The EFL Cookbook

Various
Edited by Ben technikolor Rockwood

The EFL Cookbook

by Various and Ben technikolor Rockwood

Stuff.

Table of Contents

List of Examples

Chapter 1. Introduction

With the decision to re-write Enlightenment for DR17, came the decision to build a support infrastructure that could support Enlightenment well into the future and other projects needing a solid base of libraries. This formed the basis of the Enlightenment Foundation Libraries (EFL).

The EFL has grown to encompass many of the features required by the window manager development and many of the applications that will be run on top of that WM. There are currently libraries to handle transparency, configuration, video, core event loops, and interface abstraction.

This book is an attempt to be your road map through this foundation.

Chapter 2. Imlib2

Imlib2 is the successor to Imlib. It is not just a newer version - it is a completely new library. Imlib2 can be installed alongside Imlib 1.x without any problems since they are effectively different libraries - but they Have very similar functionality.

Imlib2 can do the following:

- Load image files from disk in one of many formats
- Save images to disk in one of many formats
- Render image data onto other images
- Render images to an X-Windows drawable
- Produce pixmaps and pixmap masks of Images
- Apply filters to images
- Rotate images
- Accept RGBA Data for images
- Scale images
- Alpha blend Images on other images or drawables
- Apply color correction and modification tables and factors to images
- Render images onto images with color correction and modification tables
- Render truetype anti-aliased text
- Render truetype anti-aliased text at any angle
- Render anti-aliased lines
- Render rectangles
- Render linear multi-colored gradients
- Cache data intelligently for maximum performance
- Allocate colors automatically
- Allow full control over caching and color allocation
- Provide highly optimized MMX assembly for core routines
- Provide plug-in filter interface
- Provide on-the-fly runtime plug-in image loading and saving interface
- Fastest image compositing, rendering and manipulation library for X

If what you want isn't in the list above somewhere then likely Imlib2 does not do it. If it does, it likely does it faster than any other library you can find (this includes gdk-pixbuf, gdkrgb, etc.) primarily because of highly optimized code and a smart subsystem that does the dirty work for you and picks up the pieces for you so you can be lazy and let Imlib2 do all the optimizations for you.

Imlib2 provides a powerful engine for image manipulation and rendering. Using loaders it can handle a variety of image formats including BMP, GIF (via unGIF), JPEG, PNG, PNM, TGA, TIFF, XPM and more.

Recipe: Image Watermarking

Ben technikolor Rockwood `<benr@cuddletech.com>`

With so many individuals putting so many images online its easy to forget where they came from and hard to ensure that copyrighted material isn't inadvertently misused. Simply adding a watermark image, such as your sites logo, to each of your images can solve both these problems. But adding watermarks manual is a long and repetitive task. Imlib2 can easily be used to solve this problem. What we need to do is take an input image, and then specify a watermark image (your logo), position the watermark on the input image and then save it out to a new image which we'll use on the site. The app would look something like this:

Example 2.1. Imlib2 WaterMark Program

```
#define X_DISPLAY_MISSING
#include <Imlib2.h>
#include <stdio.h>

int main(int argc, char **argv){

    Imlib_Image image_input, image_watermark, image_output;
    int     w_input, h_input;
    int     w_watermark, h_watermark;
    char    watermark[] = "watermark.png";

    if(argc > 1)  {
        printf("Input image is: %s\n", argv[1]);
        printf("Watermark is: %s\n", watermark);
    }
    else {
        printf("Usage: %s input_image output_imagename\n", argv[0]);
        exit(1);
    }

    image_input = imlib_load_image(argv[1]);
    if(image_input) {
        imlib_context_set_image(image_input);
        w_input = imlib_image_get_width();
        h_input = imlib_image_get_height();
        printf("Input size is: %d by %d\n", w_input, h_input);
        image_output = imlib_clone_image();
    }

    image_watermark = imlib_load_image(watermark);
    if(image_watermark) {
        imlib_context_set_image(image_watermark);
        w_watermark = imlib_image_get_width();
        h_watermark = imlib_image_get_height();
        printf("WaterMark size is: %d by %d\n",
                w_watermark, h_watermark);
    }

    if(image_output) {
        int dest_x, dest_y;

        dest_x = w_input - w_watermark;
        dest_y = h_input - h_watermark;
        imlib_context_set_image(image_output);

        imlib_blend_image_onto_image(image_watermark, 0,
                0, 0, w_watermark, h_watermark,
                dest_x, dest_y, w_watermark, h_watermark);
        imlib_save_image(argv[2]);
        printf("Wrote watermarked image to filename: %s\n", argv[2]);
    }


        return(0);
}
```

Looking at the example, we first do some really basic argument checking, accepting an input image as the first argument and an output image name for our watermarked copy. Using `imlib_load_image()` we load the input image and then grab its dimensions using the get functions. With the `imlib_clone_image()` function we can create a copy of the input image, which will be the base of our watermarked output. Next we load the watermark image, and notice that we then use `imlib_context_set_image()` to change the context from the input image (image_input) to the watermark image (image_watermark). Now we grab the images dimensions as well. In the final block we do two simple calculations to determine the positioning of the watermark on the output image, in this case I want the watermark on the bottom right-hand corner. The magic function that really does the work in this program is `imlib_blend_image_onto_image()`. Notice that we change context to the output image before proceeding. The blend function will, as the name suggests, blend two images together which we refer to as the source and destination image. The blend function blends a source image onto the current image context which we designate as the destination. The arguments supplied to `imlib_blend_image_onto_image()` can look tricky, we need to tell it which source to use (the watermark), whether to merge the alpha channel (0 for no), the dimensions of the source image (x, y, w, h) and the dimensions of the destination image (x, y, w, h). You'll notice that in the example we set the x and y positions of the source (watermark) image to 0 and then use the full width. The destination (input image) is set to the bottom right hand corner minus the dimensions of the watermark, and then we specify the width and height of the watermark. Finally, we use the `imlib_save_image()` function to save the output image.

While this example should be significantly improved for real use, it outlines the basics of Imlib2 blending to solves a very common problem efficiently.

Recipe: Image Scaling

dan dj2 sinclair `<zero@perplexity.org>`

As more people gain the ability to put images on the Internet it is often desired to scale those images to a smaller size to reduce bandwidth. This can easily be solved using a simple Imlib2 program.

This recipe takes the input image name, the new width, new height and the output image name and scales the input image by the given values, saving it back to the output image.

Example 2.2. Image Scaling

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img, out_img;
    int w, h;

    if (argc != 5) {
        fprintf(stderr, "Usage: %s [in_img] [w] [h] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);
```

```
    w = atoi(argv[2]);
    h = atoi(argv[3]);
    out_img = imlib_create_cropped_scaled_image(0, 0, imlib_image_get_width(),
                                                imlib_image_get_height(), w, h );
    if (!out_img) {
        fprintf(stderr, "Failed to create scaled image\n");
        return 1;
    }

    imlib_context_set_image(out_img);
    imlib_save_image(argv[4]);
    return 0;
}
```

There is minimal argument checking done by this example, just make sure we have the correct number of arguments.

The source image is loaded with a call to `imlib_load_image()` which will load the image data into memory. If the call fails, NULL will be returned. Once we have the image data we need to set the image to be the current context. This lets Imlib2 know which image the operations will be preformed upon. This is done by calling `imlib_context_set_image()`. Once the image is set as the current context we can proceed with the scale. This is done by calling `imlib_create_cropped_scaled_image()` which takes as arguments, the starting x position, starting y position, the source width, source height, and the scaled width and scaled height. The reason we pass in the source information is that this function can also crop your image if desired. To crop, just modify the x, y, source width and source height as desired. This will result in a new image being produced out_img. If the scale fails, NULL will be returned. We then set the out_img to be the current context image and issue the save command, `imlib_save_image()`.

Although this program is simple, it shows the simplicity of image scaling using the Imlib2 API.

Recipe: Free rotation

dan dj2 sinclair `<zero@perplexity.org>`

It is sometimes desirable to rotate an image to some specific angle. Imlib2 makes this process easy. This example attempts to shows how its done. If you wish to rotate the image on angles of 90 degrees, see the 90 degree rotation recipe as this recipe will leave a black border around the image.

Example 2.3. Free rotation

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img, out_img;
    float angle = 0.0;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s [in_img] [angle] [out_img]\n", argv[0]);
```

```
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);

    angle = (atof(argv[2]) * (M_PI / 180.0));
    out_img = imlib_create_rotated_image(angle);
    if (!out_img) {
        fprintf(stderr, "Failed to create rotated image\n");
        return 1;
    }

    imlib_context_set_image(out_img);
    imlib_save_image(argv[3]);
    return 0;
}
```

After some simple argument checking we get into the Imlib2 work. We begin by loading the specified image into memory with `imlib_load_image()` giving the image name as a parameter. We then take that image and make it the current context with `imblib_context_set_image`. Contexts are used by Imlib2 so it knows what image to work on. Whenever you wish to make imlib calls on an image it must be set as the current context. We then convert the given angle from Degrees to Radians as Imlib2s rotation function works in Radians. The rotation is then done with `imlib_create_rotated_image()`. The rotation function will return the new image. In order to save the new image we need to set it as the current context, again with `imlib_context_set_image()`. Then a simple call to `imlib_save_image()` giving the name of the output file saves the new, rotated image.

The rotation function in Imlib2 will place a black border around the image to fill in any blank space. This border is calculated so that the rotated image can fit in the output. This will case borders around the output image even if you rotate by 180 degrees.

Recipe: 90 degree Image rotation

dan dj2 sinclair `<zero@perplexity.org>`

With a digital camera it is sometimes desirable to rotate you image by: 90, 180 or 270 degrees. This recipe will show how to do this easily with Imlib2. This recipe, also, will not put the black borders around the image as is seen in the free rotate example.

