

My Project

Generated by Doxygen 1.8.3.1

Tue Sep 16 2014 17:02:00

Contents

1	EFL	1
2	Ecore Examples	3
3	ecore_time - Differences between time functions	5
4	ecore timers - Scheduled events	7
5	ecore idle state - Idlers, enterers and exiters	9
6	ecore_job - Queuing tasks	11
7	Handling events example	13
8	ecore events and handlers - Setup and use	15
9	ecore fd handlers - Monitoring file descriptors	17
10	ecore poller - Repetitive polling tasks	19
11	Ecore_Con - DNS lookup	21
12	Ecore_Con_Url - downloading a file	23
13	Ecore_Con_Url - Managing cookies	25
14	Ecore_Con_Url - customizing a request	27
15	Ecore_Con - Creating a server	29
16	Ecore_Con - Creating a client	33
17	tutorial_ecore_pipe_gstreamer_example	37
18	tutorial_ecore_pipe_simple_example	39
19	Ecore animator example	41
20	Ecore_Thread - API overview	43

21	Ecore Evas Callbacks	45
22	Ecore_Evas window size hints	47
23	Ecore Evas Object example	49
24	Ecore Evas basics example	51
25	Ecore_Evas buffer example	53
26	Ecore_Evas (image) buffer example	55
27	Ecore_exe	57
28	ecore_imf - How to handle preedit and commit string from Input Method Framework	59
29	Edje Examples	61
30	Edje basics example	63
31	Edje Nested Part (hierarchy) example	67
32	Swallow example	69
33	Edje Text example	71
34	Table example	73
35	Box example - basic usage	75
36	Box example - custom layout	77
37	Dragable parts example	79
38	Perspective example	81
39	Edje signals and messages	83
40	Edje Color Class example	87
41	Edje Animations example	91
42	Multisense example	95
43	Edje basics example 2	97
44	Swallow example 2	99
45	Edje Signals example 2	101
46	Edje animations example 2	103

47 EET Examples	105
48 Very basic Eet example	107
49 Example of the various ways to interface with an Eet File	109
50 Simple data example	111
51 Nested data example	113
52 File descriptor data example	115
53 File descriptor data example, with Eet unions and variants	117
54 Eet data cipher/decipher example	119
55 Eina Examples	121
56 Eio Examples	123
57 eio_dir_copy() tutorial	125
58 eio_dir_stat_ls() tutorial	127
59 eio_file_ls() tutorial	129
60 eio_monitor_add() tutorial	131
61 eio_dir_direct_ls() tutorial	133
62 Eldbus Examples	135
63 Emotion Examples	137
64 Emotion - Basic library usage	139
65 Eo Tutorial	141
65.1 Purpose	141
65.2 Description	141
65.3 How to use it?	141
65.4 Important to know	142
65.5 How to create a class - H side?	143
65.6 How to create a class - C side?	144
66 EPhysics Examples	147
67 EPhysics - Bouncing Ball	149
67.1 A test struct	150

67.2	World Initialization	150
67.3	Render geometry	150
67.4	Adding boundaries	150
67.5	Adding a ball	151
67.6	Making it jump	151
68	test_bouncing_ball.c	153
68.1	ephysics_test.h	153
68.2	test_bouncing_ball.c	153
69	EPhysics - Bouncing Text	155
69.1	Creating the text	156
69.2	Creating the body	156
69.3	Binding	156
70	test_bouncing_text.c	157
70.1	ephysics_test.h	157
70.2	test_bouncing_text.c	157
71	EPhysics - Camera	159
71.1	Camera Data Struct	160
71.2	Adding a Camera	160
71.3	Updating the floor	160
72	test_camera.c	163
72.1	ephysics_test.h	163
72.2	test_camera.c	163
73	EPhysics - Camera Track	165
73.1	Track Data Struct	166
73.2	Adding a Camera	166
73.3	Updating the floor	166
74	test_camera_track.c	169
74.1	ephysics_test.h	169
74.2	test_camera_track.c	169
75	EPhysics - Collision Detection	171
75.1	Collision Data Struct	172
75.2	Adding the Callbacks	172
75.3	Collision Function	172
76	test_collision_detection.c	173
76.1	ephysics_test.h	173

76.2 test_collision_detection.c	173
77 EPhysics - Collision Filter	175
77.1 Adding the Callbacks	176
78 test_collision_filter.c	177
78.1 ephysics_test.h	177
78.2 test_collision_filter.c	177
79 EPhysics - Delete Body	179
79.1 Adding the Callbacks	179
80 test_delete.c	181
80.1 ephysics_test.h	181
80.2 test_delete.c	181
81 EPhysics - Constraint	183
81.1 Adding a constraint	183
82 test_constraint.c	185
82.1 ephysics_test.h	185
82.2 test_constraint.c	185
83 EPhysics - Forces	187
83.1 Adding a Force	188
84 test_forces.c	189
84.1 ephysics_test.h	189
84.2 test_forces.c	189
85 EPhysics - Growing Balls	191
85.1 Adding the growing/shrinking	192
86 test_growing_balls.c	193
86.1 ephysics_test.h	193
86.2 test_growing_balls.c	193
87 EPhysics - Gravity	195
87.1 Setting Gravity	196
87.2 Stopping a Body	196
88 test_no_gravity.c	197
88.1 ephysics_test.h	197
88.2 test_no_gravity.c	197

89 EPhysics - Logo	199
89.1 Logo Data Struct	199
89.2 Adding the letters	199
89.3 Adding the letter E	200
90 ephysics_logo.c	203
90.1 ephysics_logo.c	203
91 EPhysics - Rotating Forever	205
91.1 Rotating	205
92 test_rotating_forever.c	207
92.1 ephysics_test.h	207
92.2 test_rotating_forever.c	207
93 EPhysics - Velocity	209
93.1 Velocity Data Struct	209
93.2 Adding the Callbacks	209
93.3 Velocity Function	209
94 test_velocity.c	211
94.1 ephysics_test.h	211
94.2 test_velocity.c	211
95 EPhysics - Shapes	213
95.1 Adding a Shape	214
96 test_shapes.c	215
96.1 ephysics_test.h	215
96.2 test_shapes.c	215
97 EPhysics - Sleeping Threshold	217
97.1 Adding Max Sleeping Time	217
97.2 Adding a Sleeping Threshold	217
97.3 Adding a Damping	217
98 test_sleeping_threshold.c	219
98.1 ephysics_test.h	219
98.2 test_sleeping_threshold.c	219
99 EPhysics - Slider	221
99.1 Adding a Slider	222
100test_slider.c	223

100.1ephysics_test.h	223
100.2test_slider.c	223
101Evas Examples	225
102Simple Evas canvas example	227
103Evas' init/shutdown routines example	229
104Some image object functions examples	231
105Some more image object functions examples (2nd block)	233
106Evas events (canvas and object ones) and some canvas operations example	235
107Evas objects basic manipulation example	239
108Evas aspect hints example	241
109Evas alignment, minimum size, maximum size, padding and weight hints example	243
110Evas box example	245
111Evas object stacking functions (and some event handling)	251
112Evas Map - Overview	257
113Evas object smart objects	259
114Evas object smart interfaces	263
115Evas text object example	265
116Table Smart Object example	267
117Authors	269
118pkgconfig	273
118.1Introduction	273
118.2Usage	273
118.3Troubleshooting	273
119Module Index	275
119.1Modules	275
120Module Documentation	277
120.1Eo	277
120.2Evas	278

120.3Eet	279
120.4Eina	280
120.5Embryo	281
120.6Evil	282
120.7Escape	283
120.8Ecore	284
120.9Eio	285
120.10Eldbus	286
120.11Efreet	287
120.12Eeze	288
120.13Edje	289
120.14Emotion	290
120.15Thumb	291
121Example Documentation	293
121.1banshee.c	293
121.2client.c	293
121.3complex-types-client-eina-value.c	293
121.4complex-types-server.c	293
121.5complex-types.c	293
121.6connman-list-services.c	293
121.7ecore_animator_example.c	294
121.8ecore_con_client_simple_example.c	294
121.9ecore_con_lookup_example.c	294
121.10ecore_con_server_simple_example.c	294
121.11ecore_con_url_cookies_example.c	294
121.12ecore_con_url_download_example.c	294
121.13ecore_con_url_headers_example.c	294
121.14ecore_evas_basics_example.c	294
121.15ecore_evas_buffer_example_01.c	295
121.16ecore_evas_buffer_example_02.c	295
121.17ecore_evas_callbacks.c	295
121.18ecore_evas_object_example.c	295
121.19ecore_evas_window_sizes_example.c	295
121.20ecore_event_example_01.c	295
121.21ecore_event_example_02.c	295
121.22ecore_exe_example.c	295
121.23ecore_exe_example_child.c	295
121.24ecore_fd_handler_example.c	296
121.25ecore_fd_handler_gnutls_example.c	296

121.26	core_idler_example.c	296
121.27	core_job_example.c	296
121.28	core_pipe_gstreamer_example.c	296
121.29	core_pipe_simple_example.c	296
121.30	core_poller_example.c	296
121.31	core_thread_example.c	296
121.32	core_time_functions_example.c	296
121.33	core_timer_example.c	297
121.34	dje-basic.c	297
121.35	dje-box.c	297
121.36	dje-box2.c	297
121.37	dje-color-class.c	297
121.38	dje-drag.c	297
121.39	dje-perspective.c	297
121.40	dje-signals-messages.c	297
121.41	dje-swallow.c	297
121.42	dje-table.c	297
121.43	dje-text.c	297
121.44	et-basic.c	298
121.45	et-data-cipher_decipher.c	298
121.46	et-data-file_descriptor_01.c	298
121.47	et-data-file_descriptor_02.c	298
121.48	et-data-nested.c	298
121.49	et-data-simple.c	298
121.50	et-file.c	298
121.51	ina_accessor_01.c	298
121.52	ina_array_01.c	298
121.53	ina_array_02.c	298
121.54	ina_error_01.c	298
121.55	ina_file_01.c	299
121.56	ina_hash_01.c	299
121.57	ina_hash_02.c	299
121.58	ina_hash_03.c	299
121.59	ina_hash_04.c	299
121.60	ina_hash_05.c	299
121.61	ina_hash_06.c	299
121.62	ina_hash_07.c	299
121.63	ina_hash_08.c	299
121.64	ina_inarray_01.c	299
121.65	ina_inarray_02.c	299

121.60	ina_inlist_01.c	300
121.61	ina_inlist_02.c	300
121.62	ina_inlist_03.c	300
121.63	ina_iterator_01.c	300
121.70	ina_list_01.c	300
121.71	ina_list_02.c	300
121.72	ina_list_03.c	300
121.73	ina_list_04.c	300
121.74	ina_log_01.c	300
121.75	ina_log_02.c	300
121.76	ina_log_03.c	300
121.77	ina_magic_01.c	301
121.78	ina_model_01.c	301
121.79	ina_model_02.c	301
121.80	ina_model_03.c	301
121.81	ina_model_04_animal.c	301
121.82	ina_model_04_child.c	301
121.83	ina_model_04_human.c	301
121.84	ina_model_04_main.c	301
121.85	ina_model_04_parrot.c	301
121.86	ina_model_04_whistler.c	301
121.87	ina_simple_xml_parser_01.c	301
121.88	ina_str_01.c	302
121.89	ina_strbuf_01.c	302
121.90	ina_stringshare_01.c	302
121.91	ina_tiler_01.c	302
121.92	ina_value_01.c	302
121.93	ina_value_02.c	302
121.94	ina_value_03.c	302
121.95	io_file_ls.c	302
121.96	motion_basic_example.c	302
121.97	motion_signals_example.c	302
121.98	motion_test_main.c	303
121.99	physics_logo.c	303
121.100	as-aspect-hints.c	303
121.101	as-box.c	303
121.102	as-buffer-simple.c	303
121.103	as-events.c	303
121.104	as-hints.c	303
121.105	as-images.c	303

121.100as-images2.c	303
121.107as-init-shutdown.c	303
121.108as-map-utils.c	304
121.109as-object-manipulation.c	304
121.110as-smart-interface.c	304
121.111as-smart-object.c	304
121.112as-stacking.c	304
121.113as-table.c	304
121.114as-text.c	304
121.115ono-dial.c	304
121.116rver.c	304
121.117mple-signal-emit.c	304
121.118st_bouncing_ball.c	305
121.119st_bouncing_text.c	305
121.120st_camera.c	305
121.121st_camera_track.c	305
121.122st_collision_detection.c	305
121.123st_collision_filter.c	305
121.124st_constraint.c	305
121.125st_delete.c	305
121.126st_forces.c	305
121.127st_growing_balls.c	305
121.128st_no_gravity.c	305
121.129st_rotating_forever.c	306
121.130st_shapes.c	306
121.131st_sleeping_threshold.c	306
121.132st_slider.c	306
121.133st_velocity.c	306

Index

306

Chapter 1

EFL

The Enlightenment Project covers more than simple window management. It also includes the EFL, or Enlightenment Foundation Libraries: a framework which provides a great deal of functionality. Below you can find documentation auto-generated daily from GIT source for these libraries:

- ecore_main operating system abstraction and integration.
- edje_main layout and theme library with super powers.
- eet_main binary data parser and serializer.
- eeze_main hardware device manipulation and notification.
- efreet_main freedesktop.org (xdg) menu and desktop integration.
- eina_main data types and low-level/basic abstractions.
- eio_main asynchronous input/output
- eldbus_main d-bus integration.
- embryo_main embedded script language.
- emotion_main to play music and videos.
- eo_main generic object system.
- ephyscis_main physics simulation integration and visual effects.
- escape_main playstation3 portability layer.
- ethumb_main to generate thumbnail images of files.
- evas_main drawing canvas.
- evil_main microsoft windows portability layer.

Chapter 2

Ecore Examples

Here is a page with some Ecore examples explained:

- [ecore_time](#) - Differences between time functions
- [ecore timers](#) - Scheduled events
- [ecore idle state](#) - Idlers, enterers and exiters
- [ecore_job](#) - Queuing tasks
- [Handling events example](#)
- [ecore events and handlers](#) - Setup and use
- [ecore fd handlers](#) - Monitoring file descriptors
- [ecore poller](#) - Repetitive polling tasks
- [Ecore_Con](#) - DNS lookup
- [Ecore_Con_Url](#) - downloading a file
- [Ecore_Con](#) - Creating a server
- [Ecore_Con](#) - Creating a client
- [Ecore Evas Callbacks](#)
- [Ecore Evas Object example](#)
- [Ecore Evas basics example](#)
- [Ecore_Evas](#) window size hints
- [Ecore_Evas](#) buffer example
- [Ecore_Evas](#) (image) buffer example
- [Ecore_exe](#)
- [ecore_imf](#) - How to handle preedit and commit string from Input Method Framework

Chapter 3

ecore_time - Differences between time functions

This example shows the difference between calling `ecore_time_get()`, `ecore_loop_time_get()` and `ecore_time_unix_get()`.

It initializes `ecore`, then sets a timer with a callback that, when called, will retrieve the system time using these 3 different functions. After displaying the time, it sleeps for 1 second, then call display the time again using the 3 functions.

Since everything occurs inside the same main loop iteration, the internal `ecore` time variable will not be updated, and calling `ecore_loop_time_get()` before and after the `sleep()` call will return the same result.

The two other functions will return a difference of 1 second, as expected. But `ecore_time_unix_get()` returns the number of seconds since 00:00:00 1st January 1970, while `ecore_time_get()` will return the time since a unspecified point, but that never goes back in time, even when the timezone of the machine changes.

Note

The usage of `ecore_loop_time_get()` should be preferred against the two other functions, for most time calculations, since it won't produce a system call to get the current time. Use `ecore_time_unix_get()` when you need to know the current time and date, and `ecore_time_get()` when you need a monotonic and more precise time than `ecore_loop_time_get()`.

Chapter 4

ecore timers - Scheduled events

This example shows how to setup timer callbacks. It starts a timer that will tick (expire) every 1 second, and then setup other timers that will expire only once, but each of them will affect the first timer still executing with a different API, to demonstrate its usage. To see the full code for this example, click [here](#).

