hard"))
             render_method = RENDER_METHOD_3D_HARDWARE;
        }
     }
   /* init X */
   e_display_init(NULL);
   /* setup handlers for system signals */
   e_ev_signal_init();
   /* setup the event filter */
   e_event_filter_init();
   /* setup the X event internals */
   e_ev_x_init();

   /* program does its data setup here */
   setup();
```

```
        /* and now loop forever handling events */
        e_event_loop();
    }
```

Example 3:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
void setup(void);

/* callbacks for evas handling */
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);

/* globals */
Evas_Object o_flower;
Evas_Object o_bg;
Evas_Object o_text;
Evas_Object o_grad;
Evas_Object o_rect;
Evas_Object o_poly;
Evas_Object o_line;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;

static void
e_idle(void *data)
{
   evas_render(evas);
}

static void
e_window_expose(Eevent * ev)
{
   Ev_Window_Expose      *e;

   e = (Ev_Window_Expose *)ev->event;
   evas_update_rect(evas, e->x, e->y, e->w, e->h);
}

/* meat */
void
setup(void)
{
   Window win, ewin;
   int i;
   Evas_Gradient grad;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);

   /* create a 400x300 evas rendering in software - conveience function
that */
```

```
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
                       MAX_EVAS_COLORS, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                       FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE | XEV_BUTTON | XEV_MOUSE_MOVE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_rectangle(evas);
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);
   evas_set_color(evas, o_bg, 255, 255, 255, 255);
   evas_show(evas, o_bg);

   o_flower = evas_add_image_from_file(evas, "flower.png");
   evas_move(evas, o_flower, 10.0, 30.0);
   evas_show(evas, o_flower);

   o_text = evas_add_text(evas, "notepad", 18, "Here is a line of text!");
   evas_move(evas, o_text, 120.0, 50.0);
   evas_set_color(evas, o_text, 0, 0, 0, 255);
   evas_show(evas, o_text);

   o_grad = evas_add_gradient_box(evas);
   evas_move(evas, o_grad, 150.0, 100.0);
   evas_resize(evas, o_grad, 200.0, 120.0);
   grad = evas_gradient_new();
   evas_gradient_add_color(grad, 255, 255, 255, 255, 10);
   evas_gradient_add_color(grad, 255, 255, 0,   255, 10);
   evas_gradient_add_color(grad, 255, 0,   0,   255, 10);
   evas_gradient_add_color(grad, 0,   0,   128, 255, 10);
   evas_gradient_add_color(grad, 0,   0,   128, 0,   10);
   evas_set_gradient(evas, o_grad, grad);
   evas_gradient_free(grad);
   evas_set_angle(evas, o_grad, 290.0);
   evas_show(evas, o_grad);

   o_rect = evas_add_rectangle(evas);
   evas_move(evas, o_rect, 20.0, 130.0);
   evas_resize(evas, o_rect, 50.0, 70.0);
   evas_set_color(evas, o_rect, 20.0, 50.0, 100.0, 130.0);
   evas_show(evas, o_rect);

   o_poly = evas_add_poly(evas);
   evas_add_point(evas, o_poly, 0.0, 0.0);
   evas_add_point(evas, o_poly, 150.0, 80.0);
   evas_add_point(evas, o_poly, 210.0, 150.0);
   evas_add_point(evas, o_poly, 80.0, 110.0);
   evas_add_point(evas, o_poly, 20.0, 30.0);
   evas_set_color(evas, o_poly, 200.0, 40.0, 0.0, 130.0);
   evas_move(evas, o_poly, 20.0, 220.0);
   evas_show(evas, o_poly);

   o_line = evas_add_line(evas);
   evas_set_line_xy(evas, o_line, 220.0, 240.0, 390.0, 380.0);
   evas_set_color(evas, o_line, 30.0, 80.0, 80.0, 200.0);
   evas_show(evas, o_line);
}

int
main(int argc, char **argv)
{
   /* command line parsing */
```

```
        {
         int i;

         for (i = 1; i < argc; i++)
           {
              if (!strcmp(argv[i], "soft"))
                render_method = RENDER_METHOD_ALPHA_SOFTWARE;
              if (!strcmp(argv[i], "x11"))
                render_method = RENDER_METHOD_BASIC_HARDWARE;
              if (!strcmp(argv[i], "hard"))
                render_method = RENDER_METHOD_3D_HARDWARE;
           }
        }
      /* init X */
      e_display_init(NULL);
      /* setup handlers for system signals */
      e_ev_signal_init();
      /* setup the event filter */
      e_event_filter_init();
      /* setup the X event internals */
      e_ev_x_init();

      /* program does its data setup here */
      setup();
      /* and now loop forever handling events */
      e_event_loop();
   }
```

