The Enlightenment Foundation Libraries

A Big Picture

Gustavo Lima Chaves
glima@profusion.mobi

June 28, 2010

Contents

1 Introduction

The Enlightenment Foundation Libraries, or simply EFL, are a set of software libraries that grew up to support the Enlightenment desktop shell and the applications using its same technology. They have historically been built with high optimizations in mind, targeted not only to desktop computers, but also to low-end devices.

In this document a big picture of this set of libraries and their correlation is going to be presented, so that software developers totally unaware of them can rapidly learn the basics and start using the EFL.

2 The e-libs' basics

EFL developers tend to name their libraries with something beginning with the letter "e". In this document we'll talk about the following ones: evas, ecore, eet, edje and elementary. This section gives a brief overview of these 5 libraries, before we get a little deeper inside each of them.

2.1 Evas

Evas is a fundamental piece in the set – it is the **canvas** library, which manages the graphical objects one wishes to exhibit and deals directly with back-end engines closer to the hardware display drivers. Of course, it abstracts any need to know the characteristics of your display system or which graphics calls are in fact used to draw on the screen.

Unlike other canvas objects (or widgets) provided by most of the graphical user interface libraries (GUIs) out there, evas is specially powerful. It is a **stateful** canvas in that it keeps track of which objects must and mustn't be rendered on screen. It does the so-called **retained mode drawing**, in opposition to the **immediate mode drawing**. The programmer has no need of dealing with objects repainting or keeping their state. Evas optimizes the rendering pipeline to minimize the effort in redrawing changes made to the canvas and so takes this work out of the programmer's hand, saving a lot of time and energy.

Evas has no notion of **time** and **animations**, things that are made available to the programmer by combining it with other e-libs.

This library is intended to deal with **raster graphics**. Its powerful (super and sub-sampled) smooth-scaling algorithms guarantee fancy graphics even when different sizes of an image are needed. It ships with loaders of gif, jpeg, png, tiff and xpm image files. It can save them back, possibly after painting something on top of them or applying one of its transformations, into jpeg, png and tiff formats. Evas can also draw anti-aliased text, alpha-blend objects and much more.

Finally, evas supports **many** different back-end engines. This, paired with its highly optimized drawing methods, permit that the library can be used on a large variety of systems, including low-end embedded devices.

2.2 Ecore

Ecore is a like a "swiss knife" library for the e-world. One can resume its purpose as a library that gives developers higher level interfaces for low-level stuff (and also convenience functions). It also provides, for example, facilities evas was not meant to have, like event handling and timers. Moreover, ecore is the library which provides the **main loop** for the applications using EFL technology (though one is not restricted to it).

This library also provides wrappers on top of evas, simplifying a bit the chain of functions needed to instantiate and manage a canvas. Other facilities found on ecore are sockets abstraction, IPC, configuration handling, etc.

2.3 Edje

Edje is a special library even between the EFL ones. It is a powerful and pioneer **layout engine** and graphical design tool based on evas.

Note:

On the words of its creator:

"Edje is an attempt to find a middle ground between theming and programming without turning the theme itself into just yet another program." – Carsten Haitzler (The Rasterman)

It provides an abstraction layer between the application code and its interface, besides allowing for extremely flexible dynamic layouts and animations. More precisely, it interprets files compiled from a declarative language which allows one to describe a graphical user interface without writing a single line of the working programming language's code. In other words, your application is split into two parts: a graphical part, which knows nothing about (functionality) code and the functionality, which knows nothing about its GUI.

This brings more freedom to both programmers and interface designers. Once a contract between the program's back-end and its interface is established (which is done mainly in terms of signals, to be better explained further), developers can easily change the back-end's functionality independently of the GUI and vice-versa.

This concept, for ages already supported by the EFL, has more and more drawn people's attention and has recently been called **declarative user interfaces**.

In terms of implementation, one can say that edje is a **state machine**. When declaring an interface visual element, the designer describes one or more **states** that element can be at. These states can differ in many parameters, for example:

- object's position and size,
- object's color and opacity,
- object's visibility,
- object's response to input events, etc.

Naturally, edje holds, internally, a geometry state machine and a state graph of what is visible (or not), where, at what size, with what colors, etc.

Speaking of input events, a great feature of this library is that one can specify, besides element states, actions triggered by different input actions and **transitions** to optionally occur after them. Input events may be mouse ones (hover, click, etc), signals from the application's back-end, etc. There are some built-in transition timelines (being their total time a parameter) which take one object from origin to target states. Examples of them are: linear, accelerated and sinusoidal transitions (see section 5.6).

Edje allows for **real** theming capabilities: applications can have interfaces **completely** different one from another and with different behavior, too. One describes an interface in the form of an **edje data collection** (EDC), which is a plain text file with simple syntax to describe the interface's visual elements and their behavior. This is so that the visual elements' descriptions, along with their image sources, when applicable, and the fonts used by text elements are all packaged together into a **single** file. This way, whole themes of interfaces can be shipped around with great facility.

A designer has the ability to animate, layout and control the look-and-feel of any program using edje as its basic GUI constructor. Naturally, if one needs a more specific element behavior or look-and-feel, new visual objects can be built from scratch, with the help of a developer, who will be using evas directly.

2.4 Eet

Eet is a tiny library whose API lets the programmer write an arbitrary set of chunks of data to a file (optionally compressing them, very much like a zip file) while allowing fast random-access reading of this file later on. EET files, as they are called, are perfect for storing data that are written once (or rarely) and read many times, especially when the programmer wants to avoid the necessity of reading all the stored data at once.

Two common use cases for this library are:

- storing configuration data and
- **storing themes**.

EDC files are processed by a binary distributed with edje, **the edje compiler**, which packs the interface's description and data altogether in the form of an EET file. People have historically named these interface files with the `.edj` extension. For now on, we'll refer to these specific EET files as **EDJ files**.

Besides we cite just these two use cases, many more exist for eet. One can store **arbitrary** data chunks at EET files. There are developers storing javascript scripts in it, for example.