Example 2.4. 90 degree Image rotation

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img;
    int dir = 0;
```

```
    if (argc != 3) {
        fprintf(stderr, "Usage: %s [in_img] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
    }
    imlib_context_set_image(in_img);
    imlib_image_orientate(1);
    imlib_save_image(argv[2]);
    return 0;
}
```

After some minimal error checking we load the image to be rotated with a call to `imlib_load_image()`. This function accepts a filename and returns the Imlib_Image object, or NULL on load error. Once the image is loaded we set it as the current context image, the image Imlib2 will do its operations upon, with `imlib_context_set_image()`. The rotation is done through the call to: `imlib_image_orientate()`. The parameter to _orientate changes the amount of rotation. The possible values are: [1, 2, 3] meaning a clockwise rotation of [90, 180, 270] degrees respectively. Once the image is rotated we call `imlib_save_image()` giving the filename of the new image to have Imlib2 save the rotated image.

With this example in your hands you should be able to quickly rotate images on 90 degree intervals using Imlib2.

Recipe: Image Flipping

dan dj2 sinclair <zero@perplexity.org>

Imlib2 contains functions to do image flipping. This can be done either horizontally, vertically or diagonally. This recipe will show how to implement this functionality.

Example 2.5. Image Flipping

```
#define X_DISPLAY_MISSING

#include <Imlib2.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char ** argv) {
    Imlib_Image in_img;
    int dir = 0;

    if (argc != 4) {
        fprintf(stderr, "Usage: %s [in_img] [dir] [out_img]\n", argv[0]);
        return 1;
    }

    in_img = imlib_load_image(argv[1]);
    if (!in_img) {
        fprintf(stderr, "Unable to load %s\n", argv[1]);
        return 1;
```

```
    }
    imlib_context_set_image(in_img);

    dir = atoi(argv[2]);
    switch(dir) {
        case HORIZONTAL:
            imlib_image_flip_horizontal();
            break;

        case VERTICAL:
            imlib_image_flip_vertical();
            break;

        case DIAGONAL:
            imlib_image_flip_diagonal();
            break;

        default:
            fprintf(stderr, "Unknown value\n");
            return 1;
    }
    imlib_save_image(argv[3]);
    return 0;
}
```

This example does some minimal argument checking to begin, then loads the input image using the `imlib_load_image()` function, passing the filename to load. `imlib_load_image()` will either return the Imlib_Image object, or NULL if the load fails. Once we have the image object we set it as the current context image with a call to `imlib_context_set_image()`. This tells Imlib2 that this is the image we want to work with and all Imlib2 operations will work with this image. With the image context setup we decide on the type of flip we want to preform. This is done with one of the calls: `imlib_image_flip_horizontal()`, `imlib_image_flip_vertical()`, and `imlib_image_flip_diagonal()`. The diagonal flip essentially grabs the top left corner and makes it the bottom right corner. The top right becoming the bottom left. Once the image is flipped we call `imlib_save_image()` giving it the new filename and we're done.

This should give an example of image flipping with Imlib2. It will need enhancements before being put into a real app but the base is there.

Chapter 3. EVAS

Evas is a hardware-accelerated canvas API for X-Windows that can draw anti-aliased text, smooth super and sub-sampled images, alpha-blend, as well as drop down to using normal X11 primitives such as pixmaps, lines and rectangles for speed if your CPU or graphics hardware are too slow.

Evas abstracts any need to know much about the characteristics of your XServer's display, what depth or what magic visuals, it has. The most you need to tell Evas is how many colors (at a maximum) to use if the display is not a truecolor display. By default it is suggested to use 216 colors (as this equates to a 6x6x6 color cube - exactly the same color cube Netscape, Mozilla, gdkrgb etc. use so colors will be shared). If Evas can't allocate enough colors it keeps reducing the size of the color cube until it reaches plain black and white. This way, it can display on anything from a black and white only terminal to 16 color VGA to 256 color and all the way up through 15, 16, 24 and 32bit color.

Recipe: Key Binds, using EVAS Key Events

Ben technikolor Rockwood

Many applications can benifit from providing key binds for commonly used operations. Whether accepting text in ways the EFL doesn't normally expect or just a way to bind the + key to raise the volume of a mixer, keybinds can add just the bit of functionality that makes your app a hit.

The following code is a simple and complete application that is useful in exploring keybinds using EVAS event callbacks. It creates a black 100 by 100 pixel window in which you can hit keys.

Example 3.1. Key grabbing using EVAS Events

```
#include <Ecore_Evas.h>
#include <Ecore.h>

#define WIDTH 100
#define HEIGHT 100

        Ecore_Evas  *   ee;
        Evas        *   evas;
        Evas_Object *   base_rect;


static int
main_signal_exit(void *data, int ev_type, void *ev)
{
   ecore_main_loop_quit();
   return 1;
}

void mouse(void *data, Evas *e, Evas_Object *obj, void *event_info) {
        Evas_Event_Mouse_Down *ev;

        ev = (Evas_Event_Mouse_Down *)event_info;
        printf("You pressed button: %d\n", ev->button);
}

void key_down(void *data, Evas *e, Evas_Object *obj, void *event_info) {
        Evas_Event_Key_Down *ev;

        ev = (Evas_Event_Key_Down *)event_info;
        printf("You hit key: %s\n", ev->keyname);
```

```
}


void move(void *data, Evas *e, Evas_Object *obj, void *event_info) {
        printf("Entered callback: mouse move\n");
}



int main(){
        ecore_init();
        ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT,
                        main_signal_exit, NULL);

   ee = ecore_evas_software_x11_new(NULL, 0,  0, 0, WIDTH, HEIGHT);
        ecore_evas_title_set(ee, "EVAS Callback Test");
        ecore_evas_borderless_set(ee, 0);
        ecore_evas_show(ee);

   evas = ecore_evas_get(ee);
        evas_font_path_append(evas, "data/");

   base_rect = evas_object_rectangle_add(evas);
        evas_object_resize(base_rect, (double)WIDTH, (double)HEIGHT);
        evas_object_color_set(base_rect, 0, 0, 0, 255);
        evas_object_focus_set(base_rect, 1);
        evas_object_show(base_rect);

        evas_object_event_callback_add(base_rect,
                        EVAS_CALLBACK_MOUSE_DOWN, mouse, NULL);
        evas_object_event_callback_add(base_rect,
                        EVAS_CALLBACK_MOUSE_MOVE, move, NULL);
        evas_object_event_callback_add(base_rect,
                        EVAS_CALLBACK_KEY_DOWN, key_down, NULL);

        ecore_main_loop_begin();

        ecore_evas_shutdown();
        ecore_shutdown();

        return 0;
}
```

You can compile this example in the following manner:

Example 3.2. EVAS Keybind Compile

```
gcc `evas-config --libs --cflags` `ecore-config --libs --cflags` \
> key_test.c -o key_test
```

Chapter 4. Ecore

What is Ecore? Ecore is the core event abstraction layer and X abstraction layer that makes doing selections, Xdnd, general X stuff, and event loops, timeouts and idle handlers fast, optimized, and convenient. It's a separate library so anyone can make use of the work put into Ecore to make this job easy for applications.

Ecore is completely modular. At its base is the event handlers and timers, and initialization and shutdown functions. The abstraction modules for Ecore include:

- Ecore X
- Ecore FB
- Ecore EVAS
- Ecore TXT
- Ecore Job
- Ecore IPC
- Ecore Con
- Ecore Config

Ecore is so modular and powerful that it can be extremely useful even in non-graphics programing by itself. As an example, several web servers have been written that were based solely on Ecore and the Ecore_Con module for abstract socket communication.

Recipe: Ecore Config Introduction

dan dj2 sinclair `<zero@perplexity.org>`

The Ecore_Config module provides the programmer with a very simple way to setup configuration files for their program. This recipe will give an example of how to integrate the beginnings of Ecore_Config into your program and use it to get configuration data.

Example 4.1. Simple Ecore_Config program

```
#include <Ecore_Config.h>

int main(int argc, char ** argv) {
    int i;
    float j;
    char *str;

    if (ecore_config_init("foo") != ECORE_CONFIG_ERR_SUCC) {
        printf("Cannot init Ecore_Config");
        return 1;
    }

    ecore_config_int_default("/int_example", 1);
    ecore_config_string_default("/this/is/a/string/example", "String");
    ecore_config_float_default("/float/example", 2.22);

    ecore_config_load();

    i = ecore_config_int_get("/int_example");
    str = ecore_config_string_get("/this/is/a/string/example");
    j = ecore_config_float_get("/float/example");
```

```
    printf("str is (%s)\n", str);
    printf("i is (%d)\n", i);
    printf("j is (%f)\n", j);

    free(str);

    ecore_config_shutdown();
    return 0;
}
```

As you can see from this example the basic usage of Ecore_Config is simple. The system is initialized with a call to `ecore_config_init`. The program name setting control where Ecore_Config will look for your configuration database. The directory and file name are: ~/.e/apps/PROGRAM_NAME/config.db.

For each configuration variable you are getting from Ecore_Config, you can assign a default value in the case that the user does not have a config.db file. The defaults are assigned with the ecore_config_*_default where * is one of the Ecore_Config types. The first parameter is the key under which this is to be accessed. These keys must be unique over your program. The value passed is of the type appropriated for this call.

The `ecore_config_load` call will read the values from the config.db file into Ecore_Config. After which we can access the files with the ecore_config_*_get methods (again * is the type of data desired). These routines take the key name for this item and return the value associated with that key. Each function returns a type that corresponds to the function call name.

`ecore_config_shutdown` is then called to shutdown the Ecore_Config system before the program exits.

Example 4.2. Compilation command

```
gcc -o ecore_config_example ecore_config_example.c `ecore-config --cflags --libs`
```

To compile the program you can use the ecore-config script to get all of the required linking and library information for Ecore_Config. If you run this program as is you will receive the values put into ecore_config as the defaults as output. Once you know the program is working, you can create a simple config.db file to read the values.

Example 4.3. Simple config.db script (build_cfg_db.sh)

```
#!/bin/sh

DB=config.db

edb_ed $DB add /int_example int 2
edb_ed $DB add /this/is/a/string/example str "this is a string"
edb_ed $DB add /float/example float 42.10101
```

When build_cfg_db.sh is executed it will create a config.db file in the current directory. This file can then be copied into ~/.e/apps/PROGRAM_NAME/config.db where PROGRAM_NAME is the value passed into ecore_config_init. Once the file is copied in place, executing the test program again will show the values given in the config file instead of the defaults. You can specify as many, or as few of the configuration keys in the config file and Ecore_Config will either show the user value or the default value.

Recipe: Ecore Ipc Introduction

dan dj2 sinclair <zero@perplexity.org>

The Ecore_Ipc library provides a robust and efficient wrapper around the Ecore_Con module. Ecore_Ipc allows you to set up your server communications and handles all of the tricky stuff under the hood. This recipe will give a simple example of an Ecore client and an Ecore server.

When working with Ecore_Ipc, when writing a client or a server app an Ecore_Ipc_Server object will be created. This is because in either case it is a server being manipulated, either the one being setup, or the one being communicated with. After that, everything is easy.

Example 4.4. Ecore_Ipc client: preamble

```
#include <Ecore.h>
#include <Ecore_Ipc.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
int handler_server_add(void *data, int ev_type, void *ev);
int handler_server_del(void *data, int ev_type, void *ev);
int handler_server_data(void *data, int ev_type, void *ev);
```

The Ecore.h file is included so we can have access to the exit signal type. The functions will be explained later when their callbacks are hooked up.

Example 4.5. Ecore_Ipc client: main setup

```
int main(int argc, char ** argv) {
    Ecore_Ipc_Server *server;

    if (!ecore_init()) {
        printf("unable to init ecore\n");
        return 1;
    }

    if (!ecore_ipc_init()) {
        printf("unable to init ecore_con\n");
        ecore_shutdown();
        return 1;
    }
    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);
```

As mentioned earlier, even though we are writing a client app, we still use an Ecore_Ipc_Server object. Using Ecore_Ipc requires the setup of Ecore itself. This is done with a simple call to `ecore_init`. Ecore_Ipc is then setup with a call to `ecore_ipc_init`. If either of these return 0, the appropriate action is taken to undo any initialization take to this point. The ECORE_EVENT_SIGNAL_EXIT callback is hooked up so we can exit gracefully if required.

Example 4.6. Ecore_Ipc client: main creating client