To demonstrate this, let's define some constants that will determine at which time each timer will expire:

These constants should tell by themselves what will be the behavior of the program, but I'll explain it anyway. The first timer is set to tick every 1 second, but all the other timers until the 6th one will be started concurrently at the beginning of the program. Each of them will expire at the specified time in these constants:

- The timer2, after 3 seconds of the program being executed, will add a delay of 3 seconds to timer1;
- The timer3 will pause timer1 at 8.2 seconds;
- timer4 will resume timer1 at 11.0 seconds;
- timer5 will change the interval of timer1 to 2 seconds;
- timer6 will stop timer1 and start timer7 and timer8, with 1.1 and 1.2 seconds of interval, respectively; it also sets the precision to 0.2 seconds;
- timer7 and timer8 will just print their expiration time.

As almost all the other examples, we create a context structure to pass to our callbacks, so they can have access to the other timers. We also store the time of the program start in `_initial_time`, and use the function `__get_current_time` to retrieve the current time relative to that time. This will help demonstrate what is going on.

Now, the behavior and relationship between the timers that was described above is dictated by the following timer callbacks:

It's possible to see the same behavior as other Ecore callbacks here, returning `ECORE_CALLBACK_RENEW` when the timer needs to continue ticking, and `ECORE_CALLBACK_CANCEL` when it needs to stop its execution. Also notice that later on our program we are checking for the timers pointers in the context to see if they are still executing before deleting them, so we need to set these timer pointers to `NULL` when we are returning `ECORE_CALLBACK_CANCEL`. Otherwise the pointer would still be not `NULL`, but pointing to something that is invalid, since the timer would have already expired without renewing.

Now the main code, which will start the timers:

This code is very simple. Just after starting the library, it will save the current time to `_initial_time`, start all timers from 1 to 6, and begin the main loop. Everything should be running right now, displaying the time which each timer is expiring, and what it is doing to affect the other timers.

After returning from the main loop, every timer is checked to see if it's still alive and, in that case, deleted, before finalizing the library. This is not really necessary, since `ecore_shutdown()` will already delete them for you, but it's good practice if you have other things going on after this point that could restart the main loop.

Chapter 5

ecore idle state - Idlers, enterers and exiters

This example demonstrates how to manage the idle state of the main loop. Once a program knows that the main loop is going to enter in idle state, it could start doing some processing until getting out of this state.

To exemplify this, we also add events and a timer to this program, so we can see the idle exiter callback being called before processing the event and/or timer, the event/timer callback being called (processed), then the idle enterer being called before entering in idle state again. Once in idle, the main loop keeps calling the idler callback continuously until a new event or timer is received.

First, we declare a struct that will be used as context to be passed to every callback. It's not useful everywhere, since this example is very simple and doesn't do anything other than printing messages, but using this context will make it a little bit more real. Our context will be used to delete the timer, idler, idle enterer and exiter, and the event handler, and also to count how many times the idler was called.

Then we start declaring callbacks for the idle enterer, idle exiter and the idler itself. Idle enterer and exiter callbacks just print a message saying that they were called, while the idler, in addition to printing a message too, also sends an event every 10 times that it is called, incrementing the context count variable. This event will be used to make the main loop exit the idle state and call the event callback.

These callbacks return `ECORE_CALLBACK_RENEW`, since we want them to keep being called every time the main loop changes to/from idle state. Otherwise, if we didn't want them to be called again, they should return `ECORE_CALLBACK_CANCEL`.

The next function declared is the event callback `_event_handler_cb`. It will check if the idler was called more than 100 times already (`ctxt->count > 100`), and will delete the idler, idle enterer and exiter, the timer (if it still exists), and request that the main loop stop running. Then it returns `ECORE_CALLBACK_DONE` to indicate that the event shouldn't be handled by any other callback.

Finally, we add a callback to the timer, that will just print a message when it is called, and this will happen only once (`ECORE_CALLBACK_CANCEL` is being returned). This timer callback is just here to show that the main loop gets out of idle state when processing timers too.

The **main** function is simple, just creates a new type of event that we will use to demonstrate the event handling together with the idle state, adds the callbacks that we declared so far, fill the context struct, and starts running the main loop.

Note

We use timer and event callbacks to demonstrate the idle state changing, but it also happens for file descriptor handlers, pipe handlers, etc.

Chapter 6

ecore_job - Queuing tasks

This example shows how an `Ecore_Job` can be added, how it can be deleted, and that they always execute in the added order.

First, 2 callback functions are declared, one that prints strings passed to it in the `data` pointer, and another one that quits the main loop. In the `main` function, 3 jobs are added using the first callback, and another one is added using the second one.

Then the second added job is deleted just to demonstrate the usage of `ecore_job_del()`, and the main loop is finally started. Run this example to see that `job1`, `job3` and `job_quit` are ran, in this order.

Chapter 7

Handling events example

This example shows the simplest possible way to register a handler for an ecore event, this way we can focus on the important aspects. The example will start the main loop and quit it when it receives the `ECORE_EVENT_SIGNAL_EXIT` event. This event is triggered by a `SIGTERM`(pressing ctrl+c).

So let's start with the function we want called when we receive the event, instead of just stopping the main loop we'll also print a message, that's just so it's clear that it got called:

Note

We return `ECORE_CALLBACK_DONE` because we don't want any other handlers for this event to be called, the program is quitting after all.

We then have our main function and the obligatory initialization of ecore:

We then get to the one line of our example that makes everything work, the registering of the callback:

Note

The `NULL` there is because there is no need to pass data to the callback.

And the all that is left to do is start the main loop:

Full source code for this example: [ecore_event_example_01.c](#).

Chapter 8

ecore events and handlers - Setup and use

This example shows how to create a new type of event, setup some event handlers to it, fire the event and have the callbacks called. After finishing, we delete the event handlers so no memory will leak.

See the full source code for this example [here](#).

Let's start the example from the beginning:

First thing is to declare a struct that will be passed as context to the event handlers. In this structure we will store the event handler pointers, and two strings that will be used by the first event handler. We also will use a global integer to store the event type used for our event. It is initialized with 0 in the beginning because the event wasn't created yet. Later, in the main function we will use `ecore_event_type_new()` to associate another value to it. Now the event handler callbacks:

This is the first event handler callback. It prints the event data received by the event, and the data passed to this handler when it was added. Notice that this callback already knows that the event data is an integer pointer, and that the handler data is a string. It knows about the first one because this is based on the type of event that is going to be handled, and the second because it was passed to the `ecore_event_handler_add()` function when registering the event handler.

Another interesting point about this callback is that it returns `ECORE_CALLBACK_DONE` (0) if the event data is even, swallowing the event and don't allowing any other callback to be called after this one for this event. Otherwise it returns `ECORE_CALLBACK_PASS_ON`, allowing the event to be handled by other event handlers registered for this event. This makes the second event handler be called just for "odd" events.

The second event handler will check if the event data is equal to 5, and if that's the case, it will change the event handler data of the first event handler to another string. Then it checks if the event data is higher than 10, and if so, it will request the main loop to quit.

An interesting point of this example is that although the second event handler requests the main loop to finish after the 11th event being received, it will process all the events that were already fired, and call their respective event handlers, before the main loop stops. If we didn't want these event handlers to be called after the 11th event, we would need to unregister them with `ecore_event_handler_del()` at this point.

Now some basic initialization of the context, and the Ecore library itself:

This last line is interesting. It creates a new type of event and returns a unique ID for this event inside Ecore. This ID can be used anywhere else in your program to reference this specific type of event, and to add callbacks to it.

It's common if you are implementing a library that declares new types of events to export their respective types as `extern` in the header files. This way, when the library is initialized and the new type is created, it will be available through the header file to an application using it add some callbacks to it. Since our example is self-contained, we are just putting it as a global variable.

Now we add some callbacks:

This is very simple. Just need to call `ecore_event_handler_add()` with the respective event type, the callback function to be called, and a data pointer that will be passed to the callback when it is called by the event.

Then we start firing events:

This `for` will fire 16 events of this type. Notice that the events will be fired consecutively, but any callback will be called yet. They are just called by the main loop, and since it wasn't even started, nothing happens yet. For each event fired, we allocate an integer that will hold the number of the event (we are arbitrarily creating these numbers just for demonstration purposes). It's up to the event creator to decide which type of information it wants to give to the event handler, and the event handler must know what is the event info structure for that type of event.

Since we are not allocating any complex structure, just a simple integer, we don't need to pass any special free function to `ecore_event_add()`, and it will use a simple `free` on our data. That's the default behavior.

Now finishing our example:

We just start the main loop and watch things happen, waiting to shutdown Ecore when the main loop exits and return.

Chapter 9

ecore fd handlers - Monitoring file descriptors

This is a very simple example where we will start monitoring the stdin of the program and, whenever there's something to be read, we call our callback that will read it.

Check the full code for this example [here](#).

This seems to be stupid, since a similar result could be achieved by the following code:

```
while (nbytes = read(STDIN_FILENO, buf, sizeof(buf)))
{
    buf[nbytes - 1] = '\0';
    printf("Read %zd bytes from input: \"%s\"\n", nbytes - 1, buf);
}
```

However, the above code is blocking, and won't allow you to do anything else other than reading the input. Of course there are other methods to do a non-blocking reading, like setting the file descriptor to non-blocking and keep looping always checking if there's something to be read, and do other things otherwise. Or use a `select` call to watch for more than one file descriptor at the same time.

The advantage of using an `Ecore_Fd_Handler` is that you can monitor a file descriptor, while still iterating on the Ecore main loop. It will allow you to have timers working and expiring, events still being processed when received, idlers doing its work when there's nothing happening, and whenever there's something to be read from the file descriptor, your callback will be called. And it's everything monitored in the same main loop, no threads are needed, thus reducing the complexity of the program and any overhead caused by the use of threads.

Now let's start our program. First we just declare a context structure that will be passed to our callback, with pointers to our handler and to a timer that will be used later:

Then we will declare a `prepare_callback` that is called before any `fd_handler` set in the program, and before the main loop `select` function is called. Just use one if you really know that you need it. We are just putting it here to exemplify its usage:

Now, our `fd handler`. In its arguments, the `data` pointer will have any data passed to it when it was registered, and the `handler` pointer will contain the `fd handler` returned by the `ecore_main_fd_handler_add()` call. It can be used, for example, to retrieve which file descriptor triggered this callback, since it could be added to more than one file descriptor, or to check what type of activity there's in the file descriptor.

The code is very simple: we first check if the type of activity was an error. It probably won't happen with the default input, but could be the case of a network socket detecting a disconnection. Next, we get the file descriptor from this handler (as said before, the callback could be added to more than one file descriptor), and read it since we know that it shouldn't block, because our `fd handler` told us that there's some activity on it. If the result of the read was 0 bytes, we know that it's an end of file (EOF), so we can finish reading the input. Otherwise we just print the content read from it:

Also notice that this callback returns `ECORE_CALLBACK_RENEW` to keep being called, as almost all other Ecore callbacks, otherwise if it returns `ECORE_CALLBACK_CANCEL` then the file handler would be deleted.

Just to demonstrate that our program isn't blocking in the input read but still can process other Ecore events, we are going to setup an `Ecore_Timer`. This is its callback:

Now in the main code we are going to initialize the library, and setup callbacks for the file descriptor, the prepare callback, and the timer:

Notice that the use of `ecore_main_fd_handler_add()` specifies what kind of activity we are monitoring. In this case, we want to monitor for read (since it's the standard input) and for errors. This is done by the flags `ECORE_FD_READ` and `ECORE_FD_ERROR`. For the three callbacks we are also giving a pointer to our context structure, which has pointers to the handlers added.

Then we can start the main loop and see everything happening:

In the end we are just deleting the fd handler and the timer to demonstrate the API usage, since Ecore would already do it for us on its shutdown.

Chapter 10

ecore poller - Repetitive polling tasks

This example shows how to setup, and explains how an `Ecore_Poller` is called. You can [see the full source code here](#).

In this example we store the initial time of the program just to use as comparison to the time when the poller callbacks are called. It will be stored in `_initial_time`:

Then next step is to define the poller callback. This callback assumes that a `data` pointer is passed to it on creation, and is a string just used to identify the poller. The callback prints this string and the time since the program started, and returns `ECORE_CALLBACK_RENEW` to keep being called.

Now in the main function we initialize `Ecore`, and save the initial time of the program, so we can compare it later with the time that the pollers are being called:

Then we change the poll interval to 0.3 seconds (the default is 0.125 seconds) just to show the API usage.

Finally, we create two pollers, one that will be called every 4 ticks, and another one that will be called every 8 ticks. This means the the first poller interval will be around 1.2 seconds, and the second one will be around 2.4 seconds. But the most important point is: since the second poller interval is a multiple of the first one, they will be always synchronized. `Ecore` calls pollers that are in the "same tick" together. It doesn't go back to the main loop and check if there's another poller to execute at this time, but instead it calls all the pollers registered to this "tick" at the same time. See the description of `ecore_poller_add()` for more details. This is easy to see in the time printed by both of them.

If instead of two synchronized pollers, we were using two different timers, one with interval of 1.2 seconds and another one with an interval of 2.4 seconds, there would be no guarantee that they would be totally in sync. Some delay in the execution of another task, or even in the task called in the callback, could make them get out of sync, forcing `Ecore`'s main loop to wake up more than necessary.

Well, this is the code that create these two pollers and set the poll interval, then starts the main loop:

If you hit CTRL-C during the execution of the program, the main loop will quit, since there are some signal handlers already set by default to do this. So after the main loop begin call, we change the second poller's interval to 16 ticks, so it will happen each 4.8 seconds (or each 4 times that the first poller is called).

This means: the program is started, the first poller is called each 4 ticks and the second is called each 8 ticks. After CTRL-C is used, the second poller will be called each 16 ticks.

The rest of the program is just deleting the pollers and shutting down the library.

Chapter 11

Ecore_Con - DNS lookup

This is a very simple example that shows how to make a simple DNS lookup using `ecore_con_lookup()`.

It's possible to see in the beginning of the main function that we are using the arguments passed via command line. This is the address that we are going to make the DNS lookup on.

The next step is to initialize the libraries, and just call `ecore_con_lookup()`. This function will get the string that contains the address to be resolved as first parameter, then a callback that will be called when the resolve stage is done, and finally a data pointer that will be passed to the callback.

This function is asynchronous, and the callback will be called only on success. If there was an error during the resolve stage, there's no way to know about that. It's only possible to know about errors when setting up the lookup, by looking at the return code of the `ecore_con_lookup()` function.

The callback `_lookup_done_cb` passed as argument to `ecore_con_lookup()` just prints the resolved canonical name, IP, address of the `sockaddr` structure, and the length of the socket address (in bytes).

Finally, we start the main loop, and after that we finalize the libraries and exit.

This is the code for this simple example:

Chapter 12

Ecore_Con_Url - downloading a file

This is a simple example that shows how to download a file using Ecore_Con_Url. The full source code for this example can be found at [ecore_con_url_download_example.c](#).

First we are setting some callbacks for events that will be sent when data arrives in our connection (the data is the content of the file being downloaded), and when the download is completed. The `_url_progress_cb` and `_url_complete_cb` are these callbacks:

Notice that we also declared a struct that will hold how many bytes were downloaded through this object. It will be set in the `main` function using `ecore_con_url_data_set()`.

In the next step, on the `main` function, we open a file where we are going to save the content being downloaded:

With the file successfully open, let's create our Ecore_Con_Url object. For this, we initialize the libraries and create the object:

Then we allocate and set the data struct to the connection object, and set a file descriptor from our previously open file to it. We also add the event handlers (callbacks) to the events that will be emitted on data being received and download complete:

Finally we start our request, and run the main loop:

The rest of this code was just freeing resources, with some labels to be used for error handling.

Chapter 13

Ecore_Con_Url - Managing cookies

This example shows how to use an Ecore_Con_Url and enable it to receive/send cookies. These cookies can be set by the server, saved to a file, loaded later from this file and sent again to the server. The complete example can be found at [ecore_con_url_cookies_example.c](#)

First we are setting some callbacks for events that will be sent when data arrives in our connection (the data is the content of the file being downloaded), and when the download is completed. The `_url_data_cb` and `_url_complete_cb` are these callbacks:

In the `main` function we parse some parameter from the command line. These parameters are the url that we are connecting to, and cookie use policy.