Example 4:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
double get_time (void);
void setup(void);

/* callbacks for evas handling */
/* timeout called every now and again for animation */
static void e_timeout(int val, void *data);
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);

/* globals */
Evas_Object o_cube;
Evas_Object o_bg;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;

/* callbacks */
static void
e_timeout(int val, void *data)
{
   int i;
   double v;
   static double start = 0.0;

   if (start == 0.0) start = get_time();
   v = (get_time() - start) / 10;
   evas_set_color(evas, o_cube,
                  ((int)((sin(v    ) + 1.0) * 127.5)) & 0xff,
                  ((int)((sin(v * 2) + 1.0) * 127.5)) & 0xff,
                  ((int)((sin(v / 2) + 1.0) * 127.5)) & 0xff,
                  ((int)((sin(v / 8) + 1.0) * 127.5)) & 0xff);
   e_add_event_timer("e_timeout()", 0.10, e_timeout, val + 1, NULL);
}

static void
e_idle(void *data)
{
   evas_render(evas);
}

static void
e_window_expose(Eevent * ev)
{
   Ev_Window_Expose      *e;

   e = (Ev_Window_Expose *)ev->event;
   evas_update_rect(evas, e->x, e->y, e->w, e->h);
}
```

```
/* utils */
double
get_time(void)
{
   struct timeval      timev;

   gettimeofday(&timev, NULL);
   return (double)timev.tv_sec + (((double)timev.tv_usec) / 1000000);
}

/* meat */
void
setup(void)
{
   Window win, ewin;
   int i, w, h;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);

   /* create a 400x300 evas rendering in software - conveience function
that */
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
                       MAX_EVAS_COLORS, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                       FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE | XEV_BUTTON | XEV_MOUSE_MOVE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_image_from_file(evas, "bg.png");
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);
   evas_show(evas, o_bg);

   o_cube = evas_add_image_from_file(evas, "cube.png");
   evas_get_image_size(evas, o_cube, &w, &h);
   evas_move(evas, o_cube, (400 - (double)w) / 2.0, (400 - (double)h) /
2.0);
   evas_show(evas, o_cube);

/*
   evas_set_color(evas, o_cube, 255, 0, 0, 255);
   evas_set_color(evas, o_cube, 255, 255, 0, 255);
   evas_set_color(evas, o_cube, 0, 0, 255, 255);
   evas_set_color(evas, o_cube, 255, 255, 255, 128);
   evas_set_color(evas, o_cube, 128, 128, 128, 255);
 */
/*
   evas_set_color(evas, o_cube, 128, 0, 0, 128);
 */
}

int
main(int argc, char **argv)
{
   /* command line parsing */
     {
```

```
        int i;

        for (i = 1; i < argc; i++)
          {
             if (!strcmp(argv[i], "soft"))
               render_method = RENDER_METHOD_ALPHA_SOFTWARE;
             if (!strcmp(argv[i], "x11"))
               render_method = RENDER_METHOD_BASIC_HARDWARE;
             if (!strcmp(argv[i], "hard"))
               render_method = RENDER_METHOD_3D_HARDWARE;
          }
      }
    /* init X */
    e_display_init(NULL);
    /* setup handlers for system signals */
    e_ev_signal_init();
    /* setup the event filter */
    e_event_filter_init();
    /* setup the X event internals */
    e_ev_x_init();

