Introduction to the Enlightenment foundation libraries.

An overview of EFL

Kostis Kapelonis

Introduction to the Enlightenment foundation libraries.: An overview of EFL

Kostis Kapelonis

Abstract

The target audience of this document are UNIX programmers who are interested in the Enlightenment Foundation Libraries (EFL). You must already know C programming. You will not however learn *how* to program using the EFL. Instead, you will learn *why* you should program with the EFL. If you ever wanted to evaluate the EFL but did not see any advantages over previous graphic libraries then this document is for you!

Table of Contents

1. Introduction 1
    A little History 1
    Related documentation 2
    Obtaining the EFL libraries 2
2. The EFL structure 4
    Organization of the Libraries 4
    Brief description of each EFL library 5
3. Understanding the Evas Canvas 9
    What "Image-based" Means 9
    What "State-aware" Means 12
    Available Programming Facilities 16
4. Understanding the Edje Layout Engine 19
    Edje as a Layout engine 19
    Edje as Animation/Effects Library 24
    Edje as an IDL 30
    Edje as Logic and Appearance separator 32
    Edje as a Theming Framework 36
    Using Edje to preview your GUIs 39
    Choosing Edje over Evas 41
5. Understanding the Ecore Infrastructure Library 44
    Using Ecore in your programs 44
    Programming Facilities 44
    Configuration with Ecore 45
    The Event Loop 47
6. End matter 49
    The rest of the EFL libraries 49
    Get involved 49
    Resources 50

List of Figures

2.1. The EFL Stack viewed by the Marketing Department. 4
2.2. The EFL Stack viewed in the programmer way. 4
3.1. Workflow in your favourite Canvas widget 10
3.2. Workflow in Evas. 12
4.1. Space wasted after resizing. 20
4.2. Relative positioning in Edje. 20
4.3. An example of relative positioning in Edje. 21
4.4. Adapting automatically to a new window size. 23
4.5. Computer generated animation using keys. 27
4.6. Building the UI statically in the application. 32
4.7. Loading the UI dynamically in the application. 34
4.8. Separation of GUI and code in Edje. 35
4.9. Changing skin via a theme. 38
4.10. Changing completely the appearance via a theme. 39
4.11. Preview an Edje Theme. 39
4.12. Live signal testing in Edje. 40
4.13. Swallowing an object programmed in C into a UI described in Edje. 42

List of Tables

3.1. Evas versus the competition ... 9
3.2. Evas primitives ... 16
4.1. Top Level Edje blocks ... 31
4.2. The Edje Appearance/Logic separation ... 34
4.3. Edje themes (viewed by users) ... 37
4.4. Edje vs Evas ... 42
5.1. Ecore configuration primitives ... 46
5.2. Threads vs Event based programming ... 48

List of Examples

3.1. A simple Canvas in object-oriented style (even for C) 13
3.2. What happens in reality 14
3.3. Moving an object 15
3.4. Moving an object with a state-aware Canvas 16
4.1. Relative coordinates in Edje 22
4.2. A simple animation (in C) 25
4.3. The same simple animation in Edje 27
4.4. The same simple animation in Edje (fixed size version) 29
4.5. Sending a signal from Edje to C code. 36

Chapter 1. Introduction

In any field, one of the primary reasons why good ideas may not catch up is inertia. People are happy with what they have. They have spent time and gained expertise so that they can fit available tools to their needs, or even create workarounds when faced with different problems. When new tools become available people are reluctant to use them. In the case when the new tools are introducing new concepts and ideas, their adoption is even slower. People need some time to discover the new abilities before actually using them.

Any UNIX programmer who goes beyond the command line interface soon discovers that he/she needs separate graphic libraries called toolkits. These toolkits can make the X Window System look and act in different ways. Since X does not favor any particular toolkit, several solutions exist today. Of course some toolkits are more mature than others. Some are commercial and some free. Some are for C and others for C++. You can find many resources across the Internet which compare X11 toolkits. Most toolkits share however one common factor. They are all affected by industry concepts (rapid prototyping). They are built so that programmers can quickly build a (mostly) useful but not always pretty application. Animations are non-existent. Themes are an afterthought (sometimes even a hack). Performance tuning comes second to feature abundance. Each toolkit advertises several widgets which apply to specific cases but are useless to others.

A handful of graphics toolkits are extremely popular. They are mature, well developed, and well documented. Some even enjoy the backing of major companies. Two of them are actually the foundation for modern UNIX desktop environments. Using these toolkits is straightforward. The resulting applications look and function in a way familiar to the majority of computer users. But what if you want something more? What if you want animations, transparent components, and themeable applications? Your options are fairly limited. Rather than using OpenGL for pure 2D or even resorting to low level Xlib functions, jump right into EFL and discover what you can do with this set of powerful libraries.

We believe that EFL is not just another toolkit. You can use EFL as a toolkit (the EWL/ETK libraries), but you will probably miss all the fun. EFL has some new concepts, which if applied correctly will make your applications do things you could only imagine so far.

A little History

Back in 1999, the Enlightenment window manager (version 16 then) was offering people with eye-candy much ahead of its time while still retaining usability. Several graphical effects (windows sliding in, shading animation, the ripples and waves) were quite revolutionary for a window manager. Theme support was extensive. While most applications supporting themes actually mean the colours and skins of the interface, the Enlightenment interface could be almost everything. Themes could be completely different considering most aspects. Several themes were so complex that would eventually make Enlightenment resemble different systems (Mac OS Aqua, BeOS).

Enlightenment was also at some point integrated with Gnome. While Gnome has progressed since then and now comes with its own window manager (Metacity or the older Sawfish), the fact is that Enlightenment was installed on many desktops at the time. Version 16 is still the latest stable version of Enlightenment. It is updated even today with patches. This means that today Enlightenment v16 is a very stable window manager. Since its resource requirements are roughly the same, what used to seem a heavyweight window manager is very light indeed with modern personal computers.

So why don't we have a new version of Enlightenment today? And why has the development version (v17) not yet been released after so many years? The fact is that the Enlightenment developers, having gathered experience from v16, do not want v17 to be a simple window manager. Instead version 17 of Enlightenment is a "desktop shell". The infrastructure of this desktop shell comes in the form of separate graphic libraries called the Enlightenment Foundation Libraries (EFL for short). Enlightenment is now based on the EFL libraries, which can also be used for normal applications apart from the window man-

ager.

While this concept is fairly interesting, successful marketing of it is harder. Most people who visit the Enlightenment pages think that this is the start of a new desktop environment (like KDE and GNOME) and that EFL is simply another X11 toolkit (like QT and GTK+). This document will try to address these common misconceptions and attempt to give you a glimpse of what EFL is capable of.

Related documentation

The document you are reading attempts to explain some basic concepts of EFL rather that act as a programming guide. If you are looking for technical information regarding EFL, take a look at the guides written by the developers themselves. Available guides are:

- The EFL Cookbook
- The EWL Book Reference
- The Edje Book

The EFL Cookbook contains common recipes for most EFL components. A good read in order to discover some of the endless possibilities of EFL.

The EWL Book provides background on the common widgets provided by EFL. If you need to use EFL as a GUI toolkit, this is the proper document.

The Edje book contains almost everything you need to develop themeable EFL application which use the Edje Layout engine. Make sure that you know a bit about Evas (the EFL canvas) before reading this document.

If, on the other hand, you need to dive straight into coding, consult the reference documents for core EFL libraries. These are generated automatically by Doxygen, so they are the closest thing to code documentation apart from code itself. Available reference guides are:

- The Ecore Reference
- The Edje Reference
- The Eet Reference
- The Evas Reference
- The EWL Reference

All of the documents mentioned above can be downloaded from the *Documentation* page at the Enlightenment website.

Note

You can always look directly into the source code. Most of it is well commented. The syntax is clear and straightforward. If you become familiar with functionality not already documented, consider sending a patch with needed documentation.

Obtaining the EFL libraries

Once you want to actually start using the EFL libraries you have the following options:

- Getting binary packages for your distribution.
- Downloading snapshots of selected libraries.
- Downloading directly from CVS.
- Running/installing the E-live CD.

Downloading binary packages (e.g. rpm and deb) for your distribution is the preferred way. Since Enlightenment and libraries are not released yet officially, this is not possible. You have to track unofficial packages prepared by enthusiastic users. While this might be the safest way to obtain EFL, it actually means that you depend on the packager. If packages are not updated regularly, you are on your own.

The recommended way, if you are not adventurous enough, is downloading the source packages from freedesktop.org. These are updated regularly by EFL developers once they think that the libraries are in a stable state safe for a snapshot. Note that not all available EFL libraries are posted there. You miss experimental stuff, and, of course, you can only update when new versions are added. Still, the sources compile cleanly, and you can always get more bleeding edge stuff from CVS if you want it.

If you would prefer the latest and greatest, you can download the source directly from the CVS servers. This is is the preferred way if you want to submit bugs or patches and generally get involved in development. CVS builds are known to break. If you download your snapshot while a developer is undertaking major changes, it is pretty common that your copy is non-functional. CVS also suffers from problems outside of the control of core developers. Availability of the CVS server and load has lately been a problem (hopefully solved).

Finally, you can download the E-live CD. This is a Live CD based on Enlightenment and Debian. It is a great way to see EFL in action without touching the system or to show off to your friends. If you are really impressed you can install the CD to your PC and get a Debian GNU/Linux system with the EFL libraries pre-installed.

Chapter 2. The EFL structure

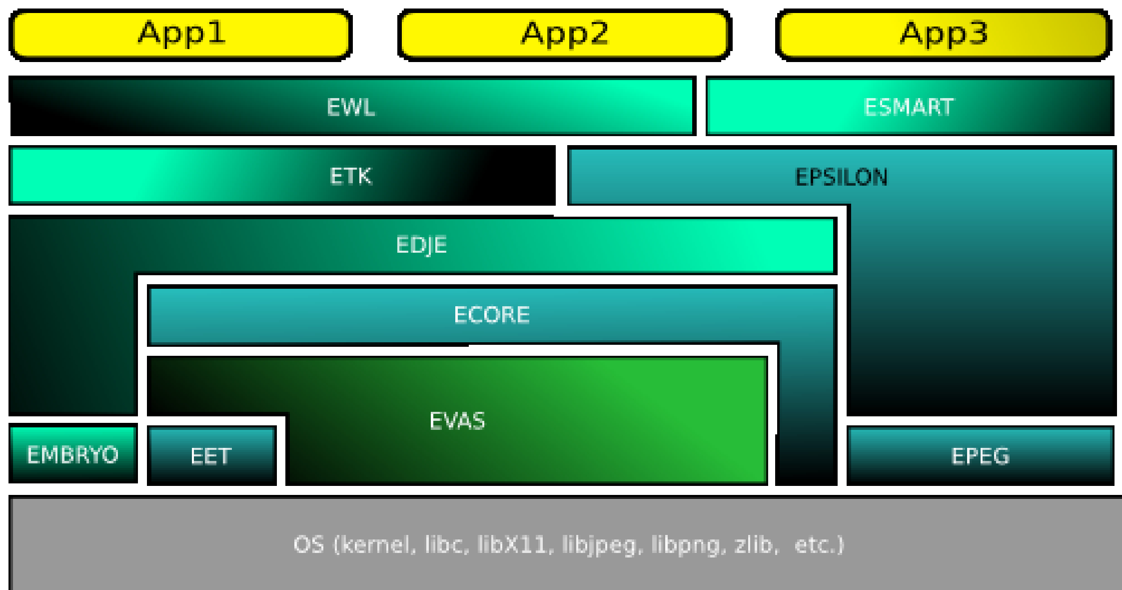
While most EFL libraries are under constant development, some of them are clearly more mature than others. Several libraries are considered stable, but that should not mean that they haven't got rough edges. Some libraries have even been declared obsolete (!!!) before the 1.0 release of the EFL. Others are constantly changing so it is hard to either use them in an application or even to understand their scope.

Since this is a concepts document, we will focus only on the foundation libraries that have existed long enough to be useful. If you really get interested in EFL development, joining the developer mailing lists will keep you informed on all the latest ideas and code that at any given moment is central to EFL core developers.

Organization of the Libraries

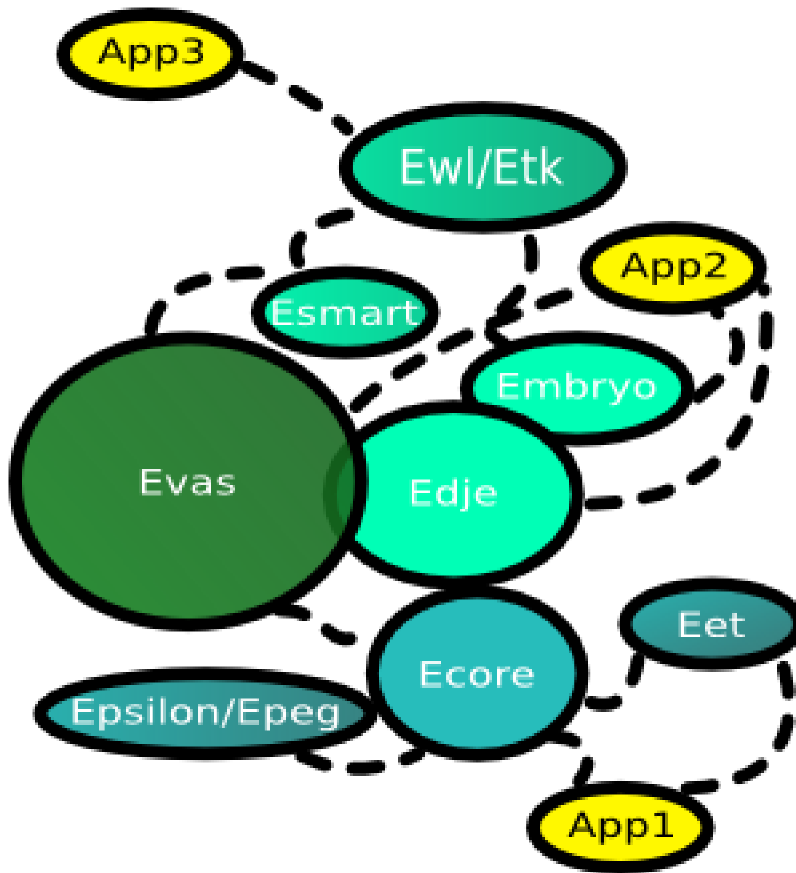
Several solutions which come in multiple libraries are often advertised in the familiar box stack picture. Low-level libraries are shown at the bottom, while high-level libraries sit on top of them. This implies that the higher levels strictly depend on the low level ones. It also implies that most programmers would always program on higher levels and never actually deal with the infrastructure. This approach may be useful for marketing purposes, but sometimes it confuses developers rather than help them understand. See figure below.

Figure 2.1. The EFL Stack viewed by the Marketing Department.



This in not true with EFL. There is one core library which can certainly used by itself, the Evas canvas, and most other EFL libraries add significant value to Evas (Edje, Embryo, Epsilon, etc.). At the same time, low-level libraries can be used by themselves even in non-GUI programs (Ecore, Eet). Thus, to better visualize the EFL, we choose a network structure with Evas and Edje in the center and everything else around. The next picture shows this.

Figure 2.2. The EFL Stack viewed in the programmer way.



This makes clear that the libraries do *not* form a hierarchy tree that makes every level dependent on the lowest levels. You are not supposed to understand any dependencies from the lines drawn among the libraries. The only thing this denotes is that several libraries can be used by themselves. Application 1 even represents a non-GUI program. You can program console application using only the EFL infrastructure.

From the size of each module you can also understand the importance of a library and whether it is central to the EFL or not. The Evas canvas is clearly the heart of EFL, while the Edje layout engine is the infrastructure that empowers the graphical abilities of Evas. This will also give you an idea on the order that we discuss each library in the next chapters of this document.

Brief description of each EFL library

Before we explain what each library offers in more detail and what new ideas it brings to development, here is a small list with a brief description of each one. Notice that this is not a final list. The 1.0 release of EFL may (and probably will) contain a few more.

The Evas canvas

Evas is a powerful canvas. While most developers have used the Canvas widget of their favourite X11 toolkit, Evas is taking the canvas idea a huge step forward. It actually keeps track of what is rendered on screen. Unlike the usual canvas widget, Evas knows that you created a rectangle at a specific point on screen and it knows how to move it or resize it without having to rely on external data structures. Evas objects are essentially "draw and forget" objects. No need for the programmer to keep state. No need for the

programmer to deal with redrawing and repainting. All this is gone. Evas is also highly optimized and can even run in embedded systems with constrained memory and low power processors. Apart from the X11 back-end, it also runs on OpenGL, Xrender, and frame-buffer devices.