After that we initialize the libraries and create a handler to our request using the given url:

We also set the event handlers for this request and add a header to it, that will inform our custom user agent:

Now we start playing with cookies. First, let's call `ecore_con_url_cookies_init()` to inform that we want cookies enabled. We also set a file from which we are loading previously set (old) cookies, in case that we don't want to clear old cookies or old session cookies.

After that we set the file where we are going to save all valid cookies in the Ecore_Con_Url object. This includes previously loaded cookies (that weren't cleared) and new cookies set by the response header "Set-Cookie" that comes with the response to our request:

And finally, before performing the request, we check the command passed as argument in the command line and use it to choose between clearing old cookies, clearing just old session cookies, or ignoring old session cookies.

After that we just finish our code as expected:

Notice that in this code, if we want to clear old cookies, we also don't load them from the file. This is a bit confusing and the API isn't clear, but `ecore_con_url_cookies_file_add()` will load cookies from the specified files just when the operation is really performed (i.e. `ecore_con_url_get()` is called). So if `ecore_con_url_cookies_clear()` is called before `ecore_con_url_get()`, the old cookies may not have been loaded yet, so they are not cleared. To avoid having old cookies loaded, don't add any cookie file with `ecore_con_url_cookies_file_add()`.

The function `ecore_con_url_cookies_clear()` is just useful to clear cookies that are already loaded/valid in the Ecore_Con_Url object (from a previous request, for example).

Chapter 14

Ecore_Con_Url - customizing a request

This is a simple example that shows how to make a custom request using Ecore_Con_Url. The full source code for this example can be found at [ecore_con_url_headers_example.c](#).

The first part of the example is setting the callbacks to be called when an #ECORE_CON_EVENT_URL_DATA or #ECORE_CON_EVENT_URL_COMPLETE event is received. These are the callbacks that are going to be used with this:

The `main` code is as simple as the Ecore_Con_Url example. It contains some checks for the arguments to see if a GET or POST request is required:

Then we start our required libraries and configure a global option to use pipelined requests:

Now we create our request object, but using `ecore_con_url_custom_new()` to use a POST or GET method depending on the command line arguments. And we also add the event handlers for our callbacks:

In order to demonstrate our API, some options are set to this request before actually performing it:

Depending on what kind of request was asked (GET or POST), we use one of the specific functions to perform it:

After that, we just check for errors, start the main loop, free resources and finally exit:

Chapter 15

Ecore_Con - Creating a server

In this example we are going to create a server that listens for connections from clients through a TCP port. You can get the full source code at [ecore_con_server_simple_example.c](#).

We begin our example in the main function, to demonstrate how to setup things, and then go to the callbacks that are needed for it to run properly.

In the `main` function, after initializing the libraries, we use `ecore_con_server_add()` to startup the server. Look at the reference documentation of this function: it supports many types of server, and we are going to use `#ECORE_CON_REMOTE_TCP` (a TCP based server). Other arguments to this function are the address where we are listening on, the port, and a data pointer that will associate that data with the server:

Notice that we are listening only on `127.0.0.1`, which is the internal loopback interface. If the server needs to listening on all of its ips, use `0.0.0.0` instead.

We also need to set event handlers to be called when we receive any data from the clients, when a new client connects to our server, or when a client disconnects. These callbacks are:

More details about what these callbacks do will be given later.

Now, before running the main loop, we also want to set some limits to our server. To avoid it to be overloaded with too many connections to handle, we are going to set a maximum of 3 clients connected at the same time. This number is used just to demonstrate the API. A good number to be used here would need to be determined by tests done on the server, to check the load supported by it.

Any other client trying to connect to this server, after the limit is reached, will wait until one of the connected clients disconnect and the server accepts the new connection.

Another important thing to do is setting a timeout, to avoid that a client hold a connection for too long without doing anything. This timeout will disconnect the idle client, allowing that other clients that may be waiting to connect finally can do it.

Then we just start the main loop:

After exiting the main loop, we print the list of connected clients, and also free the data associated with each respective client. This data was previously associated using `ecore_con_client_data_set()`:

Then before exiting we show the total uptime of the server:

Now let's go back to the used callbacks.

The first callback, `_add`, is registered to the event `#ECORE_CON_EVENT_CLIENT_ADD`, which will be called whenever a client connects to the server.

This callback will associate a data structure to this client, that will be used to count how many bytes were received from it. It also prints some info about the client, and send a welcome string to it. `ecore_con_client_flush()` is used to ensure that the string is sent immediately, instead of being buffered.

A timeout for idle specific for this client is also set, to demonstrate that it is independent of the general timeout of the server.

Before exiting, the callback will display a list of all clients still connected to this server. The code for this callback follows:

The second callback is `_del`. It is associated with `#ECORE_CON_EVENT_CLIENT_DEL`, and is called whenever a client disconnects from this server.

It will just print some information about the client, free the associated data structure, and call `ecore_con_client_del()` on it before exiting the callback. Here's its code:

The last callback will print any data received by this server from its clients. It also increments the "bytes received" counter, `sdata`, in the data structure associated with this client. The callback code follows:

The important parts of this example were described above. If you need to see the full source code for it, there's a [link to the code](#) in the beginning of this page.

This example will start a server and start accepting connections from clients, as demonstrated in the following diagram:

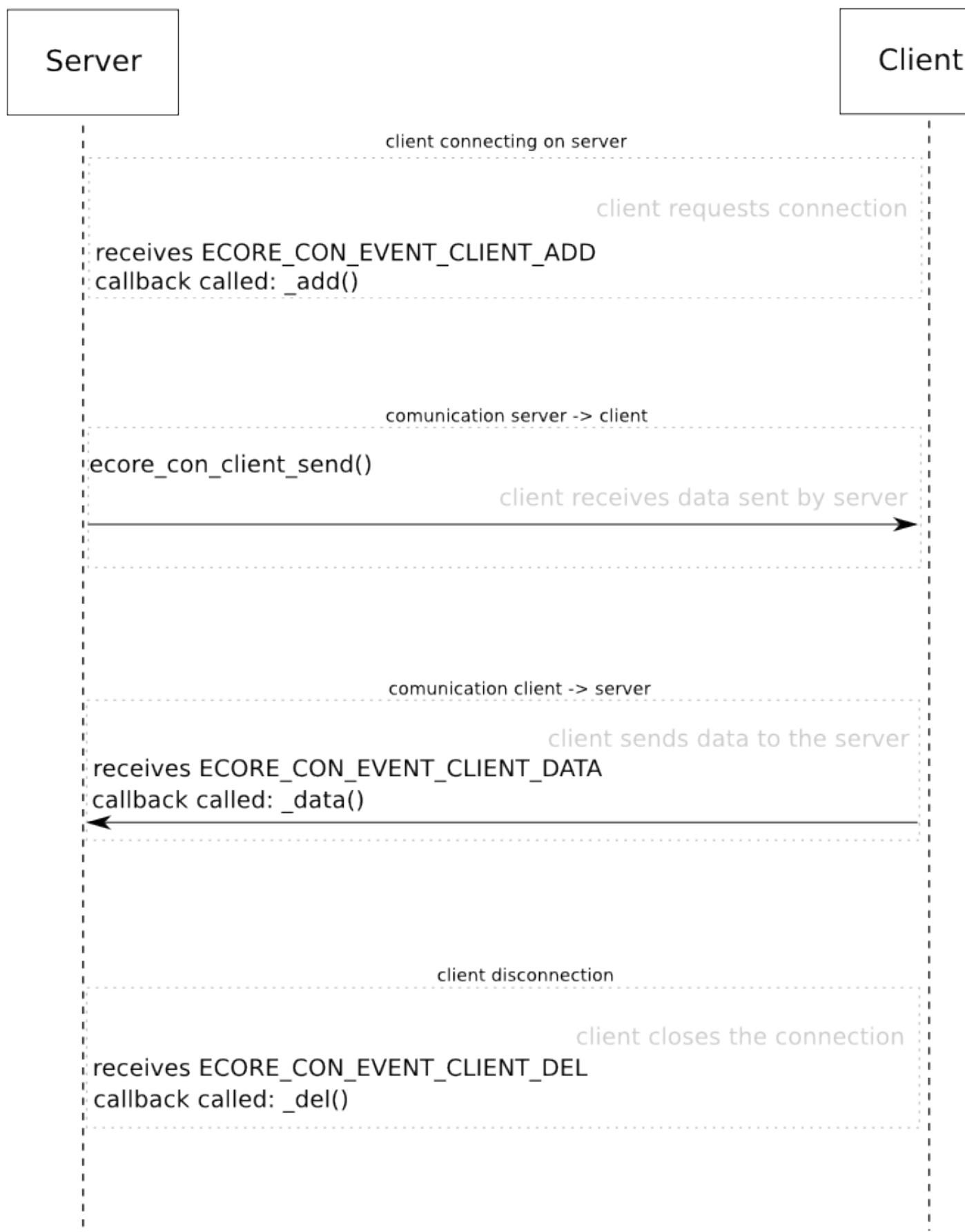


Figure 15.1: width=

Note

This example contains a serious security flaw: it doesn't check for the size of data being received, thus allowing to the string to be exploited in some way. However, it is left like this to make the code simpler and just demonstrate the API usage.

Chapter 16

Ecore_Con - Creating a client

Following the same idea as the [Ecore_Con - Creating a server](#) , this example will demonstrate how to create a client that connects to a specified server through a TCP port. You can see the full source code at [ecore_con_client_simple_example.c](#).

Starting from the `main` function, after reading the command line argument list and initializing the libraries, we try to connect to the server:

After doing this, everything else in `main` is setting up callbacks for the client events, starting the main loop and shutting down the libraries after it.

Now let's go to the callbacks. These callbacks are very similar to the server callbacks (our implementation for this example is very simple). On the `__add` callback, we just set a data structure to the server, print some information about the server, and send a welcome message to it:

The `__del` callback is as simple as the previous one. We free the data associated with the server, print the uptime of this client, and quit the main loop (since there's nothing to do once we disconnect):

The `__data` callback is also similar to the server data callback. it will print any received data, and increase the data counter in the structure associated with this server:

You can see the server counterpart functions of the ones used in this example in the [Ecore_Con - Creating a server](#).

This example will connect to the server and start communicating with it, as demonstrated in the following diagram:

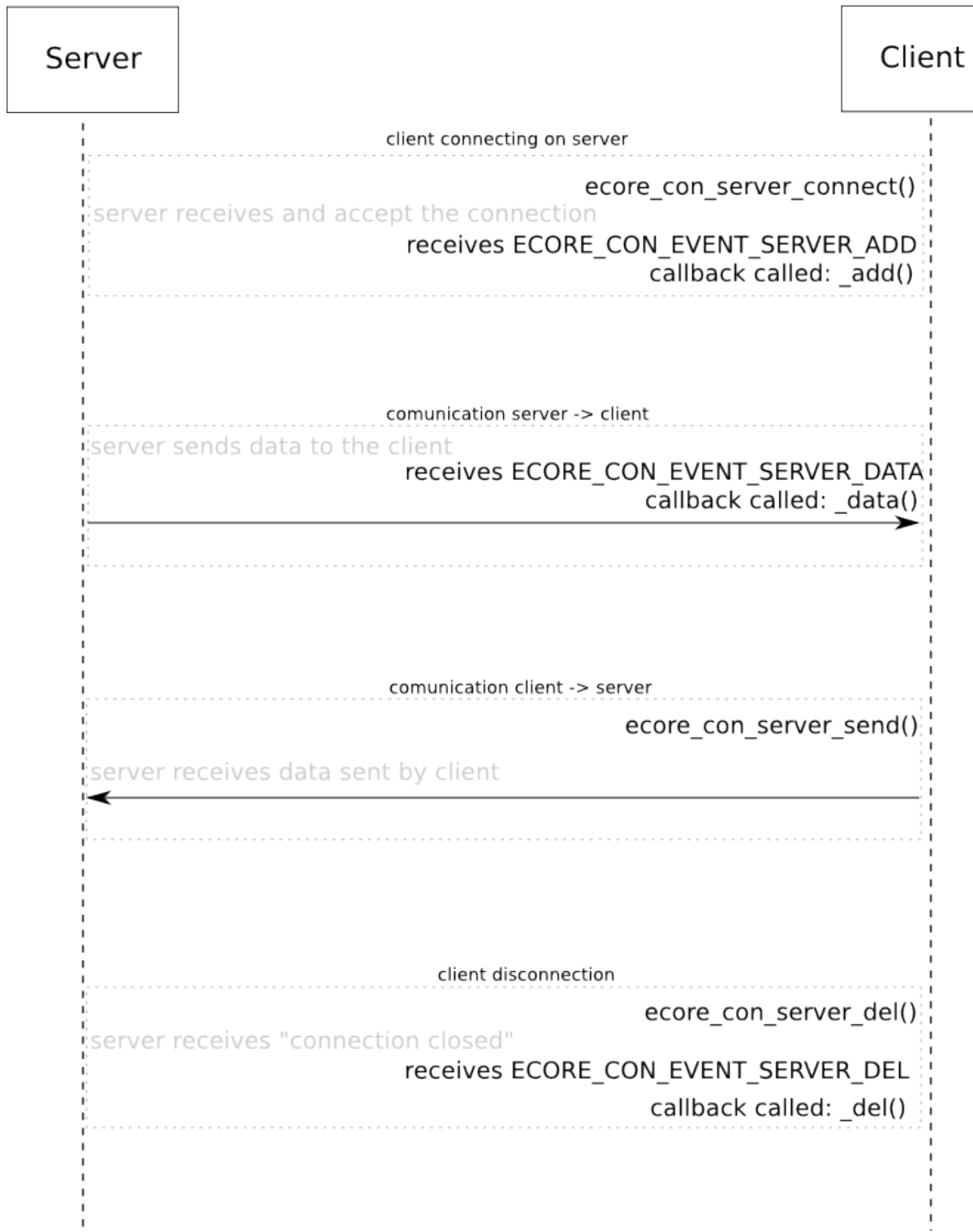


Figure 16.1: width=

Note

This example contains a serious security flaw: it doesn't check for the size of data being received, thus allowing to the string to be exploited in some way. However, it is left like this to make the code simpler and just demonstrate the API usage.

Chapter 17

tutorial_ecore_pipe_gstreamer_example

Here is an example that uses the pipe wrapper with a Gstreamer pipeline. For each decoded frame in the Gstreamer thread, a handle is called in the ecore thread.

Chapter 18

tutorial_ecore_pipe_simple_example

This example shows some simple usage of `ecore_pipe`. We are going to create a pipe, fork our process, and then the child is going to communicate to the parent the result of its processing through the pipe.

As always we start with our includes, nothing special:

The first thing we are going to define in our example is the function we are going to run on the child process, which, as mentioned, will do some processing and then will write the result to the pipe:

Note

The sleep was added so the parent process would think the child process was doing something interesting...

Next up is our function for handling data arriving in the pipe. It copies the data to another buffer, adds a terminating NULL and prints it. Also if it receives a certain string it stops the main loop(effectively ending the program):

And now on to our main function, we start by declaring some variables and initializing `ecore`:

And since we are talking about pipes let's create one:

Now we are going to fork:

Note

duh...

The child process is going to do the our fancy processing:

Note

It's very important to call `ecore_pipe_read_close()` here so that the child process won't read what it is writing to the pipe itself.

And the parent is going to run ecore's main loop waiting for some data:

Note

Calling `ecore_pipe_write_close()` here isn't important but since we aren't going to write in the pipe it is good practice.

And finally when done processing(the child) or done receiving(the parent) we delete the pipe and shutdown ecore:

Chapter 19

Ecore animator example

For this example we are going to animate a rectangle growing, moving and changing color, and then move it back to its initial state with a different animation. We are also going to have a second rectangle moving along the bottom of the screen. To do this we are going to use `ecore_evas`, but since that is not the focus here we won't go into detail about it.