```
    server = ecore_ipc_server_connect(ECORE_IPC_REMOTE_SYSTEM,
                                        "localhost", 9999, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_ADD,
                                        handler_server_add, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_DEL,
                                        handler_server_del, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_SERVER_DATA,
                                        handler_server_data, NULL);
```

In this example we are creating a remote connection to the server named "localhost" on the port 9999. This is done with the `ecore_ipc_server_connect` method. The first parameter is the type of connection being made, which can be one of: ECORE_IPC_REMOTE_SYSTEM, ECORE_IPC_LOCAL_SYSTEM, or ECORE_IPC_LOCAL_USER. If OpenSSL was available when Ecore_Ipc was compiled, ECORE_IPC_USE_SSL can be or'd with the connection type to create an SSL connection.

The three calls to `ecore_event_handler_add` setup the callbacks for the different types of events we will be receiving from the server. A server was added, a server was deleted, or the server sent us data.

Example 4.7. Ecore_Ipc client: main end

```
    ecore_ipc_server_send(server, 3, 4, 0, 0, 0, "Look ma, no pants", 17);

    ecore_main_loop_begin();

    ecore_ipc_server_del(server);
    ecore_ipc_shutdown();
    ecore_shutdown();
    return 0;
}
```

For the purposes of this example, the client is sending a message on startup to the server, which the server will respond to. The client message is sent with the `ecore_ipc_server_send` command. `ecore_ipc_server_send` takes the server to send to, the message major, message minor, a reference, a reference to, a response, the data and a size. These parameters, except for the server are up the the client and can refer to anything required. This hopefully gives the maximum flexibility in creating client/server IPC apps.

After the server message is sent we enter into the main ecore loop and wait for events. If the main loop is exited we delete the server object, shutdown Ecore_Ipc with a call to `ecore_ipc_shutdown`, and shutdown ecore through `ecore_shutdown`.

Example 4.8. Ecore_Ipc client: sig_exit_cb

```
int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}
```

The `sig_exit_cb` just tells ecore to quit the main loop.

Example 4.9. Ecore_Ipc client: the callbacks

```
int handler_server_add(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Server_Add *e = (Ecore_Ipc_Event_Server_Add *)ev;
    printf("Got a server add %p\n", e->server);
    return 1;
}

int handler_server_del(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Server_Del *e = (Ecore_Ipc_Event_Server_Del *)ev;
    printf("Got a server del %p\n", e->server);
    return 1;
}

int handler_server_data(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Server_Data *e = (Ecore_Ipc_Event_Server_Data *)ev;
    printf("Got server data %p [%i] [%i] [%i] (%s)\n", e->server, e->major,
                                e->minor, e->size, (char *)e->data);
    return 1;
}
```

These three callbacks, `handler_server_add`, `handler_server_del`, and `handler_server_data` are body of the client handling all events related to the server we are connected to. Each of the callbacks has an associated event structure, Ecore_Ipc_Event_Server_Add, Ecore_Ipc_Event_Server_Del and Ecore_Ipc_Event_Server_Data containing information on the event itself.

When we first connect to the server the `handler_server_add` callback will be executed allowing any setup to be accomplished.

If the server breaks the connection the `handler_server_del` callback will be executed allowing any required cleanup.

When the server sends data to the client the `handler_server_data` callback will the executed. Which in this example just prints some information about the message itself and the message body.

And thats the client. The code itself is pretty simple thanks to the abstractions provided by Ecore.

Example 4.10. Ecore_Ipc server: preamble

```
#include <Ecore.h>
#include <Ecore_Ipc.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
```

```
int handler_client_add(void *data, int ev_type, void *ev);
int handler_client_del(void *data, int ev_type, void *ev);
int handler_client_data(void *data, int ev_type, void *ev);
```

As with the client, the Ecore.h header is included to get access the to the exit signal. The Ecore_Ipc.h header is required for apps making use of the Ecore_Ipc library. Each sig handler will be explained with its code.

Example 4.11. Ecore_Ipc server: main setup

```
int main(int argc, char ** argv) {
    Ecore_Ipc_Server *server;

    if (!ecore_init()) {
        printf("Failed to init ecore\n");
        return 1;
    }

    if (!ecore_ipc_init()) {
        printf("failed to init ecore_con\n");
        ecore_shutdown();
        return 1;
    }

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);
```

This is the same as the client setup above.

Example 4.12. Ecore_Ipc server: main creating server

```
    server = ecore_ipc_server_add(ECORE_IPC_REMOTE_SYSTEM, "localhost", 9999, NULL
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_ADD, handler_client_add, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_DEL, handler_client_del, NULL);
    ecore_event_handler_add(ECORE_IPC_EVENT_CLIENT_DATA, handler_client_data, NULL
```

Unlike the client, for the server we add a listener to port 9999 on the machine "localhost" through the call `ecore_ipc_server_add`. This will create and return the server object to us. We then hook in the required signal handlers, the difference to the client being we want CLIENT events this time instead of SERVER events.

Example 4.13. Ecore_Ipc client: main end

```
    ecore_main_loop_begin();

    ecore_ipc_server_del(server);
    ecore_ipc_shutdown();
    ecore_shutdown();
```

```
    return 0;
}
```

This again is identical to the client shutdown, minus the sending of data to the server.

Example 4.14. Ecore_Ipc server: sig_exit callback

The `sig_exit_cb` is again identical to that seen in the client.

Example 4.15. Ecore_Ipc server: the callbacks

```
int handler_client_add(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Add *e = (Ecore_Ipc_Event_Client_Add *)ev;
    printf("client %p connected to server\n", e->client);
    return 1;
}

int handler_client_del(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Del *e = (Ecore_Ipc_Event_Client_Del *)ev;
    printf("client %p disconnected from server\n", e->client);
    return 1;
}

int handler_client_data(void *data, int ev_type, void *ev) {
    Ecore_Ipc_Event_Client_Data *e = (Ecore_Ipc_Event_Client_Data *)ev;
    printf("client %p sent [%i] [%i] [%i] (%s)\n", e->client, e->major,
                                e->minor, e->size, (char *)e->data);

    ecore_ipc_client_send(e->client, 3, 4, 0, 0, 0, "Pants On!", 9);
    return 1;
}
```

The event callbacks are similar to those seen in the client app. The main difference is that the events are _Client_ events instead of _Server_ events.

The add callback is when a client connects to our server, with the del callback being its opposite when the client disconnects. The data callback is for when a client sends data to the server.

At the end of the `handler_client_data` callback we do a call to `ecore_ipc_client_send`. This sends data to the client. As with sending data to the server, the parameters are: the client to send to, major number, minor number, reference, reference to, response, data and the data size.

Example 4.16. Ecore_Ipc: compilation

```
CC = gcc
```

```
all: server client

server: server.c
    $(CC) -o server server.c `ecore-config --cflags --libs`

client: client.c
    $(CC) -o client client.c `ecore-config --cflags --libs`

clean:
    rm server client
```

As with other ecore apps, it is very easy to compile an Ecore_Ipc app. As long as your Ecore was compiled with Ecore_Ipc, simply invoking the 'ecore-config --cflags --libs' command will add all of the required library paths and linker information.

As seen in this example, Ecore_Ipc is an easy to use library to create client/server apps.

Chapter 5. EDB & EET

EDB is a database convenience library wrapped around the Berkeley DB 2.7.7 by Sleepycat Software. It is intended to make accessing database information portable, easy, fast and efficient.

EET is a tiny library designed to write arbitary chunks of data to a file and optionally compress each chunk (very much like a zip file) and allows for fast random-access reading of the file later on. It does not do zip as a zip itself has more complexity than is needed, and it was much simpler to impliment this once here.

EDB provides an excellent method of storing and retrieving application configuration information, although it can be used for more extensively than that. Ebits, the predecessor to Edje, even used EDB as a container for Ebits themes prior to EET. An Edb consists of a series of key/value pairs, which can consist of a variety of data types, including integers, floating point values, strings, and binary data. The simplified API provides simple, complete, and unified functions for managing and accessing your database.

In addition to the library, a variety of tools are available to access and modify your EDBs. The edb_ed tool provides a simple command line interface that can easily be scripted, especially useful for use with the GNU autotools suite. The edb_vt_ed tool provides an easy to use curses interface. Finally, edb_gtk_ed provides an elegant and easy GUI interface, especially useful for end user editing of configuration data contained in EDBs.

Eet is extremely fast, small and simple. Eet files can be very small and highly compressed, making them very optimal for just sending across the internet without having to archive, compress or decompress and install them. They allow for lightning-fast random-acess reads once created, making them perfect for storing data that is written once (or rarely) and read many times, but the program does not want to have to read it all in at once.

It also can encode and decode data structures in memory, as well as image data for saving to Eet files or sending across the network to other machines, or just writing to arbitary files on the system. All data is encoded in a platform independant way and can be written and read by any architecture.

Chapter 6. Esmart

Esmart provides a variety of EVAS smart objects that provide significant power to your EVAS and EFL based applications.

Recipe: Esmart Trans Introduction

dan dj2 sinclair <zero@perplexity.org>

Transparency is increasingly becoming a common trait of applications. To this end, the Esmart_Trans object has been created. This object will do all of the hard work to produce a transparent background for your program.

Esmart trans makes the integration of a transparent background into your application very easy. You need to create the trans object, and then make sure you update it as the window is moved or resized.

Example 6.1. Includes and declarations

```
#include <stdio.h>
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Esmart/Esmart_Trans_X11.h>

int sig_exit_cb(void *data, int ev_type, void *ev);
void win_del_cb(Ecore_Evas *ee);
void win_resize_cb(Ecore_Evas *ee);
void win_move_cb(Ecore_Evas *ee);

static void _freshen_trans(Ecore_Evas *ee);
void make_gui();
```

Every application that uses an Esmart_Trans object is going to require the Ecore, Ecore_Evas and the Esmart/Esmart_Trans header files. The next four declarations are callbacks from ecore for events on our window, exit, delete, resize, and move respectively. The last two declarations are convenience functions being used in the example and do not need to be in your program.

Example 6.2. main

```
int main(int argc, char ** argv) {
    int ret = 0;

    if (!ecore_init()) {
        printf("Error initializing ecore\n");
        ret = 1;
        goto ECORE_SHUTDOWN;
    }

    if (!ecore_evas_init()) {
        printf("Error initializing ecore_evas\n");
        ret = 1;
        goto ECORE_SHUTDOWN;
    }
```

```
    make_gui();
    ecore_main_loop_begin();

    ecore_evas_shutdown();