2.5 Elementary

Elementary is a basic widget set library built on top of other e-libs (mainly evas, ecore, and edje) that is easy to use. It was created, originally, aiming development for mobile touch-screen devices.

This is just one of the ways, while using EFL technologies, of creating visual elements besides using pure edje (or even pure evas). Elementary provides many built-in widgets to the programmer, like:

- icons,
- buttons,
- scroll bars,
- labels, etc.

All these widgets are themeable, of course.

Elementary also provides higher level abstractions and control. For example, the library enables operations which affect the whole application's window, with functions to lower it, raise it, maximize it, rotate it and the like. Also, with elementary the user may set the "finger size" (finger-clickable widgets' sizes) intended for use on a touchscreen application, affecting all the widgets with finger interaction.

3 Ecore-evas

Ecore-evas is one of ecore's modules you'll certainly want to use. It is a set of convenience functions around evas, which save you from lots of low level interaction with drawing engines, canvas update and maintenance calls and the like.

All graphic engines evas supports have respective high level functions, found at ecore-evas, to:

- instantiate a canvas, bound to that back-end engine, with a given geometry, and
- retrieve the application's window, if the graphic back-end implements windows.

Besides that, ecore-evas gives you generic canvas manipulation functions, regardless of the engine being used. For example, you can:

- set callback functions on canvas events like resize, move, gain/loss of focus,
- perform actions on the canvas like move, resize, set the title, set fullscreen mode, etc.

Ecore-evas, finally, integrates the calls of canvas updating (re-calculation of object states and re-renderization) into ecore's main loop, so that the user mustn't deal with it.

4 Evas and its objects

The basic evas object unit one is going to interact with is the `Evas_Object`. This is a C opaque type which represents a generic visual element evas handles and draws. `Evas_Objects` have, naturally, a **type** associated with them, so that specialization is possible.

Evas has a set of built-in object types[1]:

- rectangle,
- line,
- polygon,
- text,
- gradient and
- image.

The ones which are most used are rectangles, text and images. In fact, with these ones one can create 2D interfaces of arbitrary complexity and EFL makes it easy!

However, people are not at all limited to these objects. Evas lets you build objects of arbitrary complexity with the concept of **smart objects** (SOs). Beginning from a basic `Evas_Object`, one can specify a basic interface to this new object (function pointers like `add`, `del`, `move`, `resize`, `show`, etc.) that form the common set of evas interaction routines over its objects. More importantly, SOs can have other objects as their **children**. Smart objects' behavior can be so that the outer operations executed on them reflect on a custom manner over each of the child objects.

Besides having to be built a little differently than simple built-in objects, smart objects end up with the same opaque type for use by the programmer: `Evas_Object`. When building a widget set, programmers have this instantiation steps hidden away from the API users, so that the creation of evas smart objects end up as simple as the base ones (just one function call).

Evas comes with three built-in smart object types, whose descriptions follow.

Box Objects A box container is a smart object intended to have child elements, displaying them at **sequential order**. It can layout the child elements in various different ways, though. It has, in its API, functions to set the desired layout, with the possibility of using layouting functions other than the built-in ones.

Table Objects A table container is a smart object also intended to have child elements, displaying them in a **table** (or grid) form. Again, one can choose between some built-in layouts, but there is no support for custom layouts for these objects.

[1]Actually there are two implementations of gradients and an object for multi-line text entries, too.

Edje Objects This is how, at programming level, edje and evas fit together. Visual elements whose appearance and behavior are controlled by edje are nothing more than evas smart objects. Edje encapsulates all the instantiation steps of these objects inside itself. We'll see, further, that after instantiation, an edje (smart) object must have an EDJ file set to it. This operation also requires that a **group**[2], declared inside that EDJ, is associated with this object. This group tells about all of the child elements this edje object has. These can be any evas object, including the smart ones (note the recursive aggregation possibilities, here).

Elementary's widgets, as one might expect, are also evas smart objects.

Finally, any `Evas_Object` may have **hint fields** set. Their objective is to guide the e-libs' functionality in some way while layouting and displaying that object. These hints may affect, for example:

- object sizes (both horizontal and vertically),
- object alignment inside a container and
- object padding.

Hints are not always enforced (and may be treated just like hints, then). Actual sizes of an object, for example, are properties separate from the size hints. The latter will never be used in calculations which involve the object's sizes.

SOs may implement any custom behavior for the values they get from these fields. For example, box objects use alignment hints to align its lines/columns inside its container, padding hints to set the padding between each individual child, etc. Edje uses size hints to layout its objects, too. We'll see, further, that edje exposes some of these hint fields at the EDC language.

[2]"Groups", as will be explained further, are declaration blocks of edje data collections which pack together visual elements, or "parts".

5 Playing Edje with Edje Data Collections

We shall begin a deeper study of edje concurrently with the presentation of the language it interacts with. As previously said, the way one describes, to edje, a graphical user interface is through a simple declarative language. A text file is written containing (possibly) several sections, which can describe:

- the images and fonts to be used in the interface,
- what visual elements should the interface have and how they should be laid out,
- actions (or "programs") to occur when the interface is interacted with.

Programs can be further supplemented by employing scripting languages, which add some programability to the interface's behavior[3].

These text files, which we call edje data collections, are commonly named with the `.edc` extension. They are not parsed at this form at run time, though. They must be previously compiled into a binary, more succinct form, which includes the binary blobs of the images, the fonts, etc. As said before, all the interface's descriptions and data are packed together in an EET file, which is customarily named with the `.edj` extension. EDJ files are the output of the `edje_cc` compiler, whose basic invocation is something like:

```
edje_cc input_file.edc output_file.edj
```

At run-time, the application loads the EDJ file(s) through edje's API. Then, edje automatically generates the evas objects necessary to display the interface.