The Edje Layout Engine

The Edje Layout Engine is a breakthrough (in the humble opinion of the unbiased author of this document!). It can not be compared with another preexisting library (at least in the open source field) because it is simply unique. Edje allows you to describe a graphical user interface without writing a single line of C code. You describe what your interface looks like and *how* you want it to appear and act on screen. Edje then takes the tough job to translate all this into low level code and bundle everything in a single theme file. This theme file is distributed along with the executable of your application which contains the functionality of the application. Your application is essentially split into two parts. A graphical part which knows nothing about C code and the functionality which knows nothing about GUI. This split makes the code development and theme creation really easy compared with current solutions. It also allows you (the developer) to easily change the functionality back-end with out touching the GUI and vice-versa. You can ship to your users a new theme for the same app, or upgrade just the functionality without changing the GUI. Users can also create new themes without getting in contact with C code.

The Ecore glue library

Ecore is a library which provides all the low-level stuff of your application. You can compare it with Glib from GTK+. It contains data structures, IPC mechanisms, easy networking, and generally all helper functions you would expect so that you don't have to reinvent the wheel. It glues together major EFL libraries so that it gives you a start point for your first applications. It even abstracts some functionality of common UNIX and X capabilities. Not to mention that you can always use Ecore for an application that isn't even graphical. Before starting to implement your own data structure/X11 wrapper function/networking code/e.t.c., check the Ecore documentation. Maybe it is already there with a clean, simple EFL API.

The Eet Storage library

Most themes delivered via the Internet nowadays are rather simple in their structure. Although they appear to be single files, in reality they are a bunch of files compressed using the usual compression methods (e.g. zip or .tar.gz). The application that uses the themes uncompresses them (usually upon installation and not on the fly) and a directory containing all the images of the theme is added to the application directory. Eet takes a completely different approach. The theme file is a single file which contains compressed images and the description of the interface too. This information is read by the Edje layout library to create the interface of your application on the fly. The theme is never uncompressed unless it is modified by a themer (and then compressed again). Moving around themes means moving around files. You can also use Eet as a general storage format for images, text, or even arbitrary data chunks. Eet is best known for storing themes but nothings prevents you from using it for all your storage needs.

The Embryo Scripting Language

People always want more power. Look what happened with HTML. Although HTML pages are static, soon enough JavaScript appeared

along with Flash. Pages became dynamic. Then the Ajax buzzword came along and we have to believe that all web pages should look and act like local applications. While all this functionality is an overkill for the hypertext-based Internet, having the ability to script your interface is not always a bad idea. This is what the Embryo scripting brings to Edje. Edje themes can now contain small Embryo scripts which allow for more interesting interfaces. Although Embryo is not C code, some pure themers may think that the layout engine is contaminated with programming code, therefore Embryo scripts are optional. If you are a themer, you can use Edje as you think it should be used, while if you are more on the programming side you will soon discover the value of Embryo scripts for your interfaces. It's all about choice.

The Epeg Thumbnailing Library

Epeg is a small thumbnailing library specially optimized for Jpeg files. It seems that nowadays most people keep their huge picture collections in the Jpeg format. The fact that it was embraced by camera manufacturers means that more and more Jpeg images will be stored on digital media and hard disks. To search all these images, thumbnails are used, so a library exactly for this purpose will serve our needs well. Epeg can be used standalone in your programs or again via the EFL infrastructure (Esmart, Ecore, Epsilon)

The Epsilon Thumbnailing Library

Epsilon is a more general thumbnailing library. It supports PNG, JPEG, and all other formats offered my imlib2 (see below) and Evas. It is closer to EFL than Epeg, and it can even take advantage of Epeg for Jpeg images if it is present on the system. Epsilon essentially abstracts the thumbnailing process for images, while Epeg is clearly targeted to the Jpeg format. Again, this is all about choice.

The Esmart Utility Library

With all the power that Edje gives to the programmer, one would think that she should be happy. Esmart is a helper library that gives some additional capabilities to Edje without blowing it up to a full toolkit (yet). Esmart gives the programmer the ability to have containers for the GUI (think the vbox and hbox of GTK+). It allows for transparent applications which seem to be cool these days (think Eterm). We mean applications which adapt their background to the X root window. Transparency itself is included already in Evas for the canvas and its objects. Esmart also wraps around epsilon for thumbnailing support and finally adds some more toolkit facilities (text entries, file dialogs, and draggable objects). Esmart is required for the Entrance application (a replacement of xdm/gdm/kdm based on EFL). You can use it for your application easily because all Esmart capabilities (as well as Ecore facilities and Evas engines) exist on separate library files.

The EWL Widget Set

Finally, if you want to use EFL as a toolkit similar to GTK+ and QT and just want to have a widget set available for your applications, you can use EWL. It offers the familiar toolkit facilities based on the EFL libraries and ties everything up under widgets which have different purposes. If you have programmed on any X11 toolkit, then programming for EWL should be a well-known experience.

The Imlib2 Image Library

Imlib2 is not strictly an EFL library but since some of the EFL applications and libraries use it, we include it here for reference. Imlib2 is a highly optimized and ultra fast imaging library which predates the EFL. You are not expected to use it directly unless you are a hard-core graphics programmer who knows the internals of graphics

systems and image formats. Imlib2 ties seamlessly to Evas and allows you to really tinker with images at a low level. Notice also that Imlib2 is *not* the second version of Imlib, but rather a completely different library (Imlib1 is by the way obsolete). Several applications outside the EFL/Enlightenment world use Imlib2 (such as the feh image viewer).

As already mentioned, several other EFL libraries are under development (such as Emotion, a video library based on xine), but for the time being we will discuss only the core ones in the next chapters.

Chapter 3. Understanding the Evas Canvas

By now you should know that Evas is a canvas. But what makes it different from the other solutions that already exist? Each mature toolkit provides its own Canvas widget which if used correctly can really give the edge to your application. Let's see a quick summary of Evas features versus the traditional solutions.

Table 3.1. Evas versus the competition

Your Canvas widget	The Evas Canvas
An extra widget of the toolkit	The base of the toolkit (EWL/ETK)
Part of an Application	*THE* application itself
Shape/Vector based	Image based
State un-aware	State aware
X11 back-end	X11, framebuffer, Xrender, OpenGL back-end
Desktop usage	Desktop/Embedded usage

While most users will perceive Evas-based applications as impressive applications with low requirements for computer resources, for developers things are different. The way you program Evas is a refreshing change from what you have seen before. Most of the points presented on the table are fairly easy to understand.

The last point regarding the speed of Evas is the easiest to recognize once you start programming with EFL. Graphic routines are highly optimized and carefully profiled (even for the software back-end) so that Evas really feels fast for users. If you ever hated Canvas widgets for their lack of speed, problematic scrolling, and low-quality resizing, Evas might be just what you are looking for.

We devote two separate sections on the key features that we think make Evas truly evolutionary.

What "Image-based" Means

Since computers became capable of displaying graphics, pixel-based images comprise the bulk of visual output presented on user screens. Images stored in pixels take a lot of space to store, but are rather easy and fast to load and display. The pixel information is stored on a big 2D array (in the simplest case) and displaying graphics means simply copying the colour values from file to screen (not exactly but you get the idea).

Apart from the problem with storage space, pixel-based graphics suffer also from fixed quality and resolution dependence. If the user has a big screen, graphics will look small, while in the opposite case graphics will look really cramped since the screen estate is constrained. This problem is especially evident with application icons which can not be easily resized and keep their sharpness. Current solutions are forced to include different sizes of icons for 3 or 4 possible resolutions in order to accommodate for different uses. This not only increases disk space requirements but also adds complexity to loading and locating the appropriate icon file. Well known is the fact that wallpapers for common desktop environments come in different sizes. But what happens if you use 1600x1200 resolution and your favourite wallpaper comes in 1024x768?

The software platforms are slowly but steadily moving to vector graphics. Vector graphics are destined

to replace pixel-based graphics in many fields. Files which contain vector-based graphics do not store any pixel information at all. They rather contain a *mathematical representation* of an image. When the file is displayed the computer plots the mathematical function in real time on screen, and the user sees an image which is actually the result of rendered graphics.

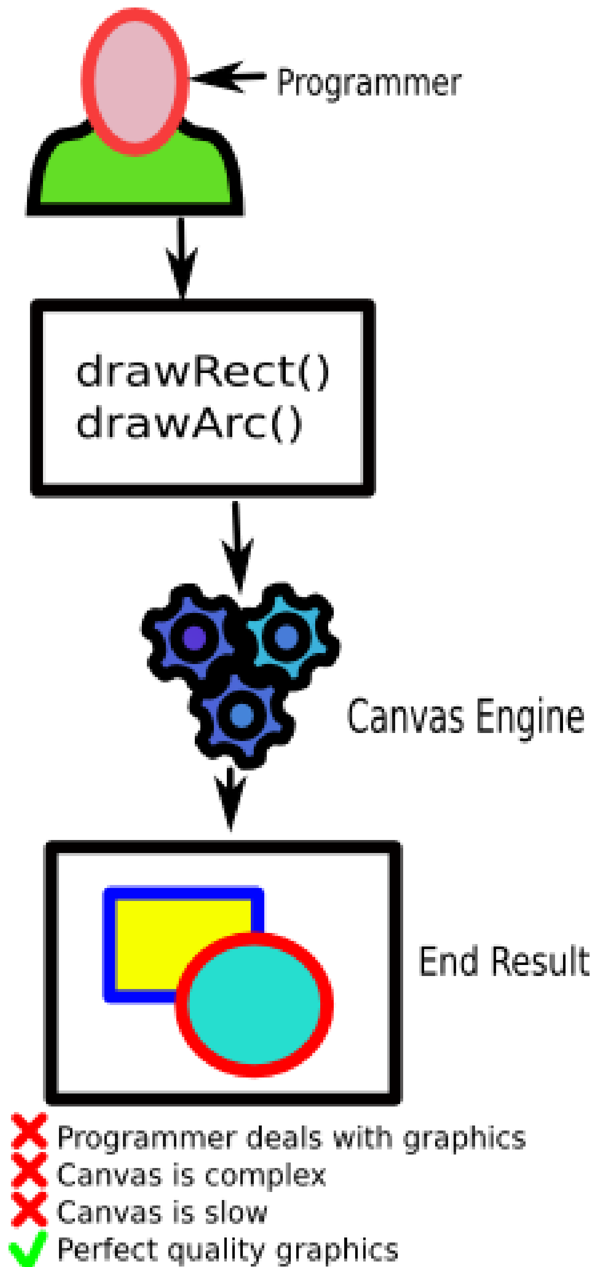
The big advantage of vector graphics is the fact that they are truly resolution independent. Because the graphical representation is computed in real-time for each display, graphics will look the same (and always sharp) no matter the canvas they are rendered on. If you have 1600x1200 display your image will be rendered with these coordinates while if you have 1024x768 graphics will again be recomputed to adapt to the new resolution. Both open-source desktop environments support vector-based icons (stored in the svg format) which showcase that icon resizing can result in big, sharp icons with no loss of quality.

On the downside of course, this real-time rendering puts a load on the computer processor, since it has to calculate the result of the function each time the image is displayed. A pixel-based image will be loaded in a fraction of time that the same image will be displayed if stored in vector format. The vector-based file will be significantly smaller though, than the pixel-based one since only the math behind the image is stored rather than all the pixels.

Vector-based graphics will never be used for digital photos of course since these need pixels for information. Computer interfaces on the other size are a natural candidate for vector graphics. Being resolution independent is a big plus for applications. And since processors get faster each year, the rendering overhead will slowly disappear.

This is is the reason why many Canvas widgets are vector based. They implement a complex mathematical layout engine (can be even postscript) which allows the programmer to display complex graphics with anti-aliasing, sharp edges and curved lines all in a rectangular window which is resolution agnostic. This might be perfect for mathematical and plotting applications (or even for custom widgets) but not for complete interfaces. This kind of canvas widgets can also not be used on embedded systems which have limited processing capacity and vector graphics will be inherently slow. The next figure outlines the shortcomings.

Figure 3.1. Workflow in your favourite Canvas widget

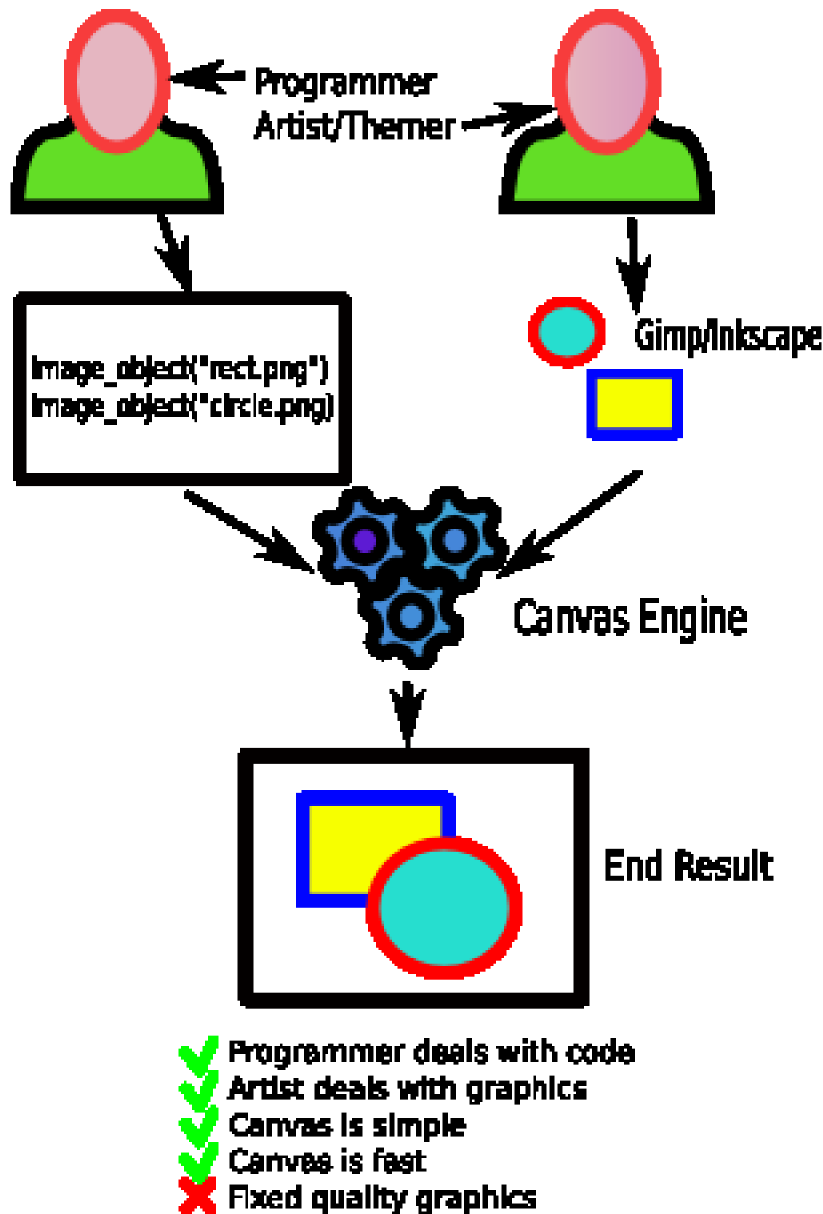


Evas takes a completely different approach. While most Canvas widgets can also display pixel-based images as an extra feature, Evas is actually targeted on them. The Evas primitive list is rather short. Text, rectangles, lines, polygons, and images are the most basic primitives. You won't find arcs, circles, arrows, bezier curves, or any other traditional Canvas methods if you search the Evas documentation. To create an Evas based interface you store all elements in normal pixel-based files (e.g. in PNG format) and have Evas load and resize them when the interface is activated. The interface elements themselves can be processed with Gimp as normal images or even be created as vectors in Inkscape and later converted to pixels and distributed as a theme.

This means that graphics are no longer a job of the programmer but become a job of the Gimp artist/ Guru where they belong. Evas does not suffer from complex layout mathematical languages and therefore can be used on embedded systems too. Evas graphics are not resolution independent, but since Evas has rather sophisticated resizing algorithms (as imlib2) for most cases graphics will maintain their qual-

ity. The following figure shows the main idea.