ECORE_SHUTDOWN:
    ecore_shutdown();

    return ret;
}
```

The main routine for this example is pretty simple. Ecore and Ecore_Evas are both initialized, with appropriate error checking. We then create the gui and start the main ecore event loop. When ecore exits we shut everything down and return.

Example 6.3. exit and del callbacks

```
int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}

void win_del_cb(Ecore_Evas *ee) {
    ecore_main_loop_quit();
}
```

The exit and del callbacks are the generic ecore callbacks.

Example 6.4. _freshen_trans

```
static void _freshen_trans(Ecore_Evas *ee) {
    int x, y, w, h;
    Evas_Object *o;

    if (!ee) return;

    ecore_evas_geometry_get(ee, &x, &y, &w, &h);
    o = evas_object_name_find(ecore_evas_get(ee), "bg");

    if (!o) {
        fprintf(stderr, "Trans object not found, bad, very bad\n");
        ecore_main_loop_quit();
    }
    esmart_trans_x11_freshen(o, x, y, w, h);
}
```

The `_freshen_trans` routine is a helper routine to update the image that the trans is shown. This will be called when we need to update our image to whats currently under the window. The function grabs the current size of the ecore_evas, and then gets the object with the name "bg" (this name is the same as the name we give our trans when we create it). Then, as long as the trans object exists, we tell esmart to freshen the image being displayed.

Example 6.5. resize_cb

```
void win_resize_cb(Ecore_Evas *ee) {
    int w, h;
    int minw, minh;
    int maxw, maxh;
    Evas_Object *o = NULL;

    if (ee) {
        ecore_evas_geometry_get(ee, NULL, NULL, &w, &h);
        ecore_evas_size_min_get(ee, &minw, &minh);
        ecore_evas_size_max_get(ee, &maxw, &maxh);

        if ((w >= minw) && (h >= minh) && (h <= maxh) && (w <= maxw)) {
            if ((o = evas_object_name_find(ecore_evas_get(ee), "bg")))
                evas_object_resize(o, w, h);
        }
    }
    _freshen_trans(ee);
}
```

When the window is resized we need to update our evas to the correct size and then update the trans object to display that much of the background. We grab the current size of the window `ecore_evas_geometry_get` and the min/max size of the window. As long as our currently desired size is within the min/max bounds set for our window, we grab the "bg" (same as title again) object and resize it. Once the resizing is done, we call the `_freshen_trans` routine to update the image displayed on the bg.

Example 6.6. move_cb

```
void win_move_cb(Ecore_Evas *ee) {
    _freshen_trans(ee);
}
```

When the window is moved we need to freshen the image displayed as the transparency.

Example 6.7. Setup ecore/ecore_evas

```
void make_gui() {
    Evas *evas = NULL;
    Ecore_Evas *ee = NULL;
    Evas_Object *trans = NULL;
    int x, y, w, h;

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);

    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 300, 200);
    ecore_evas_title_set(ee, "trans demo");
```

```
    ecore_evas_callback_delete_request_set(ee, win_del_cb);
    ecore_evas_callback_resize_set(ee, win_resize_cb);
    ecore_evas_callback_move_set(ee, win_move_cb);

    evas = ecore_evas_get(ee);
```

The first portion of make_gui is concerned with setting up ecore and ecore_evas. First the exit callback is hooked into ECORE_EVENT_SIGNAL_EXIT, then the Ecore_Evas object is created with the software X11 engine. The window title is set and we hook in the callbacks written above, delete, resize and move. Finally we grab the evas for the created Ecore_Evas.

Example 6.8. Creating Esmart_Trans object

```
    trans = esmart_trans_x11_new(evas);
    evas_object_move(trans, 0, 0);
    evas_object_layer_set(trans, -5);
    evas_object_name_set(trans, "bg");

    ecore_evas_geometry_get(ee, &x, &y, &w, &h);
    evas_object_resize(trans, w, h);

    evas_object_show(trans);
    ecore_evas_show(ee);

    esmart_trans_x11_freshen(trans, x, y, w, h);
}
```

Once everything is setup we can create the trans object. The trans is to be created in the evas returned by `ecore_evas_get`. This initial creation is done by the call to `esmart_trans_x11_new`. Once we have the object, we move it so it starts at position (0, 0) (the upper left corner), set the layer to -5 and name the object "bg" (as used above). Then we grab the current size of the ecore_evas and use that to resize the trans object to the window size. Once everything is resized we show the trans and show the ecore_evas. As a final step, we freshen the image on the transparency to what is currently under the window so it is up to date.

Example 6.9. Simple makefile

```
CFLAGS = `ecore-config --cflags` `evas-config --cflags` `esmart-config --cflags`
LIBS = `ecore-config --libs` `evas-config --libs` `esmart-config --libs` \
            -lesmart_trans_x11

all:
    gcc -o trans_example trans_example.c $(CFLAGS) $(LIBS)
```

In order to compile the above program we need to include the library information for ecore, ecore_evas and esmart. This is done through the -config scripts for each library. These -config scripts know where each of the includes and libraries resides and sets up the appropriate linking and include paths for the compilation.

Recipe: Esmart Container Introduction

dan dj2 sinclair <zero@perplexity.org>

There is usually a desire while designing an apps UI to group common elements together and have their layout depend on one another. To this end the Esmart Container libary has been created. It has been designed to handle the hard parts of the layout, and in the cases where it does not do what you need, the layout portions of the container are extensible and changeable.

This recipe will give the basics of using an Esmart container. The final product is a program that will let you see some of the different layout combinations of the default container. The layout will be done by Edje with callbacks to the program to change the container layout, and to tell if the user clicked on a container element.

Example 6.10. Includes and declarations

```
#include <Ecore.h>
#include <Ecore_Evas.h>
#include <Edje.h>
#include <Esmart/Esmart_Container.h>
#include <getopt.h>

static void make_gui(const char *theme);
static void container_build(int align, int direction, int fill);
static void _set_text(int align, int direction);
static void _setup_edje_callbacks(Evas_Object *o);
static void _right_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source);
static void _left_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source);
static void _item_selected(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source);

static Ecore_Evas *ee;
static Evas_Object *edje;
static Evas_Object *container;

char *str_list[] = {"item 1", "item 2",
                    "item 3", "item 4",
                    "item 5"};
```

As with other EFL programs we need to include Ecore, Ecore_Evas, Edje and as this is a container example, the Esmart/Esmart_Container header. Getopt will be used to allow for some command line processing.

Next comes the function prototypes which will be described later when we get to their implementations. Then a few global variables to be used throughout the program. The str_list array is the content to be stored in the container.

Example 6.11. main

```
int main(int argc, char ** argv) {
    int align = 0;
    int direction = 0;
```

```
    int fill = 0;
    int ret = 0;
    int c;
    char *theme = NULL;

    while((c = getopt(argc, argv, "a:d:f:t:")) != -1) {
        switch(c) {
            case 'a':
                align = atoi(optarg);
                break;

            case 'd':
                direction = atoi(optarg);
                break;

            case 'f':
                fill = atoi(optarg);
                break;

            case 't':
                theme = strdup(optarg);
                break;

            default:
                printf("Unknown option string\n");
                break;
        }
    }

    if (theme == NULL) {
        printf("Need a theme defined\n");
        exit(-1);
    }
```

The beginning of the main function gets the options out of the command line arguments and sets up the default display. As you can see, we require a theme to display. This could be made more intelligent, searching default install directories and the users application directories, but this example takes the easy way out and forces the theme to be a command line option.

Example 6.12. Initialization

```
    if (!ecore_init()) {
        printf("Can't init ecore, bad\n");
        ret = 1;
        goto EXIT;
    }
    ecore_app_args_set(argc, (const char **)argv);

    if (!ecore_evas_init()) {
        printf("Can't init ecore_evas, bad\n");
        ret = 1;
        goto EXIT_ECORE;
    }

    if (!edje_init()) {
        printf("Can't init edje, bad\n");
        ret = 1;
        goto EXIT_ECORE_EVAS;
```

```
    }
    edje_frametime_set(1.0 / 60.0);

    make_gui(theme);
    container_build(align, direction, fill);

    ecore_main_loop_begin();
```

After receiving the command line arguments, we then proceed to initializing the required libraries, Ecore, Ecore_Evas and Edje. We take the additional step of setting the Edje frame rate.

Once the initialization is complete we create the initial GUI for the app. I have separated the building of the container out into a separate function to make the container code easier to locate in the example.

Once everything is created we call `ecore_main_loop_begin` and wait for events to occur.

Example 6.13. Shutdown

```
    edje_shutdown();

EXIT_ECORE_EVAS:
    ecore_evas_shutdown();

EXIT_ECORE:
    ecore_shutdown();

EXIT:
    return ret;
}
```

The usual end routine, be good programmers and shutdown everything we started.

Example 6.14. Window callbacks

```
static int sig_exit_cb(void *data, int ev_type, void *ev) {
    ecore_main_loop_quit();
    return 1;
}

static void win_del_cb(Ecore_Evas *ee) {
    ecore_main_loop_quit();
}

static void win_resize_cb(Ecore_Evas *ee) {
    int w, h;
    int minw, minh;
    int maxw, maxh;
    Evas_Object *o = NULL;

    if (ee) {
        ecore_evas_geometry_get(ee, NULL, NULL, &w, &h);
        ecore_evas_size_min_get(ee, &minw, &minh);
        ecore_evas_size_max_get(ee, &maxw, &maxh);
```

```
        if ((w >= minw) && (h >= minh) && (h <= maxh) && (w <= maxw)) {
            if ((o = evas_object_name_find(ecore_evas_get(ee), "edje")))
                evas_object_resize(o, w, h);
        }
    }
}
```

Next we setup some generic callbacks to be used by the UI. This will be the exit, destroy and resize callbacks. Again, the usual EFL style functions.

Example 6.15. make_gui

```
static void make_gui(const char *theme) {
    Evas *evas = NULL;
    Evas_Object *o = NULL;
    Evas_Coord minw, minh;

    ee = NULL;
    edje = NULL;
    container = NULL;

    ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, sig_exit_cb, NULL);

    ee = ecore_evas_software_x11_new(NULL, 0, 0, 0, 300, 400);
    ecore_evas_title_set(ee, "Container Example");

    ecore_evas_callback_delete_request_set(ee, win_del_cb);
    ecore_evas_callback_resize_set(ee, win_resize_cb);
    evas = ecore_evas_get(ee);

    // create the edje
    edje = edje_object_add(evas);
    evas_object_move(edje, 0, 0);

    if (edje_object_file_set(edje, theme, "container_ex")) {
        evas_object_name_set(edje, "edje");

        edje_object_size_min_get(edje, &minw, &minh);
        ecore_evas_size_min_set(ee, (int)minw, (int)minh);
        evas_object_resize(edje, (int)minw, (int)minh);
        ecore_evas_resize(ee, (int)minw, (int)minh);

        edje_object_size_max_get(edje, &minw, &minh);
        ecore_evas_size_max_set(ee, (int)minw, (int)minh);
        evas_object_show(edje);