Because of the great simplicity in having all the interface information in only one file, changing an EFL-based application's theme is a mere question of loading it with a different EDJ file. These interchangeable interface versions must only adhere to the same "contract", that is, the way they communicate with the software's back-end. For this reason, people in the e-world often call EDJ files directly with the term **theme**. We are going to use these terms interchangeably here, too.

The syntax for EDC files follows a simple structure of brace-enclosed blocks that can contain properties, more blocks, or both. We shall, now, describe the main building blocks of this language. For each block, we give an example of use at the beginning of its section.

A thorough description of this language can always be consulted at the Edje Data Collection Reference web page, found at `http://docs.enlightenment.org/auto/edje/edcref.html`.

[3]This topic will be readdressed later.

5.1 Macros

`edje_cc` uses the C pre-processor to expand macros. Then, all if its macros may be used, like:

- `#define`
- `#include`
- `#ifdef/#endif`, etc.

5.2 Top-level blocks

The main top level blocks which can go into an EDC file are the ones declaring:

- images,
- fonts,
- data (in string form),
- styles and
- collections.

When declared globally, they are visible throughout the whole edje data collection. Most of them can be defined at more restricted scopes, which can be more convenient in some cases.

5.2.1 Images

```
images {
    image: "filename1.ext" COMP;
    image: "filename2.ext" LOSSY 99;
}
```

The `images` block is used to list each image file that will be used in the theme along with its compression method (if any). This information block, besides the EDC file's top level, can occur inside lower level blocks, easing the maintenance of the file list when the theme is split among multiple files[4].

The general rule to include each image is:

```
image: "[path_to_image_file]" [compression_method] [compression_level];
```

[4]Remember the `#include` macro.

The full path to the directory holding the images can be defined later with `edje_cc`'s "`-id`" option. These image files must match one of the types evas was compiled with support to. Usually png and jpeg formats are supported, but it depends on how evas was compiled.

The compression methods may be the following:

`RAW` uncompressed,

`COMP` lossless compression and

`LOSSY [0-100]` lossy compression with quality from 0 to 100.

5.2.2 Fonts

```
fonts {
    font: "file_name1.ext" "font_alias";
    font: "file_name2.ext" "other_font_alias";
}
```

The `fonts` block is used to list each font file with an **alias** used later in the theme, besides including the font itself into it. Similarly to the `images` one, `fonts` blocks may be included inside other blocks. The full path to the directory holding the fonts can be defined later with `edje_cc`'s "`-fd`" option.

Evas (and consequentially edje) can be compiled with fontconfig support. When it's done, system fonts may be accessed by name and style parameters, like in "`Sans Serif:style=Bold`".

5.2.3 Data

```
data {
    item: "key" "value";
    file: "other_key" "file_name.ext";
}
```

The `data` block is used to pass arbitrary parameters (in the form of strings) from the theme to the application. This can be useful, for example, in a scenario where the software back-end programmer had in mind more them one GUI schema, being them discrepant to the point that visual elements (which translate, in edje, to groups and parts, as will be further explained) exist in one of them but not on the other. The themes could signal that in the form of a data field, and the back-end would them interact with the right visual parts for every case.

The properties defined for this block have two possible forms:

`item: "[key]" "[value]";` This defines a new data item whose value will be the string specified in the `value` field.

file: "[key]" "[filename]"; This defines a new data item whose value will be the contents of the specified file formatted as a single string of text. Naturally, this property only works with plain text files.

This block may also occur inside `group` blocks, thus, having only that scope.

5.2.4 Styles

```
styles {
    style {
        name: "style_name";
        base: "font_size=14 color=#ffffff valign=baseline";
        tag: "br" "  \n";
    }
}
```

The `styles` block contains a list of one or more `style` blocks, which are used to create **style tags** for advanced `TEXTBLOCK` formatting.

`TEXTBLOCK`s are one type of visual elements native to edje which, naturally, exhibit text. We'll get back to them further in this text.

The properties to be filled at a `style` block are:

name: "[style_name]"; The name of this style, to be referenced later in the theme.

base: "[style_properties_string]"; The default style properties that will be applied to the complete text. The syntax of this string is going to be presented further.

tag: "[tag_name]" "[style_properties_string]"; Style to be applied only to text between tags in the form `<tag_name>`, `</tag_name>`.

The `styles` block can also occur inside lower level blocks.

5.2.5 Collections

```
collections {
    images { /*...*/ }
    fonts { /*...*/ }
    styles { /*...*/ }

    group { /*...*/ }
}
```

The `collections` block is used to group `images`, `fonts`, `styles` and `group` blocks altogether, defining a "groups scope". Additional `collections` blocks do not prevent overriding of groups with equal names, though.

5.3 Packing cohesive visual elements together: the `group` block

```
group {
    name: "name_used_by_the_application";
    alias: "another_name";
    min: 100 200;
    max: 100 9999;

    data { /*...*/ }
    script { /*...*/ }
    images { /*...*/ }
    fonts { /*...*/ }
    styles { /*...*/ }

    parts { /*...*/ }
}
```

A `group`, broadly, defines the contents of an edje smart object. An edje SO, which is a very elaborate kind of SO, may correspond to a GUI's whole screen, a widget, or simply a part of a screen. The child elements of an edje object are declared as EDC `parts`. All of evas' built-in objects are supported, obviously, and so are smart objects. Actually, a part may be another group itself, reflecting on edje data collections the recursive object aggregation possibilities evas gives us.

Being an edje object a really smart one, in what it has edje's state machine supporting it, one can declare, at EDCs, **programs**. As said before, these are actions occurring after user (or back-end) interaction with the visual objects composing the edje SO. The `programs` block is presented at section 5.6, while the `script` one is explained at section 5.7.

Some properties one can define at a `group` are:

`name: "[group_name]";` The name that will be used by the application to load the resulting edje object or by the EDC writer himself, when nesting groups. If more than one group, **with the same name**, exist in an EDC file, the last one's definition is going to override the others'.

`alias: "[additional_group_name]";` Additional name to serve as this group's identifier. Defining multiple aliases is supported.