All of this is just setup, not what we're interested in right now.

Now we are going to set the frametime for our animation to one fiftieth of a second, this will make our program consume more resources but should make our animation extra smooth:

And now we get right to the business of creating our `ecore_animator`:

Note

We are telling our animation to last 10 second and to call `_advance_frame` with `rect` as data.

So far we setup the first and second animations, the third one however is a bit different, this time we won't use a timeline animation, that's because we don't want our animation to stop:

Next we set a few timers to execute `_start_second_anim`, `_freeze_third_anim` and `_thaw_thir_anim` in 10, 5 and 10 seconds respectively:

And now we tell `ecore` to begin the main loop and free some resources once it leaves the main loop:

Here we have the callback function for our first animation, which first takes `pos` (where in the timeline we are), maps it to a `SPRING` curve that which will wobble 15 times and will decay by a factor of 1.2:

Now that we have the frame we can adjust the rectangle to its appropriate state:

And now the callback that will run 10 seconds after the program starts(5 seconds after the first animation finishes) and starts our second animation:

Note

For this animation we made the frametime much larger which means our animation might get "jerky".

The callback for our second animation, our savvy reader no doubt noted that it's very similar to the callback for the first animation. What we change for this one is the type of animation to BOUNCE and the number of times it will bounce to 50:

And for our last animation callback something simpler, we just move our rectangle right by one pixel until it reaches the end of the screen and then start at the beginning again:

Our next two functions respectively freezes and thaw our third animation, so that it won't happen for the 5 seconds after the first animation ends and the second animation begins:

Chapter 20

Ecore_Thread - API overview

Working with threads is hard. Ecore helps to do so a bit easier, but as the example in [ecore_thread_example.c](#) shows, there's a lot to consider even when doing the most simple things.

We'll be going through this thorough example now, showing how the different aspects of Ecore_Thread are used, but users are encouraged to avoid threads unless it's really the only option, as they always add more complexity than the program usually requires.

Ecore Threads come in two flavors, short jobs and feedback jobs. Short jobs just run the given function and are more commonly used for small tasks where the main loop does not need to know how the work is going in between. The short job in our example is so short we had to artificially enlarge it with `sleep()`. Other than that, it also uses threads local data to keep the data we are working with persistent across different jobs ran by the same system thread. This data will be freed when there are no more jobs are pending and the thread is terminated. If the data doesn't exist in the thread's storage, we create it and save it there for future jobs to find it. If creation fails, we cancel ourselves, so the main loop knows that we didn't just exit normally, meaning the job could not be done. The main part of the function checks in each iteration if it was canceled by the main loop, and if it was, it stops processing and clears the data from the storage (we assume `cancel` means no one else will need this, but this is really application dependent).

Feedback jobs, on the other hand, run tasks that will inform back to the main loop its progress, send partial data as is processed, just ping saying it's still alive and processing, or anything that needs the thread to talk back to the main loop.

Finally, one more feedback job, but this one will be running outside of Ecore's pool, so we can use the pool for real work and keep this very light function unchecked. All it does is check if some condition is met and send a message to the main loop telling it it's time to close.

Every now and then the program prints its status, counting threads running and pending jobs.

In our main loop, we'll be receiving messages from our feedback jobs using the same callback for both of them.

The light job running out of the pool will let us know when we can exit our program.

Next comes the handling of data sent from the actual worker threads, always remembering that the data belongs to us now, and not the thread, so it's our responsibility to free it.

Last, the condition to exit is given by how many messages we want to handle, so we need to count them and inform the condition checking thread that the value changed.

When a thread finishes its job or gets canceled, the main loop is notified through the callbacks set when creating the task. In this case, we just print what happen and keep track of one of them used to exemplify canceling. Here we are pretending one of our short jobs has a timeout, so if it doesn't finish before a timer is triggered, it will be canceled.

The main function does some setup that includes reading parameters from the command line to change its behaviour and test different results. These are:

- `-t <some_num>` maximum number of threads to run at the same time.
- `-p <some_path>` adds `some_path` to the list used by the feedback jobs. This parameter can be used multiple times.
- `-m <some_num>` the number of messages to process before the program is signalled to exit.

Skipping some bits, we init Ecore and our application data.

If any paths for the feedback jobs were given, we use them, otherwise we fallback to some defaults. Always initializing the proper mutexes used by the threaded job.

Initialize the mutex needed for the condition checking thread

And start our tasks.

To finalize, set a timer to cancel one of the tasks if it doesn't end before the timeout, one more timer for status report and get into the main loop. Once we are out, destroy our mutexes and finish the program.

Chapter 21

Ecore Evas Callbacks

Our example is remarkably simple, all it does is create an `Ecore_Evas` and register a callback for a bunch of events. What's interesting here is knowing when each of these callbacks will be called, however since that depends on the underlying windowing system there are no guarantees that all of the callbacks will be called for your windowing system. To know which callbacks will be called for your windowing system run the example and redirect the output to a file, and take a look at it.

Note

Make sure you minimize, resize, give and remove focus to see more callbacks called.

The example is constituted of two main parts, first is the implementation of callbacks that will be called for each event(all our callbacks do is print their own name) and the second is the main function where we register the event callbacks and run the main loop:

Chapter 22

Ecore_Evas window size hints

On this example, we show you how to deal with `Ecore_Evas` window size hints, which are implemented **per Evas engine**.

We start by defining an initial size for our window and, after creating it, adding a background white rectangle and a text object to it, to be used to display the current window's sizes, at any given time:

The program has a command line interface, responding to the following keys:

Use the `'m'` key to impose a minimum size of half the initial ones on our window. Test it by trying to resize it to smaller sizes than that:

The `'x'` key will, in turn, set a maximum size on our window – to two times our initial size. Test it by trying to resize the window to bigger sizes than that:

Window base sizes will override any minimum sizes set, so try it with the `'b'` key. It will set a base size of two times the initial one:

Finally, there's a key to impose a "step size" on our window, of 40 pixels. With than on (`'s'` key), you'll see the window will always be bound to **multiples** of that size, for dimensions on both axis:

The full example follows.

Chapter 23

Ecore Evas Object example

This example creates an `Ecore_Evas`(a window) and associates a background and a custom cursor for it.

We'll start looking at the association, which is quite simple. We choose to associate using `ECORE_EVAS_OBJECT_ASSOCIATE_BASE` to have it be resized with the window, since for a background that is what's most useful:

Note

If we didn't associate the background we'd need to listen to resize of `Ecore_Evas` and manually resize the background or have artifacts on our window.

We then check that the association worked:

Next we are going to set a custom cursor, for our cursor we are going to use a small green rectangle. Our cursor is going to be on layer 0 (any lower and it would be below the background and thus invisible) and clicks will be computed as happening on pixel 1, 1 of the image:

We then check every one of those parameters:

Here you have the full-source of the code:

Chapter 24

Ecore Evas basics example

This example will illustrate the usage of some basic Ecore_Evas functions. This example will list the available evas engines, check which one we used to create our window and set some data on our Ecore_Evas. It also allows you to hide/show all windows in this process (we only have one, but if there were more they would be hidden), to hide the windows type 'h' and hit return, to show them, type 's' and hit return.

The very first thing we'll do is initialize ecore_evas:

Once inited we query which engines are available:

We then create an Ecore_Evas(window) with the first available engine, on position 0,0 with size 200,200 and no especial flags, set its title and show it:

We now add some important data to our Ecore_Evas:

And since our data is dynamically allocated we'll need to free it when the Ecore_Evas dies:

We now print which Evas engine is being used for our example:

We are going to add a background to our window but before we can do that we'll need to get the canvas(Evas) on which to draw it:

We then do a sanity check, verifying if the Ecore_Evas of the Evas is the Ecore_Evas from which we got the Evas:

Now we can actually add the background:

To hide and show the windows of this process when the user presses 'h' and 's' respectively we need to know when the user types something, so we register a callback for when we can read something from `stdin`:

The callback that actually does the hiding and showing is pretty simple, it does a `scanf`(which we know won't block since there is something to read on `stdin`) and if the character is an 'h' we iterate over all windows calling `ecore_evas_hide` on them, if the character is an 's' we call `ecore_evas_show` instead:

Once all is done we run our main loop, and when that is done(application is exiting) we free our `Ecore_Evas` and shutdown the `ecore_evas` subsystem:

Here you have the full-source of the code:

Chapter 25

Ecore_Evas buffer example

Between the Evas examples, there is one in which one creates a canvas bound to the Evas **buffer** engine and uses its pixel contents to create an PPM image on disk. There, one does that by creating the canvas "by hand", with `evas_new()`, `evas_engine_info_set()`, etc.

On this example, we accomplish the very same task, but by using the `Ecore_Evas` helper wrapper functions on a buffer engine canvas. If you compare both codes, you'll see how much code one is saved from by using the `Ecore_Evas` wrapper functions.

The code is simple as it can be. After instantiating our canvas window, with `ecore_evas_buffer_new()`, we grab its canvas pointer and create the desired objects scene on it, which in this case is formed by 3 rectangles over the top left corner of a white background:

Since it's a buffer canvas and we're using it to only save its contents on a file, we even needn't `ecore_evas_show()` it. We make it render itself, forcefully, without the aid of Ecore's main loop, with `ecore_evas_manual_render()`:

And we're ready to save the window's shiny rendered contents as a simple PPM image. We do so by grabbing the pixels of the `Ecore_Evas`' internal canvas, with `ecore_evas_buffer_pixels_get()`:

Check that destination file for the result. The full example follows.

Chapter 26

Ecore_Evas (image) buffer example

In this example, we'll demonstrate the use of `ecore_evas_object_image_new()`. The idea is to have the same scene created for [Ecore_Evas buffer example](#) as the contents of an image object.

The canvas receiving this image object will have a white background, a red border image to delimit this image's boundaries and the image itself. After we create the special image, we set its "fill" property, place and resize it as we want. We have also to resize its underlying `Ecore_Evas` too, to the same dimensions:

Now, we re-create the scene we cited, using the sub-canvas of our image to parent the objects in question. Because image objects are created with the alpha channel enabled, by default, we'll be seeing our white rectangle beneath the scene:

And that's all. The contents of our image could be updated as one wished, and they would always be mirrored in the image's area.

Check that destination file for the result. The full example follows.

Chapter 27

Ecore_exe

Creating a processes and IPC (Inter process communication)

In this example we will show how to create a new process and communicate with it in a portable way using the Ecore_exe module.

In this example we will have two process and both will communicate with each other using messages. A father process will start a child process and it will keep sending messages to the child until it receives a message to quit. To see the full source use the links:

- [Father](#)
- [Child](#)

Let's start the tutorial. The implementation of the child it's pretty simple. We just read strings from stdin and write a message in the stdout. But you should be asking yourself right know. "If I'm receiving data from an other process why I'm reading and writing in stdin/stdout?". That's because, when you spawn a process using the Ecore_Exe module it will create a pipe between the father and the child process and the stdin/stdout of the child process will be redirected to the pipe. So when the child wants to receive or send data to the father, just use the stdin/stdout. However the steps to send data from the father to the child is quite different, but we will get there.

The child will register a fd handler to monitor the stdin. So we start registering the ecore FD handler:

If you don't remember the parameters of `ecore_main_fd_handler_add`, please check its documentation.

Now that we have our handler registered we will start the ecore's main loop:

Now let's take a look in the callback function. Its a simple function that will read from stdin 3 times and at the third time will say to the father: "quit".

You may notice that we are sending the messages to stdout, and our father will receive it. Also our string must have a "\n" because the string will be buffered in the pipe until it finds EOF or a "newline" in our case we won't have a EOF unless we close the pipe, so we use the "\n" char.

One more thing, we use `fflush(stdout)` because probably our message won't fill our entire buffer and the father would never receive the message. So we use this function to flush the buffer and the father can receive as fast as possible.

Now that we have our child ready, let's start our work in the father's source code.

We start creating the child process like this:

With the command above we are creating our child process, the first parameter is the command to be executed, the second are the pipe flags and in our case we will write and read in the pipe so we must say what we are doing in the pipe. You may notice the flag `ECORE_EXE_PIPE_READ_LINE_BUFFERED`, this means that reads are buffered until I find a newline. And the third parameter is data that we would like to send to the process in its creating. This case we are sending nothing, so just use `NULL`.

Then we check if the process was created:

After this we get the PID of the child process and just print it in the screen. The PID stands for Process identification. This is just an internal identifier of your process:

The way that `Ecore_exe` works is: when we want to read data sent from our child we must use an `ecore` event. So let's start register our event listener:

Now to send messages to our child we will use a timer, so every 1 second we will send a message to the child.

After all this we start the main loop. Now let's pass to the callback functions.

Now we will see how we actually send the data and receive it. Let's start with `_sendMessage`:

We use `ecore_exe_send` to send data to the child process, it's pretty simple. To know what the parameters stands for, check the docs.

Note

The function **`ecore_exe_send`** will never block your program, also there is no partial send of the data. This means either the function will send all the data or it will fail.

Now let's take a look in our event callback and see how we retrieve the messages.

It's just like an normal event, we get a reference to `Ecore_Exe_Event_Data`, extract the data and then show it in the screen.

And that's it, after all it's not complicated to create a process and communicate with it.

Chapter 28

ecore_imf - How to handle preedit and commit string from Input Method Framework

This example demonstrates how to connect input method framework and handle preedit and commit string from input method framework.

To input Chinese, Japanese, Korean and other complex languages, the editor should be connected with input method framework.

How to initialize and shutdown ecore imf module

- `ecore_imf_init()` should be called to initialize and load immodule.
- `ecore_imf_shutdown()` is used for shutdowning and unloading immodule.

How to create input context and register pre-edit and commit event handler

Each entry should have each input context to connect with input service framework. Key event is processed by input method engine. The result is notified to application through `ECORE_IMF_CALLBACK_PREEDIT_CHANGED` and `ECORE_IMF_CALLBACK_COMMIT` event.

The full example follows.

Chapter 29

Edje Examples

Examples:

- [Edje basics example](#)
- [Edje Nested Part \(hierarchy\) example](#)
- [Swallow example](#)
- [Swallow example 2](#)
- [Edje Text example](#)
- [Table example](#)
- [Edje Color Class example](#)
- [Edje signals and messages](#)
- [Box example - basic usage](#)
- [Box example - custom layout](#)
- [Dragable parts example](#)
- [Perspective example](#)
- [Edje Animations example](#)
- [Multisense example](#)
- [Edje basics example 2](#)
- [Edje Signals example 2](#)
- [Edje animations example 2](#)

Chapter 30

Edge basics example

In this example, we illustrate how to start using the Edge library, with the very basic one needs to instantiate an Edge object.

We place, in the canvas, an Edge object along with a **red** border image to delimit its geometry. After we instantiate the Edge object, we **have** to set a file and a group, within that file, to bind to it. For this example, we're using an EDC file which declares two parts (blue and green rectangles) and an item data:

We start by trying to access an **unexistent** group in the file, so that you can see the usefulness of `edge_object_load_error_get()` and `edge_load_error_str()`. Check that the error message will tell you just that – a group which didn't exist in the file was called for:

Then, we finally bind our Edge object to `"example_group"`, printing a message afterwards:

What follows is a series of Edge API calls which are of general use. The first of them is `edge_object_data_get()`, which we use to get the value we have put in the `"example_data"` data field, in our EDC object declaration:

Then, we exemplify `edge_object_part_exists()`:

The next call is to query `"part_one"'s` geometry, relative to the whole Edge object's area. The part will be situated in the middle of the Edge object's, because it has a restricted forced size (we set its minimum size equal to its maximum, for that) and, by default, parts are aligned to the center of their containers:

We can grab a direct pointer on the rectangle implementing `"part_one"`, by using `edge_object_part_object_get()`. Since we are not allowed to set properties on it, we just check its color, to assure its really blue, as declared in the EDC:

The "min" and "max" EDC properties can be queried with the following calls:

The next two calls are to make **size calculations** on our object. Because of the minimum size declared for "part_one" part's default state description, that will be our exact minimum size calculated for the group (remember the "min" declaration at group level is just a **hint**, not an enforcement). We then exercise the `edge_object_size_min_restricted_calc()` function, passing a minimum size of 500, in each axis. Since we have **no** object bigger than that, it will be the minimum size calculated, in the end:

"part_two" part is there with a purpose: since it extrapolates the Edge object's boundaries, the `edge_object_parts_extends_calc()` function will report origin coordinates for the rectangle grouping both parts with **negative** values, indicating it extrapolates to the upper left of our group, just as we see it.