    } else {
        printf("Unable to open (%s) for edje theme\n", theme);
        exit(-1);
    }
    _setup_edje_callbacks(edje);
    ecore_evas_show(ee);
}
```

The GUI consists of the Ecore_Evas containing the canvas itself, and the Edje that we will be using to control our layout. The `make_gui` function sets up the callbacks defined above and creates the Ecore_Evas.

Once we have the Evas and the callbacks are defined, we create the Edje object that will define our layout. The `edje_object_add` call is used to create the object on the Evas, and once thats done, we take the theme passed in by the user and set our Edje to use said theme, the "container_ex" parameter is the name of the group inside the EET that we are to use.

Once the theme file is set to the Edje, we use the values in the theme file to setup the size ranges for the app, and show the Edje. We then setup the callbacks on the Edje and show the Ecore_Evas.

Example 6.16. Edje Callbacks

```
static void _setup_edje_callbacks(Evas_Object *o) {
    edje_object_signal_callback_add(o, "left_click",
                        "left_click", _left_click_cb, NULL);
    edje_object_signal_callback_add(o, "right_click",
                        "right_click", _right_click_cb, NULL);
}
```

The program will have two main callbacks attached to the Edje, one for the left click signal and one for the right click signal. These will be used to switch the direction/alignment of the container. The second and third parameters of the callbacks need to match the data emitted with the signal from Edje, this will be seen later when we look at the EDC file. The third parameter is the function to call, and the last, any data we wish to be passed into the callback.

Example 6.17. container_build

```
static void container_build(int align, int direction, int fill_policy) {
    int len = 0;
    int i = 0;
    const char *edjefile = NULL;

    container = esmart_container_new(ecore_evas_get(ee));
    evas_object_name_set(container, "the_container");
    esmart_container_direction_set(container, direction);
    esmart_container_alignment_set(container, align);
    esmart_container_padding_set(container, 1, 1, 1, 1);
    esmart_container_spacing_set(container, 1);
    esmart_container_fill_policy_set(container, fill_policy);

    evas_object_layer_set(container, 0);
    edje_object_part_swallow(edje, "container", container);
```

The `container_build` function will create the container and set our data elements in said container. The creation is easy enough with a call to `esmart_container_new` giving back the Evas_Object that is the container. Once the container is created we can set a name on the container to make reference easier.

Next, we set the direction, which is either (CONTAINER_DIRECTION_VERTICAL or CONTAINER_DIRECTION_HORIZONTAL) [or in this case, an int being passed from the command line as the two directions map to 1 and 0 respectively]. The direction tells the container which way the elements will be drawn.

After the direction we set the alignment of the container. The alignment tells the container where to draw the elements. The possible values are: CONTAINER_ALIGN_CENTER, CONTAINER_ALIGN_LEFT, CONTAINER_ALIGN_RIGHT, CONTAINER_ALIGN_TOP and CONTAINER_ALIGN_BOTTOM. With the default layout, left and right only apply to a vertical container, and top and bottom only apply to a horizontal container, although center applies to both.

If we wanted to use a different layout scheme then the default, we could place a call to `esmart_container_layout_plugin_set(container, "name")` where the name is the name of the plugin to use. The default setting is the container named "default".

Once the directions and alignment are set, the spacing and padding of the container are specified. The padding specifes the space around the outside of the container taking four numeric parameters: left, right, top and bottom. The spacing parameter specifies the space between elements in the container.

We then continue and set the fill policy of the container. This specifies how the elements are positioned to fill the space in the container. The possible values are: CONTAINER_FILL_POLICY_NONE, CONTAINER_FILL_POLICY_KEEP_ASPECT, CONTAINER_FILL_POLICY_FILL_X, CONTAINER_FILL_POLICY_FILL_Y, CONTAINER_FILL_POLICY_FILL and CONTAINER_FILL_POLICY_HOMOGENOUS.

Once the container is fully specified we set the containers layer, and then swallow the container into the edje and the part named "container".

Example 6.18. Adding Elements to the Container

```
len = (sizeof(str_list) / sizeof(str_list[0]));
for(i = 0; i < len; i++) {
    Evas_Coord w, h;
    Evas_Object *t = edje_object_add(ecore_evas_get(ee));

    edje_object_file_get(edje, &edjefile, NULL);
    if (edje_object_file_set(t, edjefile, "element")) {
        edje_object_size_min_get(t, &w, &h);
        evas_object_resize(t, (int)w, (int)h);

        if (edje_object_part_exists(t, "element.value")) {
            edje_object_part_text_set(t, "element.value", str_list[i]);
            evas_object_show(t);
            int *i_ptr = (int *)malloc(sizeof(int));
            *i_ptr = (i + 1);

            edje_object_signal_callback_add(t, "item_selected",
                                    "item_selected", _item_selected, i_ptr);

            esmart_container_element_append(container, t);
        } else {
            printf("Missing element.value part\n");
            evas_object_del(t);
        }
    } else {
        printf("Missing element part\n");
        evas_object_del(t);
    }
}
evas_object_show(container);
```

```
    _set_text(align, direction);
}
```

Now that we have a container, we can add some elements to be displayed. Each of the entries in the str_list array defined at the beginning of the program will be added into the container as a text part.

For each element we create a new Edje object on the Evas. We then need to know the name of the theme file used to create our main Edje, so we call `edje_object_file_get` which will set edje file to said value.

We then try to set the group named "element" onto the newly created element. If this fails we print an error and delete the object.

As long as we have found the group "element" we can attempt to grab the part for our element, "element.value". If this part exists, we set the text value of the part to our current string and show the part.

A callback is created through `edje_object_signal_callback_add` and attached to the new element. This will be called if the "item_selected" signal is sent from the Edje. The i_ptr value shows how data can be attached to the element, when the user clicks on an element its number will be printed to the console.

Once the element is created we add it to the container (in this case, appending the element).

To finish, the container is show and we do some extra work to display information about the container in the header through the call `_show_text`.

Example 6.19. _set_text

```
static void _set_text(int align, int direction) {
    Evas_Object *t = edje_object_add(ecore_evas_get(ee));
    const char *edjefile;

    if (direction == CONTAINER_DIRECTION_VERTICAL)
        edje_object_part_text_set(edje, "header_text_direction", "vertical");
    else
        edje_object_part_text_set(edje, "header_text_direction", "horizontal");

    if (align == CONTAINER_ALIGN_CENTER)
        edje_object_part_text_set(edje, "header_text_align", "center");

    else if (align == CONTAINER_ALIGN_TOP)
        edje_object_part_text_set(edje, "header_text_align", "top");

    else if (align == CONTAINER_ALIGN_BOTTOM)
        edje_object_part_text_set(edje, "header_text_align", "bottom");

    else if (align == CONTAINER_ALIGN_RIGHT)
        edje_object_part_text_set(edje, "header_text_align", "right");

    else if (align == CONTAINER_ALIGN_LEFT)
        edje_object_part_text_set(edje, "header_text_align", "left");
}
```

The `_set_text` routine takes the current direction and alignment of the container and sets some text in the header of the program. This is just a simple communication with the user of the current container settings.

Example 6.20. _left_click_cb

```
static void _left_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source) {
    Container_Direction dir = esmart_container_direction_get(container);
    Container_Direction new_dir = (dir + 1) % 2;
    Container_Alignment align = esmart_container_alignment_get(container);

    esmart_container_direction_set(container, new_dir);

    if (align != CONTAINER_ALIGN_CENTER) {
        if (new_dir == CONTAINER_DIRECTION_HORIZONTAL)
            align = CONTAINER_ALIGN_TOP;
        else
            align = CONTAINER_ALIGN_LEFT;
    }
    esmart_container_alignment_set(container, align);
    _set_text(align, new_dir);
}
```

When the user clicks the left mouse button on the screen this callback will be executed. We take the current container direction information and switch to the other direction. (e.g. horizontal becomes vertical and visa versa.) We also reset the alignment if we are not currently aligned center to make sure everything is valid for the new direction. The text in the header is updated to be current.

Example 6.21. _right_click_cb

```
static void _right_click_cb(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source) {
    Container_Direction dir = esmart_container_direction_get(container);
    Container_Alignment align = esmart_container_alignment_get(container);

    if (dir == CONTAINER_DIRECTION_HORIZONTAL) {
        if (align == CONTAINER_ALIGN_TOP)
            align = CONTAINER_ALIGN_CENTER;

        else if (align == CONTAINER_ALIGN_CENTER)
            align = CONTAINER_ALIGN_BOTTOM;

        else
            align = CONTAINER_ALIGN_TOP;

    } else {
        if (align == CONTAINER_ALIGN_LEFT)
            align = CONTAINER_ALIGN_CENTER;

        else if (align == CONTAINER_ALIGN_CENTER)
            align = CONTAINER_ALIGN_RIGHT;

        else
            align = CONTAINER_ALIGN_LEFT;
    }
    esmart_container_alignment_set(container, align);
    _set_text(align, esmart_container_direction_get(container));
```

```
}
```

The right click callback will cycle through the available alignments for a given direction as the user clicks the right mouse button.

Example 6.22. _item_selected

```
static void _item_selected(void* data, Evas_Object* o, const char* emmission,
                                                            const char* source) {
    printf("You clicked on the item with number %d\n", *((int *)data));
}
```

Finally the `_item_selected` callback will be executed when the user middle clicks on an item in the container. The data will contain the number set for that element in the create routine above.

Thats the end of the code for the app, next comes the required EDC for everything to display and function correctly.

Example 6.23. The Edc

```
fonts {
    font: "Vera.ttf" "Vera";
}

collections {
    group {
        name, "container_ex";
        min, 300, 300;
        max, 800, 800;

        parts {
            part {
                name, "bg";
                type, RECT;
                mouse_events, 1;

                description {
                    state, "default" 0.0;
                    color, 0 0 0 255;

                    rel1 {
                        relative, 0.0 0.1;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, 0 0;
                    }
                }
            }

            part {
                name, "header";
```

```
        type, RECT;
        mouse_events, 0;

        description {
            state, "default" 0.0;
            color, 255 255 255 255;

            rel1 {
                relative, 0.0 0.0;
                offset, 0 0;
            }

            rel2 {
                relative, 1.0 0.1;
                offset, 0 0;
            }
        }
    }

    part {
        name, "header_text_direction";
        type, TEXT;
        mouse_events, 0;

        description {
            state, "default" 0.0;
            color, 0 0 0 255;

            rel1 {
                relative, 0.0 0.0;
                offset, 0 10;
                to, "header";
            }
            rel2 {
                relative, 1.0 1.0;
                offset, 0 0;
                to, "header";
            }
            text {
                text, "direction";
                font, "Vera";
                size, 10;
            }
        }
    }

    part {
        name, "header_text_align";
        type, TEXT;
        mouse_events, 0;

        description {
            state, "default" 0.0;
            color, 0 0 0 255;

            rel1 {
                relative, 0.0 0.0;
                offset, 0 0;
                to, "header_text_direction";
            }
            rel2 {
                relative, 1.0 1.0;
                offset, 110 0;
                to, "header_text_direction";
```