`min: "[width]" "[height]";` The **size hints** for the minimum size the container of the declared parts may have.

`max: "[width]" "[height]";` The size hints for the maximum size the container of the declared parts may have.

5.3.1 Parts

```
parts {
    images { /*...*/ }
    fonts { /*...*/ }
    styles { /*...*/ }

    part {
        name: "part_name";
        type: IMAGE;
        clip_to: "another_part";

        description { /*...*/ }
    }

    programs { /*...*/ }
}
```

`parts` serve as descriptions of the basic visual elements of an edje object. For example, a part could describe a line in a border or a label on a button. As you see, `parts` must occur inside a `parts` block.

Because these are the building units of a GUI in edje, they have a significant number of configurable properties and sub-blocks. We are going to pass through the most basic ones, here, so that you can get started at using the EFL.

Some properties one can define at a `part` are:

`name: "[part_name]";` The symbol by which to reference this element from inside the edje data collection (most probably by programs) or, in some cases, from the application itself. This name must be unique within the `group`.

`type: "[TYPE]";` Here one sets the type of the visual element from among the available ones[5]. The valid types are:

- RECT
- TEXT
- IMAGE
- SWALLOW
- TEXTBLOCK
- GRADIENT
- GROUP
- BOX
- TABLE

[5]By default, a `part` is of type `IMAGE`.

These are, not surprisingly, the evas objects we have previously presented. We'll come back to the `SWALLOW` part type and concept later in the document.

`clip_to: "[another_part_name]";` By setting this property, evas only renders the area of this part that coincides with the clipper container. Overflowing content won't be displayed.

Clipper rectangles are a very common object in interface implementation. They are used to hide totally or partially any object at a given context. In edje, clippers are simple `RECT` objects. When one declares the first object whose `clip_to` property points to it, edje treats it specially. When visible, they won't actually be drawn. Their clipped objects, if visible, will have their sub-areas which coincide with the clipper's area drawn with their color **modulated** by the clipper's one. If a clipper is not visible, the objects clipped to it will not have the coinciding sub-areas drawn. We talk about objects' visibility and color at section 5.4.

The **order** in which parts are declared inside a group is very important. It defines **object stacking** for the interface being declared: when a part is written, it will be placed on top of all the others whose blocks appear before in the EDC file. Clipper logic bypasses it: objects that are clipped can be declared after their clippers and they'll still be clipped.

Lastly, edje allows one to build **scalable interfaces**. This is a powerful feature that comes in handy when, for example, we have to ship different sizes of the same interface (targeted at different screen sizes). This scaling could be based, also, on the screen DPI of that devices. The “finger size” property elementary exposes makes use of edje's scaling API functions, as some might have thought.

The way one tells edje whether and how to scale the interface is by beans of **scale factors**. These are float numbers which multiply some objects' sizes. Scaling affects the values of `min`/`max` part properties and font sizes, for example. There is a **global scaling factor**, which will affect all edje objects, and a per evas object one – the **individual scaling factor**. If the latter is set, it overrides the former's effect. This is so that different visual objects may scale with distinct proportions. The scaling factors are set through edje's API.

Besides giving the programer/designer this possibility, it also lets us be **selective** on what is going to be scaled. Some visual elements may not look good scaled, like a button border, for example. If only the button's contents get larger, it may look better. They way we tell edje which objects to scale or not is by this `part` property:

`scale: "[1 or 0]";` Specifies whether the part is scalable or not, being the default value `0` (not scalable).

The only (new) block occurring inside a `part` we're going to describe is the `description` one.

5.4 Interface objects' states

```
part {
    description {
        inherit: "another_description" 0.0;
        state: "description_name" 0.0;
        visible: 1;
        min: 0 0;
        max: -1 -1;
        color: 255 0 0 128;

        rel1 { /*...*/ }
        rel2 { /*...*/ }
    }
}
```

Every part can have one or more `description` blocks. With them, one defines the **states** a part can be at[6], each state having its own layout and style properties. From now on, you can think of a part state every time we mention the term "description" (except when the context tells us differently).

This EDC block has lots of properties, being the main ones those that follow:

`state: "[description_name]" "[an_index]";` This is the symbol to identify a description inside a given part. Multiple descriptions are used to declare different states of the same part. For a button, for example, there could be states matching contexts like "clicked", "invisible", etc. The `description_name` string actually defines a "description namespace". A complete state identifier must contain, also, a float number, ranging from 0.0 to 1.0. This schema just adds the possibility of simpler naming when one deals with really similar states.

All parts must have a description named "`default 0.0`", which edje takes as the initial state. If not explicitly declared, edje creates this state for the part, with default properties.

`inherit: "[another_description_name]" "[another_description_index]";` This is an optional property by which one makes a state inherit another states' properties, with the possibility of overriding any of them. It exists to reduce the amount of necessary code for simple state changes.

Because of the way it is implemented[7], the correct usage of this property is to declare it right after the `state` one, only once per description, naturally.

`visible: "[0 or 1]";` Takes a boolean value specifying whether the part is visible (`1`) or not (`0`) at this state, being the former the default value. Non-visible parts do not emit signals (see section 5.5).

`min: "[width]" "[height]";` The size hints for the minimum size this part may have at this state.

[6]The transitions we have talked about take parts from a given state to another one.
[7]Other properties are overridden at the time this property is parsed, at compile time, with no dependency resolution.

max: "[width]" "[height]"; The size hints for the maximum size this part may have at this state.

Note:

For **parts**, a way of fixing their sizes (only subject to scaling to have sizes changed) is to declare their **min** size hints identically to the **max** ones.

color: "[red]" "[green]" "[blue]" "[alpha]"; Sets the "main" color to the specified RGB (and alpha channel) values, which range from `0` to `255`. For transparency, the former means totally trasparent, while the latter means opaque.

Usually you'll use this attribute for `TEXT` and `RECT` objects' states, where the "main" color is their unique one. If set for objects which themselves have colors, like the `IMAGE` one, those colors get **modulated** by this one. For rectangles, the default value of this property is "`255 255 255 255`" (opaque white).