To interact with the last features exemplified in the program, there's a command line interface. A help string can be asked for with the 'h' key:

Those commands will change the scaling factors of our Edge objects. The first of them, 's', will change Edge's **global** scaling factor between 1.0 (no scaling) and 2.0 (double scale). Scaling will be applied to "part_one", only, because that's the part flagged to be scaled at EDC level:

Note, finally, that the 's' command will depend on the 'r' one to have its effects applied. The latter will change "part_one"'s **individual** scaling factor, which **overrides** Edge's global scaling factor. Only when the individual one is set to zero, will the global one take effect:

The example's window should look like this picture:

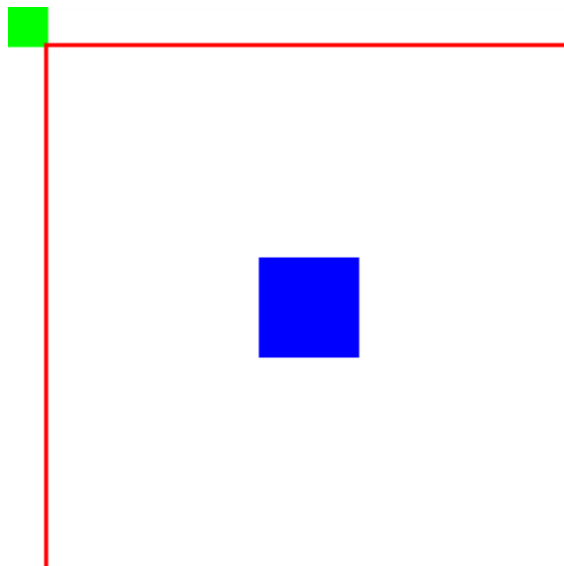


Figure 30.1: width=

The full example follows.

To compile use this command:

```
* gcc -o edge-basic edge-basic.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR  
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"  
* `pkg-config --cflags --libs evas ecore ecore-evas edge`  
*  
* edge_cc basic.edc  
*
```


Chapter 31

Edge Nested Part (hierarchy) example

Nested part feature represents the concept of hierarchy to edge.

A nested part inherits its location relatively to the parent part. Thus, parent part modifications such as move or map affect all nested parts. To declare a nested part just start a new part declaration within (before closing) the current part declaration.

Note that nested part declaration is allowed only after current part name is defined.

Here's an example of a rect nested in other rect plus inner nested rect:

The example's window should look like this picture:

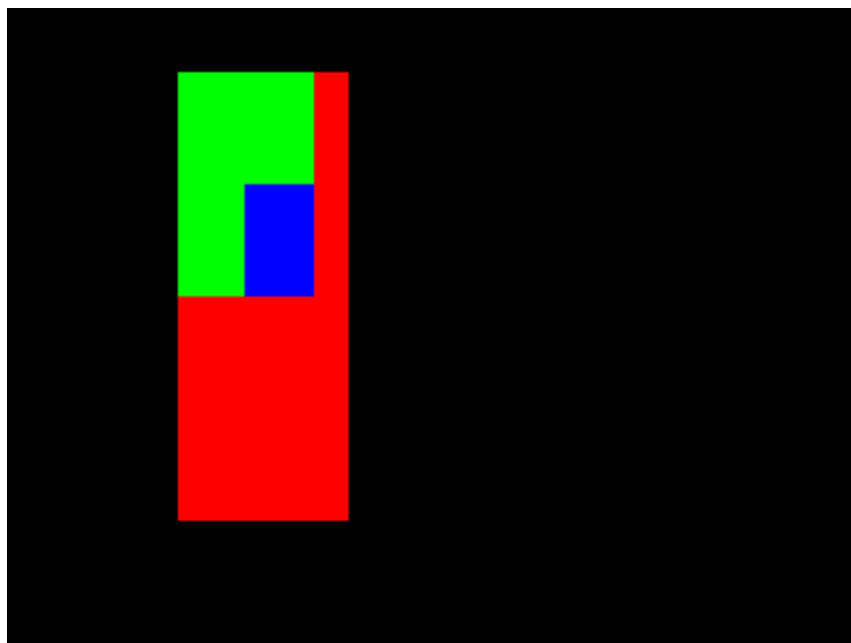


Figure 31.1: width=

Chapter 32

Swallow example

This is a simple example in which we create a rect and swallow it.

Focusing on the relevant parts of the code we go right to the creation of our rectangle. It should be noted that we don't resize or show our rect, that is because when an object is swallowed it's geometry and visibility is controlled by the theme:

The other bit of code that is relevant to us now is our check that the swallow worked:

The full source code follows:

To compile use this command:

```
* gcc -o edge-swallow edge-swallow.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIBS="-lE"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edge`
*
* edge_cc swallow.edc
*
```


Chapter 33

Edge Text example

This example shows how to manipulate TEXT and TEXTBLOCK parts from code.

The very first we are going to do is register a callback to react to changes in the text of our parts:

Note

Since edge_obj represent a group we'll be notified whenever any part's text in that group changes.

We now set the text for two our two parts:

Note

Since the "part_two" part is a TEXTBLOCK we can use formatting such as ``

And we now move on to selection issues, first thing we do is make sure the user can select text:

We then select the entire text, and print the selected text:

We now unselect the entire text(set selection to none), and print the selected text:

Our example will look like this:

one

two

Figure 33.1: width=

The full source code follows:

The theme used in this example is:

To compile use this command:

```
* gcc -o edge-text edge-text.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR=\  
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"\  
* `pkg-config --cflags --libs evas ecore ecore-evas edge`\  
*\  
* edge_cc text.edc\  
*
```

Chapter 34

Table example

In this example, we illustrate how to organize your objects on a table, using the `edje_object_part_table` functions. To be easier to understand the objects in this example will be four simple rects, when the user click over one item with the left button its is removed from the table, if any other button was used all items are removed. For each action is printed a message with the current number of rows and columns.

We started creating an EDC file with one part of the type TABLE called "**table_part**", that is the part which we will refer to access the table:

On the other hand, in the C file we first create the rectangles and added a callback for mouse down, as you can see bellow:

With the objects created we have to pack them into the table, to do this, we just have to use the function `edje_object_part_table_pack()`.

The other bit of code that is relevant to us now is our event handler for when the user click over the rectangle. Here we use the function `edje_object_part_table_unpack()` to remove the item from the table or `edje_object_part_table_clear()` to remove all items, it depends on which mouse button the user uses.

Finally, the last important thing in this example is about how to know how many columns and rows are there in the table, It should be noted that this function don't tell you how many items are there in the table, just the number of the columns and rows of the table.

The example's window should look like this picture:

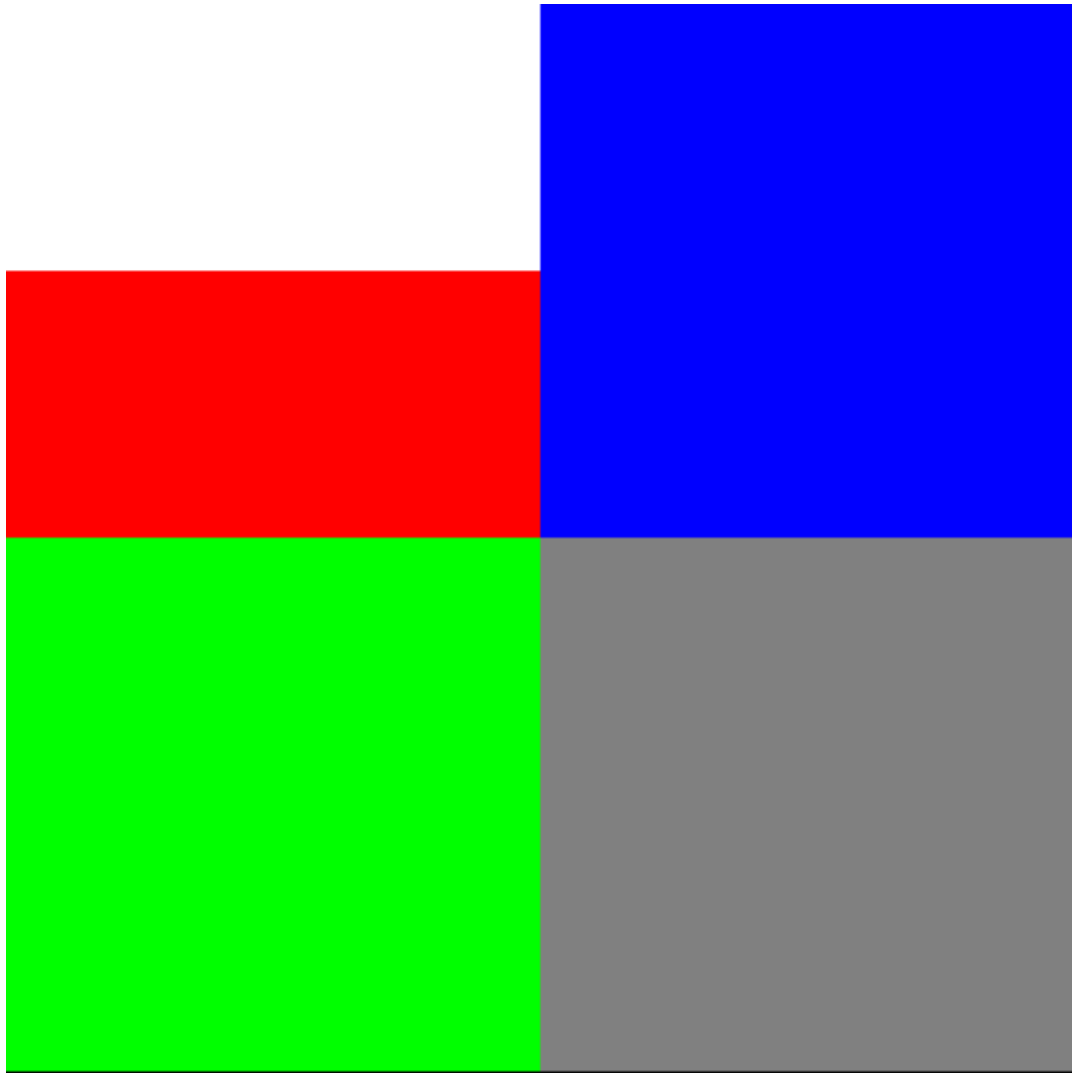


Figure 34.1: width=

The full source code follows:

To compile use this command:

```
* gcc -o edge-table edge-table.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR  
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"  
* `pkg-config --cflags --libs evas ecore ecore-evas edge`  
*  
* edge_cc table.edc  
*
```


Chapter 35

Box example - basic usage

This example shows how to append, insert and remove elements from an Edge box part. It will make use of the `edge_object_part_box` functions.

To play with this example, use mouse left click to delete an existing rectangle from the box and right mouse click to add a new rectangle just before the clicked one. Use the keyboard keys "a" to append a rectangle, "i" to prepend, and "c" to remove all rectangles from the box.

We will store our example global information in the data structure defined below, and also set some callbacks for resizing the canvas and exiting the window:

In the `main` function, we create our `Ecore_Evas`, add a background to it, and finally load our `Edge` file that contains a `Box` part. This part is named `"example/box"` in this case, and we use this name to append elements to it.

The code until now is the one that follows:

Also notice that we set the callback `_bg_key_down` for `"key down"` events on the background object, and that object is the one with focus.

Now we add some small rectangles to the box part, using the `edge_object_part_box_append()` API, and set some callbacks for `"mouse down"` events on every object. These callbacks will be used to add or delete objects from the box part.

Now let's take a look at the callbacks for key down and mouse down events:

This callback for mouse down events will get left clicks and remove the object that received that left click from the box part, and then delete it. This is done with the `edge_object_part_box_remove()` function.

However, on right clicks it will create a new rectangle object, and add it just before the right clicked object, using `edge_object_part_box_insert_before()`.

And this is the key down callback:

It will insert elements at the beginning of the box if "i" was pressed, using `edge_object_part_box_insert_at()`. It will also append objects to the box if "a" was pressed, just exactly like we did in the `main` function. And will remove all objects (deleting them) if "c" was pressed.

As you can see, this example uses the "horizontal_flow" layout for the box, where each item is put linearly in rows, in as many rows as necessary to store all of them.

The example's window should look like this picture:

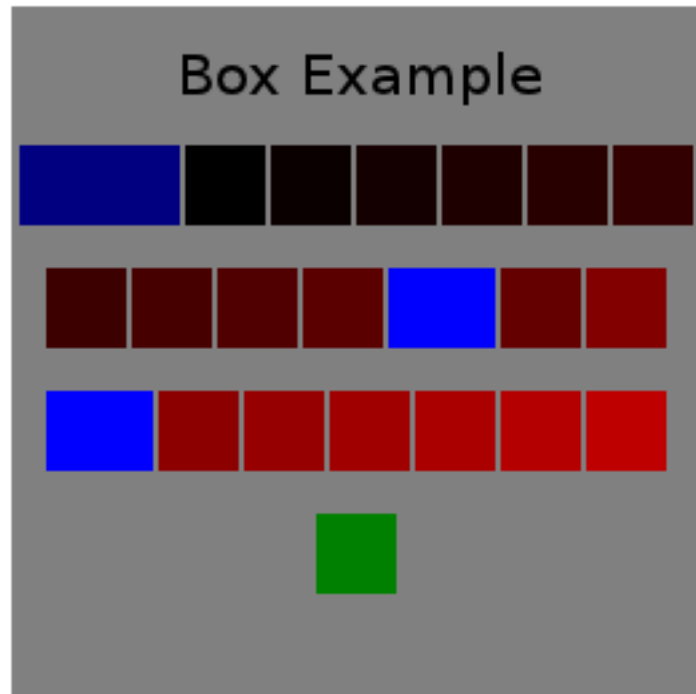


Figure 35.1: width=

The full source code follows:

To compile use this command:

```
* gcc -o edge-box edge-box.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR=\"/
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc box.edc
*
```

Chapter 36

Box example - custom layout

This example shows how to register a custom layout to be used by the Edge box part. It will use `edge_box_layout_register()` for that.

To play with this example, use the keyboard modifier keys and number keys together. The Ctrl key is used for adding elements, and Shift is used for removing them. For instance, Ctrl + 3 will insert a new rectangle object in the 3rd position of the box, while Shift + 6 will try to remove the 6th element of the box.

This example is very similar to the other box example, has a structure with global data, a callback for key down events where we create or delete rectangle objects and add or remove them to/from the box part.

But the important part is the next one:

This code implements our custom layout, which will position every object added to the box in a diagonal through the size of the box part. Notice that it just calculates the position and offset based on the size of the box and number of children, and then moves each child to the respective position.