```
            }
            text {
                text, "align";
                font, "Vera";
                size, 10;
            }
        }
    }
```

This EDC file expects to have the Vera font embedded inside it, as defined by the font section at the beginning. This means when you compile the edc you either need the Vera.ttf file in the current directory or give edje_cc the -fd flag and specify the directory to the font.

After the fonts are defined the main collections are defined. The first collection is the main portion of the app itself, the "container_ex" group. This group specifes the main window of the app. As such it contains the parts for the background, the header, and the header text. These parts are all fairly standard with some (minimal) alignment done between them.

Example 6.24. The Container Part

```
    part {
        name, "container";
        type, RECT;
        mouse_events, 1;

        description {
            state, "default" 0.0;
            visible, 1;

            rel1 {
                relative, 0.0 0.0;
                offset, 0 0;
                to, bg;
            }
            rel2 {
                relative, 1.0 1.0;
                offset, 0 0;
                to, bg;
            }
            color, 0 0 0 0;
        }
    }
}
programs {
    program {
        name, "left_click";
        signal, "mouse,clicked,1";
        source, "container";
        action, SIGNAL_EMIT "left_click" "left_click";
    }

    program {
        name, "right_click";
        signal, "mouse,clicked,3";
        source, "container";
        action, SIGNAL_EMIT "right_click" "right_click";
    }
}
```

```
    }
```

The container part is then defined. The part itself is pretty simple, just positioned relative to the background and set to receive mouse events. After the parts are defined we specify the programs for this group, of which there are two. The first program "left_click" specifies what is to happen on a click event of the first mouse button.

The action is to emit a signal, the two parameters after SIGNAL_EMIT match up to the values put in the callback in the application code.

There is a similar callback for the third mouse button as the first, just emitting a slightly different signal.

Example 6.25. The Element Group

```
    group {
        name, "element";
        min, 80 18;
        max, 800 18;

        parts {
            part {
                name, "element.value";
                type, TEXT;
                mouse_events, 1;
                effect, NONE;

                description {
                    state, "default" 0.0;
                    visible, 1;

                    rel1 {
                        relative, 0.0 0.0;
                        offset, 0 0;
                    }
                    rel2 {
                        relative, 1.0 1.0;
                        offset, 0 0;
                    }
                    color, 255 255 255 255;

                    text {
                        text, "";
                        font, "Vera";
                        size, 10;
                    }
                }
            }
        }
        programs {
            program {
                name, "center_click";
                signal, "mouse,clicked,2";
                source, "element.value";
                action, SIGNAL_EMIT "item_selected" "item_selected";
            }
        }
    }
}
```

The element group specifies how each element of the container is to be displayed. You will notice that the names given here match up to the names seached for in the application code itself while creating the elements.

There is one program in this group which will emit a signal of "item_selected" when the middle mouse button is pressed while hovering over one of the elements in the list.

Thats the end of the EDC code. To compile the app code, a makefile similar to that below could be used.

Example 6.26. Makefile

```
CFLAGS = `ecore-config --cflags` `evas-config --cflags` `esmart-config --cflags`
LIBS = `ecore-config --libs` `evas-config --libs` `esmart-config --libs` \
            -lesmart_container

container_ex: container/container_ex.c
        gcc -o container/container_ex container/container_ex.c $(CFLAGS) $(LIBS)
```

And to create the EET file, a simple 'edje_cc default.edc' should suffice as long as the Vera.ttf file is in the current directory.

Thats it, assuming everything goes as planned, you should have a simple app in which clicking the right/ left mouse buttons moves the container to different portions of the window. While clicking the middle mouse button on elements prints out the number of the element printed.

Chapter 7. Etox

Etox is a type setting and text layout library based on Evas. Etox helps you when it comes to displaying, moving, resizing, layering, clipping, aligning and coloring fonts in different styles, and more.

Chapter 8. Edje

Edje is a complex graphical design and layout library.

It's purpose is to be a sequel to "Ebits" which to date has serviced the needs of Enlightenment development for version 0.17. The original design paramteres under which Ebits came about were a lot more restricted than the resulting use of them, thus Edje was born.

Edje is a more complex layout engine compared to Ebits. It doesn't pretend to do containering and regular layout like a widget set. It still inherits the more simplistic layout ideas behind Ebits, but it now does them a lot more cleanly, allowing for easy expansion, and the ability to cover much more ground than Ebits ever could. For the purposes of Enlightenment 0.17, Edje should serve all the purposes of creating visual elements (borders of windows, scrollbars, etc.) and allow the designer the ability to animate, layout and control the look and feel of any program using Edje as its basic GUI constructor. This library allows for multiple collections of Layouts in one file, sharing the same image database and thus allowing a whole theme to be conveneintly packaged into one file and shipped around.

Edje, unlike Ebits, separates the layout and behavior logic. Edje files ship with an image database, used by all the parts in all the collections to source graphical data. It has a directory of logical part names pointing to the part collection entry ID in the file (thus allowing for multiple logical names to point to the same part collection, allowing for the sharing of data between display elements). Each part collection consists of a list of visual parts, as well as a list of programs. A program is a conditionally run program that if a particular event occurs (a button is pressed, a mouse enters or leaves a part) will trigger an action that may affect other parts. In this way a part collection can be "programmed" via its file so as to hilight buttons when the mouse passes over them or show hidden parts when a button is clicked somewhere. The actions performed in changing from one state to another are also allowed to transition over a period of time, allowing animation.

This separation and simplistic event driven style of programming can produce almost any look and feel one could want for basic visual elements. Anything more complex is likely the domain of an application or widget set that may use Edje as a conveneient way of being able to configure parts of the display.

Chapter 9. Edje EDC

Edje Data Collections (EDC) source files allow for easy creation of rich and powerful interface designs...

Chapter 10. EWL

The Enlightened Widget Library (EWL) is a widget toolkit that builds upon the foundations constructed by the other libs in the EFL. Ewl uses Evas as its rendering backend and its appearence is abstracted out for Edje to handle.

EWL is similar in feel to many of the other toolkits out there including GTK, QT or MOTIF. The APIs differ by the concepts are the same.

Recipe: EWL Introduction

dan dj2 sinclair <zero@perplexity.org>

Thought the use of the Enlightened Widget Library (EWL), a lot of power can be delivered into the programmers hands with little effort.

This introduction to EWL will show how to create a simple text viewing application with a menu bar and file dialog. The text area will have scrollbars and will also allow scrolling using either the keyboard arrow keys, or a mouse wheel.

Example 10.1. Includes and declarations

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <Ewl.h>

#define PROG    "EWL Text Viewer"

/* globals */
static Ewl_Widget *main_win = NULL;
static Ewl_Widget *fd_win = NULL;

/* pre-declarations */
static void destroy_cb(Ewl_Widget *, void *, void *);
static void destroy_filedialog_cb(Ewl_Widget *, void *, void *);
static void open_file_cb(Ewl_Widget *, void *, void *);
static void home_cb(Ewl_Widget *win, void *ev, void *data);
static void file_menu_open_cb(Ewl_Widget *, void *, void *);
static void key_up_cb(Ewl_Widget *, void *, void *);
static void wheel_cb(Ewl_Widget *win, void *ev, void *data);

static char *read_file(char *);
static void mk_gui(void);
```

The only required include to write an Ewl application is the <Ewl.h> declaration. We make the main window and the file dialog window global to facilitate easier access in the callback functions. They do not need to be global, but for the purposes of this example, its simpler if they are.

Example 10.2. main

```
/* lets go */
int main(int argc, char ** argv) {
    ewl_init(&argc, argv);
    mk_gui();
    ewl_main();
    return 0;
}
```

The main function for our text viewer is very simplistic. We start by initializing ewl through the `ewl_init` call. Ewl takes the argc and argv entries to do some command line parsing of its own. This includes such things as setting the Ewl theme to use (--ewl-theme) or setting the rendering engine (--ewl-software-x11, --ewl-gl-x11, etc.).

`ewl_init` takes care of all the dirty work of initializing the other required libs, abstracting all that away from the programmer into a simple interface.

The call to mk_gui will set up the main window and any content required.

The call to `ewl_main` sets up the main processing loop, and upon exit handles all of the applications required shutdown, hence there is no shutdown call from our main routine.

Example 10.3. mk_gui: creating the window

```
/* build the main gui */
static void mk_gui(void) {
    Ewl_Widget *box = NULL, *menu_bar = NULL;
    Ewl_Widget *text_area = NULL, *scroll = NULL;

    /* create the main window */
    main_win = ewl_window_new();
    ewl_window_set_title(EWL_WINDOW(main_win), PROG);
    ewl_window_set_name(EWL_WINDOW(main_win), PROG);
    ewl_window_set_class(EWL_WINDOW(main_win), PROG);

    ewl_object_request_size(EWL_OBJECT(main_win), 200, 300);
    ewl_object_set_fill_policy(EWL_OBJECT(main_win), EWL_FLAG_FILL_FILL);

    ewl_callback_append(main_win, EWL_CALLBACK_DELETE_WINDOW, destroy_cb, NULL);
    ewl_widget_show(main_win);
```

The first thing we need to do to get our app off of the ground is to create the main application window. This is done through the call to `ewl_window_new`. Once we have the window we can continue to set the title (as will appear in the WM bar on top of the app), name and class of the window.

Once the default information is set for the window we request a default size for the window to be 200x300 through the call to `ewl_object_request_size`. Along with the default size we could have set a minimum and maximum size for the window through the calls to `ewl_object_set_minimum_size` and `ewl_object_set_maximum` size. But as this is not required for this application they are left out.

The final setup of the window is done by setting the fill policy with `ewl_object_set_fill_policy`. This sets how Ewl will pack widgets into the window, with a possible values of:

EWL_FLAG_FILL_NONE	Do not fill or shrink in either direction
EWL_FLAG_FILL_HSHRINK	Shrink horizontally
EWL_FLAG_FILL_VSHRINK	Shrink vertically
EWL_FLAG_FILL_SHRINK	Shrink both horizontally and vertically
EWL_FLAG_FILL_HFILL	Fill horizontally
EWL_FLAG_FILL_VFILL	Fill vertically
EWL_FLAG_FILL_FILL	Fill both horizontally and vertically
EWL_FLAG_FILL_ALL	Shrink and Fill at the same time

After all the window properties are defined a callback to catch the destruction of the main window is attached with `ewl_callback_append`. The function `destroy_cb` will be called if someone requests the window to be destroyed in some fashion.

We show the main with with a call to `ewl_widget_show`. If `ewl_widget_show` is not called nothing would appear on the screen. All widgets are hidden until they are explicitly shown. The opposite to this is `ewl_widget_hide` which will remove a widget from the screen.