Figure 3.2. Workflow in Evas.



So will Evas boost your applications? If your canvas is going to plot complex mathematic functions for a scientific application then Evas won't help you much. But if you intend to create a beautiful application with interface elements scrolling around the screen, windows sliding in and out, and crazy animations composed of many picture frames, then Evas will certainly come to rescue you.

What "State-aware" Means

Canvas widgets for pre-rendered graphics are not anything new of course. Evas is a well-thought solution heavily optimized for images and is also field tested in the Enlightenment window manager and the

rest of EFL applications which are under development. If you were reading the previous chapters, shaking your head and thinking that there is nothing extraordinary about Evas, then you might want to reconsider after reading this chapter.

Evas does its own bookkeeping. It knows the whole state of the Canvas at any given moment. While most other Canvas widgets are one layer above the X abstractions (exposing windows and stuff), Evas is rather smart and sophisticated. It frees the programmer from keeping his/her own data-structures for objects shown on screen, and allows one to concentrate on the actual interaction of these objects rather than their memory management.

We could spend a lot of paragraphs talking about this subject, but you could easily understand if we show some pseudocode. We noted on the beginning that this is not a technical document, so don't expect to see details of Evas API. The code presented here is only describing concepts and not showing how Evas works.

Joe Programmer is attempting to create his first canvas application. Joe is a seasoned C programmer but also knows C++ and maybe Java. He knows that a C API can be centered around "objects" (think GTK+) and he has some minimal exposure to X11 toolkits. Before starting actual development, Joe thinks how the API of the Canvas should look. Since he does not believe that a Canvas is something extraordinary he concludes that adding some more statements to the program `main` should do the trick. Here is what a Canvas should look like in his mind.

Example 3.1. A simple Canvas in object-oriented style (even for C)

```
int main()
{
        Canvas *a_canvas;
        Rectangle *rect;
        Image *image;

        a_canvas=create_new_canvas(800,600);

        rect=create_new_rectangle();
        draw_rect_on_canvas(a_canvas,rect);
        move_rect(rect,10,50);
        resize_rect(rect, 100,100);

        image=create_new_from_file("button.png");
        draw_image_on_canvas(a_canvas,image);
        move_image(image, 150,200);

        show_rect(rect);
        show_image(image);
        show_canvas(canvas);

        //Continue with rest of the program
        [...]
}
```

Looks rather logical. With that in mind Joe starts to look at the API of the Canvas widget he uses only to find that things are a bit more complex. He learns that the canvas has a `paint` function which is run when the canvas is redrawn (either at short time periods or after an on-screen event). In order to make the Canvas draw something he must collect all objects that need to be drawn and pass them to the canvas. He also has to override/extend/call/redefine (pick your favourite) the paint function of the canvas. After some coding he finally gets the Canvas to show stuff with the following code listing:

Example 3.2. What happens in reality

```
//Include files which contain implementation
//of linked lists or other data structures.
[...]

int main()
{
        Canvas *a_canvas;
        List *objects_to_be_drawn;

        a_canvas=create_new_canvas(800,600);

        //Setup a callback. VERY IMPORTANT
        set_paint_function_of_canvas(a_canvas,my_repaint);

        rect=create_new_rectangle();
        set_coords_rect(rect,10,50);
        set_size_rect(rect, 100,100);
        //Append the rectangle to objects drawn by Canvas
        add_rect(objects_to_be_drawn,rect);

        image=create_new_from_file("button.png");
        set_coord_image(150,200);
        //Append the image to objects drawn by Canvas
        add_image(objects_to_be_drawn,image);

        show_canvas(canvas);
        repaint(canvas); //Here the my_repaint function is called.

        //Continue with rest of the program
        [...]
}

//Function which smells X11 internals (what happens after an expose event)
void my_repaint(canvas *where)
{
        canvas_object *current;
        while(objects_to_be_drawn !=EMPTY)
        {
                current=get_next_object(objects_to_be_drawn);
                draw_object(where,current); //Finally each object is drawn.
        }
}
```

While the actual code is not a lot longer than what he had in mind, Joe makes a couple of observations. First, he is forced to keep by himself all canvas objects in a separate data structure. The List `objects_to_be_drawn` holds everything that should appear on screen. Joe never actually draws directly on the Canvas. He just adds objects to this List and informs the Canvas (via a separate callback function) that it should process it for the actual drawing. The second important observation is the fact that whichever function would like to access the canvas cannot directly access the `Canvas` object. Instead it needs both the `Canvas` object *and* the `objects_to_be_drawn` List which represents its state. To actually change anything, this List should be changed first and then the Canvas should be ordered to redraw again.

Later on, Joe needs to do something simple. He wants the user to be able to click an object on the canvas

and have that object move on the top left of the screen (at x=10 and y=10). He searches again and again the Canvas API only to discover that he has to write a lot of code himself. The Canvas API is full of functions that draw *new* objects on screen but almost no functions that deal with *existing* objects on screen. After some effort and not sure about himself anymore he reaches the following code:

Example 3.3. Moving an object

```
void user_clicked_on_canvas(canvas *where)
{
        canvas_object *clicked_by_the_user;
        int x;
        int y;

        x=get_pointer_x(where);
        y=get_pointer_y(where);

        clicked_by_the_user=search_what_object_is_there(x,y,objects_to_be_drawn);

        //Let's say it is a rectangle for simplicity
        set_coords_rect(clicked_by_the_user,10,10);
        repaint(canvas); //Here the canvas is repainted;
}

canvas_object *search_what_object_is_there(x,y,objects_to_be_drawn)
{

        canvas_object *current;
        while(objects_to_be_drawn!=NULL)
        {
                current=get_next_object(objects_to_be_drawn);
                if(is_x_in_object(x,current)==1);
                {
                        if(is_y_in_object(y,current)==1);
                        return current; //Found it
                }
        }
}
```

Joe is a bit disappointed now. He sees that he has to keep the state of canvas in custom data structures, and manually control when the canvas is updated. He also sees that once objects are drawn it is hard to find them again. Complex functions which look into the custom structures must be implemented if the Canvas is to do anything useful rather than act as a simple scenery view. Joe starts to think that he is spending more code on Canvas management instead of the application code. He finally realizes the sad truth. The canvas is just a 2D array of pixels. Once an object is drawn on the Canvas it loses all character. Its pixels are the same as the rest of the Canvas which are not drawn. This Canvas is essentially dumb and "state unaware".

Joe starts to ponder why the situation is like this. If he had a "state-aware" canvas, coding would be much easier. Such a Canvas would know all of its objects. It would be able to access and move them on demand. It would not bother the programmer with Canvas management. The programmer would concentrate on the functionality and not on the Canvas widget. With such a canvas the previous example becomes trivial.

Example 3.4. Moving an object with a state-aware Canvas

```
void user_clicked_on_canvas(canvas *where)
{
        canvas_object *clicked_by_the_user;
        int x;
        int y;

        x=get_pointer_x(where);
        y=get_pointer_y(where);

        //The next  function is already implemented by the canvas itself.
        clicked_by_the_user=check_which_object_is_in(x,y,canvas);

        //Let's say it is a rectangle for simplicity
        move_rect(clicked_by_the_user,10,10);
}
```

You can see where this is heading. This ideal "state-aware" Canvas is actually Evas. If you find yourself often in Joe's shoes, include Evas in your programs and never look back again. A whole lot of infrastructure code is already written for you. The Canvas is a self-contained object which is smart enough to know what is going on under the surface.

Evas also is the same Canvas Joe was thinking as ideal (see the first code-listing example). Evas implements a Canvas "as it should be done" in the first place. Everything about Evas is logical. The author of the document is clearly biased, but he believes that if you invest time in Evas, the reward will be great. For big programs where Canvas management overruns the actual application, Evas will show the difference and allow you to build your application as you were imagining it from the beginning, rather than searching all the time the Canvas API for things you need.

Evas is here now. Power your applications with it!

Available Programming Facilities

Now that you are familiar with the Evas development model, it is time to delve a bit deeper. This section outlines the general capabilities of Evas. It helps you understand what is available and what is not. Since Evas is not released officially yet, the complete details are a moving target.

The following table lists the Evas primitives at the disposal of the programmer. These are offered in the form of "Evas Objects" in the code.

Table 3.2. Evas primitives

Primitive	Short description
Rect	Typical rectangle; useful for clipping too
Image	The most common Evas object
Text	Text object; one line only
TextBlock	Multi-line text object
Line	Line object with start and end points
Polygon	General-purpose shape

Primitive	Short description
Gradient	Linear, radial, rectangular, etc.

Many times you may have found yourself writing several lines of code in order to accomplish something, only to find out that your library already implements it. Avoiding this is easy. Reading beforehand what a library offers, rather than just diving into the code as soon as possible is the obvious solution. For example, Evas implements some basic data structures, so writing an Evas-only application does not mean that you will have to implement them yourself.

Clipping functions
: Clipping means showing only the parts of an object that are contained (constrained) by a second object. Currently Evas supports clipping only *to* Rectangles.

Object data functions
: You can attach and retrieve any type of data to an Evas primitive. Storage is performed on a key-data pair fashion.

Different display layers
: Objects can have different Z-ordering. The layer number is not fixed. Transparency works as expected. That is, creating a transparent object on the upper layer will make objects on the lower level layer visible. Evas makes transparent interfaces really easy, and more importantly fast.

Pointer functions
: Evas allows you to detect if the (mouse) pointer is within the Canvas, and what are its coordinates. You can even get the state of mouse buttons (if they are pressed or not). You can also translate from screen to Canvas coordinates and vice-versa.

Object finder functions
: Since Evas knows the state of its objects as already explained, it gives you access to a lot of functions for retrieving currently drawn objects. You can search object by name, coordinates, and pointer position. You can even get a list of objects which are contained in a specific rectangular area of the Canvas.

Simple Data Structures
: For your own convenience Evas comes with some useful Data Structures. Currently a hash-table implementation and a linked-list one are offered. Even if you use your own or exploit external third-party libraries, you should at least learn the linked List API. This is because all Evas functions which deal with collections of objects, strings, etc. actually return an Evas List.

Callback binding functions
: Since Evas is a Canvas, it also offers the usual callback binding facilities. These are employed in order to respond to several external events including resizing and moving the canvas, using the mouse, or typing on the keyboard.

Evas Smart objects
: Evas allows you to create you own *Evas objects* and have them implement the same API used for built-in objects too. Naturally, this is a great way to group several different primitives to form more complex objects without the need of custom glue and wrapper code. Not to be confused with the EFL Smart library.

Evas Engine functions
: Most programmers will use the X11 back-end in general. Evas runs on other devices too. The framebuffer is another good candidate found on embedded systems (which do not run an X server).

Text effects and style
: The Text object of Evas is not as limited as you might think. Evas includes some built-in styles, which will make text appear on screen

a little more appealing. Shadows, outlines, and glows including combinations among them are already there.

As always, you should look into the code if you want to know everything that Evas offers. Not all features are documented yet and some of course are more experimental than others.

Chapter 4. Understanding the Edje Layout Engine

The Edje Library is a complex and intimidating beast at first sight. It is hard to understand the full potential of this library, let alone describe what it does in simple terms. In a nutshell Edje is what really showcases what Evas can do, and allows the programmer to create live, animated and playful user interfaces. It is perfectly normal for you, to read this chapter and think that you still do not grasp all the capabilities of Edje.

Edje has multiple roles, and depending on the application it might seem that Edje is a complex library with multiple purposes, that can confuse the programmer who comes in contact with EFL for the first time. As time passes (and the EFL libraries are finally released!) more and more Edje applications will appear that will show what Edje can really do.

So depending on who you ask and what application you are using, Edje can be:

- A layout engine.
- An animation/effects graphic library.
- An Interface Description Language.
- A logic/appearance separation library.
- A theming framework.
- A GUI previewer (think glade).
- An abstraction over Evas

All the above are different aspects of Edje. You can view Edje as many things, but the fact is that it remains the same library no matter how you look at it. It just happens that Edje is clearly something more than a trivial EFL library. We will explore each role of Edje in turn.

Edje as a Layout engine

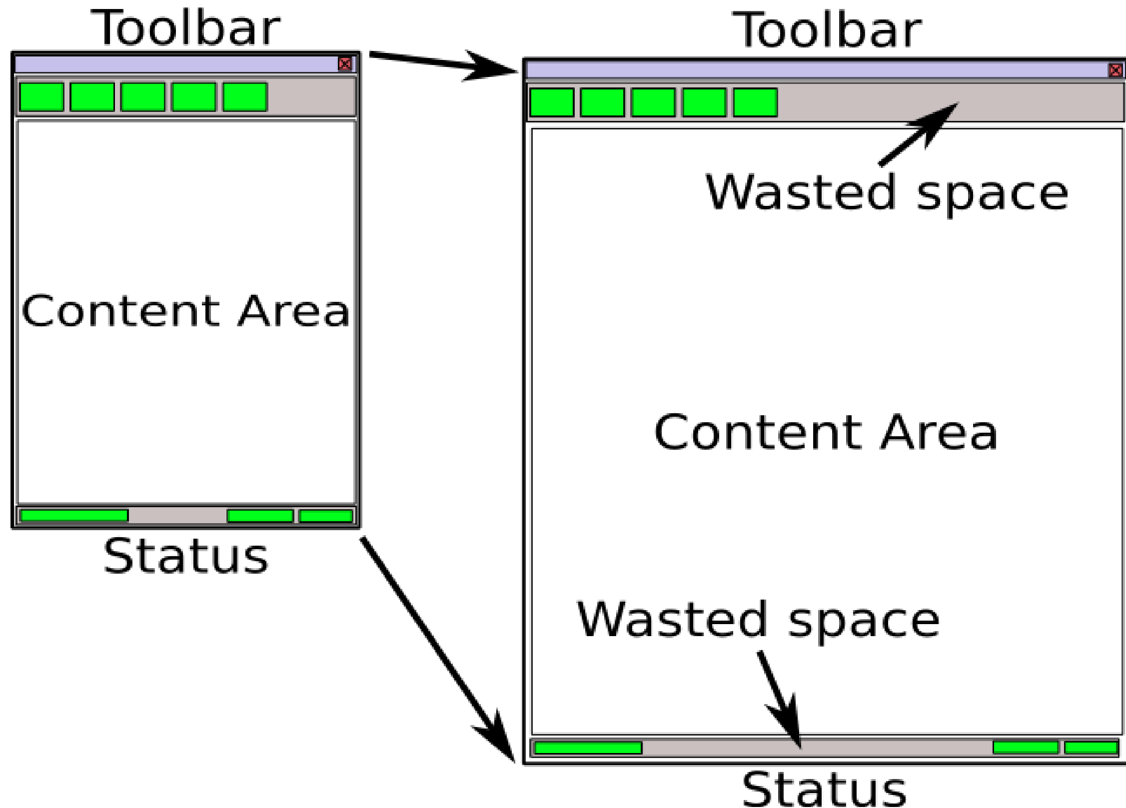
Most applications are built so that when started, they automatically request a specific size from the window manager. This size is usually the one that the application programmer has chosen so that the application fully utilizes the given screen space. Most of these times the user never actually changes the size of an application window. If the application is the sort of utility which will be used for a brief period of time the user might not even move the application to a better position leaving it to where the window manager has placed it. But what happens if the user resizes the window or even fully maximizes it?

If you are really unlucky the application won't even resize since it is a dialog. How many times have you tried to grab a window from its borders only to realize that you cannot change its size? If you are equally unlucky the application window will resize but the window contents will stay the same. This happens because the elite programmer of the application never bothered with application resizing in the first place. Go bug him/her about this.

In most cases however, the application window will resize smartly. Assuming that the application is document based, its content area along with the status bar and the toolbar will be notified of the new coordinates and change their layouts to match the new size. Notice anything strange here? Of course not since 99% percent of applications resize this way.

The problem stems from the fact that all GUI toolkits are coordinate-based. The buttons and text have fixed sizes. So although the document area will actually increase the toolbar and status bar will not. Empty space will be wasted in several parts of the window. See the following figure:

Figure 4.1. Space wasted after resizing.