Later on the `main` function, everything we need to do is to register this custom layout function with `edge`:

And use it inside the `box.edc` file:

The example's window should look like this picture:

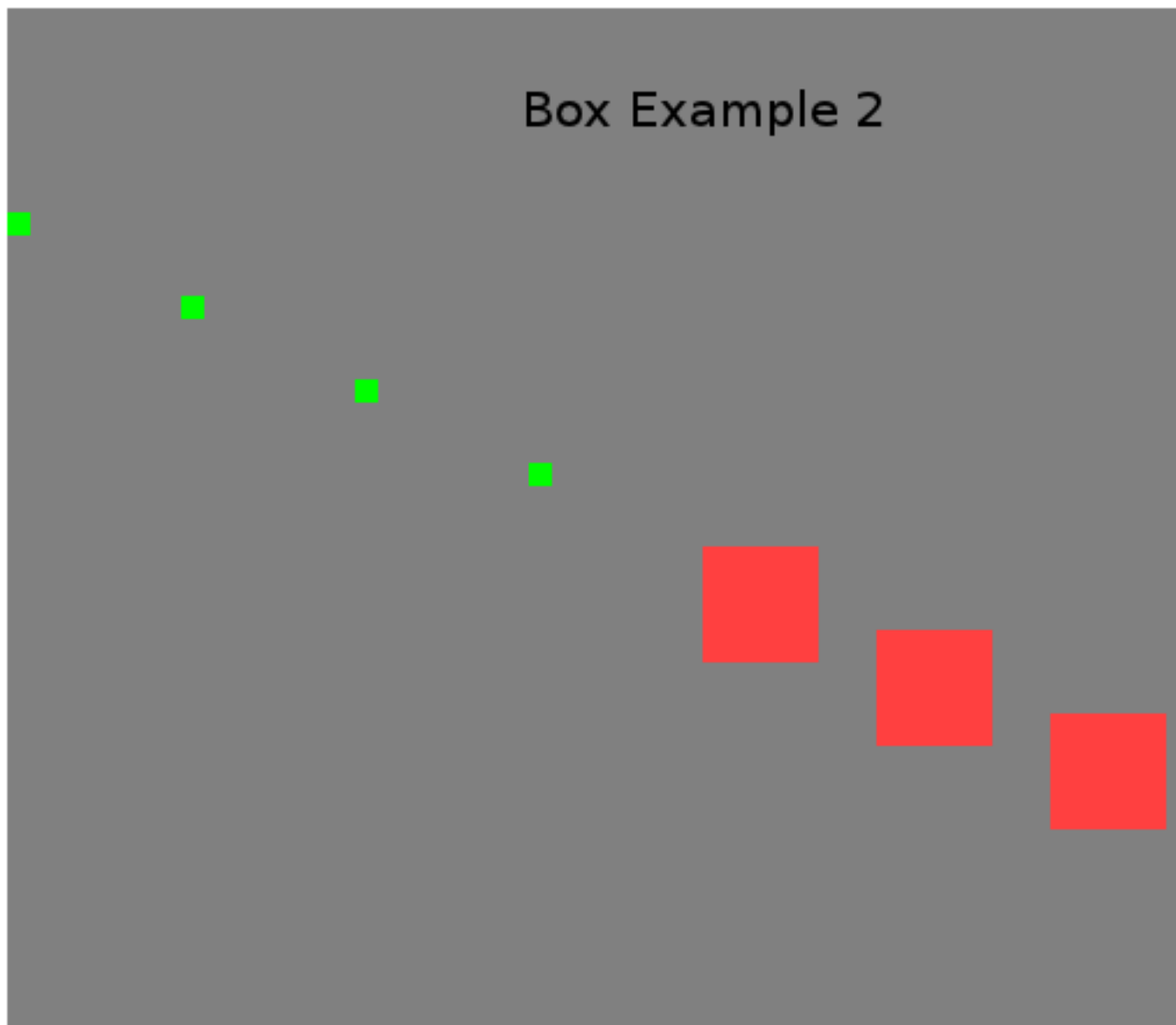


Figure 36.1: width=

The full source code follows:

To compile use this command:

```
* gcc -o edge-box2 edge-box2.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR=
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc box.edc
*
```

Chapter 37

Dragable parts example

This example shows how to manipulate a dragable part through the `edge_object_part_drag` API.

First, in the edc code, we are declaring a part which will be our movable part, called "knob". It is a normal rectangle, which contains a block called "draggable", that will define the area where this rectangle can be moved, and in which axis it can be moved.

This is our part:

Notice that it defines, through its `"x : "` and `"y : '` properties, that the part will be only moved on the y axis (vertical). Check the edc reference docs for more info about this.

Now, in our example C code, we just do the same as on the other examples, setting some global data on a structure, load the edge file and so:

We want to use the `drag_page` and `drag_step` functions, and in order to do so we need to define the step size and page size of our draggable part. They are defined as float values which represent a portion of the entire size of the draggable area:

We are going to use the keyboard to move the `knob` part, through the key down callback `_bg_key_down`, but we also want to know when the user has moved the knob by using the mouse (which is possible, since we defined that this part will receive mouse events). Thus, we set a callback for the signal "drag", which comes from the draggable part:

Now, let's take a look at our key down callback:

On this callback we define that the user will use the "up" and "down" arrows to move the draggable part, respectively, -1.0 and 1.0 times the step size. And that the "Page Up" (Prior) and "Page Down" (Next) keys will move -1.0 and 1.0 times the page size. Both of these will occur on the vertical axis, since we pass 0.0 as value to the respective horizontal axis parameters. And our draggable part also only supports being moved in the vertical axis (defined in the edc).

We also define that the "m" key will be used to explicitly position the knob part in the middle of the dragable area.

And here is the callback for the "drag" signal that is received from the theme:

The example's window should look like this picture:

Drag Example



Figure 37.1: width=

The full source code follows:

To compile use this command:

```
* gcc -o edge-drag edge-drag.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\" -DPACKAGE_LIB_DIR=\
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc drag.edc
*
```

Chapter 38

Perspective example

This example demonstrates how someone can set a perspective to be used by an Edje object, but setting a global perspective. The API for setting a perspective for just one Edje object is almost the same and it's trivial, so we are not doing that on this example.

Let's go first to the main function, where we start creating our objects and loading the theme. We also set some variables that will be used globally in our program:

A boolean is used to indicate that we are animating.

We also set the `app.x` and `app.y` to (0, 0) because the original position of our text + rectangle part will be on top left. This is a convention that we are using in this example, and setting x, y to 1, 1 would mean bottom right. We do this to later define the name of the signals that we are sending to the theme.

After this, some boilerplate code to load the theme:

Now we are going to setup a callback to tell us that the animation has ended. We do this just to avoid sending signals to the theme while it's animating.

Finally, let's create our perspective object, define its position, focal distance and z plane position, and set it as global:

Notice that if we wanted to set it just to our edje object, instead of setting the perspective as global to the entire canvas, we could just use `edje_object_perspective_set()` instead of `edje_perspective_global_set()`. The rest of the code would be exactly the same.

Now, let's take a look at what we do in our callbacks.

The callback for `key_down` is converting the arrow keys to a signal that represents where we want our text and rectangle moved to. It does that by using the following function:

Notice that, after sending the signal to the Edje object, we set our boolean to store that we are animating now. It will only be unset when we receive a signal from the theme that the animation has ended.

Now, on the `key_down` code, we just call this function when the arrows or "PgUp" or "PgDown" keys are pressed:

Notice that we also do something else when the numeric keyboard "+" and "-" keys are pressed. We change the focal distance of our global perspective, and that will affect the part that has a map rotation applied to it, with perspective enabled. We also need to call `edje_object_calc_force()`, otherwise the Edje object has no way to know that we changed the global perspective.

Try playing with these keys and see what happens to the animation when the value of the focal distance changes.

Finally we add a callback for the animation ended signal:

The example's window should look like this picture:



Figure 38.1: width=

The full source code follows:

The full .edc file

To compile use this command:

```
* gcc -o edge-perspective edge-perspective.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edge_cc perspective.edc
*
```


Chapter 39

Edje signals and messages

In this example, we illustrate how Edje signals and Edje messages work.

We place, in the canvas, an Edje object along with a **red** border image to delimit its geometry. The object's group definition is so that we have four parts:

- a blue rectangle, aligned to the right
- a white rectangle, aligned to the left
- a text part, aligned to the center
- a clipper rectangle on the blue rectangle

The left rectangle is bound to a **color class**, so that we can multiply its colors by chosen values on the go:

The `#define`'s on the beginning will serve as message identifiers, for our accorded message interface between the code and the this theme file.

Let's move to the code, then. After instantiating the Edje object, we register two **signal callbacks** on it. The first one uses **globbing**, making all of the wheel mouse actions over the left rectangle to trigger `_mouse_wheel`. Note that those kind of signals are generated **internally** (and automatically) in Edje. The second is a direct signal match, to a (custom) signal we defined in the EDC, ourselves:

That second callback is on a signal we emit on the theme, where we just translate Edje `"mouse,move"` internal events to the custom `"mouse,over"` one. When that signals reaches the code, we are, besides printing the signals' strings, sending a **message** back to the theme. We generate random values of color components and send them as an `#EDJE_MESSAGE_INT_SET` message type:

In our theme we'll be changing the `"cc"` color class' values with those integer values of the message, so that moving the mouse over the right rectangle will change the left one's colors:

Now we're also sending messages **from the Edje object**, besides signals. We do so when one clicks with the left button over the left rectangle. With that, we change the text part's text, cycling between 3 pre-set strings declared in the EDC. With each new text string attribution, we send a string message to our code, with the current string as argument:

To get the message in code, we have to register a message handler, as follows:

To interact with the last missing feature – emitting signals **from code** – there's a command line interface to exercise it. A help string can be asked for with the 'h' key:

The 't' command will send either "part_right,show" or "part_right,hide" signals to the Edje object (those being the emission part of the signal), which was set to react on them as the names indicate. We'll set the right rectangle's visibility on/off, respectively, for those two signals:

The example's window should look like this picture:

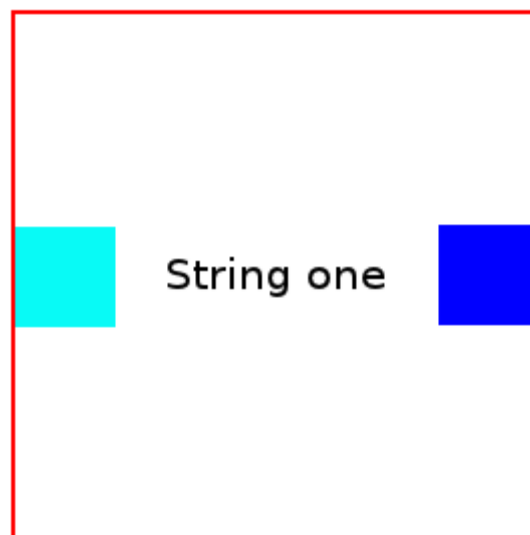


Figure 39.1: width=

The full example follows, along with its EDC file.

To compile use this command:

```
* gcc -o edge-signals-messages edge-signals-messages.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/b
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc signals-messages.edc
*
```


Chapter 40

Edge Color Class example

This example shows how to manipulate and change Color classes. In this example we will create two surfaces to show what happens when you change the color class at the process and object level.

It's a very simple example, there are two surfaces created from the same EDC, but just in one of them we will set a specific color class, although both will be affected by color class set at the process level as you will see.

It's important you know that all colors has the format R G B A. Just to be easier to understand this example, we will create a small set of colors that will be used along of the example. This piece of code is shown below:

Focusing on the relevant parts of the code we go right to the part where we set the new color class. For that we will use the functions `edge_color_class_set` (which will affect all edges) and `edge_object_color_class_set` (which affects just the specific object).

Note

- `argv[1]` is the name of a color class used in the EDC.
- The second and third colors only apply to text part.
- The color class set for the object overrides the color previously set.

After we have set the color class we will check the color classes, for that we created a function which prints all color classes and tries to get theirs values and print too.

There are two other things that are worth mentioning, we added two callbacks for the objects, one for mouse down (that we use to delete the color class) and another for the signal emitted when a color class is deleted.

And then we delete the color class:

Our example will look like this, if you run with the parameters "green_class gray pink yellow":