Example 10.4. The main container

```
/* create the main container */
box = ewl_vbox_new();
ewl_container_append_child(EWL_CONTAINER(main_win), box);
ewl_object_set_fill_policy(EWL_OBJECT(box), EWL_FLAG_FILL_FILL);
ewl_widget_show(box);
```

We could pack all of our widgets into the main window itself, but that could cause problems later if we wanted to change things easily, so instead we create a box inside the main window to hold all of our widgets.

This is done by creating a vertical box with `ewl_vbox_new`. The box is then taken and appended to the windows list of children with `ewl_container_append_child`. After attaching to the window we set the fill policy to fill both horizontal and vertical with `ewl_object_set_fill_policy`, and show the widget with `ewl_widget_show`.

The order you put your widgets into the containers will affect the way that the application is displayed. The first widget packed will be the first widget displayed. Since we specified a vertical box we will start by packing the widgets from the top to the bottom of our display.

Example 10.5. Create the menu bar

```
/* create the menu bar */
menu_bar = ewl_hbox_new();
ewl_container_append_child(EWL_CONTAINER(box), menu_bar);
ewl_object_set_fill_policy(EWL_OBJECT(menu_bar), EWL_FLAG_FILL_HSHRINK);
```

```
ewl_object_set_alignment(EWL_OBJECT(menu_bar), EWL_FLAG_ALIGN_LEFT);
ewl_box_set_spacing(EWL_BOX(menu_bar), 4);
ewl_object_set_padding(EWL_OBJECT(menu_bar), 5, 5, 5, 5);
ewl_widget_show(menu_bar);
```

The first widget we put into place is the menu bar. We will place the actual contents into the menu bar later after some of the other widgets are created but we need to put the bar itself into place first.

The calls are the same as many you have seen before, appending ourselves to our parent, setting our fill policy, showing the widget. The ones not seen before include `ewl_object_set_alignment`, this will set how the widget is aligned within its container. In this case we are using EWL_FLAG_ALIGN_LEFT, but could have used on of the other available alignments including:

- EWL_FLAG_ALIGN_CENTER
- EWL_FLAG_ALIGN_LEFT
- EWL_FLAG_ALIGN_RIGHT
- EWL_FLAG_ALIGN_TOP
- EWL_FLAG_ALIGN_BOTTOM

So the menu will be aligned with the left side of the main box.

We then specify the spacing of items inside the menu box. This will give a little more space between our menu items and is done with `ewl_box_set_spacing`. After changing the space we change the padding around the box as a whole with the call to `ewl_object_set_padding`, this will increase the amount of space left around the object as a whole.

Example 10.6. Create the scrollpane

```
/* create the scrollpane */
scroll = ewl_scrollpane_new();
ewl_container_append_child(EWL_CONTAINER(box), scroll);
ewl_object_set_fill_policy(EWL_OBJECT(scroll), EWL_FLAG_FILL_FILL);
ewl_scrollpane_set_hscrollbar_flag(EWL_SCROLLPANE(scroll),
                                    EWL_SCROLLBAR_FLAG_AUTO_VISIBLE);
ewl_scrollpane_set_vscrollbar_flag(EWL_SCROLLPANE(scroll),
                                    EWL_SCROLLBAR_FLAG_AUTO_VISIBLE);
ewl_widget_show(scroll);
```

The scrollpane is going to be the parent of our text object. The scrollpane provides us with all the magic to handle the scrollbars and the scrolling itself.

The scrollpane is created with a call to `ewl_scrollpane_new`, and we then proceed to attach the scrollpane to the main box, and set its fill policy.

The ewl_scrollpane_set_[hv]scrollbar_flag() calls tell Ewl how the scrollbars should behave. The possible values are:

- EWL_SCROLLBAR_FLAG_NONE
- EWL_SCROLLBAR_FLAG_AUTO_VISIBLE
- EWL_SCROLLBAR_FLAG_ALWAYS_HIDDEN

Once the scrollbars are setup we tell Ewl to show the widget.

Example 10.7. Create the text area

```
/* create the text area */
text_area = ewl_text_new("");
ewl_container_append_child(EWL_CONTAINER(scroll), text_area);
ewl_object_set_padding(EWL_OBJECT(text_area), 1, 1, 1, 1);
ewl_widget_show(text_area);
```

The text area will be responsible for holding the text we display in our viewer. The widget is created with a simple call to `ewl_text_new`. This will cause the text area to be created, but with the actual text blank. As with the menu bar we increase the padding around the text area to provide a bit of space from the edge of the text to any other elements.

Example 10.8. Add menu contents

```
/* create the menu */
{
    Ewl_Widget *file_menu = NULL, *item = NULL;

    /* create the file menu */
    file_menu = ewl_imenu_new(NULL, "file");
    ewl_container_append_child(EWL_CONTAINER(menu_bar), file_menu);
    ewl_widget_show(file_menu);

    /* add the open entry to the file menu */
    item = ewl_menu_item_new(NULL, "open");
    ewl_container_append_child(EWL_CONTAINER(file_menu), item);
    ewl_callback_append(item, EWL_CALLBACK_SELECT, file_menu_open_cb,
                                                            text_area);
    ewl_widget_show(item);

    /* add the quit entry to the file menu */
    item = ewl_menu_item_new(NULL, "quit");
    ewl_container_append_child(EWL_CONTAINER(file_menu), item);
    ewl_callback_append(item, EWL_CALLBACK_SELECT, destroy_cb, NULL);
    ewl_widget_show(item);
}
```

Now that the text area is created we can proceed to create the menu entries. I've done this inside its own block to limit the number of declarations at the top of the function, this isn't required for any reason.

The menu is created with a call to `ewl_imenu_new`. This takes two parameters, the first is the image to display with this menu, in this case NULL, being no image. The second parameter is the name of the menu as will appear in the menu bar.

Once the menu is created we can then proceed to add entries to the menu through a call to `ewl_menu_item_new`. This again takes two parameters, the icon to display beside this entry in the menu, and the name as it will appear in the menu.

As the items are added to the menu we make a call to `ewl_callback_append` to attach to the EWL_CALLBACK_SELECT call. The given function will be executed when the use clicks on the menu entry. In the "open" case we have passed the text_area to the open callback to allow us to easily modify its contents.

Other menus could have been added in the same fashion as this, but for this application only one menu is required.

Example 10.9. Attach callbacks

```
    ewl_callback_append(main_win, EWL_CALLBACK_KEY_UP, key_up_cb, scroll);
    ewl_callback_append(main_win, EWL_CALLBACK_MOUSE_WHEEL, wheel_cb, scroll);
}
```

Once everything is setup on the main window we attach the callbacks we wish to receive. In this case we are attaching ourselves to the EWL_CALLBACK_KEY_UP and the EWL_CALLBACK_MOUSE_WHEEL. This way we well be notified if a key on the keyboard is release, or the use rolls the mouse wheel.

Example 10.10. Destroy callback

```
/* destroy the app */
static void destroy_cb(Ewl_Widget *win, void *ev, void *data) {
    ewl_widget_destroy(win);
    ewl_main_quit();
}
```

When the main window is closed we destroy the widget that is the main window through a call to `ewl_widget_destroy`. After the window is destroyed we tell Ewl that we wish to exit by calling `ewl_main_quit`. This will cause Ewl to halt the main processing loop and the previous call to `ewl_main` will return.

Example 10.11. File menu open callback

```
/* the file menu open button callback */
static void file_menu_open_cb(Ewl_Widget *win, void *ev, void *data) {
    Ewl_Widget *fd = NULL;
    Ewl_Widget *box = NULL;
    Ewl_Widget *home = NULL;

    /* create the file dialog window */
    fd_win = ewl_window_new();
    ewl_window_set_title(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_window_set_name(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_window_set_class(EWL_WINDOW(fd_win), PROG " -- file dialog");
    ewl_object_request_size(EWL_OBJECT(fd_win), 500, 400);
    ewl_object_set_fill_policy(EWL_OBJECT(fd_win),
                EWL_FLAG_FILL_FILL | EWL_FLAG_FILL_SHRINK);
    ewl_callback_append(fd_win, EWL_CALLBACK_DELETE_WINDOW,
                            destroy_filedialog_cb, NULL);
    ewl_widget_show(fd_win);

    /* fd win container */
```

```
    box = ewl_vbox_new();
    ewl_container_append_child(EWL_CONTAINER(fd_win), box);
    ewl_object_set_fill_policy(EWL_OBJECT(box),
                EWL_FLAG_FILL_FILL | EWL_FLAG_FILL_SHRINK);
    ewl_widget_show(box);

    /* the file dialog */
    fd = ewl_filedialog_new(EWL_FILEDIALOG_TYPE_OPEN);
    ewl_callback_append(fd, EWL_CALLBACK_VALUE_CHANGED, open_file_cb, data);
    ewl_container_append_child(EWL_CONTAINER(box), fd);

    /* add a home button */
    home = ewl_button_new("Home");
    ewl_callback_append(home, EWL_CALLBACK_CLICKED, home_cb, fd);
    ewl_object_set_fill_policy(EWL_OBJECT(home), EWL_FLAG_FILL_HFILL);
    ewl_container_append_child(EWL_CONTAINER(fd), home);
    ewl_widget_show(home);

    ewl_widget_show(fd);
}
```

If a use clicks on the open entry in the file menu, the `file_menu_open_cb` will be executed. When that happens we need to create the file dialog for the use to select the file to view.

In the same fashion as the main window, we create a window to hold the file dialog and set its title, name and class. We request a default size, set its fill policy and attach a callback to handle the destruction of the window itself. We then attach a simple box into the window to hold the file dialog.

Once the window is setup, we make the call to create the file dialog. This is done with a call to `ewl_filedialog_new`, specifying the type of file dialog we wish to create. In this case we want a dialog to allow us to open a file, so we specify EWL_FILEDIALOG_TYPE_OPEN. We could have specified EWL_FILEDIALOG_TYPE_SAVE if we wished to use the dialog to save a file instead of open.

We then proceed to create an extra button to allow the user to navigate to their home directory with a single click. This is done by calling `ewl_button_new` and packing the subsequent button into the file dialog itself.

Example 10.12. File dialog destroy callback

```
/* close the file dialog */
static void destroy_filedialog_cb(Ewl_Widget *win, void *ev, void *data) {
    ewl_widget_hide(win);
    ewl_widget_destroy(win);
}
```

When we need to get rid of the file dialog we remove the widget from the screen with a call to `ewl_widget_hide`, and once it is no longer displayed we destroy the widget with a call to `ewl_widget_destroy`.