    /* program does its data setup here */
    setup();
    /* and now loop forever handling events */
    e_timeout(0, NULL);
    e_event_loop();
}
```

Example 5:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
double get_time (void);
void setup(void);

/* callbacks for evas handling */
/* timeout called every now and again for animation */
static void e_timeout(int val, void *data);
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);

/* globals */
Evas_Object o_cube[8];
Evas_Object o_bg;
Evas_Object o_clip;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;

/* callbacks */
static void
e_timeout(int val, void *data)
{
   int i;
   double v, w, h;
   static double start = 0.0;

   if (start == 0.0) start = get_time();
   v = (get_time() - start) / 10;
   for (i = 0; i < 8; i++)
     {
      evas_get_geometry(evas, o_cube[i], NULL, NULL, &w, &h);
      evas_move(evas, o_cube[i],
                200 - (w / 2) + (sin((v * 31) + ((double)i * 3.141592654 /
4)) * 150)
                ,
                200 - (h / 2) + (cos((v * 18) + ((double)i * 3.141592654 /
4)) * 130)
                );
     }
   e_add_event_timer("e_timeout()", 0.01, e_timeout, val + 1, NULL);
}

static void
e_idle(void *data)
{
   evas_render(evas);
}

static void
e_window_expose(Eevent * ev)
```

```
{
   Ev_Window_Expose      *e;

   e = (Ev_Window_Expose *)ev->event;
   evas_update_rect(evas, e->x, e->y, e->w, e->h);
}

/* utils */
double
get_time(void)
{
   struct timeval      timev;

   gettimeofday(&timev, NULL);
   return (double)timev.tv_sec + (((double)timev.tv_usec) / 1000000);
}

/* meat */
void
setup(void)
{
   Window win, ewin;
   int i, w, h;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);

   /* create a 400x300 evas rendering in software - conveience function
that */
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
                       MAX_EVAS_COLORS, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                       FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE | XEV_BUTTON | XEV_MOUSE_MOVE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_image_from_file(evas, "bg.png");
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);
   evas_show(evas, o_bg);

   o_clip = evas_add_rectangle(evas);
   evas_move(evas, o_clip, 20.0, 20.0);
   evas_resize(evas, o_clip, 180.0, 180.0);
   evas_set_color(evas, o_clip, 255, 255, 255, 255);
   evas_show(evas, o_clip);

   for (i = 0; i < 8; i++)
     {
      o_cube[i] = evas_add_image_from_file(evas, "cube.png");
      evas_set_clip(evas, o_cube[i], o_clip);
      evas_get_image_size(evas, o_cube[i], &w, &h);
      evas_set_color(evas, o_cube[i], rand() & 0xff, rand() & 0xff, rand()
& 0xff, 0xff);
      evas_resize(evas, o_cube[i], (double)(w / 2), (double)(h / 2));
      evas_set_image_fill(evas, o_cube[i], 0, 0, (double)(w / 2),
(double)(h / 2));
```

```
         evas_move(evas, o_cube[i], (400 - (double)(w / 2)) / 2.0, (400 -
(double)(h / 2)) / 2.0);
         evas_show(evas, o_cube[i]);
       }
}

int
main(int argc, char **argv)
{
   /* command line parsing */
     {
      int i;

      for (i = 1; i < argc; i++)
        {
           if (!strcmp(argv[i], "soft"))
             render_method = RENDER_METHOD_ALPHA_SOFTWARE;
           if (!strcmp(argv[i], "x11"))
             render_method = RENDER_METHOD_BASIC_HARDWARE;
           if (!strcmp(argv[i], "hard"))
             render_method = RENDER_METHOD_3D_HARDWARE;
        }
     }
   /* init X */
   e_display_init(NULL);
   /* setup handlers for system signals */
   e_ev_signal_init();
   /* setup the event filter */
   e_event_filter_init();
   /* setup the X event internals */
   e_ev_x_init();

   /* program does its data setup here */
   setup();
   /* and now loop forever handling events */
   e_timeout(0, NULL);
   e_event_loop();
}
```

Demo Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <unistd.h>
#include <Evas.h>
#include <Ecore.h>