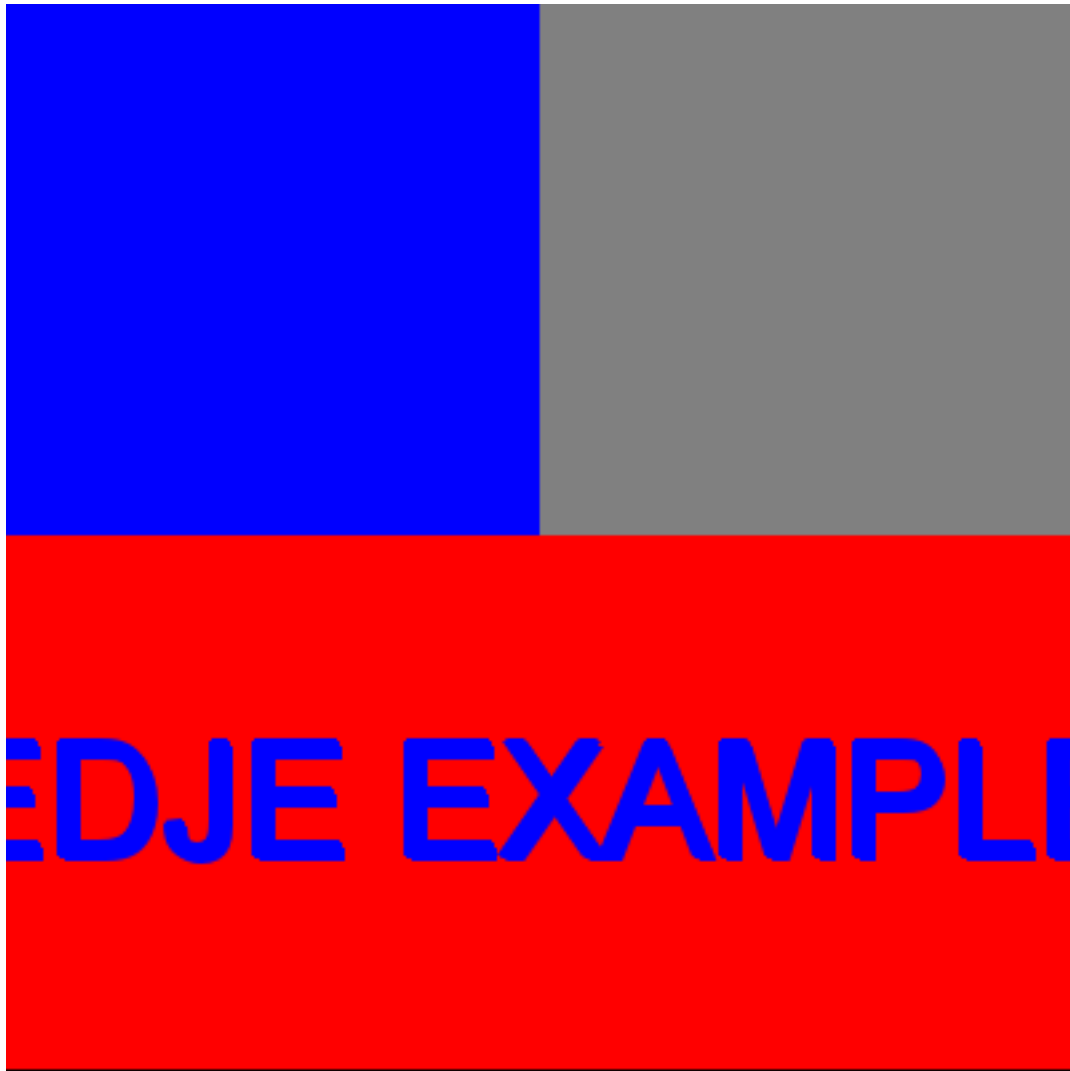


Figure 40.1: width=

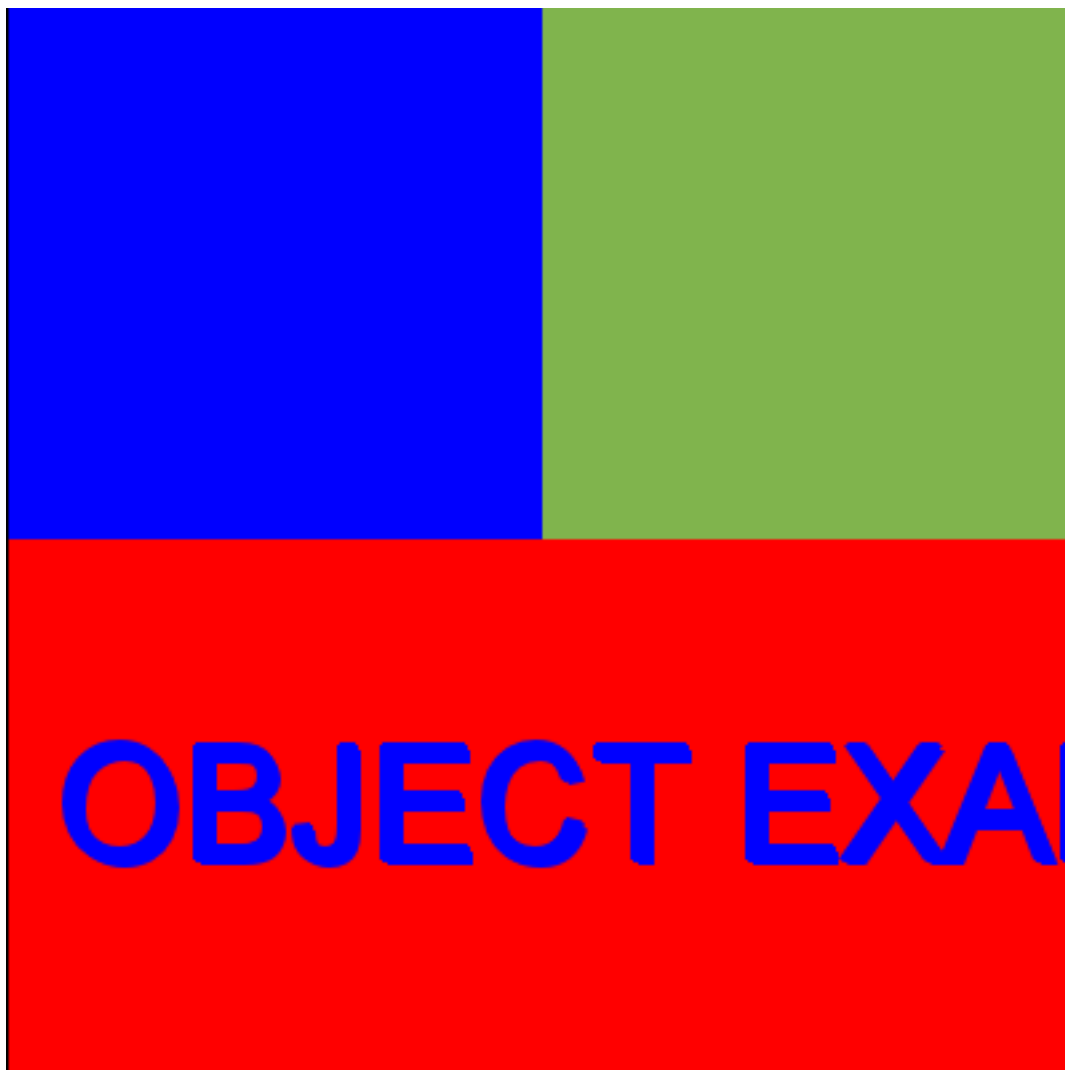


Figure 40.2: width=

The full source code follows:

The theme used in this example is:

To compile use this command:

```
* gcc -o edge-color-class edge-color-class.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc color-class.edc
*
```


Chapter 41

Edje Animations example

In this example we will figure out how to manipulate the animations on an Edje object. After reading this document you will be able to manipulate the frametime, freeze, pause and stop, all animations on an Edje object.

To play with this example you will use the keyboard. Below are listed the keys and what each does.

- '+' Increase the frametime;
- '-' Decrease the frametime;
- '=' Prints the actual frametime and says if the animations is playing;
- 'f' Freezes the animations in the Edje object;
- 'F' Freezes the animations in all objects in the running program;
- 't' Thaws the animations in the Edje object;
- 'T' Thaws the animations in all objects in the running program;
- 's' Pauses the animations;
- 'p' Plays the animations previously stopped;
- 'a' Starts the animation in the Edje object;
- 'A' Stops the animations in the Edje object;

Now that we've explained how to use our example, we will see how it is made. Let's start by looking at the piece of code responsible to the actions commented above.

Note

The actions for the keys 'f' and 'F' will have the same effect in our example, just because there is only one object in the running program, The same happens with the keys 't' and 'T'.

As you may have seen these set of functions are pretty easy to handle. The other important part of this example is the EDC file. The animations used in the code were created there

The example's window should look like this picture:

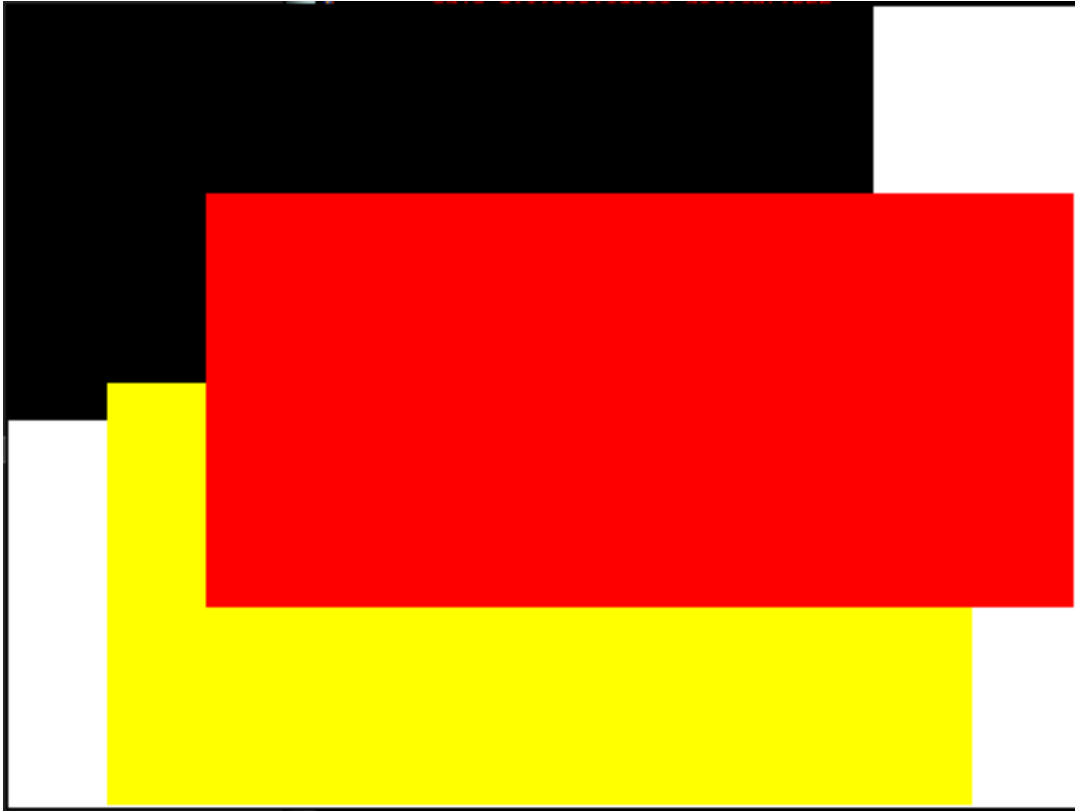


Figure 41.1: width=

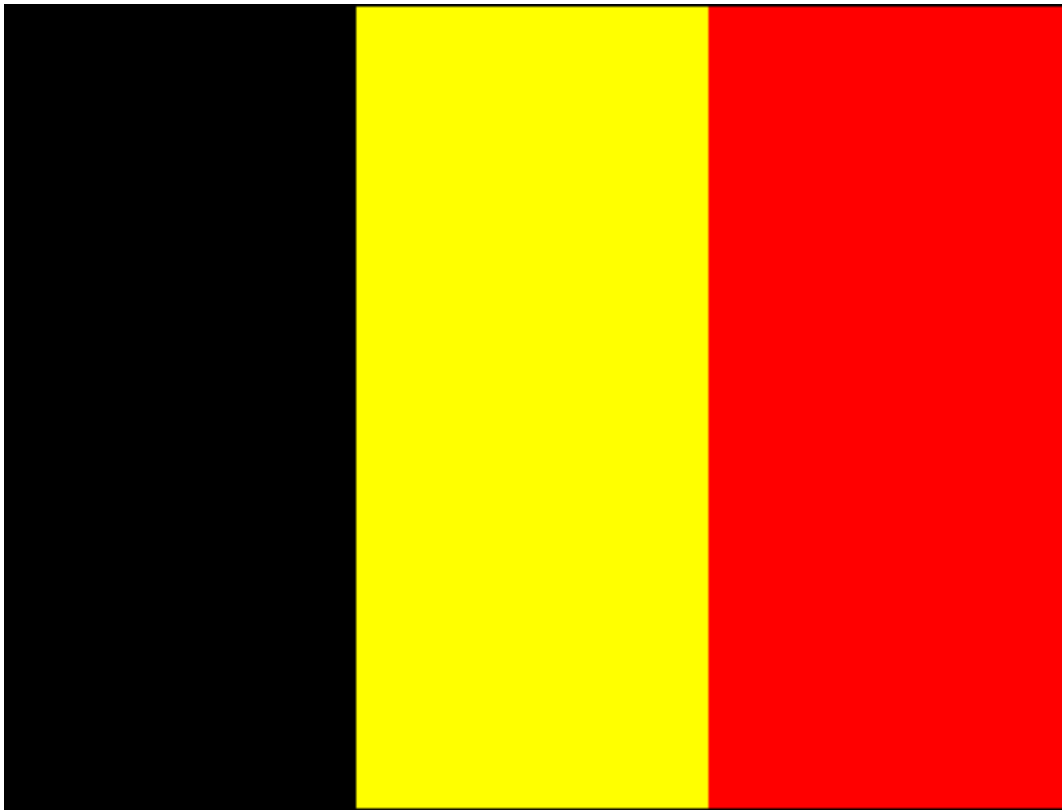


Figure 41.2: width=

The full example follows.

To compile use this command:

```
* gcc -o edge-animations edge-animations.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edge_cc animations.edc
*
```


Chapter 42

Multisense example

This is a simple example in which a rect is created and sound and tone are played on mouse down event.

Focusing on the creation of sample and tone. It should be noted that creation of sample sound is from any supported (sndfile lib) audio file, tone from of specific audible frequency range are controlled by the theme:

The full source code follows:

Chapter 43

Edje basics example 2

In this example we will show how to load an image and move it across the window.

To load the image to our program, it needs to be declared in the .edc using the images block:

Note

COMP means that we are using a lossless compression

Then to be able to use it in our window we must declare a part for this image:

Now we move to our .c file, you will notice this define:

This means how fast we want to move the image across the screen

To load our edje file we will use this command, we do just like the last example (Basic example):

If we want to move our image, we need to add a callback to be able to do this, so we define:

To get the position of the image we use this:

Now we use the if's to check in what direction the user wants to move the image then we move it:

The example's window should look like this picture:

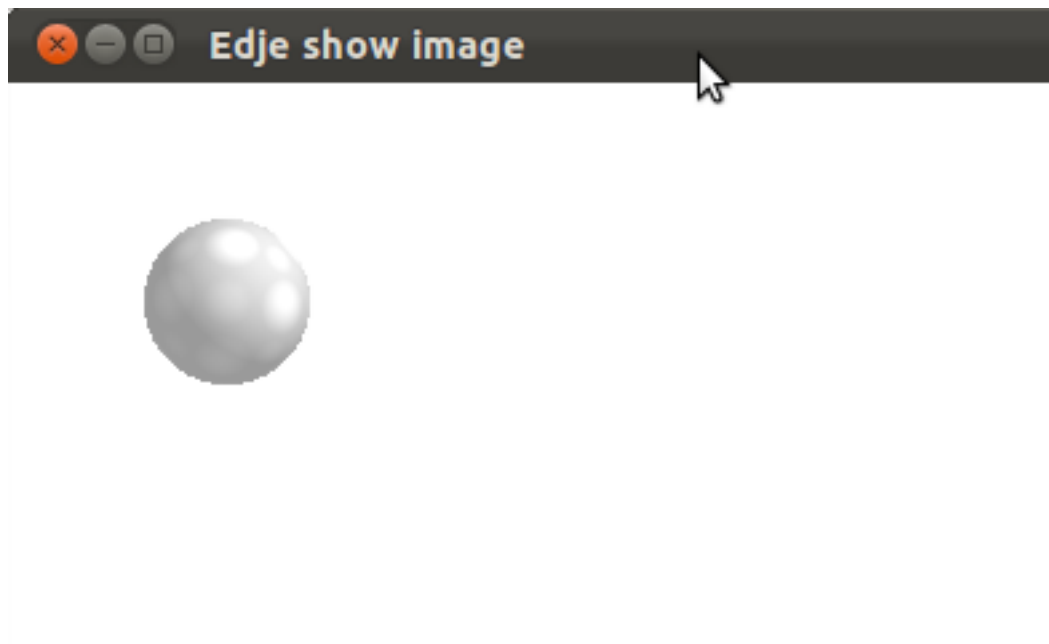


Figure 43.1: width=

The complete .edc file:

And the source code:

To compile use this command:

```
* gcc -o edge-basic2 edge-basic2.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edge_cc -id /path/to/the/image basic2.edc
*
```


Chapter 44

Swallow example 2

This example will show you how to load an image using `evas` and fill a swallow with it. It's basically almost like the last example, but with a minor change.

These variables are used to store the name of the image that is going to be used, the path to it and a variable that will store the error cause if something goes wrong.

Here it is:

Then we load the image with this command:

To check if we had some problem we use:

Now we are going to swallow it and check if worked. If you notice we are using `"part_one"` as argument. We do this because we must explicit what part of our `.edc` file we want to swallow:

The example's window should look like this picture:



Figure 44.1: width=

The complete .edc file:

And the source code:

To compile use this command:

```
* gcc -o edge-swallow2 edge-swallow2.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edje_cc swallow.edc
*
```

Chapter 45

Edje Signals example 2

In this example we will make use of signals to help to move an image away from the mouse pointer.

Signals are software interruption, this means that when it happens and if the program is sensitive to it the program will stop whatever it is doing and handle the signal.

In this example we are only sensitive to the "mouse,move" signal so we need to register a callback to it. To do this we will add a signal callback to our edje object that will detect "mouse,move" signal coming from the part "part_image" and when this happens we will call the function `_on_mouse_over` passing the evas pointer as a parameter. The evas pointer is passed as a parameter because we need to know where is the mouse pointer in the screen.