Note:

The most common use of clippers is to only hide things, not changing their colors when they must be shown. To get this behavior, the clipper rectangle's color must be set to opaque white, which is the neutral element in color multiplication. This justifies the default value for a `RECT`'s color.

5.4.1 Sizing and positioning in edje

```
% TODO: place a "to:" property in the example

description {
    rel1 {
        relative: 0.0 0.0;
        offset: 0 0;
    }
    rel2 {
        relative: 1.0 1.0;
        offset: -1 -1;
    }
    align: 0.5 0.5;
}
```

The reader may have noticed two new EDC blocks occurring inside the `description` block's example snippet: `rel1` and `rel2`. These are EDC blocks you'll use a lot when writing an edje interface. That is because it's with them one describes, to edje, the size and position of the parts at a given state.

`rel1` and `rel2` specify, respectively, the coordinates of **the top left and the bottom right corners** of the part's container relatively to another objects' **top left coordinate**. This reference object may be the enclosing group's container or any other part's container, when convenient (given that it's a part in the same group as the one in question). The area defined by a container is, normally, exactly the area an object will occupy at a given state.

These are the properties of these blocks:

`to: "[another_part_name]";` Makes the part's corner to be positioned **relatively to another part's top left one**. This affects the behavior of the `relative` property.

By default (if there's no explicit "`to`" property), the corner will be relative to the enclosing group's (top left) one.

`relative: "[x_axis_relative]" "[y_axis_relative]";` This property specifies, for each axis, how much of the relative "`to`" object's **size**, in that axis, edje must sum, **from the origin corner**, in order to place **this part's corner**. These quantities (or distances) are given as **percentages**[8], which, at the end, translate to an integer number of pixels. If not set, it is assumed that this property has the values "`0.0 0.0`", for `rel1` blocks, and the values "`1.0 1.0`", for `rel2` ones.

`offset: "[x_axis_offset]" "[y_axis_offset]";` This property gives quantities to sum from the origin corner's coordinates, for each axis, **besides the amount specified by the `relative` property's logic**, to place this part's corner. The quantities here, though, are **absolute** (integer values, naturally). If not set, it is assumed that this property has the values "`0 0`", for `rel1` blocks, and the values "`-1 -1`", for `rel2` ones.

When placing an object on the canvas, edje gives us the possibility of relativizing it **independently for each axis**. When this is desired, the "`to`" property must be exchanged for one (or both) of the following ones:

`to_x: "[another_part_name]";` When set, this property will make edje to consider the (top left) corner belonging to the object named "`another_part_name`" as the the origin one, when applying the `relative` block's functionality, **but just for the x axis**.

`to_y: "[another_part_name]";` The same said for `to_x`, but applied to the **y axis**.

If **just one** of the two descriptions above are declared, edje will use the enclosing group's top left corner to the calculations on the missing axis.

[8]Actually, they are not strictly percentages, as any float number, negative or positive, will do. Edje will multiply this value with the corresponding size. When negative, it'll mean going leftwards, on the x axis, and going upwards, on the y axis.