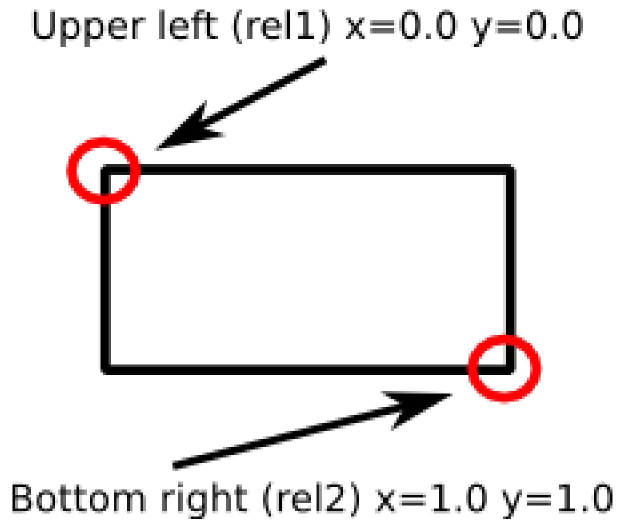
Toolkit programmers know this and have provided application developers will several facilities in order to control where this extra space goes. The buzzwords are containers/boxes/glue space/auto-fill/constraints e.t.c. Most developers either don't use these and just hardcode their user interface, or even if they use them they are never actually happy about the behavior of the application after resizing.

The truth is that all these methods are complex to use and understand. In any case the fact that not all elements of an application resize evenly when its window is maximized is problematic. Your latest application might look cool on your 19" LCD monitor with 1600x1200, but it looks ugly at your friends 17" LCD at 1024x768 and unusable at your aunt's 15" CRT and 800x600.

Edje allows you (if you want it of course) to give relative coordinates to *all* your interface elements. You can create a truly resolution independent application. You use Edje to *describe* the relative size and location of parts in your application. Each time the application window is resized all its elements will be also resized proportionally.

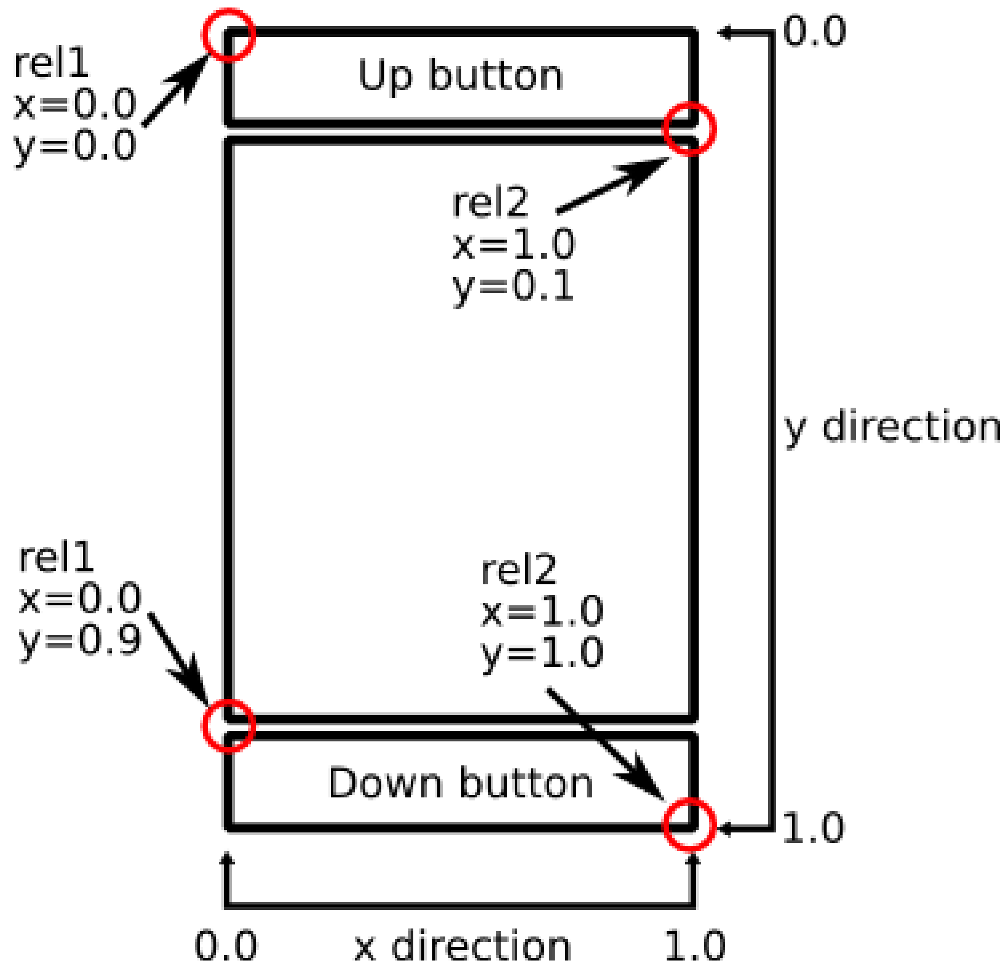
For each interface element you only need to describe its upper left and bottom right corner as it will appear in the application window. The figure below shows this. Let's say that your application is a "scroller" one which will show some content and will allow the user to go up and down by clicking arrows that reside on the sides of the window. You quickly draw two arrow images (one up and one down) in Inkscape and export them as .png images. Then you decide that you want each arrow to take 10% of bottom or top screen space.

Figure 4.2. Relative positioning in Edje.



To tell Edje where a *part* of your interface resides you describe the relative coordinates of the top-left and bottom-right corners of the area it occupies on screen. Coordinates are normalized using the window size of your application (that is why they are called relative). So top-left corner of the whole window has x=0.0 and y=0.0 while bottom-right corner is at x=1.0 and y=1.0. Negative and greater than 1.0 values are actually valid ones. These describe elements outside of the viewable window and are well suited for animations (explained later). Next figure shows the relative coordinates of our example according to this approach.

Figure 4.3. An example of relative positioning in Edje.



This visual representation of the window content would be translated in Edje with the following code.

Example 4.1. Relative coordinates in Edje

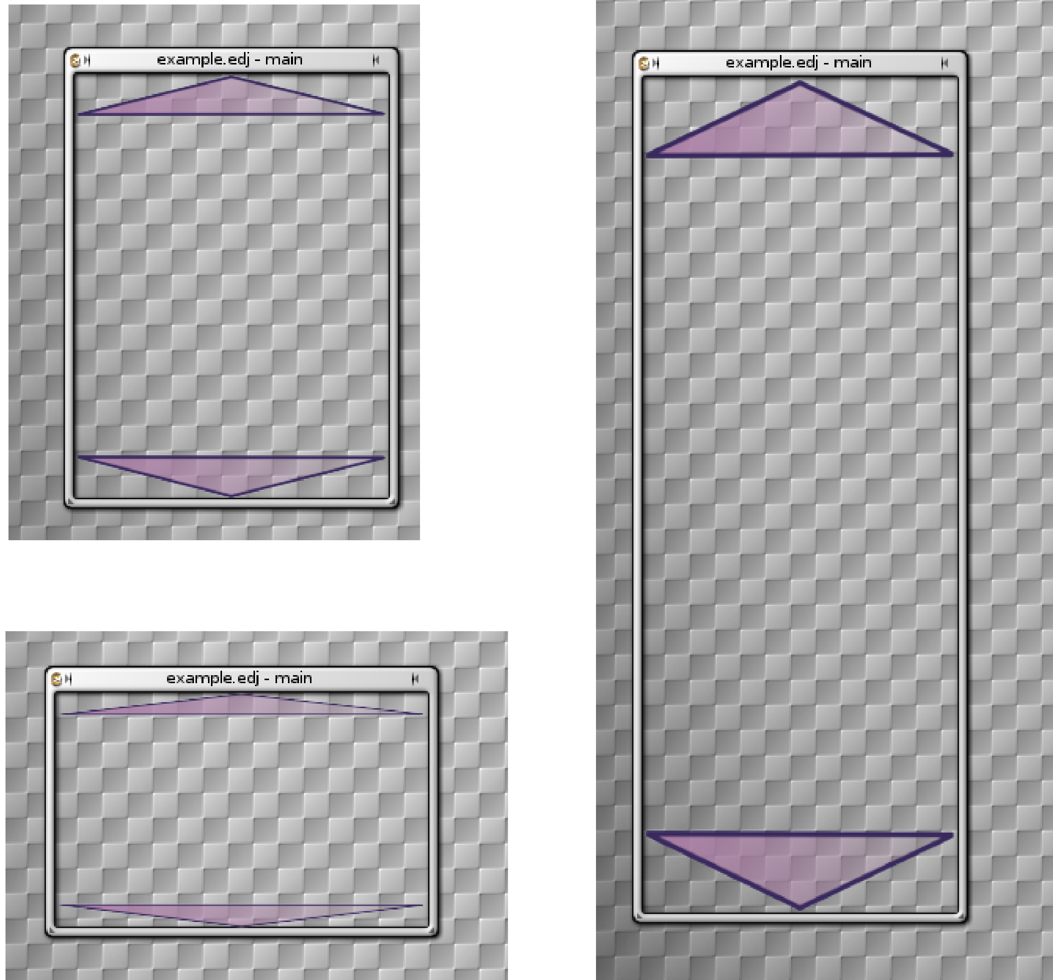
```
part
{
        name, "up_button";
        type, IMAGE;
        description
        {
                state, "default" 0.0;
                rel1
                {
                        relative, 0.0 0.0;
                        offset, 0 0;
                }
                rel2
                {
                        relative, 1.0 0.1;
                        offset, 0 0;
                }
                image
```

```
                {
                        normal, "up.png";
                }
        }
}
part
{
        name, "down_button";
        type, IMAGE;
        description
        {
                state, "default" 0.0;
                rel1
                {
                        relative, 0.0 0.9;
                        offset, 0 0;
                }
                rel2
                {
                        relative, 1.0 1.0;
                        offset, 0 0;
                }
                image
                {
                        normal, "down.png";
                }
        }
}
```

The key point here is that the `rel1` block corresponds to the top-left corner of something while the `rel2` is the bottom-right one. Take a moment to compare this code with the image presented before. You should fully understand how the text describes the interface and what do the float values (0.0 0.1 0.9 and 1.0) mean. Ignore the `offset` lines. The `name` and `type` keywords are self explaining.

So what have we accomplished so far? Without writing a single C function we have a fully dynamic interface. Check the figure below. No matter how the user resizes the window, the interface will automatically "adapt" to its new size. Window coordinates? Screen coordinates? World coordinates? translation between them? No code for them at all needs to be written by the programmer. It is already in Edje (or rather Evas).

Figure 4.4. Adapting automatically to a new window size.



Of course a lot of people will probably step up and shout that this should only work if our graphics were vector based. This would make scaling images a lossless procedure and would always produce perfect quality visuals for the interface. If you read the Evas chapter you will already know what to answer to them. Evas/Edje has really sophisticated image resizing algorithms, so in most cases (provided that your image resources are reasonably sized) the loss in quality will never be evident to the end-user. And unlike vector graphics the Edje implementation is really fast.

Edje as Animation/Effects Library

We have seen how Edje can be used to describe the location of interface parts in the application window. This is nice for static applications but EFL is all about motion. Edje allows you to describe animation for your user interface. And all this in a very natural way.

Let's compare the Edje way with the traditional one. This time we will examine the adventures of Mary Developer. Mary wants to create a simple animation for her latest application. A ball sprite which starts from the top-left corner of the window and goes all the way down to the bottom right. We choose this animation path because we assume that these are the positive directions of x and y axes of the application window. Mary wants the animation to last 5 seconds and present 20 frames per second for the transition. The canvas is a rectangle with x=300 and y=300.

Mary carefully studies the documentation of her graphic toolkit. She learns about timers and how to use

them (and maybe a little about threads). She spends some time calculating some timing results and finally she crafts the following code:

Example 4.2. A simple animation (in C)

```
//Include files which contain implementation
//of linked lists or other data structures.
[...]
//Include files which contain timers
[...]

int main()
{
        Canvas *a_canvas;
        List *objects_to_be_drawn;
        timer *animation_timer;

        //Canvas is 300x300
        a_canvas=create_new_canvas(300,300);

        //Do not forget the paint function!
        //Setup a callback. VERY IMPORTANT
        set_paint_function_of_canvas(a_canvas,my_repaint);

        //We assume that ball.png is 10x10 pixels
        image=create_new_from_file("ball.png");
        set_coord_image(0,0);
        //Append the image to objects drawn by Canvas
        add_image(objects_to_be_drawn,image);

        show_canvas(canvas);
        repaint(canvas); //Here the my_repaint function is called.

        //Setup a timer to create animation
        //Schedule the timer to run every 50ms (20 frames per second=1000ms)
        animation_timer=timer_create(animate,50,image);

        //Continue with rest of the program
        [...]
}

//Function which smells X-Windows internals (what happens after an expose event)
void my_repaint(canvas *where)
{
        canvas_object *current;
        while(objects_to_be_drawn !=EMPTY)
        {
                current=get_next_object(objects_to_be_drawn);
                draw_object(where,current); //Finally each object is drawn.
        }
}

//The animation function which is controlled by a timer
//Return 0 if the timer is finished or 1 if
//it is to be rescheduled again
int animate(void *data)
{
        canvas_object *ball;
        int x;
```

```
        int y;

        ball=(canvas_object *)data;
        //Again let's say we know it is an image for simplicity
        x=get_image_x(ball);
        y=get_image_y(ball);

        //Advance coordinates by 3 to each direction
        //3 is found by dividing the canvas size (300) by total
        //number of frames we want to show (5 seconds * 20 frames each =100)
        //Therefore 300/100 =3
        x=x+3;
        y=y+3;

        //Update the new coordinates of the object
        set_coords_rect(ball,x,y);
        //Make sure that canvas is updated too
        repaint(canvas);

        //We need a check here to detect if the ball has reached
        //the bottom right corner. If yes the timer should be stopped.
        //One could also count the number of frames shown so far.
        //There are more elegant ways to deal with this.
        //The fact remains that a check should exist at one form or another.
        if(x==300 || y==300) return 0;

        //The ball has a long way to go.
        //Reschedule the timer
        return 1;
}
```

That is a lot of code for a simple animation. No wonder why most of today's application seem so static and motionless. Animation libraries do exist but they are only used in specific application domains (usually games). With Edje this is no longer true. Animation comes to the desktop!

Apart from the large size of the code above there is also another important problem. If you have ever coded like this you should see it right away. Mary wants the animation to last 5 seconds with 20 frames per second so she has to make calculations to find the number of total frames shown and the progress of the ball sprite. These values are currently hardcoded (the 50ms timer delay, and 3 pixel movement respectively). Theses values are only correct in the case of a 300x300 canvas (also hardcored). But what happens if the user resizes the application window? These values should be recalculated. Mary has to write additional code which automatically changes these values so that the animation is smooth no matter the size of the window.

All this becomes too complicated for a simple application and also forces the programmer to deal with canvas management more that she should. There has to be a better way. Actually there is and can already be found in most advanced 3D (and maybe 2D) animation/cad/modelling programs. If you have ever used one of them (Blender and Synfig are such open source applications for 3D and 2D respectively) you should already be familiar with this. The concept is called key framing in almost all (3D) animation suites. We would like to digress a bit at this point and mention a little history. You can always skip ahead if you want.

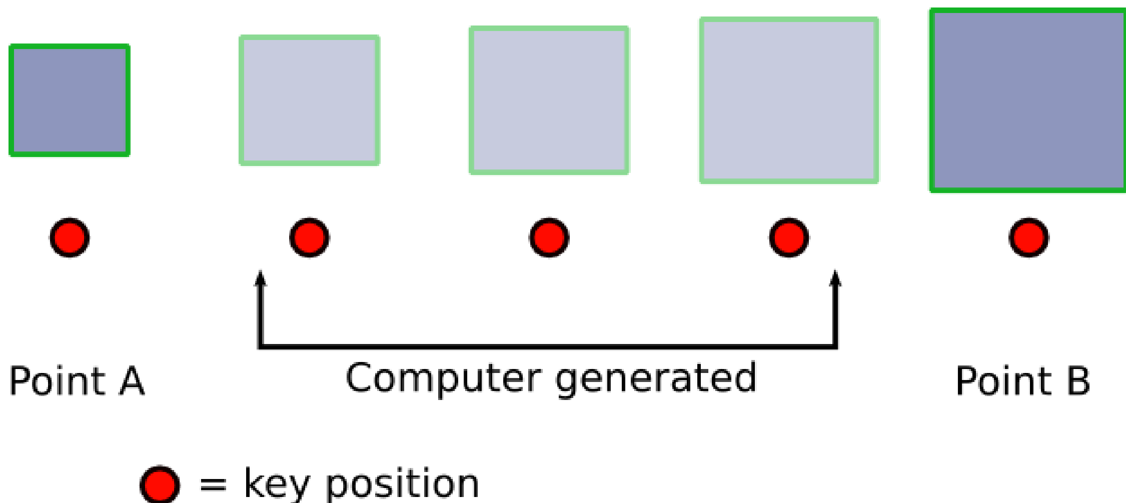
Before computing reached the masses, any kind of animation was a daunting task. The artist had to draw each frame separately. Having cinema quality (24 frames per second) animation meant that a single artist could produce only very short clips. More artists would be needed for longer films. People came up with two partial solutions. The first one was to lower the number of frames to 10 up to 15 (in best cases). This meant that the application was less smoother but less frames had to be drawn. This kind of animation can be even seen today in animation films destined to be shown on TV to young audiences. The second solution was to draw a small number of frames which had the backgrounds and focus the ac-

tual work on the parts of the screen that changed during animation (characters moving and talking). Of course this meant that full motion action was not an option. Japanese animation (a.k.a. Anime) has taken this concept to extremes featuring extremely detailed static backgrounds and blocky moving characters.