Example 10.13. File dialog open button callback

```
/* the file dialog open button callback */
```

```
static void open_file_cb(Ewl_Widget *win, void *ev, void *data) {
    char *text = NULL;
    text = read_file((char *)ev);

    if (text) {
        ewl_text_text_set(EWL_TEXT(data), text);
        free(text);
    }
    text = NULL;

    ewl_widget_hide(fd_win);
}
```

This callback will be executed when the user clicks the open button in the file dialog, or if the user double clicks on a file in a directory. The event passed (the ev parameter) will be the full path to the file that the use has selected.

In our case, we take that file and pass it to the function to read in the file and return the text of the file. Then using that text, as long as it is defined, we call `ewl_text_text_set` which will set the text of the given text object.

As the user has now finished their selection they file dialog is hidden from view.

Example 10.14. File dialog home button callback

```
/* the fd home button is clicked */
static void home_cb(Ewl_Widget *win, void *ev, void *data) {
    char *home = NULL;
    Ewl_Filedialog *fd = (Ewl_Filedialog *)data;

    home = getenv("HOME");
    if (home)
        ewl_filedialog_set_directory(fd, home);
}
```

If the user clicks on the "Home" button in the file dialog we want to display the contents of their home directory to them. We set the file dialog as the user data to the callback, so we cast that back to the Ewl_Filedialog and grabbing the home directory from the environment. The call to `ewl_filedialog_set_directory` changes the current directory the file dialog is displaying to be the users home directory.

Example 10.15. Read text file

```
/* read a file */
static char *read_file(char *file) {
    char *text = NULL;
    FILE *f = NULL;
    int read = 0, st_ret = 0;
    struct stat s;

    f = fopen(file, "r");
    st_ret = stat(file, &s);
```

```
    if (st_ret != 0) {
        if (st_ret == ENOENT)
            printf("not a file %s\n", file);
        return NULL;
    }

    text = (char *)malloc(s.st_size * sizeof(char));
    read = fread(text, sizeof(char), s.st_size, f);

    fclose(f);
    return text;
}
```

This is just a simple routine to take the given file, open it and read its contents into memory. Probably not the best idea for a real app, but is sufficient for this example program.

Example 10.16. Key press callback

```
/* a key was pressed */
static void key_up_cb(Ewl_Widget *win, void *ev, void *data) {
    Ewl_Event_Key_Down *e = (Ewl_Event_Key_Down *)ev;
    Ewl_ScrollPane *scroll = (Ewl_ScrollPane *)data;

    if (!strcmp(e->keyname, "q")) {
        destroy_cb(win, ev, data);

    } else if (!strcmp(e->keyname, "Left")) {
        double val = ewl_scrollpane_get_hscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_hscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 0)
            ewl_scrollpane_set_hscrollbar_value(EWL_SCROLLPANE(scroll),
                                                                  val - step);

    } else if (!strcmp(e->keyname, "Right")) {
        double val = ewl_scrollpane_get_hscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_hscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 1)
            ewl_scrollpane_set_vscrollbar_value(EWL_SCROLLPANE(scroll),
                                                                  val + step);

    } else if (!strcmp(e->keyname, "Up")) {
        double val = ewl_scrollpane_get_vscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_vscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 0)
            ewl_scrollpane_set_vscrollbar_value(EWL_SCROLLPANE(scroll),
                                                                  val - step);

    } else if (!strcmp(e->keyname, "Down")) {
        double val = ewl_scrollpane_get_vscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_vscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 1)
            ewl_scrollpane_set_vscrollbar_value(EWL_SCROLLPANE(scroll),
                                                                  val + step);
```

```
    }
}
```

The `key_up_cb` will be called whenever the user releases a key on the keyboard. The callback will receive an Ewl_Event_Key_Down structure containing the information on the key press itself. In our case we just need the keyname entry which is the name of the key that was pressed.

If the user presses the "q" key we just call the destroy callback and be done with it.

The "Left", "Right", "Up" and "Down" relate the the arrow keys on the users keyboard. If any of these keys are pressed we force the scrollpane to scroll in a specified direction.

In order to manipulate the scrollpane we need to know where it currently is in the file and the amount of distance each increment/decrement should travel. Luckily Ewl makes this easy. The call to ewl_scrollpane_get_[hv]scrollbar_value() will return the current value of the scroll bar. This is a double value in the range of [0, 1] inclusive. A value of 0 meaning the scrollbar is at the top and a value of 1 being at the bottom. The left and right work the same way, but 0 is absolute left and 1 is absolute right.

The second piece of information is obtained through the call to ewl_scrollpane_get_[hv]scrollbar_step(). The step is the distance the scrollpane will travel with one iteration. So using these two values we can then move the scrollbar in the correct direction with the call to ewl_scrollpane_set_[hv]scrollbar_value().

Example 10.17. Mouse wheel callback

```
/* the mouse wheel callback */
static void wheel_cb(Ewl_Widget *win, void *ev, void *data) {
    Ewl_Event_Mouse_Wheel *e = (Ewl_Event_Mouse_Wheel *)ev;
    Ewl_ScrollPane *scroll = (Ewl_ScrollPane *)data;

    if (e->z > 0) {
        double val = ewl_scrollpane_get_vscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_vscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 1)
            ewl_scrollpane_set_vscrollbar_value(EWL_SCROLLPANE(scroll),
                                                        val + step);
    } else {
        double val = ewl_scrollpane_get_vscrollbar_value(EWL_SCROLLPANE(scroll));
        double step = ewl_scrollpane_get_vscrollbar_step(EWL_SCROLLPANE(scroll));

        if (val != 0)
            ewl_scrollpane_set_vscrollbar_value(EWL_SCROLLPANE(scroll),
                                                        val - step);
    }
}
```

The final callback we need to deal with is for the mouse wheel. The event data passed to the mouse wheel callback is a Ewl_Event_Mouse_Wheel structure. From this structure we can see which way the user rolled the wheel by examining the "z" entry. If this value is 1, the use ran the wheel forward, if it is -1 the user ran the wheel backwards.

So we use the same logic from the key press callback to manipulate the scrollpane when the wheel is turned.

Example 10.18. Complication

```
zero@oberon [ewl_intro] -< gcc -Wall -o ewl_text main.c \
 `ewl-config --cflags --libs`
```

Compiling an ewl app is as simple as calling ewl-config and getting the --cflags and --libs.

Thats it. With that you should have a fully functioning Ewl application including menus, a file dialog and a text area with horizontal and vertical scrollbars. This example is just scratching the surface of the power contained inside of the Ewl toolkit with many other types of widgets available for use.

Chapter 11. Evoak

Evoak is a canvas server. This is similar to an X server that serves out a display and graphics operations. Evoak serves out a single canvas to be shared by multiple applications (clients) allowing each client to manipulate its set of objects on the canvas.

Recipe: Evoak hello client

dan dj2 sinclair <zero@perplexity.org>

This recipe is a very simple introduction to the world of Evoak programming. Building on grand traditions before, it displays the Canadian version of 'Hello World' on an Evoak canvas.

Example 11.1. Includes and pre-declarations

```
#include <Evoak.h>
#include <Ecore.h>

static unsigned int setup_called = 0;

static int canvas_info_cb(void *, int, void *);
static int disconnect_cb(void *, int, void *);
static void setup(Evoak *);
```

We need to include the Evoak header obviously, and the Ecore header is required to have access to the callback functions.

Example 11.2. main

```
int main(int argc, char ** argv) {
    Evoak *ev = NULL;

    if (!evoak_init()) {
        fprintf(stderr, "evoak_init failed");
        return 1;
    }

    ecore_event_handler_add(EVOAK_EVENT_CANVAS_INFO, canvas_info_cb, NULL);
    ecore_event_handler_add(EVOAK_EVENT_DISCONNECT, disconnect_cb, NULL);

    ev = evoak_connect(NULL, "evoak_intro", "custom");

    if (ev) {
        ecore_main_loop_begin();
        evoak_disconnect(ev);
    }

    evoak_shutdown();
    return 0;
}
```

Evoak needs to be setup initially with a call to `evoak_init`. This will setup all the internal libraries and requirements for Evoak.

As long as Evoak initializes correctly, we hook up two callbacks, the first is for canvas information and the second is for if we get disconnected from the Evoak server. These will be discussed later when the actual callbacks are displayed.

Once the callbacks are in place we need to connect to the Evoak canvas server. This is done through the call to `evoak_connect`. The parameters to `evoak_connect` are: the server to connect to, the client name and the client class. If the first argument is NULL, as in this example, the default Evoak server is connected too. The second argument to `ecore_connect` is the clients name, this value should be something unique as it will be used to distinguish the client from other clients. The final argument, the class, is the type of client, some of the possible values being: "background", "panel", "application" or "custom".

If the call to `evoak_connect` fails a NULL value will be returned. So, as long as we receive a Evoak object back, we start the main ecore loop. When ecore finishes we call `evoak_disconnect` to disconnect from the Evoak server.

We finish off by calling `evoak_shutdown` to clean up after ourselves.

Example 11.3. Canvas info callback

```
static int canvas_info_cb(void *data, int type, void *ev) {
    Evoak_Event_Canvas_Info *e = (Evoak_Event_Canvas_Info *)ev;

    if (!setup_called) {
        setup_called = 1;
        setup(e->evoak);
    }
    return 1;
}
```

A canvas info callback will be made when our client receives information about the Evoak server canvas. Wit this canvas information we can then proceed to setup our clients contents. This is contained inside of a setup_called flag as we only want to initialize once.

Example 11.4. Disconnect callback

```
static int disconnect_cb(void *data, int type, void *ev) {
    printf("disconnected\n");
    ecore_main_loop_quit();
    return 1;
}
```

The disconnect callback will the called when the client has been disconnected from the Evoak server. In this case, the simple solution of exiting is used.

Example 11.5. setup routine

```
static void setup(Evoak *ev) {
    Evoak_Object *o = NULL;

    evoak_freeze(ev);

    o = evoak_object_text_add(ev);
    evoak_object_text_font_set(o, "Vera", 12);
    evoak_object_color_set(o, 255, 0, 0, 255);
    evoak_object_text_text_set(o, "Hello Evoak, eh.");
    evoak_object_show(o);

    evoak_thaw(ev);
}
```

The setup routine will be called once to setup the display of our client. For this example, the client only draws the text 'Hello Evoak, eh'.

The first thing we do is call `evoak_freeze`, this should keep us from getting any unwanted callbacks while we are setting up our interface. At the end of the function we call the reciprocal `evoak_thaw` to undo the previous freeze.

We then proceed to create a text object with `evoak_object_text_add` and taking that text object, set the font, colour and text contents with the calls to, `evoak_object_text_font_set`, `evoak_object_color_set`, and `evoak_object_text_text_set` respectively.

Example 11.6. Compilation

```
zero@oberon [evoak_intro] -> gcc -o hello_evoak main.c \
 `evoak-config --cflags --libs`
```

As with many of the other EFL based libraries, compiling an Evoak application is as simple as calling the evoak-config program and getting the --cflags and --libs contents.

Thats it, this was a really simple introduction into Evoak and the surface remains unscratched as to the potential available for client applications.