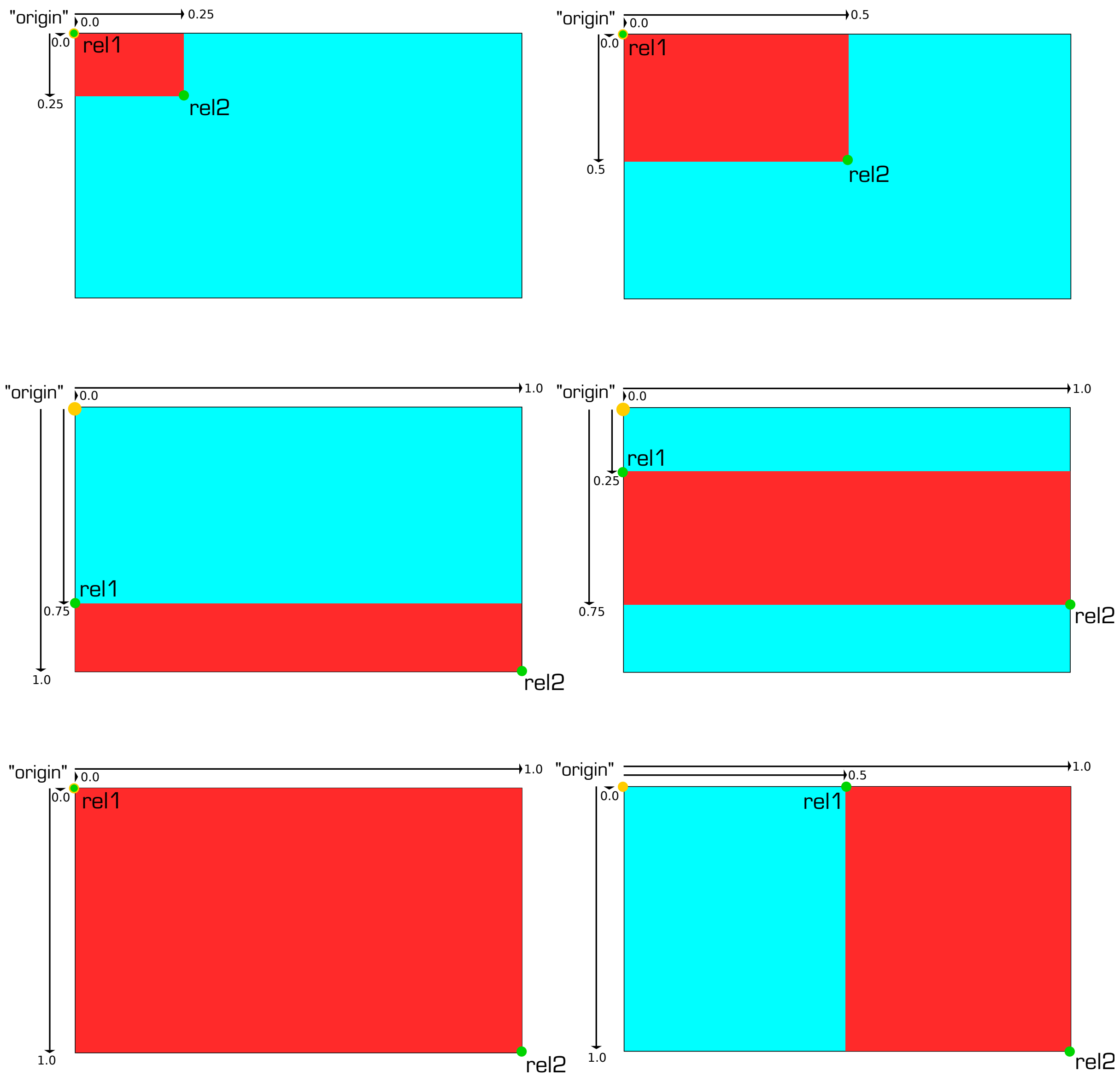

Figure 1: Here, we have six situations of the red object being placed relatively to the blue one, illustrating the use of the `relative` property.

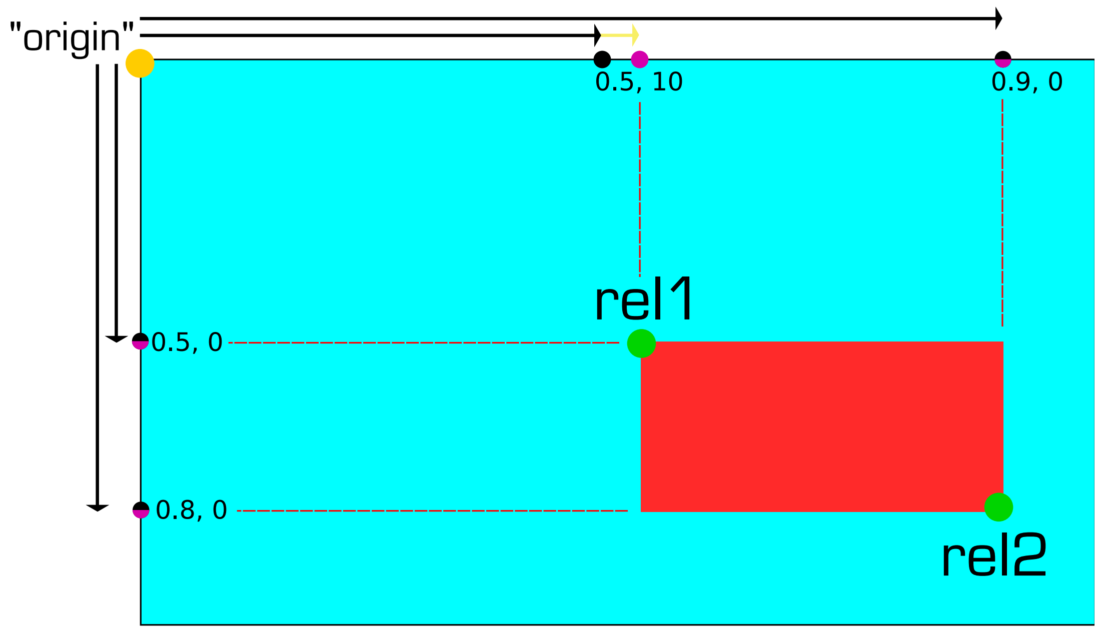

Figure 2: One more positioning of the red rectangle relative to the blue one. Black dots represent where the `relative` property would take to (for each axis), while the purple ones represent the fixed offset added by the `offset` property. If they are too close one to each other, a two-colored dot is shown.

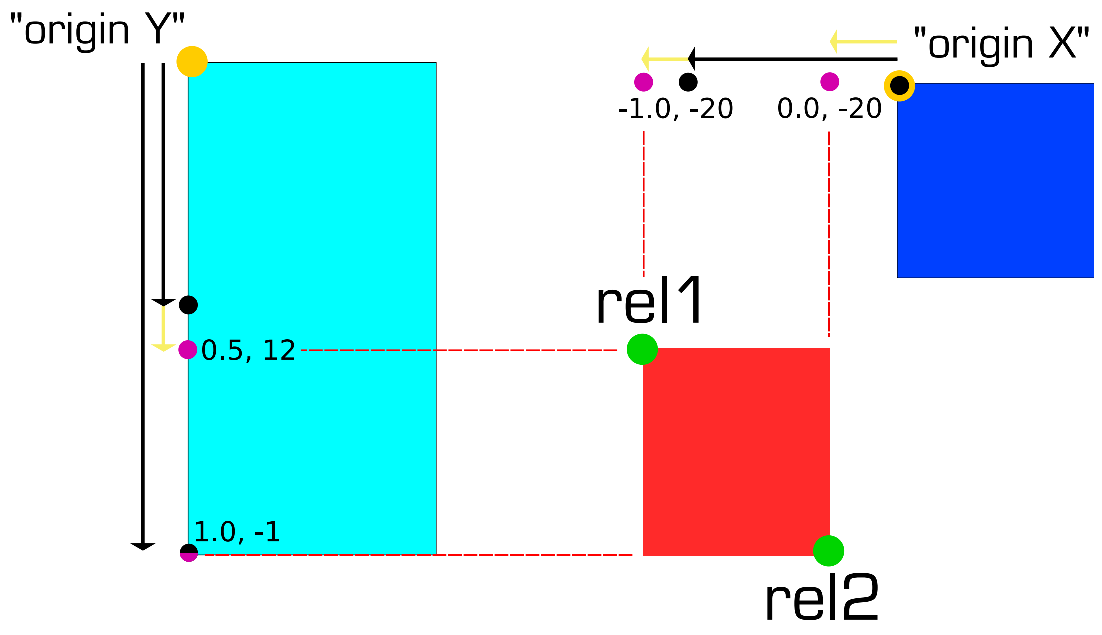

Figure 3: Now the `to_x` and `to_y` options are exemplified, together with a negative "percentage".