3D animation with the help of computers was a revolution. An artist could spend her time creating complex models of characters and places and then have the computer render the scene from different perspectives. Each frame was calculated from the computer. High quality animation became an option since the computer could render 30 or even 60 frames per second with no additional effort from the user. A single artist with a single workstation could create anything. In the case of professional studios with a team of artists and clusters of computers (render farms) high quality films would take over the movie and gaming industry in less than a decade. This happens because the user no longer defines frames but instead defines key positions and commands the computer to compute the animation between different keys.

The concept works elegantly both in 3D and 2D. The user defines a starting point A for an object and selects some characteristics of the object to be recorded. Location of the object at this point is stored by the computer but other parameters such as size, rotation, color, textures and even shape can be recorded too. Then a second point B is defined where the object has different values on the recorded parameters. Finally the computer jumps in, creating automatically the middle frames changing the values in a relative way depending on the distance between point A and B. The result is a smooth transition as seen in the next figure. In the case of different shapes this is a very efficient way to create morphing, a technique commonly used in commercials, games and movies.

Figure 4.5. Computer generated animation using keys.



Edje implements this concept in 2D and allows the programmer to specify several key position for objects displayed on the canvas. Since Evas is a stateful canvas, as already mentioned in the previous chapter, it can compute automatically the frames in between and create smooth transitions between several states of the canvas.

Joe Programmer sees how Mary Developer struggles with her simple animation. He has become an experienced programmer in EFL since the last chapter so he offers to help Mary and introduces her to EFL. He listens to her requirements (20 fps and 5 second animation) and after some coding he presents her the following listings:

Example 4.3. The same simple animation in Edje

The C code of the program:

```
int main()
{
        //Code that creates a canvas and an Edje object
        //like any other object (text, image, rectangle e.t.c)
        [...]

        //Specify frames per second to 20
        edje_frametime_set(1.0/20.0);

        //Continue with rest of the program
        [...]
}
```

Inside the Edje description of the interface: (by now you know it is not C)

```
part
{
        name, "ball";
        type, IMAGE;
        description
        {
                state, "default" 0.0;
                rel1 {
                        relative, 0.0 0.0;
                        offset, 0 0;
                }
                rel2
                {
                        relative, 0.05 0.05;
                        offset, 0 0;
                }
                image
                {
                        normal, "ball.png";
                }
        }
        description
        {
                state, "finish" 0.0;
                rel1 {
                        relative, 0.95 0.95;
                        offset, 0 0;
                }
                rel2
                {
                        relative, 1.0 1.0;
                        offset, 0 0;
                }
                image
                {
                        normal, "ball.png";
                }
        }
}
program
{
        name, "animate_ball";
        signal, "show";
        action, STATE_SET "finish" 0.0;
        transition, LINEAR 5.0;
        target, "ball";
```

```
}
```

Notice the key positions. Point A is the state called `default`. This is the name Edje uses for the starting point of an interface element. You cannot change this name. Point B is the state called `finish` which defines the ball sprite to be at the bottom right of the Canvas window. Animation is accomplished by the `program` block which states that we want a linear transition lasting 5 seconds for the ball element (defined by the `target` keyword) which changes the state to `finish` (defined by the `action` keyword). We want this animation to be launched when the program loads and the canvas is shown (defined by the `signal` keyword).

Also notice the lack of extra calculations. Mary's requirement of a 5 second transition with 20fps is directly transfered to code. The number of frames shown and the delay between them is automatically calculated by Edje. No need for the programmer to worry about low level stuff. Finally notice the lack of any hardcoded values for sizes. The canvas can be anything and the ball sprite will be 5% of the canvas size. This means that no matter the size of the application window, Edje will adjust sizes, delays and frames to satisfy the requirements (20fps for 5 seconds). Extra code is not needed for that. The end result is that the user can resize the application in any way she likes and the animation quality will be preserved.

Edje is all about power. We stressed in the previous section that you can describe all elements of your interface in a relative way *if you want*. If you don't want this, you can still revert to fixed size graphics. Let's assume that Mary likes Edje-based animations but she wants to keep the ball sprite to its original size (10x10) no matter what. That is, the canvas can be any size, the animation will continue to last 5 seconds at 20fps but the ball sprite will have a fixed size. This is where the `offset` keyword comes in. It allows you to describe sizes in pixels instead of percentages. Mary can rewrite the Edje description of the interface as:

Example 4.4. The same simple animation in Edje (fixed size version)

```
part
{
        name, "ball";
        type, IMAGE;
        description
        {
                state, "default" 0.0;
                rel1 {
                        relative, 0.0 0.0;
                        offset, 0 0;
                }
                rel2
                {
                        relative, 0.0 0.0;
                        offset, 10 10;
                }
                image
                {
                        normal, "ball.png";
                }
        }
        description
        {
                state, "finish" 0.0;
                rel1 {
```

```
                        relative, 1.0 1.0;
                        offset, -10 -10;
                }
                rel2
                {
                        relative, 1.0 1.0;
                        offset, 0 0;
                }
                image
                {
                        normal, "ball.png";
                }
        }
}
program
{
        name, "animate_ball";
        signal, "show";
        action, STATE_SET "finish" 0.0;
        transition, LINEAR 5.0;
        target, "ball";
}
```

Remember that positive values on the x direction go rightwards and on y direction downwards. You should understand what the number 0.0,1.0,+10,-10 do. The `offset` is a great way of course to create depressed buttons in your interfaces. They also allow you to fine-tune the size of your interface elements if you do not like the relative approach.

What you saw so far was a trivial example of Edje animation. Here the only variable we recorded between the two states of the ball sprite was the location. But Edje allows you to record many more including size, color, text, alpha value, image e.t.c. Also you can have more than two states. You can create complex animation with multiple states acting as intermediate stations between the beginning and end of an animation. Edje also simplifies the creation of animations created by a big number of successive images (e.g. rotating logos). You can do many things in Edje! We have barely scratched the surface.

Last but not least we should mention that apart from the `LINEAR` transition, Edje also includes `ACCELERATE`, `DECELERATE` and `SINUSOIDAL` transition effects.

Edje as an IDL

Almost all C code that we have shown so far is imaginary. It doesn't show any of the API the EFL provide. This is intentional of course. This way you can concentrate on the high level concepts instead of being puzzled with why and how the API works. But this does not apply to the Edje "code".

All 3 examples of Edje snippets already mentioned are actual Edje code. You cannot simply copy-paste them for your tests because some infrastructure blocks are missing. All code however is exactly as you would write it *inside* your programs. If you think that the listings are a bit abstract, its because Edje is abstract, not because we have removed some important code in any way.

With that in mind, you can see that with Edje you *describe* your graphical interface. Your code says *what* your interface looks like, and *what* it is doing once loaded but not *how* it does it. This part of complexity is handled by Edje. As you saw in the previous section you can forget about timers and manual moving and sizing of interface elements. Edje works for you by abstracting all this low level code. You can keep coding the application logic rather than the application canvas.

In this sense Edje works as an Interface Description Language (IDL). [1] The concept is not entirely new.

The Mozilla foundation uses XUL for describing the interface of Mozilla and Mozilla-based applications. Microsoft also introduced XAML shortly after XUL appeared on the scene. Both XUL and XAML are XML based (notice the X pattern). Edje is not XML of course.

The similarity ends there. Edje can do things that XUL and XAML were never meant to do. If this analogy helps you understand what Edje is all about, that is fine. But you will be mistaken if you decide that Edje is the EFL answer to XUL and/or XAML. The architectures also differ a lot in how the interface definition fits into the final program.

Edje code goes into a normal text file with the *.edc* extension. The name stands for Edge Collections. We mentioned that Edje code looks like C but it is not C. You can search the gory details of Edje syntax in the technical documents already released about Edje. We will not focus on the syntax of Edje files. We will deal however with what types of blocks (starting and ending with *{}*) can be included in an Edje file. These are the elements that you can *collect* in an Edje file (hence the name).

Table 4.1. Top Level Edje blocks

Block type	**Description**
Images	Image files that will be used in the interface
Fonts	Font definitions for text and textblock objects
Data items	Simple data entries in key value pairs
Text styles	Styles definitions for text and textblock objects
Color classes	Color definitions to be shared by objects
Collections	The description of the interface

From the name you can assume that the Collections block is the most important one. All Edje code that you have seen so far belongs to the collections block. The *parts* and *program* blocks already demonstrated are all child blocks inside the collection one. This is why we said before that some infrastructure is missing from the Edje code we have presented so far. All Edje code listings shown do not stand on their own, but must be included in a collection block (or to be more precise in a parts block, inside a groups block, inside the collections block). You can look up the source code or the Edje book to find the exact hierarchy of blocks inside the collections one. The other blocks (images, fonts, e.t.c) have a simpler (flat) hierarchy.

Your typical Edje experience will probably go like this:

When you start playing with Edje and want to discover its abilities you will mainly use rectangles. So your first *.edc* files will just contain the collections block. You will play a bit with animations and transitions and then decide that you want to include image resources in your interface. This is where the Images block comes in.

Rectangles and images are fine for graphics but your application will sooner or later need some text. If you have lots of Text objects you will be tired of defining the font manually for each one. So you will use the Fonts block to define a font once and use it everywhere. As your application grows you will also be tired of defining colors for each element in the collections block so you will create some common definitions in the Colors block.

Once you start using TextBlocks for lots of text (and not just single lines) you will appreciate the usage of the Style blocks which you will construct by yourself or borrow from other EFL applications (this is open source after all). Finally once you start playing with moving some of the interface properties from hardcoded values in the C code of the application, you will try to assimilate them in the Edje file in the Data Items block. The framerate of the Edje animations could be for example in the Edje file itself rather than in the C code as shown before (more on this separation on the next chapter).

[1] Do not confuse this term with Interface Definition Languages commonly found in distributed architectures and RPC platforms.

To finish this chapter we will mention the fact that not all Edje functionality has been revealed. Apart from the blocks shown in the table which imply a static interface, Edje Collection files can also contain scripts that make everything a bit more dynamic. Scripts are written in Embryo (another EFL library). But since someone must first learn (X)HTML and then JavaScript, we will not say anything about Embryo in this document. Try to become familiar with just Edje right now. Just keep an open mind and remember that there is more than meets the eye. If you think that Edje files are always trivial think again. You can also open some .edc files from other EFL libraries or from the Enlightenment Window Manager to see what we mean.

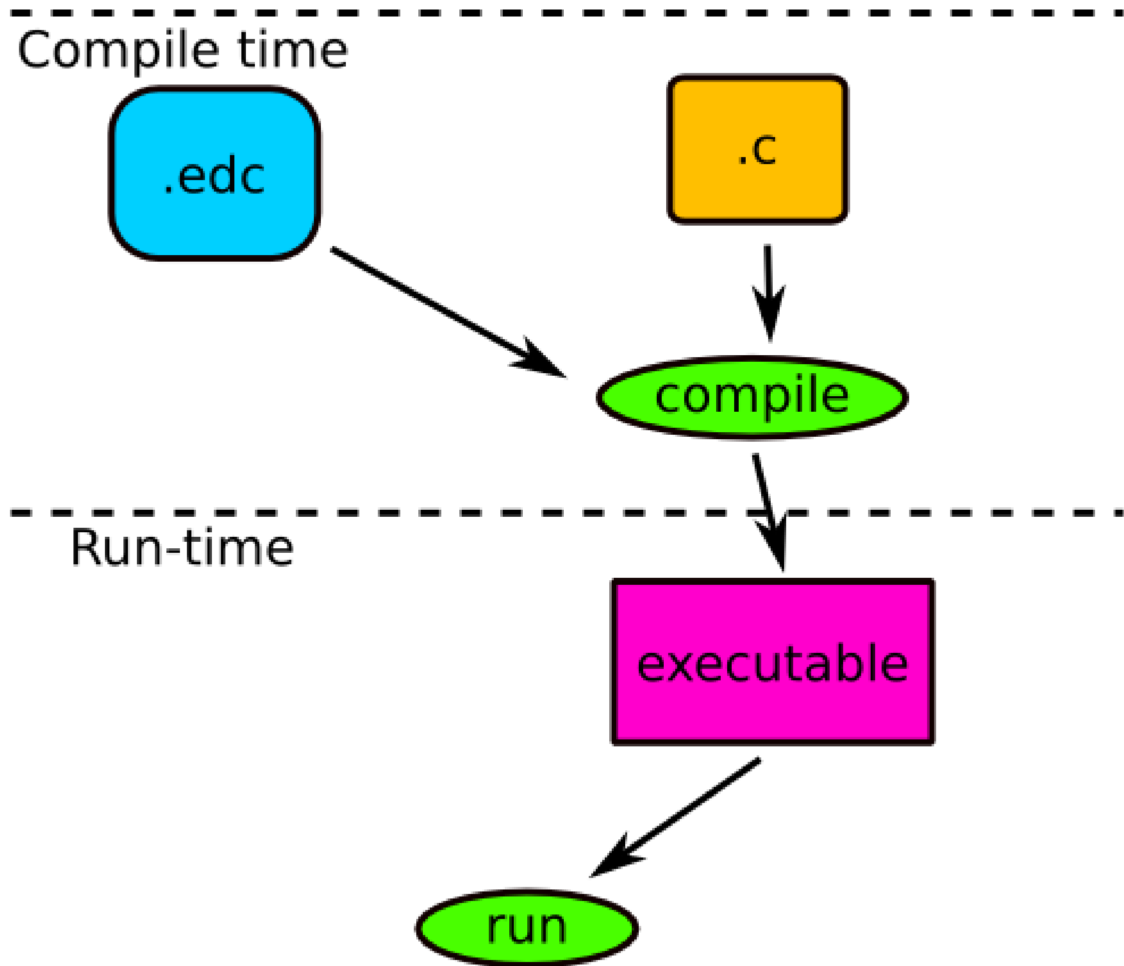
Edje as Logic and Appearance separator

So after reading the previous chapter you have a (mostly blurred) idea of the contents of an .edc file. This file describes your graphical user interface in Edje. It may contain relative positions of objects, some simple animations and maybe some images. But what do you do when you write it?

If you are an experienced programmer you will probably visualize two workflows for .edc files. The static and the dynamic one.

The static approach is the most trivial one. You assume that Edje code segments are just special macros (in C). You include in your C code the "Edje.h" header file which contains the implementation of these macros and you compile your program normally as any other C application. This results into a big executable which includes everything. Both the user interface and the application logic are in one file. See the following figure.

Figure 4.6. Building the UI statically in the application.



This approach clearly helps the developer. Instead of writing lots of C code for Evas she can quickly describe what she want to do in Edje, and be more productive because of abstraction. For a user however things remain unchanged. A single executable is the whole application. Nothing can be changed without the source code. The great benefits of this approach are of course simplicity (no extra tools, just macros and the compiler) and speed (everything is compiled in the end).