#define MAX_EVAS_COLORS (216)
#define MAX_FONT_CACHE (512 * 1024)
#define MAX_IMAGE_CACHE (1 * (1024 * 1024))
#define FONT_DIRECTORY "./"
#define RENDER_ENGINE RENDER_METHOD_ALPHA_SOFTWARE
/* #define RENDER_ENGINE RENDER_METHOD_BASIC_HARDWARE */
/* #define RENDER_ENGINE RENDER_METHOD_3D_HARDWARE */

/* general functions */
double get_time (void);
void setup(void);

/* callbacks for evas handling */
/* timeout called every now and again for animation */
static void e_timeout(int v, void *data);
/* when the event queue goes idle call this */
static void e_idle(void *data);
/* when the window gets exposed call this */
static void e_window_expose(Eevent * ev);
/* when the mouse moves in the window call this */
static void e_mouse_move(Eevent * ev);
/* when a mouse button goes down in the window call this */
static void e_mouse_down(Eevent * ev);
/* when a mouse button is released in the window call this */
static void e_mouse_up(Eevent * ev);

/* globals */
Evas_Object o_bg, o_logo, o_logo_sh, o_text;
Evas_Object o_b1, o_b1_sh;
Evas_Object o_b2, o_b2_sh;
Evas_Object o_b3, o_b3_sh;
Evas evas;
Evas_Render_Method render_method = RENDER_ENGINE;
int max_colors = MAX_EVAS_COLORS;
double start = 0.0;
double end = 10.0;
Window main_win;

/* callbacks */
static void
e_timeout(int v, void *data)
{
   int i;
   double val;
   double x, y, z, r;
   int mouse_x, mouse_y;
   int w, h;

   mouse_x = 0;
   mouse_y = 0;

   if (start == 0.0) start = get_time();
   val = get_time() - start;

   r = (end - val) / end;
   evas_set_color(evas, o_text, 0, 0, 0, 255 - (255 * r));

   if (val > end)
     val = 0.0;
```

```
        else
          val *= ((end - val) * (end - val)) / (end * end);

        val += 3.0;

        r = 48;
        z = ((2 + sin(val * 6 + (3.14159 * 0))) / 3) * 64;
        x = (r + 32) + (cos(val * 4 + (3.14159 * 0)) * r) - (z / 2);
        y = (r + 32) + (sin(val * 6 + (3.14159 * 0)) * r) - (z / 2);
        evas_resize(evas, o_b1, z, z);
        evas_set_image_fill(evas, o_b1, 0, 0, z, z);
        evas_move(evas, o_b1, x, y);
        evas_resize(evas, o_b1_sh, z, z);
        evas_set_image_fill(evas, o_b1_sh, 0, 0, z, z);
        evas_move(evas, o_b1_sh,
                x - ((mouse_x - (x + (z / 2))) / 16) + (z / 2),
                y - ((mouse_y - (y + (z / 2))) / 16) + (z / 2));
        z = ((2 + sin(val * 6 + (3.14159 * 0.66))) / 3) * 64;
        x = (r + 32) + (cos(val * 4 + (3.14159 * 0.66)) * r) - (z / 2);
        y = (r + 32) + (sin(val * 6 + (3.14159 * 0.66)) * r) - (z / 2);
        evas_resize(evas, o_b2, z, z);
        evas_set_image_fill(evas, o_b2, 0, 0, z, z);
        evas_move(evas, o_b2, x, y);
        evas_resize(evas, o_b2_sh, z, z);
        evas_set_image_fill(evas, o_b2_sh, 0, 0, z, z);
        evas_move(evas, o_b2_sh,
                x - ((mouse_x - (x + (z / 2))) / 16) + (z / 2),
                y - ((mouse_y - (y + (z / 2))) / 16) + (z / 2));
        z = ((2 + sin(val * 6 + (3.14159 * 1.33))) / 3) * 64;
        x = (r + 32) + (cos(val * 4 + (3.14159 * 1.33)) * r) - (z / 2);
        y = (r + 32) + (sin(val * 6 + (3.14159 * 1.33)) * r) - (z / 2);
        evas_resize(evas, o_b3, z, z);
        evas_set_image_fill(evas, o_b3, 0, 0, z, z);
        evas_move(evas, o_b3, x, y);
        evas_resize(evas, o_b3_sh, z, z);
        evas_set_image_fill(evas, o_b3_sh, 0, 0, z, z);
        evas_move(evas, o_b3_sh,
                x - ((mouse_x - (x + (z / 2))) / 16) + (z / 2),
                y - ((mouse_y - (y + (z / 2))) / 16) + (z / 2));