Note:

It is important to highlight that `rel1` and `rel2` give the placement of the part's corners in **pixel coordinates**. However, the `relative` property deals with part **sizes**. Because pixels are indexed starting from zero, you'll commonly see and use **offset corrections** on `rel2` blocks. The code snippet which opens section 5.4.1 is a common EDC idiom: a part's state in which it will have the exact size of its enclosing group. This also justifies the default `offset` property values for `rel2` blocks.

Figure 1 shows the use of the `relative` property. Figures 2 and 3 illustrate a more elaborate use of the `rel1`/`rel2` blocks. You could also think of a corner's final pixel coordinate tuple as a simple account:

```
x_dest = x_orig + x_axis_relative * x_width + x_axis_offset
y_dest = y_orig + y_axis_relative * y_height + y_axis_offset
```

We left one of the `description`'s properties to be presented after the ones you have seen because it relates specifically to parts positioning:

`align: "[x_axis_alignment]" "[y_axis_alignment]";` Edje objects will have their sizes limited by the ones declared at the `min`/`max` properties, if they are set. However, every part has a container, which is the area reserved for by the `rel1`/`rel2` declarations. When not limited by size hints, the parts will have the exact sizes of these containers. But if they have those limits set, there may be space **left over** or **lacking** in these containers. These are the contexts for which the `align` property apply. This property will place the part relatively, along both axis, inside its container, when it can't have the actual container's size.

The values may range from `0.0` to `1.0` and the default value is "`0.5 0.5`".

An account, which gives the coordinates of the top left pixel of a part being placed with alignment may help to figure out the `align` property's behavior:

```
x_dest = x_orig + (container_width - part_width) * x_axis_alignment

y_dest = y_orig + (container_height - part_height) * y_axis_alignment
```

Figure 4 helps to understand it, too.

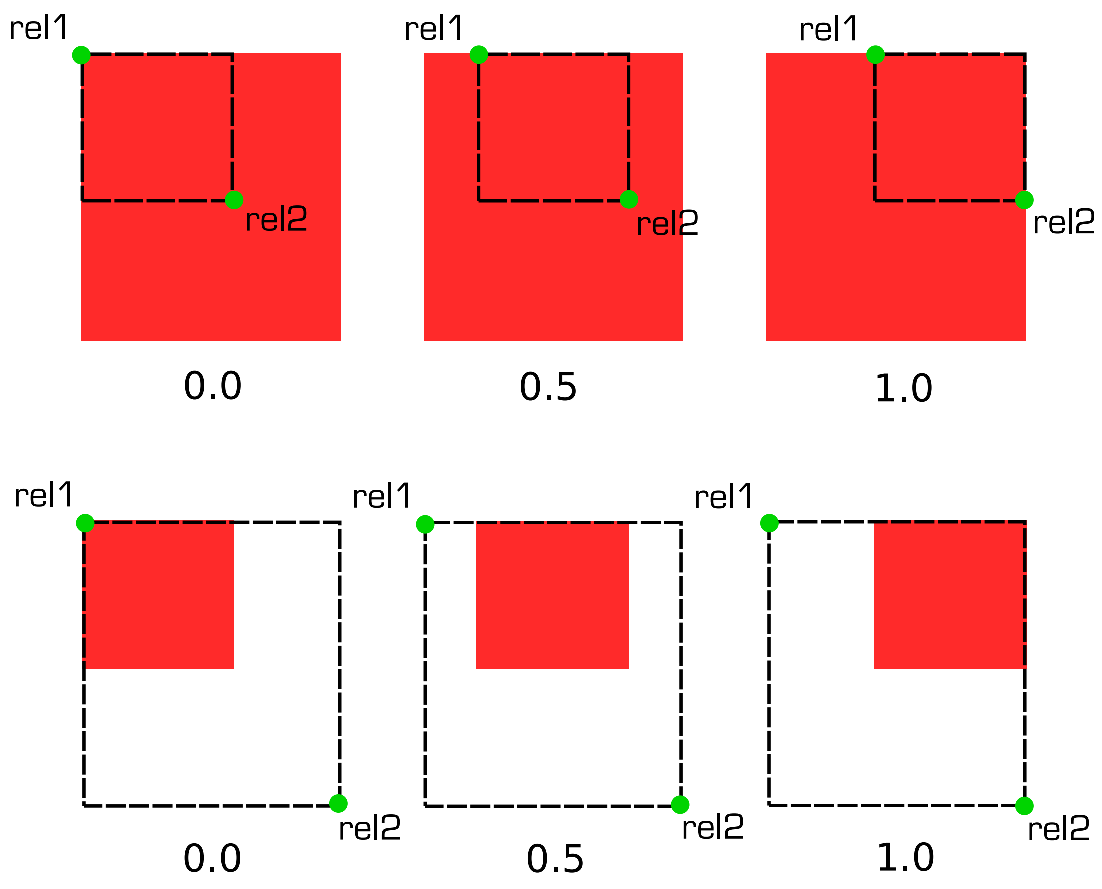

Figure 4: The top row shows a (red) object whose minimum size is bigger than its respective container. The numbers below it illustrate its placement, relative to the container, for those three x axis alignment values. The bottom row illustrates, analogously, the case in which the object's maximum size is smaller than the container's. For all the examples, the alignment in the y axis is zero.

5.5 Edje signals

Edje signals are pieces of information transiting back and forth the application's code and its interface and also inside edje, internally. They extend the possibilities of interfacing with this library, beyond its API.

In terms of implementation, they are merely function callbacks with a defined signature. It includes two string arguments, which we call **emission** and **source**.

The signal's emission string must tell what the signal means in some way. For example, "`mouse,in`" would be suitable to indicate the event of the mouse cursor being over a given interface object. It is common practice to use this kind of syntax for the emission string: incrementally refined information tokens separated by commas.