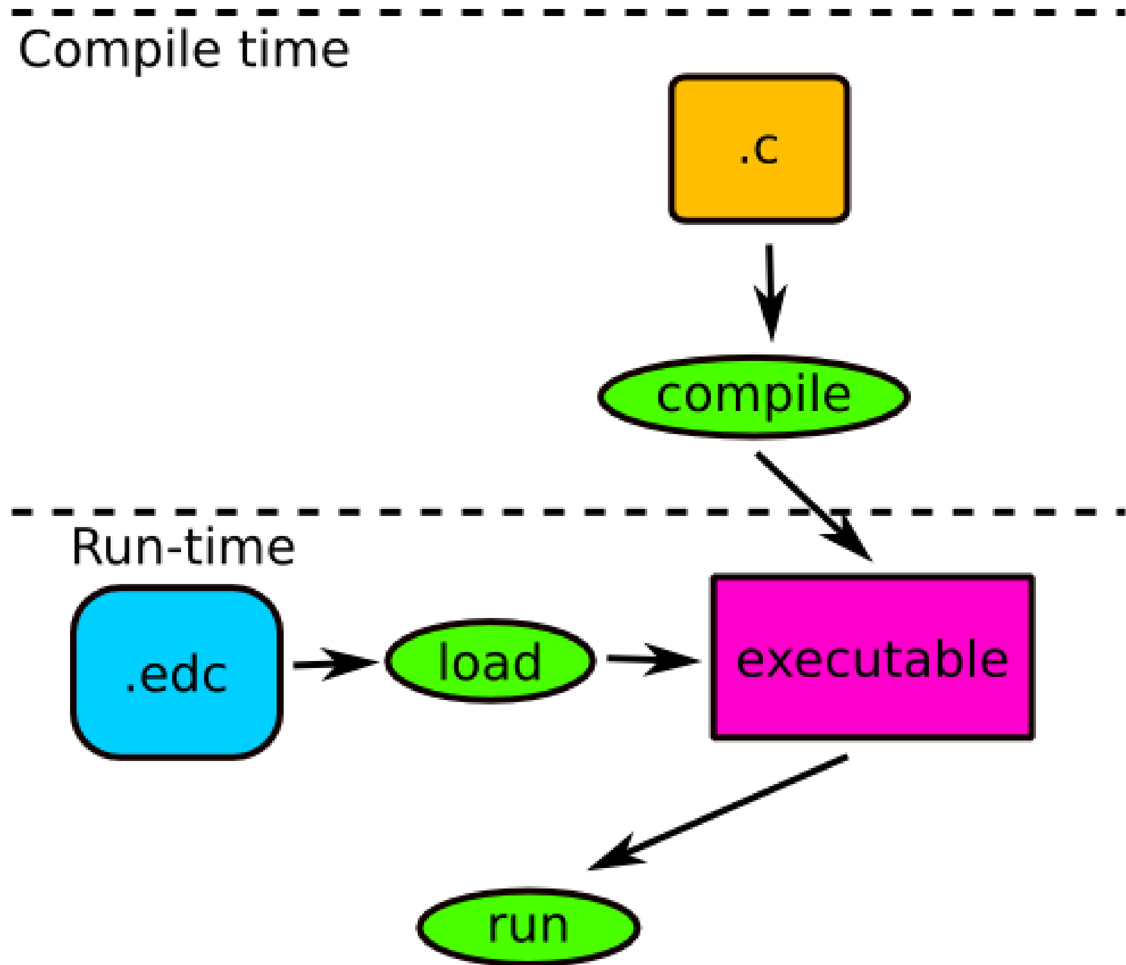
The dynamic approach would be to separate the graphical representation of the application from the actual functionality[2]. So you distribute two files for your application. The binary executable which is the code and the text .edc file which describes the interface. The application binary runs and dynamically during *runtime* it loads the graphical user interface described in the .edc file. This idea already exists for applications but the text file is mostly written in XML so that it is more human readable.

The dynamic approach has a clear advantage regarding flexibility. The user can change the text file representing the graphics of the application and after she runs it for a second time the application adapts to the changes. No recompiling is necessary. See the following figure. A lot of theme frameworks also depend on this technique. The obvious drawback of this is lack of speed. The binary file of the application must contain a text parser for reading the text file which has the application appearance. In the case of XML a full featured XML parser needs to be included. There are separate libraries in the Open Source community which implement XML parsers and thus free the programmer from this burden, but the fact remains that this is added complexity with controversial benefits. It is also clear that while this approach is more convenient for the user, it means more effort for the programmer in order to accomplish the re-

[2]You should be already familiar with this concept if you have ever done web programming where XHTML/CSS and Server Side Code are all contained in different files

quired flexibility.

Figure 4.7. Loading the UI dynamically in the application.



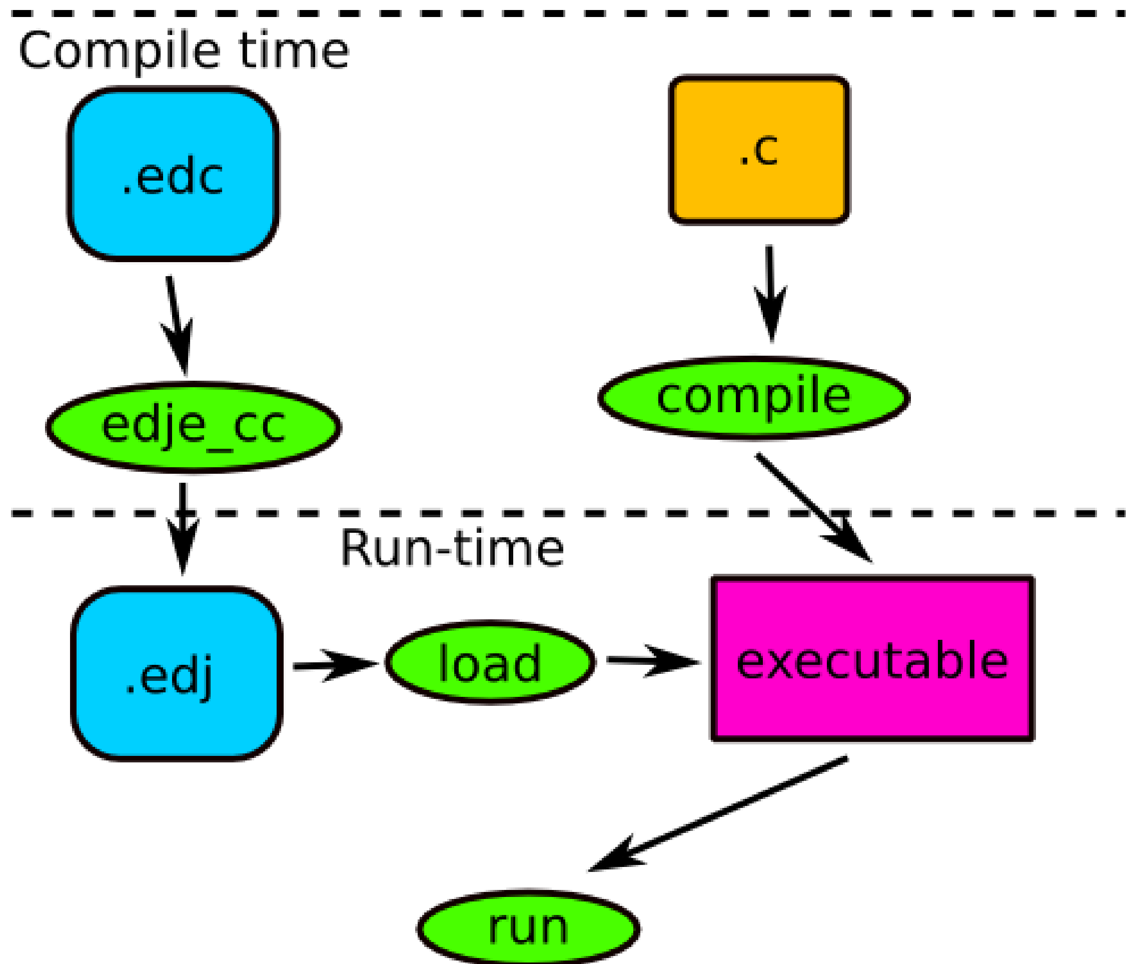
With Edje you don't have to choose between the two approaches (static, dynamic). The Edje approach is a third one which combines the best of both worlds and does away with the disadvantages! The following table summarizes the Edje approach:

Table 4.2. The Edje Appearance/Logic separation

	Static way	**Dynamic way**	**Edje way**
Files Distributed	1 binary	1 binary + 1 text	2 binaries
UI resides	Inside main executable	In separate text file	In separate binary file
Speed	Fast	Slow	Fast
UI code	compiled (built-in)	text (interpreted)	compiled (loaded)
UI is determined	During compile time	During run time	During run time
UI Flexibility	No	Yes	Yes

In Edje the .edc text file is *compiled* into a .edj file via the edje_cc compiler which is offered along the Edje framework. The resulting .edj which is binary is loaded during run-time into the main executable resulting into the final application which is presented to the user. See the following figure. You should see now why the solution combines the advantages of both approaches. You have a binary file with your UI (fast) that is added dynamically (flexibly) into your application. We are not aware of any other technology in the Open source world which does the same thing.

Figure 4.8. Separation of GUI and code in Edje.



You may think that the figures presented so far at too high level and do not actually explain *how* all this is accomplished. A great number of programmers want to get down and dirty with code to really understand a concept. Rather than presenting long code listings with an Edje application we will describe things a bit more concisely. So let's say that you have been enlightened with Edje and want to write your next application with it. What do you actually do?

First you write your .edc file which contains the User interface. Then you write the C code of your application. The C code would only contain initialization of a Canvas and nothing more (more on this later). When the UI is initialized you designate the name of the .edj (compiled Edje UI) that is to be loaded. This should be something like `edje_file_load("my_UI.edj")`. The name of the function does not really matter. Currently Edje is just a normal Canvas Object. You create and put it in a Canvas like any other object. The name of the associated .edj file is a property of the Edje object.

Next you rename your UI text file to my_UI.edc. You compile it with edje_cc and the resulting file is a binary one named my_UI.edj. The end user installs the binary and the .edj file (probably somewhere in /usr/share). When the application runs it dynamically loads the my_UI.edj file and draws itself of screen. If another my_UI.edj file replaces the previous one, the application will use this instead (themes anyone?).

It is rather simple actually. The only thing left to explain is how communication is handled between the UI placed into Edje and the C code of the application. No mystery here. If you already know to how to use GTK+ callbacks or QT Signals/Slots you are already set.

In the programs part of an .edc file you can define apart from animation (as shown previously) signals that are emitted from the User Interface in Edje back to the main C code of your application in response to user events. In your .edc file you would include in the programs block code like this:

Example 4.5. Sending a signal from Edje to C code.

```
program
{
        name, "user_clicked_button";
        signal, "mouse,down,*";
        source, "ok_button";
        action, SIGNAL_EMIT "finished" "ok";
}
```

This should be easy to understand. The asterisk in the `signal` like denotes *any* mouse button. So this program would run when the user clicks (mouse down) on the source component (named "ok_button). The `finished` and `ok` parameters are additional options which are passed back to C code. Since an OK button has only one function (usually) these extra options are not needed but are just shown here for educational purposes.

That is all that you have to do in the .edc file of your application. To decide what happens when these signals are emitted you write in the C code function callbacks which register which signals they want to listen to. As you might expect we will not show any API for this but you can rest assured that is just normal C code and nothing extraordinary. The Edje API provides you also with functions which can change the Edje User Interface programmatically. Thus, you are given the ability to respond to Edje signals with changes in the Edje object itself.[3]

So in the end you have an application binary full of functionality (C functions) and an .edj file which contains the User Interface. These two communicate dynamically while the application runs. A media player as a full example is described in the Edje book. You should consult it if you want to see in detail how a media player would work the Edje way. All the functionality for music files (load, play, stop, pause) are in C code, while everything that is graphical (stop, play buttons e.t.c) are in the Edje object.

After reading this section you may probably think that Edje would be a good framework for themes. You are right of course! Edje was built with theming in mind and we devote a separate section for this Edje ability.

Edje as a Theming Framework

[3]Not to be confused with introspection

Several applications nowadays have a fixed user interface. If a user does not like the looks of the UI she cannot do anything at all. To accommodate for different preferences, applications begun to have a component based UI with toolbars, windows, frames and panels which could be resized by the user. But all these applications still retained their boring (grayish) color that is not favored by all users. Lately most user applications introduced changes to how the application looks as well. Colors, skins, pixmaps became changeable as well. The whole idea is called theming. Several often used applications (such as media players, web browsers, window managers) are expected to support themes natively. The problem is that theming does not always gets the attention it deserves. Theming is just a hack for several application. Users just overwrite the resource files (images) that are used from the application. Other application shamelessly advertise theme support while in reality they mean the ability to change the colour/skin in the best case.

Edje brings Themes to all applications that use it. Instead of writing custom code to your application you can use the Edje framework more easily. All Edje based applications are themeable with zero additional effort. If you have ever user the Enlightenment Window manager version 16 you have already some idea on what is possible. With the EFL you actually get theming on steroids.

There are many kinds of people would like to change how the application looks. Many will just need simple color changes, others will write complete themes. Some others will want to change the position of interface elements as well. You cannot predict what people will want to do with your application. If we leave aside programmers who will just download the source code and start sending patches for what they don't like, Edje caters for all non-programmers who come across your application. Look at the following table:

Table 4.3. Edje themes (viewed by users)

	Casual Users	**Artists**	**Expert Users/Themers**
Time allocated for theming	Some minutes	Some hours	Some days
Wanted changes	colours	colours and images	the complete UI
Edje knowledge	None	Limited	Extensive
Contributions	New colour combinations	New textures/pixmaps	New themes (.edj files)

Changing the colours in a theme is trivial. The user finds the appropriate colourclass lines inside the .edc file and after appropriate changes, she recompiles its back to .edj. Nothing more to mention here.

An artist however would want to change all pixmaps of an application. Now with other theme frameworks the artist would have to either hunt down all images files inside the distributed application, or download the source code of the application and get the raw images of the user interface. Neither happens with Edje. The images block inside an .edc file does not contain just the pointers to image files which will be used by the application. It defines what images will be bundled *inside* the .edj file. Compiling an .edc file with edje_cc creates a binary file with all .edc code that describes the interface *and* the image files which are mentioned inside the interface descriptions. This makes the .edj file self-contained. An .edj file is a theme by itself. No more tarballs or zip files for themes. An .edj file is actually an EET file (another EFL library) which is a general purpose storage library for storing arbitrary information inside a single file. The .edj file is architecture independent and the programmer can choose during creation if the images will be compressed inside it or not and what would be the level of compression in the former case.

So for an artist things are simplified. She uses the edje_decc decompiler to obtain from the .edj file that comes with the application all the original resources along with the .edc file. Without looking at any line of code (Edje or C) she can start replacing the image files (.png, .jpeg e.t.c) with her own graphics. When finished, the edje_cc recreates an .edj file with the new images and exact same interface structure as before. Starting the application will bring the new interface in effect. The procedure is not getting simpler than this. Only if she changes the sizes of image files, she actually needs to edit the .edc file.

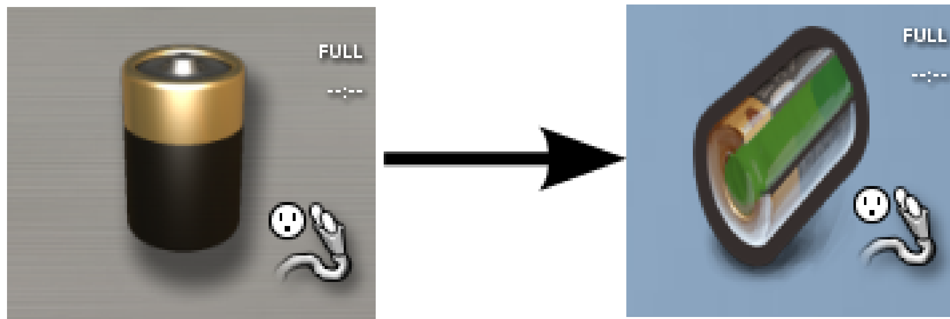
Of course one can edit the .edc file as well the image files. This gives great power to the themer and most expert users will appreciate this feature. Since an .edc file contains the complete description of the interface one can change everything regarding the application. Not only the colours or the images used but the whole structure of the UI. Location of elements, size, animations, texts, everything is configurable. One can even remove interface elements (the respective C functions in the code will remain unused) or export the same signal with a different method. The original interface for example could export a "maximize" signal when a certain button is pushed. The themer changes the .edc file to trigger the same signal during application loading. The result is that that the application is maximized as soon as it starts. And this change was inside the theme.

We saw in the previous section that all functionality is included in the C code and the Edje .edj file contains the interface which handles communication with signals. This decoupling of logic and appearance gives great power to the themer. The experienced themer can change anything at will to the point that the resulting application looks nothing like the original one. Edje does away with the concept of skins. Edje technology is a revolution when it comes to themes.