        e_add_event_timer("e_timeout()", 0.01, e_timeout, val + 1, NULL);
    }

    static void
    e_idle(void *data)
    {
        evas_render(evas);
    }

    static void
    e_window_expose(Eevent * ev)
    {
        Ev_Window_Expose      *e;

        e = (Ev_Window_Expose *)ev->event;
        evas_update_rect(evas, e->x, e->y, e->w, e->h);
    }

    static void
    e_mouse_move(Eevent * ev)
    {
        Ev_Mouse_Move      *e;

        e = (Ev_Mouse_Move *)ev->event;
        evas_event_move(evas, e->x, e->y);
    }

    static void
    e_mouse_down(Eevent * ev)
    {
        Ev_Mouse_Down      *e;
```

```
   e = (Ev_Mouse_Down *)ev->event;
   evas_event_button_down(evas, e->x, e->y, e->button);
}

static void
e_mouse_up(Eevent * ev)
{
   Ev_Mouse_Up      *e;

   e = (Ev_Mouse_Up *)ev->event;
   evas_event_button_up(evas, e->x, e->y, e->button);
}

static void
e_window_configure(Eevent * ev)
{
   Ev_Window_Configure *e;

   e = (Ev_Window_Configure *)ev->event;
   if (e->win == main_win)
     {
      e_window_resize(evas_get_window(evas), e->w, e->h);
      evas_set_output_size(evas, e->w, e->h);
     }
}
/* callbacks evas will call for events within the evas */

static void
mouse_down(void *_data, Evas _e, Evas_Object _o, int _b, int _x, int _y)
{
   double x, y;

   evas_get_geometry(_e, _o, &x, &y, NULL, NULL);
   evas_move(_e, _o, x + 16.0, y + 16.0);
}

static void
mouse_up (void *_data, Evas _e, Evas_Object _o, int _b, int _x, int _y)
{
   double x, y;

   evas_get_geometry(_e, _o, &x, &y, NULL, NULL);
   evas_move(_e, _o, x - 16.0, y - 16.0);
   start = 0.0;
}

static void
mouse_move (void *_data, Evas _e, Evas_Object _o, int _b, int _x, int _y)
{
}

static void
mouse_in (void *_data, Evas _e, Evas_Object _o, int _b, int _x, int _y)
{
}

static void
mouse_out (void *_data, Evas _e, Evas_Object _o, int _b, int _x, int _y)
{
}

/* utils */
double
get_time(void)
{
   struct timeval      timev;

   gettimeofday(&timev, NULL);
   return (double)timev.tv_sec + (((double)timev.tv_usec) / 1000000);
}
```

```
/* meat */
void
setup(void)
{
   Window win, ewin;
   int i;

   /* setup callbacks for events */
   e_event_filter_handler_add(EV_WINDOW_EXPOSE,
e_window_expose);
   e_event_filter_handler_add(EV_MOUSE_MOVE,                e_mouse_move);
   e_event_filter_handler_add(EV_MOUSE_DOWN,                e_mouse_down);
   e_event_filter_handler_add(EV_MOUSE_UP,                  e_mouse_up);
   e_event_filter_handler_add(EV_WINDOW_CONFIGURE,
e_window_configure);
   /* handler for when the event queue goes idle */
   e_event_filter_idle_handler_add(e_idle, NULL);
   /* create a 400x300 toplevel window */
   win = e_window_new(0, 0, 0, 400, 400);
   e_window_set_events(win, XEV_CONFIGURE);
   main_win = win;