Each signal must come from some place, be it a `part`, a `program` (explained at section 5.6) or the the application's back-end. The place from where a signal comes is all the source string is about. If a user left clicks on an image element whose part is named "`button`", a signal with emission string "`mouse,clicked,1`" and source string "`button`" is generated from that part. This is an example of an **edje's internal signal**. Other contexts at which the library emits signals are:

- when a theme is loaded,
- when the mouse cursor moves over an object,
- when an object is resized, moved, etc.

The way an application's back-end receives signals from its interface is by registering to them. In C, this is done by the `edje_object_signal_callback_add()` edje's API function.

5.6 Programs and transitions

```
parts {
    programs {
        program {
            name: "program_name";
            signal: "signal_name";
            source: "part_name";
            action: STATE_SET "state_name" 0.3;
            transition: LINEAR 0.5;
            target: "another_part";
            target: "and_another_part";
            after: "another_program_name";
            after: "and_another_program_name";
        }
    }
}
```

Programs, broadly, define how your interface reacts when it is interacted with. This interaction may be in the form of user input, signals emitted from the application's back-end or internal edje signals. In fact, user input is translated, by edje, into edje signals too, so that **most of the interaction with an edje interface, if not directly made by edje's API, is by means of edje signals**.

Programs are one of the ways of changing edje objects' states. Between other possible actions, programs may also emit signals, like previously said.

The main properties one can set at this block are:

`name: "[program_name]";` The program's unique identifier.

`signal: "[emission_string]";` Programs may be triggered by signals, be they internal or sent from the application's code. When this possibility of invocation is desired, this is the property which **must** be filled. The arriving signal's emission string must match the "`emission_string`" one in order to the program to run.

These signal name strings may be globbed with "`*`" wildcard characters. For example, if we have "`mouse,clicked,*`", clicking any mouse button will rise signals that match this pattern and, thus, start the program.

`source: "[source_string]";` More precisely, the `source` property's string will be paired with the "`emission_string`" one at the time edje tries to match the arriving signal with the one the program is triggered with.

These signal source strings may be globbed like the emission ones. For example, if we have "`button-*`", signals from any part or program whose name is prefixed with that string are accepted (if their emission strings match, too).

If left empty, this property will implicitly contain the empty string.

`action: "[type]" "[param1]" "[param2]";` Programs may provoke **actions**. The main ones, which were already mentioned, are to change objects' states and to emit signals. Here goes the syntax for these two contexts:

`action: STATE_SET "[state_name]" INDEX;` Take the target parts to the named state.

`action: SIGNAL_EMIT "[emission]" "[source]";` Emit the given signal.

`transition: "[type]" "[duration]";` If the program has the `STATE_SET` action, edje gives the possibility of gradual "mutation" of the visual object from initial to final states. The individual steps can occur, through time, with different built-in patterns. These patterns are what we call edje transitions. Their names, listed below, must go into the `type` field, while at the `duration` one a float number must be given. This is the number of seconds the transition will take.

`LINEAR` The "frames" will be displayed with equal time.

`SINUSOIDAL` The rate of frame displaying changes in a sinusoidal fashion, i. e., faster, slower, then faster again.

`ACCELERATE` Frame times always decreasing during the transition.

`DECELERATE` The opposite of the last type.

`target: "[target_part]";` If the program has the `STATE_SET` action, this is the property with which one specifies a `part` to act upon. Multiple target parts may be declared (with multiple `target` entries in the program). `SIGNAL_EMIT` actions do not have target parts, naturally.

`after: "[program_occurring_after]";` This is a way of chaining programs sequentially. This comes in handy when, for example, you have to perform more the one action at a given interaction context. The program whose name matches the `program_occurring_after` string will start as soon as this one ends. Multiple after statements may be specified per program and all of them will run **in parallel**.

5.7 Scripting and edje

We'll give, now, a few words on how edje relies on scripting environments to supplement the capabilities of an edje object's behavior. This section will probably need updates soon, because these scripting environments have been worked on[9].

We start this section by presenting the way scripts are invoked externally from edje objects.

5.7.1 Edje messages

At the beginning of section 5.6, there is a reason why we've said that **most of** the interaction with edje interfaces occur in that forms. There is a **third** possibility: **edje messages**.

Messages differ from signals in (at least) two ways:

- They are **unidirectional**. Signals, when emitted from edje, can be caught inside it and trigger programs, besides being caught from the application back-end, when registered to. It works analogously, when the back-end is emitting a signal. Messages, on the other hand, are meant to be passed in only one of those directions.
- They may carry more types of data, besides strings, like integer and float numbers.

At the application back-end's side, messages will be processed if (message) handler functions are set for edje objects. At the interface's side, messages must be handled by edje's **scripting environment**.

[9]Lua scripting inside edje is going to be working shortly.

5.7.2 Embryo

The edje library has, as dependency, a small library called **embryo**. Embryo implements a scripting language, which we call by this same name, with C-like syntax. It was based on SMALL, which is now called Pawn. Embryo gives us some built-in functions and a way of defining small functions inside EDC files. These functions are the contents of the `script` blocks.

The run-time environment provided by embryo is totally **sandboxed**. This limits its interfacing language to act only upon interface elements, basically.

Besides occurring at `group` scopes, **where message handler functions must be declared**, when applicable, `script` blocks may also occur inside `programs`. The code snippet which follows illustrates this use.

```
program {
   name: "part_name";
   signal: "show";
   source: "*";
   script {
      set_int(is_mouse_down, 0);
      set_int(enabled, 1);
   }
```

We're not going into embryo language's details here. For this context, all we say is that `set_int` is embryo's way of assigning to a global integer variable, which in this case should have been declared at a group level `script` block. As you see, embryo facilitates the task of keeping state in the interface.

When a program has such block, it mustn't have the `action` one. The functions listed at this `script` block form the program's action in this case. Naturally, besides setting (or referencing) global variables, global functions could be used.