Unfortunately the EFL killer application which will showcase the full power of Edje themes does not exist yet. The most mature EFL application is the Enlightenment window manager itself. A handful of themes are already available. These can give you a glimpse of what Edje is capable of. Animated backgrounds, window borders and window buttons are easier then ever. Entrance (a replacement of xdm/kdm/gdm) also has a range of themes full of diversity. Retractable panels, animated buttons and draggable subwindows are seen in the Entrance themes.

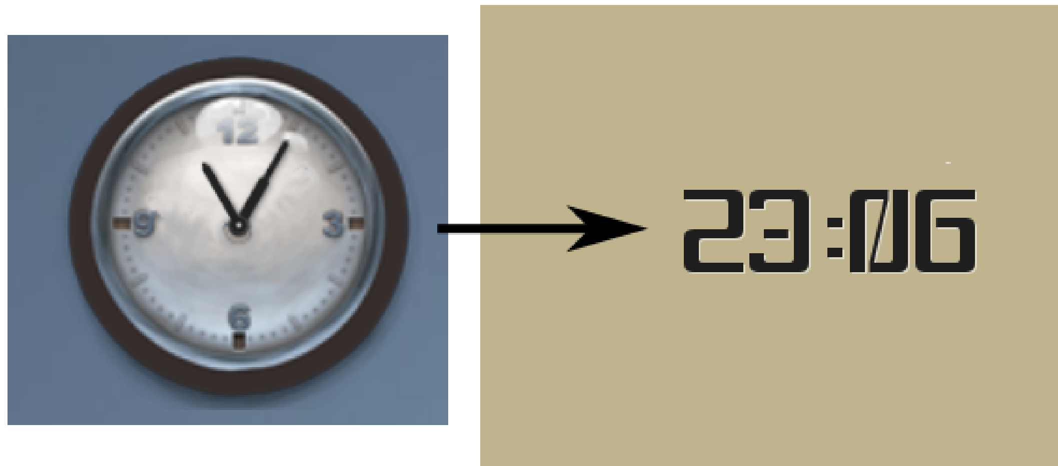
So our only example of Edje power will be in the form of an Enlightenment module. You can of course change just the pictures of the Edje file to give a new look to a component that functions in the same way as before. See the next figure. This shows the battery module for Enlightenment. The .edc file defines several images for the different states (percent full) of the battery. By changing these images an artist can create its own look for the battery graphic which will be displayed on the screen.

Figure 4.9. Changing skin via a theme.



Another included Enlightenment module is the clock. Changing the graphics of the clock is a trivial task. The background of the clock stays the same and one needs different images for the clock hands. An experienced Edje themer can however change things in more depth. Keeping in mind that time handling stays in C code and all graphics are in Edje one can create a theme that converts the analog clock to digital. See the following figure. Notice that this is a just a new theme. There is no special code for analog and digital clocks in Enlightenment. There is also some Embryo scripting involved (which we have not discussed), but the fact remains that Edje themes give you abilities which were previously available only by changing the source code of the application.

Figure 4.10. Changing completely the appearance via a theme.



In summary theming with Edje works flawlessly. The Edje file (.edj) distributed with an application is the theme itself. Self contained and self describing, the .edj file can be changed by anyone. The binary file of the application remains untouched and nobody needs to look at the C code. No recompilation is required. It is possible also to be able to change themes on the fly, but this requires effort from the application programmer to include a way to change the .edj file used by the application via an on-screen option.

We believe that once you start distributing and changing Edje themes, all existing theme frameworks will start to look limited and clumsy compared to Edje elegance, speed and flexibility.

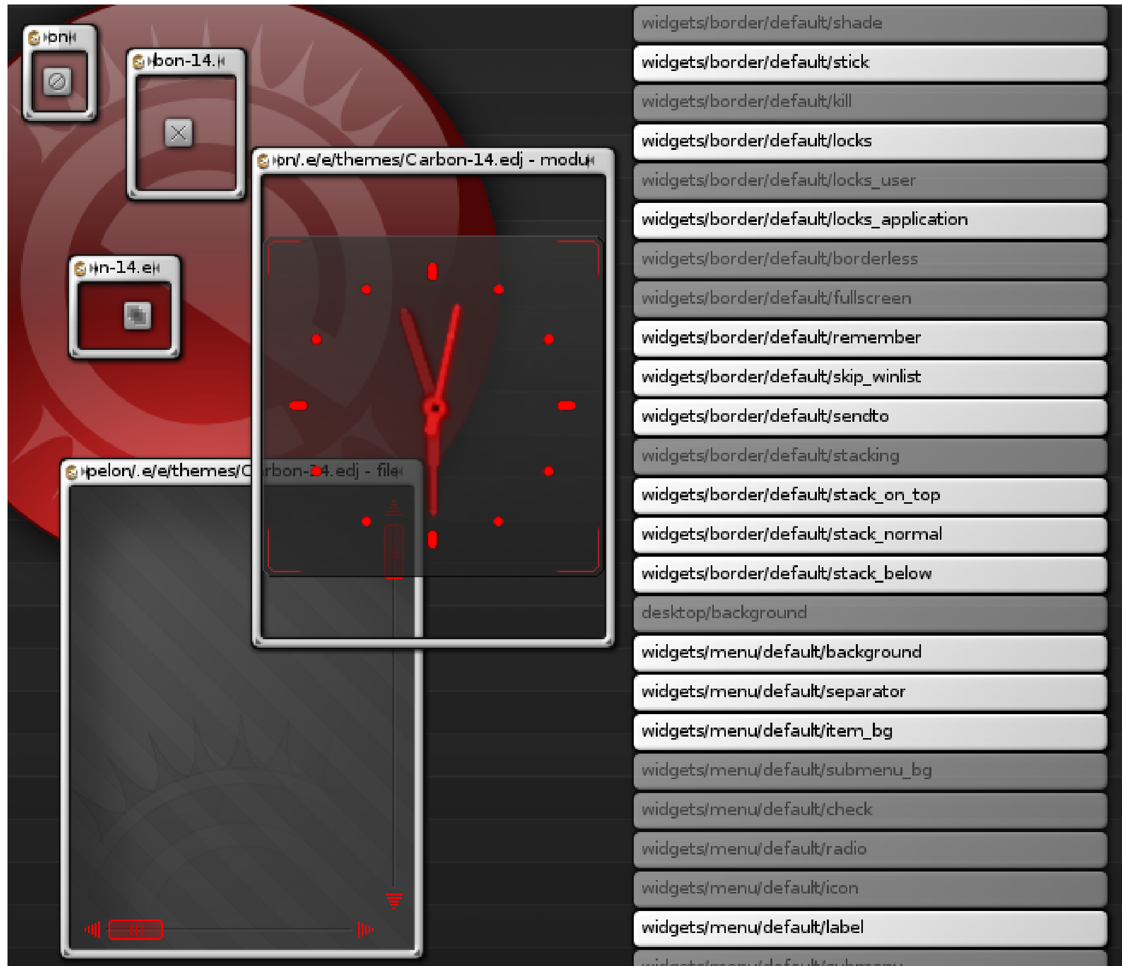
Using Edje to preview your GUIs

Judging by the previous chapters, you should have reached the conclusion that Edje accomplishes a lot. What more could you ask from Edje? This question is easily answered. A GUI previewer for debugging and inspecting your Edje (.edj) themes.

You might have come across with Glade or QT Designer/Builder. These tools allow you to create your interface in a natural way by dragging components onto your application and setting their properties from dialogs. The only code that remains is the functionality. Such a tool would not really work with Edje, since the power of Edje is found in dynamic interfaces with components that move around and change their appearance. A visual GUI builder *would work* for EWL/ETK, the toolkit part of the EFL libraries.

Edje provides you with something more straightforward. A way to preview .edj files. Without writing a single C line of code in an executable you can launch the edje executable passing as argument the name of an .edj file. You will get an instant preview of the interface. See the figure below. You can resize the interface to your liking. Clicking, dragging, and scrolling works as in the finished application, but of course all signals go nowhere so you cannot get any functionality.

Figure 4.11. Preview an Edje Theme.



This preview program works really well for both themers and developers alike. Themers who have an existing application and change its theme (the .edj) file can concentrate on the interface and not on the application. Changing bit by bit the application theme they can preview it in a simpler way (decoupled from the application) so that they are sure that it works as expected. When finished they can run the application and see how their interface works once deployed.

The Edje preview program allows one to view individual groups inside an Edje file. So instead of loading a really big application every time they are changing the theme and going to a deep level menu in order to find what they have changed they can instantly preview only the interface part they are interested in.(Provided the application programmer has split the interface into many groups).

For developers, Edje debugging becomes easier with the Edje previewer. Apart from the fact that they can construct and test the interface without writing any C code, the Edje previewer has one additional benefit. Once you load the interface and start playing around with it, Edje will print on the console which signals are emitted while you are clicking around. You can see exactly which part of the interface receives the event and what event it actually handles. Then you can write your C code accordingly. That is, the Edje previewer tells you exactly what events you need to capture (and register callbacks) in order to respond to user actions. The next figure shows this:

Figure 4.12. Live signal testing in Edje.

Nothing more needs to be said here. The Edje previewer provides you with live testing for your themes/ Interfaces.

Choosing Edje over Evas

We will finish our discussion on Edje with a more advanced section. Using Edje is clearly a breath of fresh air in the case of themes and animation, but is it always the best tool for the job?

Everything that can be done in Edje can also be done in Evas. Edje is actually using the Evas Canvas for graphics, the EET library for .edj files and the Embryo scripting language for its effects. Edje helps the programmer to be more productive with all the facilities it provides. But that does not prevent the programmer from writing an Evas-only application using only C. Evas is a powerful Canvas on its own as was discussed before.

Edje is an abstraction over Evas that saves the programmer from writing the same code all the time (relative coordinates, themes e.t.c). This abstraction of course comes at a price. Some of the flexibility that Evas provides is lost in the Edje abstraction layer. This may sound strange since in all the previous sections we tried to stress the importance of Edje and the revolutionary features it brings to the table. We still believe this. There is however a (small) percentage of people who will find Edje a bit limited and will stick to C code and Evas or even C code and their favourite Canvas widget. They will prefer to deal with low level graphic code in order to get the maximum flexibility that not even Edje can provide.

The last sentence of the previous paragraph needs some explanation. Let's say that you want to create a custom Canvas object. What do you do with Edje? Edje will not manage this object since it does not know anything about it. You are forced to revert to pure Evas code. Does this mean that you have to leave the benefits of Edje behind? Not at all. The final concept of Edje that we will examine is Edje swallowing.

Everything we have said so far about Edje, refers to using a single Edje object for your application. The C code contains only functionality but not graphics. Your whole interface is the Edje object which fills the entire application window and everything that is GUI specific is described in the .edc file.

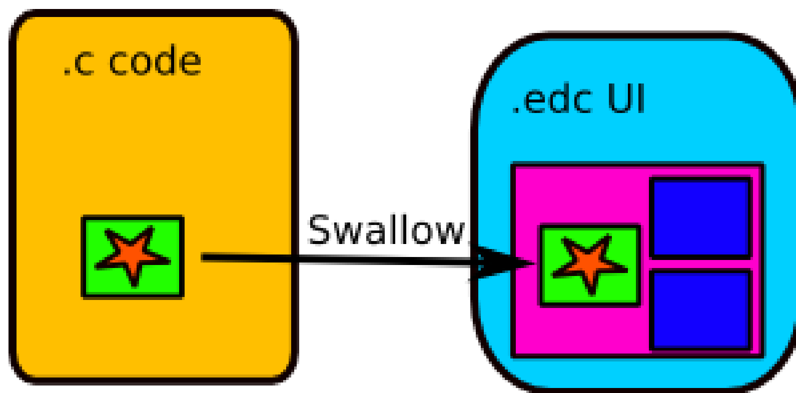
If you have a really tricky application which creates multiple canvas objects on the fly (their number is not known beforehand) and moves them around in positions calculated mathematically (dynamically) you are of course welcome to leave Edje and use only Evas instead. You can do whatever you want keeping in mind that everything will be implemented in C code. But this is the extreme case. Most cases you will want to keep Edje around and shift some (but not all) graphic code back into C where more

flexibility is possible and you can calculate several things dynamically.

This can easily happen in two steps. First you split your single Edje object into multiple. Multiple Edje collections can be implemented by making use of the groups block in the .edc file. You keep one Edje collection for your main interface and several small ones for graphic parts which behave in a more dynamic way. Then in the main Edje collection you create your parts as before but instead of using a predefined Edje part (TEXT,RECTANGLE, IMAGE e.t.c) you mark it instead with type = SWALLOW. Then you compile your .edc file normally.

The SWALLOW Edje object is not a specific object. Instead it defines a container which can be programmed dynamically in C code regarding its contents. You can fill it with another Edje collection loaded from the same .edj file, another Edje collection loaded from another .edj file or even a custom Evas object created dynamically in C. Once the SWALLOW object is filled Edje will modify its geometry according to how the application resizes or moves. The technique is illustrated in next figure:

Figure 4.13. Swallowing an object programmed in C into a UI described in Edje.



The result is a mixed approach. You lose some Edje benefits. Themers cannot change swallowed parts of your interface since they are not inside the .edc file. You cannot use the Edje previewer to see how the interface works since these parts are created during runtime (in the case of C code). Changing anything for these parts requires manual changing of the source code.

On the other hand you have perfect flexibility. You keep Edje around for most parts of your interface and you employ sophisticated C code for specific parts which are powerful as pure Evas code would be. The choice is your to make. You can have as many SWALLOW parts as you want. The next table outlines all possible approaches.

Table 4.4. Edje vs Evas

	Pure Evas	**Pure Edje**	**Edje and Evas**
UI Complexity in	Evas (.c)	Edje (.edc)	Both
Edje objects	None	One	Many
Dynamic UI morphing	Maximum	Limited	Available
Swallowed Edje parts	Not possible	None	One or more
UI and code separation	None	High	Medium
Programming effort	High	Low	Medium
Theme support	None	Extensive	Partial

To fully grasp Edje you need of course to write some applications that actually use it. We could spend more time about Edje, its relationship with Evas or even Embryo but by now you should have realized the power that Edje gives you. Evas may be the base of all EFL libraries. Evas may be a powerful Canvas. Evas may be fast and optimized. But in the end Edje is the leverage you need in order to completely do away with low level Canvas code and build powerful graphic applications. Before Edje programmers were faced with a hard decision. Either use a limited toolkit for rapid prototyping of a WIMP[4] application or a powerful but low level Canvas for custom graphics. Edje is right in the middle giving you the power that you need without forcing you one way or another. You can still use only Evas or only EWL/ETK for the two extremes, but you also keep the best of both worlds with Edje.

[4] Windows, Icons, Menus, Pointers (that is 99% of applications)

Chapter 5. Understanding the Ecore Infrastructure Library

Ecore is the glue that holds all the EFL libraries together. You can use most EFL libraries by themselves but Ecore gives you the plumbing you need to make them work all together in an elegant way. It also gives you additional programming constructs that will make EFL development a little easier.

Using Ecore in your programs

Unlike Evas and Edje, Ecore is a non-GUI library. Although it deals with graphics in a sense (abstracting some X functions for example) you can perfectly program a command line application in Ecore. Ecore is the foundation of the EFL family implementing job handling, networking functions, timers, configuration management, IPC, and several other aspects of an application that you would have to write yourself if your program used only Evas/Edje.

You might think that Ecore is to EFL what Glib is to GTK+. This is only partially true. While Glib is the non-GUI part of GTK+ which is even used by non-GTK+ apps, in practice all GTK+ applications *depend* on Glib. Ecore in contrast, is an add-on library that *boosts* your EFL application. It should be clear from the previous chapters that you can use *only* Evas for an application.

As with Evas engines all different modules of Ecore (IPC, Jobs, X abstraction e.t.c.) come in different libraries in your system so you can selectively choose what you want to link with your application. If your system does not employ the DBUS daemon for example you can leave out the ecore_dbus module during linking. For a console application you can leave out the ecore_x module and so on.

The next section briefly covers the Ecore capabilities.

Programming Facilities

This section is a general list of Ecore capabilities.

Job Handling
: You can add and remove jobs (that call your functions) in the event loop of your application. The event loop and what ecore does with it are explained in a separate section.