   /* create a 400x300 evas rendering in software - conveience function
that */
   /* also creates the window for us in the right colormap & visual */
   evas = evas_new_all(e_display_get(), win, 0, 0, 400, 400,
render_method,
                       max_colors, MAX_FONT_CACHE, MAX_IMAGE_CACHE,
                       FONT_DIRECTORY);
   /* get the window ID for the evas created for us */
   ewin = evas_get_window(evas);

   /* show the evas window */
   e_window_show(ewin);
   /* set the events this window accepts */
   e_window_set_events(ewin, XEV_EXPOSE | XEV_BUTTON | XEV_MOUSE_MOVE);
   /* show the toplevel */
   e_window_show(win);

   /* now... create objects in the evas */
   o_bg = evas_add_image_from_file(evas, "background.png");
   evas_move(evas, o_bg, 0, 0);
   evas_resize(evas, o_bg, 400, 400);

   o_logo_sh = evas_add_image_from_file(evas, "logo_shadow.png");
   o_logo = evas_add_image_from_file(evas, "logo.png");

   o_b1_sh = evas_add_image_from_file(evas, "bubble_shadow.png");
   o_b2_sh = evas_add_image_from_file(evas, "bubble_shadow.png");
   o_b3_sh = evas_add_image_from_file(evas, "bubble_shadow.png");

   o_b1 = evas_add_image_from_file(evas, "bubble.png");
   o_b2 = evas_add_image_from_file(evas, "bubble.png");
   o_b3 = evas_add_image_from_file(evas, "bubble.png");

   o_text = evas_add_text(evas, "notepad", 20, ". . .  the one you
love.");
   evas_set_color(evas, o_text, 0, 0, 0, 160);

     {
      int w, h;

      evas_get_image_size(evas, o_logo, &w, &h);
      evas_move(evas, o_logo, (400 - w) / 2, 400 - h - 16);
      evas_move(evas, o_logo_sh, ((400 - w) / 2) + 16, 400 - h - 16 + 16);
      w = evas_get_text_width(evas, o_text);
      h = evas_get_text_height(evas, o_text);
      evas_move(evas, o_text, (400 - w) / 2, 400 - h - 4);
     }
```

```
   evas_show(evas, o_bg);
   evas_show(evas, o_logo_sh);
   evas_show(evas, o_logo);
   evas_show(evas, o_b1_sh);
   evas_show(evas, o_b2_sh);
   evas_show(evas, o_b3_sh);
   evas_show(evas, o_b1);
   evas_show(evas, o_b2);
   evas_show(evas, o_b3);
   evas_show(evas, o_text);

   evas_callback_add(evas, o_logo, CALLBACK_MOUSE_DOWN, mouse_down, NULL);
   evas_callback_add(evas, o_logo, CALLBACK_MOUSE_UP, mouse_up, NULL);
   evas_callback_add(evas, o_logo, CALLBACK_MOUSE_MOVE, mouse_move, NULL);
   evas_callback_add(evas, o_logo, CALLBACK_MOUSE_IN, mouse_in, NULL);
   evas_callback_add(evas, o_logo, CALLBACK_MOUSE_OUT, mouse_out, NULL);
}

int
main(int argc, char **argv)
{
   /* command line parsing */
     {
      int i;

      for (i = 1; i < argc; i++)
        {
           if (!strcmp(argv[i], "soft"))
             render_method = RENDER_METHOD_ALPHA_SOFTWARE;
           else if (!strcmp(argv[i], "x11"))
             render_method = RENDER_METHOD_BASIC_HARDWARE;
           else if (!strcmp(argv[i], "hard"))
             render_method = RENDER_METHOD_3D_HARDWARE;
           else
             max_colors = atoi(argv[i]);
        }
     }
   /* init X */
   e_display_init(NULL);
   /* setup handlers for system signals */
   e_ev_signal_init();
   /* setup the event filter */
   e_event_filter_init();
   /* setup the X event internals */
   e_ev_x_init();

   /* program does its data setup here */
   setup();
   /* call the animator once to start it up */
   e_timeout(0, NULL);
   /* and now loop forever handling events */
   e_event_loop();
}
```