Idle Handlers
: You can define what your application does in its idle time. You can even define what it should do *before entering* and *after exiting* an idle state. Again this functionality is explained in a separate section.

Configuration management
: Ecore does away with the traditional text based UNIX configuration files. Instead it employs configuration keys stored in binary (powered by the Eet storage library). You can even register listeners that are called when configuration changes. A separate section is devoted to storing/loading configuration too.

Process Spawning functions
: An abstraction over UNIX `fork`, `popen`, `exec`, `getpid` system calls. You also have the ability to send individual UNIX signals to the processes spawned this way.

Data structures
: A hash table, a linked list and a tree data structure are offered. A doubly linked list is available too. Notice that these implementations do not require Evas to be present (as we said before Evas includes

	code for some simple data structures too).
File monitoring	You are offered the ability to monitor a file descriptor and have a callback function when there is activity on the file managed by this descriptor (e.g. reading/writing).
Searching a file in a set of directories	Similar to the `PATH` variable on UNIX system where an executable is searched on a predefined set of directories, Ecore allows you to do this for any file your application wants. Of course the directories contained in the search set are defined by the programmer.
Ecore plugins	Loading additional Ecore plugins during run-time. The usual `dlopen` stuff.
Timers	Programmable timers. Complete with interval changing *after* the timer has already started.
Network Connections	An abstraction over BSD sockets. Can also use local UNIX sockets for localhost communication.
Framebuffer Utilities	Several helper functions for Framebuffer devices. Calibration stuff mostly. Backlight/Display and double click interval too.
Generic IPC	IPC is Inter Process Communication. An abstraction over UNIX IPC.
Ecore X	An abstraction over X. You don't need to use X specific functions to set the class/icon/title of your application any more. Display, windows, properties, synchronization/flushing, pixmap, geometry functions are available which wrap around the respective X function.
Ecore dbus	Dbus bindings. Dbus in an emerging technology offered as a freedesktop.org standard. It is already used by Gnome and recently adopted by KDE. Its objective is system-wide IPC.

Configuration with Ecore

The first Ecore feature we will focus on is configuration for an application. Any non-trivial application needs to store somewhere the options selected by the user. Once the application is started the second time it should "remember" the user choices. This implies the saving of configuration parameters on disk (filesystem) between runs.

The traditional way to store configuration in a UNIX system is well known. Each application reads and writes a text file which is human readable. Most times this text file contains key-value pairs on each line, but there are applications with more complicated configuration files which actually define their own configuration language schematics. Several of the heavyweight UNIX servers (web, mail, FTP) can even have configurations which span multiple files and directories.

This approach was chosen for its universality. Changing the configuration of *any* program means using *any* text editor to change the text file. Configuration files are usable across any architecture since they are text based. Contrast this to the commercial (closed-source) world where configuration is stored into cryptic binary files. These files are contained in a non-documented format known only to the company that produces the respective application. If the company dies you are out of luck. You cannot move configuration files around and several times you even need special programs (converters) just to upgrade configuration files to newer versions.

Of course the UNIX approach has its drawbacks too. First of all casual users do not really like to manually edit text files. While a UNIX administrator is happy that with a single SSH shell and his trusty VI

editor can completely (and remotely) manage the whole system, a non-expert user prefers GUI dialogs with buttons and entries for configuration parameters. Then there is the problem of the actual text format. While the configuration of most programs in a UNIX system are in text, the exact layout of the text file differs from application to application. A comment can start with `#`,`;` or `%` for example. Comments may or may not span multiple lines. The order of lines may be important or not. Some lines will be in blocks some other will not. You get the idea. The fact that other UNIX programs also use text files for input (Makefiles, Latex, e.t.c) makes things even worse. A non-expert user is afraid that she even may change the configuration file to a non valid state which will be not understood by the application.

Another problem (from the development view) is the fact that each application must now implement a text parser for its specific file format. While specialized tools exist for this purpose (lex/flex, yacc/bison), for small applications this is simply an overkill. Not to mention the fact that text parsing is almost always slower that reading binary data. For small configuration files this is not a problem but for huge applications this becomes quickly evident.

Finally a big question is what happens when someone changes the configuration file while the UNIX application is actually running. If the application does not support this, one must simply restart it so that the new changes take effect. Otherwise the application is informed that its configuration file has changed and it automatically fetches the new values from disk. There is even a well-accepted convention just for UNIX servers for this reason. Once an application of this kind receive an NOHUP UNIX signal it is expected to read again from the disk its configuration file and adapt accordingly. So in effect the NOHUP signal is the "restart" signal.

When XML appeared on the scene people started using it for many things and one of them was of course configuration files. While XML may be marketed as the universal format of the Internet we are not really sure that applications really gain from storing configuration in XML. The fact remains that the application must implement a text parser. And in this case the parser must be also an XML validator so the complexity and effort is greater for the programmer. Also, since XML files are either valid or not by definition, it is easy for a non-expert user to make an XML file unreadable after manual editing. A single syntax errors means that the application must reject the whole XML file. This does not happen with traditional UNIX applications which could dismiss lines they do not understand and load the rest of the values. Today several applications indeed use XML for storing configuration but it is not yet clear if this is an improvement or not over the past.

Ecore breaks away from the traditional UNIX approach and offers a configuration API which is based on binary format. While this decision may seem controversial at first, it is actually well thought of. Ecore provides functions used to store primitives on disk without specifying any additional details of where and how (the location or format of the file that is).

Table 5.1. Ecore configuration primitives

Primitive	Description
Integer	A simple number
Float	Floating point number
String	String-based value
Colour	RGB description of a colour
Theme	Definition of a theme
Boolean	Binary (true/false) value

Notice that you do not need to write any code on how to store and load these values. All the low-level code is handled by Ecore. The values are stored in the home directory of the user that runs your application in the file `$HOME/.e/apps/YOUR_APP/config.eet`. Notice the .eet extension. Ecore uses the Eet Storage library for configuration which is the same one used by Edje for themes (and Enlighten-

ment for backgrounds, eapps, themes, splash screens e.t.c). The only thing that matters to you as a programmer is the name of the value that you can use to load and store information. Everything else is abstracted away by Ecore.

So why this approach is better? First of all binary files are (as expected) faster that text files. Secondly you don't need to deal with any text parsing at all. Your application does not contain any low level I/O code at all. It just links to the Ecore_Config library. You need of course to provide some GUI for your users to change the values if that is required.

The fact that all configuration files are actually Eet files gives applications two advantages. Eet is designed so that its files are architecture independent. Despite being binary Eet files can be freely moved around on different systems. Additionally it is trivial to program a command line application which will read/dump the values stored on *any* configuration file. Therefore universality is accomplished in a way too. Expert users who want to bypass the GUI and directly edit in the command line can do so, since their Eet reading application will be compatible with all configuration files that use Ecore for storage. The Eet library is open source so the binary format does not need reverse engineering.

Last thing to notice is that Ecore gives you the capability of Configuration Listeners. These Listeners register to configuration parameters and when they are changed they trigger the callback function you have specified. This means that you don't need special "restart" code. Your application can be informed during run-time on configuration changes and act accordingly.

In summary, Ecore provides a unified way to store configuration values which makes things easier for the programmer. It may feel a bit strange to know that your application stores configuration in binary but you should quickly see the advantages.

The Event Loop

You might be surprised to read that Ecore provides IPC and networking abstractions but not any kind of thread API. Thread support seems to be important these days and most support libraries (as Ecore in the case of EFL) usually offer some sort of wrapper around UNIX threads. The fact is that Ecore takes a completely different approach when it comes to "parallelism". Rather than employing threads, Ecore focuses instead on a powerful event loop mechanism. If you know everything about thread versus event based programming feel free to skip ahead.

Console programs are straightforward to program when it comes to parallelism. The application is in control, while the user just waits the execution to finish. Interaction happens either before the application starts (command line arguments) or only at rare cases during runtime. In the latter case the application stops working and asks the user for required input. With GUI applications the situation is reversed. The user is now in control and the application must respond to user actions (which come in the form of events). Since a GUI application must continue working and updating its window even when the user has assigned some work to it, a form of parallelism is needed. The GUI application must do at least two things at once (GUI in the foreground, processing in the background).

Threads come in the rescue for this kind of problem. One thread is managing the GUI and one or more are processing in the background. The concept is well known but also well known is the fact that thread programming is difficult. When threads have to share common resources the programmer must be extra careful to avoid racing conditions and use locking to avoid corruption of data. Without getting into details, we should just say that a badly written threaded application has bad performance in the best case (because of locks) and serious problems in the worst (deadlocks which result in application freezing). Threads need also special libraries for implementation and special debugging support from the run-time system.

An alternative approach for this is event driven programming. Here all pending actions (jobs to do) are queued in a single "list". The running application looks at this queue, selects one job to serve, then comes back, selects another one and so on. There is only one control flow at a time which simplifies programming (no locks/racing conditions). If you have ever used the `poll()` or `select()` UNIX system calls which act on file descriptors you are already familiar with event handling. Of course even

driven programming has its drawbacks too. The performance is not always optimal if the queue is too large (compared with using threads). Threads also have gained great momentum with the coming of multiprocessors (where a thread can be served by another CPU in a truly parallel way). The following table compares the two different approaches.

Table 5.2. Threads vs Event based programming

	Threads	**Events**
Programming effort	High	Low
Locks needed	Yes	No
Possible Deadlocks	Yes	No
Debugging	Hard/Impossible	Easy/Trivial
Good for	Many independent workloads	A few intermixed workloads
Bad for	Too many shared resources	Too many workloads
Ideal in	Multi processor/core systems	Single processor/core systems

Ecore chooses the Event-driven approach. Notice than in the case of uniprocessor systems, since only one control flow is active at a time, you never have true parallelism no matter the approach you choose. It is just a matter of convenience.

The Ecore event loop already deals away with a lot of events. You do *not* need to write event handlers for several window actions such as moving/resizing/exposing. You do not even need to decide when to repaint your main window (previously shown in the Evas chapter). In general if your application is simple enough you do not need to deal with events at all. Just draw your items in Evas/Edje and everything is done automatically for you behind the scenes. When the interface changes (values updated, UI elements added/removed) the application will manage everything by itself.

So why do we devote a separate section on the Event loop in Ecore? For the simple reason that you have some capabilities not easily found in other toolkits. The first is that you can add manually jobs in the event loop of the application. These will be served by Ecore in a best effort manner. This means that they will be served eventually by the application but you cannot predict when exactly.

The second capability (which is more interesting) is that you are able to decide what your program does when idle. That is, you can specify a callback function for situations where your application waits for I/O or user input. The callback function will be run using whatever CPU percentage is available to the application. Taking this idea a bit further, Ecore additionally allows to define what happens when the application *enters* and *exits* the idle state. In result, you can program a really complex EFL application without using any threads at all and delegate all processing into "idler" functions. This gives EFL programmer great flexibility.

If you still believe that threads are essential, you might not know that your X-Server is a *single* user-level process (no ties with the kernel OS) which handles all on screen drawing via Events. If a whole implementation of the X architecture is possible with events then certainly your application can refrain from using threads too.

Closing this section, we need to note that if you need to run a function at a specific interval during run-time you are always free to use the traditional timers which are offered by Ecore.

Chapter 6. End matter

With the Ecore library we have concluded the EFL tour. You should have a fairly good idea of the capabilities of the EFL libraries by now. You may have begun to imagine how you should use them in your next project! If some implementation details escape you, do not worry. This was exactly the purpose of this document. Rather than confusing you with low level stuff and lengthy code listings of all the details we have attempted to give you a gentle introduction into the general concepts surrounding the most important of EFL libraries.

The rest of the EFL libraries

At this point you may wonder why we stopped at Ecore and did not continue with the rest of the EFL libraries. Eet and Embryo are mentioned several times in the text so it is natural to expect a separate chapter on each one them. Why stop now?

We think that it is better to just focus on Evas/Edje/Ecore only. There are three reasons for this decision. First of all, a lot of programmers who come in contact with EFL are confused by the exact nature of each library since all of them start with "E". While reading the documentation they forget which library does what and whether it is important (essential) for a program or not. The fact that most EFL applications also start with the letter "E" makes things even worse. It is easy to lose track of things with all the "E" names around you. By focusing on only 3 libraries we think it is easy for you to understand quickly what EFL is all about. You can instantly decide what is important and what is not and if you need this particular library for your project or not. While both Eet and Embryo are important EFL libraries, it should be clear that your first programs do not need to use them in order to do something useful. After all, Eet is hidden behind the Edje themes and Embryo scripts are rather advanced and in a way litter the clean-cut Edje theme files.

The second reason is that Evas and Edje are the libraries which are really groundbreaking. These are the libraries that provide you with new capabilities that you have hardly seen before. We could spend a lot of time talking about how Eet stores its data in an architecture independent form or how Epsilon/Epeg create thumbnails, but you would probably shake your head and think "I have seen this in library xyz before". But can you say the same for Evas or even Edje? Ecore is presented in this document because it is the glue library of EFL and it will save you a lot of code if you use the facilities it offers.

Finally remember that all EFL libraries are under heavy development. Several libraries have changed purposes/API/design over time. Some have even been declared obsolete. So instead of giving you a description of libraries that might not even be released eventually, we focus only on Evas/Edje/Ecore. These libraries have existed for a long time, they are stable, well documented and tested. They will almost certainly be a part of the EFL libraries when released. Several smaller EFL libraries come and go (in the form or ideas or partial implementations) but these 3 will always be there for you to play with.

Get involved

When you start using the EFL for your projects, do not forget that you can also help develop the libraries themselves. EFL, the Enlightenment window manager and assorted applications are offered under an open source licence.

If you are a programmer with heavy C skills you are most welcome to send patches for EFL code or even become a regular developer. Subscribe to the Enlightenment development list, see what are the problems at hand and decide where your want to help. You need of course to use the latest code available (offered via CVS) in order to be able to keep up with the ever changing EFL code base.

If you are an artist instead (a guru in Gimp maybe) you will be happy to learn that EFL is all about eye candy. Browse through the pages for Themes/backgrounds/modules/splash screens and see what other users have already submitted. Then either modify an existing theme to your needs or even better create

you own from scratch. You will soon find that EFL is THE platform that you can showcase how good artist you are (second only to OpenGL/textures/gaming engines). Several of the EFL programmers actually consider themselves more as an artist than you would think.

You can also help with documentation. Once you start using EFL for your themes/programs you will certainly discover some tricks/concepts/methods/functions which lack good documentation. Do not hesitate to write about your discoveries. The document that you are reading at the moment may for example be inadequate in some areas in your opinion. If you think that you can improve it in any way contact its author and show your interest!

Finally if you are just a (mortal) user do not despair! You can help too. Download the snapshots from freedesktop.org (or if you feel lucky directly from CVS) and perform some bug hunting. Programmers can only test a small set of functionality and sometimes miss some test cases. They never know what users will actually try to do with their software!. If you submit a bug make sure that you give a thorough description (all error messages, gdb backtrace e.t.c.). There is also a separate mailing list just for users (it also serves the Enlightenment version 16 users.

Resources

Rather that including links inside that main text of this document, we have decided to gather all resources in one place. This makes things easier when you have already read the document once and start searching for more information.

- The Enlightenment website [http://www.enlightenment.org]
- The Enlightenment Team [http://enlightenment.sourceforge.net/Main/The_Team/]
- E-Develop [http://www.edevelop.org/]
- Enlightenment Documentation [http://enlightenment.sourceforge.net/Libraries/Documentation/]
- Enlightenment Development (mailing list) [https://lists.sourceforge.net/lists/listinfo/enlightenment-devel]
- Enlightenment Users (mailing list) [https://lists.sourceforge.net/lists/listinfo/enlightenment-users]
- Get - E [http://www5.get-e.org/]
- Freedesktop EFL snapshots [http://enlightenment.freedesktop.org/]
- The Elive CD [http://www.elivecd.org/]
- EFL user guide [http://www5.get-e.org/EFL_User_Guide/English/]
- KDE [http://www.kde.org/]
- QT at Trolltech [http://www.trolltech.com/products/qt]
- GNOME [http://www.gnome.org/]
- The GTK+ toolkit [http://www.gtk.org/]
- Blender 3D (3D Animation) [http://www.blender.org/]
- Synfig 2D (Vector Animation) [http://www.synfig.com/]
- The Gimp (Image Manipulation) [http://www.gimp.org/]
- Inkscape (vector drawing) [http://www.inkscape.org/]
- Feh Image Viewer [http://linuxbrit.co.uk/feh/]