We can see bellow how we can listen to the signal:

Now, let's pass to the callback function. If we want to keep the ball away from the mouse pointer we need to now where is the ball and where is the mouse and we can easily discovery these things using this:

For the object position in the canvas:

For the mouse position relative to the screen:

Now that we have the position of the mouse and the object we just need to set the new location and move the object. To set the new location we do this:

You can change the formula above if you like. Because we are changing the object's position we need to do something if the new position is beyond the canvas size. So here it is:

Then now what we need to do is move the object:

Here is the complete callback function:

When you compile and run it you should see this:

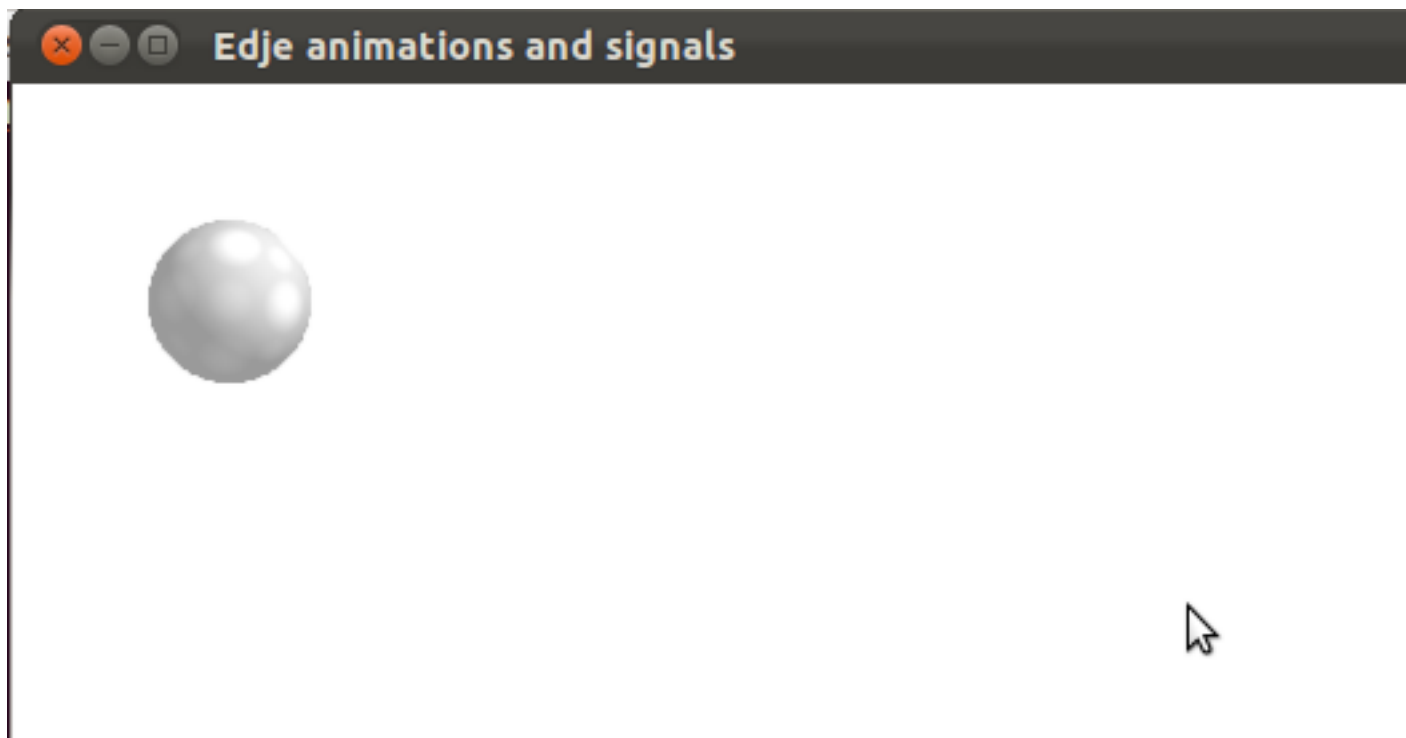


Figure 45.1: width=

The .edc file:

The source code:

To compile use this command:

```
* gcc -o signals2 signals2.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edje`
*
* edge_cc -id /path/to/the/image signalsBubble.edc
*
```

Chapter 46

Edje animations example 2

In this example we will make an animation of a bouncing ball.

Our focus now will be in the .edc file, because in the C source code we just load the edje object and do nothing else.

We want to give the freedom to the object to use the whole canvas of our program, so lets define the min, max of our group:

You will notice that in our .c file the canvas will have this size

Now we will define our part that's going to be our image, the ball:

At the description block bellow we are saying that this part has an image, defining the max and min size and it's position on the edje_object. The default state is the first state of the part, this means that when the edje object is loaded this one is going to be showed to you.

Here it is:

Now in the next description block we are saying where the ball is going to stop. Note that we have the "inherit" property. This means we are inheriting everything from default, except rel1 and rel2, because we are redefining it.

Check the code:

We defined how our object will look like and it's position during the animation, now we need to define how it's going to act during the time. To do this we will use the programs block

The first program block will start the animation, it will wait for the 'load' signal. This signal is generated when the edje object is loaded. The 'after' property is saying to this program block exactly this: "When you finish, call the program 'animation,state1' ". The 'in' property is saying, wait 0.5 seconds until you execute this program block.

Here is the code:

Now lets make the ball move and bounce it. In the second program block we are defining what we need to do with the action property. So we are saying change to the state "down-state" using the transition BOUNCE and apply this to the part "part_bubble". You can notice that BOUNCE has three parameters, the first one is saying how much time the transition will last, the second one is the factor of curviness and the last one is saying how many times and object will bounce.

The code is very easy:

Now all you have to do is compile the code and run it!

When you compile and run it you should see this:

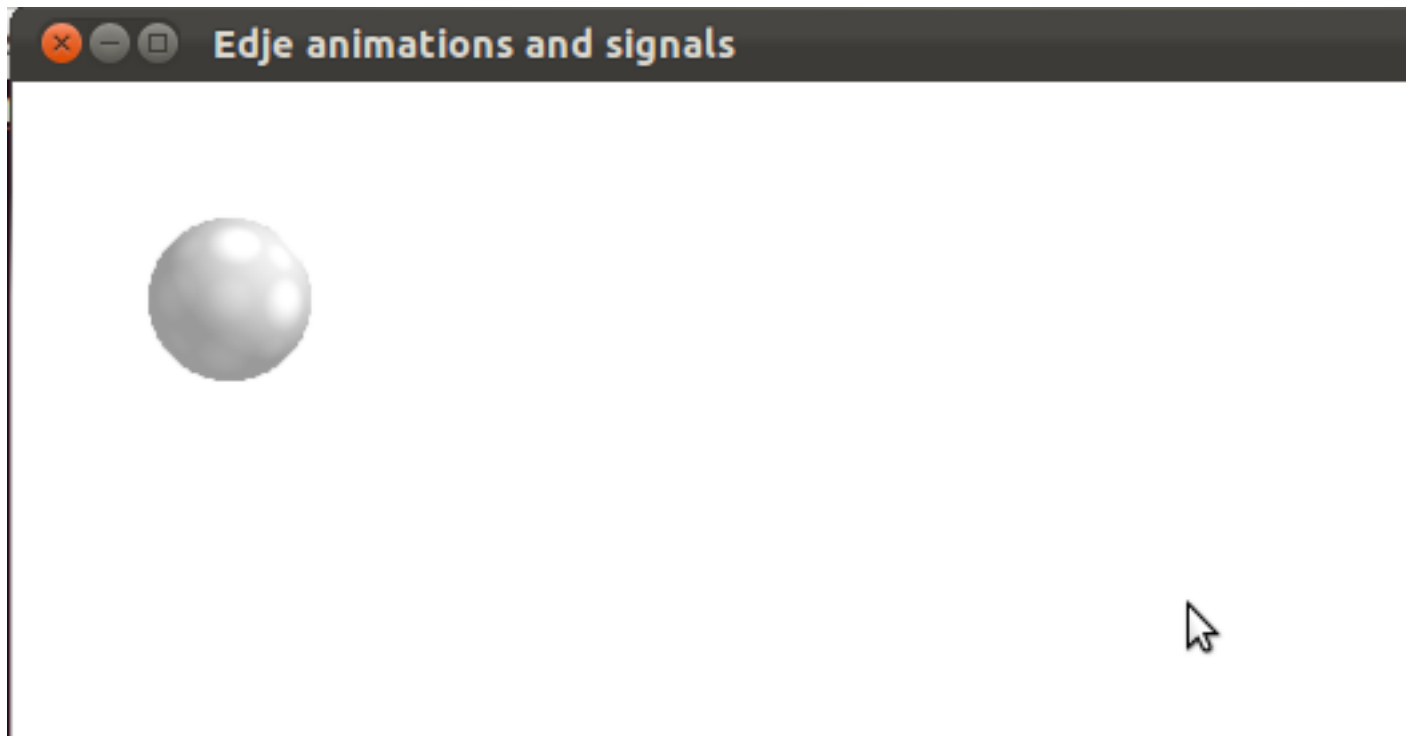


Figure 46.1: width=

The .edc file:

The source code:

To compile use this command:

```
* gcc -o animations2 animations2.c -DPACKAGE_BIN_DIR=\"/Where/enlightenment/is/installed/bin\"
* -DPACKAGE_LIB_DIR=\"/Where/enlightenment/is/installed/lib\"
* -DPACKAGE_DATA_DIR=\"/Where/enlightenment/is/installed/share\"
* `pkg-config --cflags --libs evas ecore ecore-evas edge`
*
* edge_cc animations2.edc
*
```

Chapter 47

EET Examples

Here is a page with examples.

[Simple data example](#)

[Nested data example](#)

[File descriptor data example](#)

[File descriptor data example, with Eet unions and variants](#)

[Eet data cipher/decipher example](#)

Chapter 48

Very basic Eet example

Chapter 49

Example of the various ways to interface with an Eet File

Chapter 50

Simple data example

Chapter 51

Nested data example

Chapter 52

File descriptor data example

Chapter 53

File descriptor data example, with Eet unions and variants

This is an example much like the one shown in `eet_data_file_descriptor`. The difference is that here we're attaining ourselves to two new data types to store in an Eet file – **unions** and **variants**. We don't try to come with data mapping to real world use cases, here. Instead, we're defining 3 different simple structures to be used throughout the example:

To identify, for both union and variant data cases, the type of each chunk of data, we're defining types to point to each of those structs:

We have also a mapping from those types to name strings, to be used in the Eet unions and variants `type_get()` and `type_set()` type identifying callbacks:

In this example, we have no fancy hash to store our data into profiles/accounts, but just two lists for union and variant data nodes:

Let's begin with our unions, then, which look like:

The first interesting part of the code is where we define our data descriptors for the main lists, the unions and all of structures upon which those two depend.

The code for descriptors on `Example_Struct1`, `Example_Struct2` and `Example_Struct3` is straightforward, a matter already covered on `eet_data_file_descriptor`. What is new, here, are the two type matching functions for our unions. There, we must set the `data` pointer to its matching type, on `_union_type_set` and return the correct matching type, on `_union_type_get`:

With the `#EET_DATA_DESCRIPTOR_ADD_MAPPING` calls, which follow, we make the link between our type names and their respective structs. The code handling actual data is pretty much the same as in `eet_data_file_descriptor` – one uses command line arguments to enter new data chunks (or just to visualize the contents of an Eet file), signalling if they are unions or variants. One must also pass the type of the data chunk to enter, with integers 1, 2 or

1. Then, come the fields for each type:

Variants are very similar to unions, except that data chunks need **not** contain previously allocated space for each of the possible types of data going in them:

The code declaring the data descriptors and handling the data is very similar to the unions part, and is left for the reader to check for him/herself. The complete code of the example follows.

Chapter 54

Eet data cipher/decipher example

In this example, we exemplify the usage of `eet_write_cipher()` and `eet_read_cipher()`. For it to work, **make sure** to have your Eet installation with a ciphering backend enabled.

We start by defining the information to record in an Eet file (`buffer`), the key to cipher that (`key`) and a dummy wrong key to try to access that information, later (`key_bad`).

After opening our file, we simply use the first cited function to write our string ciphered:

Then, after closing it on purpose, we open it again, to retrieve the encrypted information back, in a readable format:

Note that we do it twice, being the last time with the wrong key. In this last case, if the information is read back and matches the original `buffer`, something wrong is going on (we made it to fail on purpose). The former access is OK, and must work.

What we do in sequence is just to delete the file. The complete code of the example follows.

Chapter 55

Eina Examples

Examples:

- [eina_accessor_01.c](#)
- [eina_array_01.c](#)
- [eina_array_02.c](#)
- [eina_error_01.c](#)
- [eina_file_01.c](#)
- [eina_hash_01.c](#)
- [eina_hash_02.c](#)
- [eina_hash_03.c](#)
- [eina_hash_04.c](#)
- [eina_hash_05.c](#)
- [eina_hash_06.c](#)
- [eina_hash_07.c](#)
- [eina_hash_08.c](#)
- [eina_inarray_01.c](#)
- [eina_inarray_02.c](#)
- [eina_inlist_01.c](#)
- [eina_inlist_02.c](#)
- [eina_inlist_03.c](#)
- [eina_iterator_01.c](#)
- [eina_list_01.c](#)
- [eina_list_02.c](#)
- [eina_list_03.c](#)
- [eina_list_04.c](#)
- [eina_log_01.c](#)
- [eina_log_02.c](#)

- [eina_log_03.c](#)
- [eina_magic_01.c](#)
- [eina_model_01.c](#)
- [eina_model_02.c](#)
- [eina_model_03.c](#)
- [eina_model_04_animal.c](#)
- [eina_model_04_child.c](#)
- [eina_model_04_human.c](#)
- [eina_model_04_main.c](#)
- [eina_model_04_parrot.c](#)
- [eina_model_04_whistler.c](#)
- [eina_simple_xml_parser_01.c](#)
- [eina_str_01.c](#)
- [eina_strbuf_01.c](#)
- [eina_stringshare_01.c](#)
- [eina_tiler_01.c](#)
- [eina_value_01.c](#)
- [eina_value_02.c](#)
- [eina_value_03.c](#)

Tutorials:

- [tutorial_benchmark_page](#)
- [tutorial_binshare_page](#)
- [tutorial_eina_string](#)
- [tutorial_error_page](#)
- [tutorial_log_page](#)
- [tutorial_matrixsparse_page](#)
- [tutorial_quadtree_page](#)
- [tutorial_strbuf](#)
- [tutorial_ustringshare_page](#)

Chapter 56

Eio Examples

Here is a page with some Eio examples explained:

- [eio_file_ls.c](#)

Tutorials:

- [eio_dir_copy\(\)](#) tutorial
- [eio_dir_stat_ls\(\)](#) tutorial
- [eio_file_ls\(\)](#) tutorial
- [eio_dir_direct_ls\(\)](#) tutorial
- [eio_monitor_add\(\)](#) tutorial

Chapter 57

eio_dir_copy() tutorial

To use `eio_dir_copy()`, you basically need the source and destination files (or directories), and set three callbacks:

- The notification callback, which allows you to know if a file or a directory is copied, and the progress of the copy.
- The end callback, which is called when the copy is finished.
- The error callback, which is called if an error occurred. You can then retrieve the error type as an `errno` error.

Warning

It is the user's duty to provide the "right target". It means that copying to `'.'` will copy the content directly inside `'.'` and not in a subdirectory.

Here is a simple example:

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

static void
_test_notify_cb(void *data, Eio_File *handler, const Eio_Progress *info)
{
    switch (info->op)
    {
        case EIO_FILE_COPY:
            printf("[%s] %f%%\n", info->dest, info->percent);
            break;
        case EIO_DIR_COPY:
            printf("global [%li/%li] %f%%\n", info->current, info->max, info->percent);
            break;
    }
}

static void
_test_done_cb(void *data, Eio_File *handler)
{
    printf("copy done\n");
    ecore_main_loop_quit();
}

static void
_test_error_cb(int error, Eio_File *handler, void *data)
{
    fprintf(stderr, "error: [%s]\n", strerror(error));
    ecore_main_loop_quit();
}

int
main(int argc, char **argv)
{
    Eio_File *cp;

    if (argc != 3)
    {
        fprintf(stderr, "eio_cp source_file destination_file\n");
    }
}
```

```
        return -1;
    }

    ecore_init();
    eio_init();

    cp = eio_dir_copy(argv[1], argv[2],
                     _test_notify_cb,
                     _test_done_cb,
                     _test_error_cb,
                     NULL);

    ecore_main_loop_begin();

    eio_shutdown();
    ecore_shutdown();

    return 0;
}
```

Chapter 58

eio_dir_stat_ls() tutorial

- The filter callback, which allow or not a file to be seen by the main loop handler. This callback run in a separated thread.
- The main callback, which receive in the main loop all the file that are allowed by the filter. If you are updating a user interface it make sense to delay the insertion a little, so you get a chance to update the canvas for a bunch of file instead of one by one.
- The end callback, which is called in the main loop when the content of the directory has been correctly scanned and all the file notified to the main loop.
- The error callback, which is called if an error occured or if the listing was cancelled during it's run. You can then retrieve the error type as an errno error.

Here is a simple example that implement a stupidly simple replacement for find:

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

static Eina_Bool
_test_filter_cb(void *data, Eio_File *handler, const Eina_File_Direct_Info *info)
{
    fprintf(stderr, "ACCEPTING: %s\n", info->path);
    return EINA_TRUE;
}

static void
_test_main_cb(void *data, Eio_File *handler, const Eina_File_Direct_Info *info)
{
    fprintf(stderr, "PROCESS: %s\n", info->path);
}

static void
_test_done_cb(void *data, Eio_File *handler)
{
    printf("ls done\n");
    ecore_main_loop_quit();
}

static void
_test_error_cb(void *data, Eio_File *handler, int error)
{
    fprintf(stderr, "error: [%s]\n", strerror(error));
    ecore_main_loop_quit();
}

int
main(int argc, char **argv)
{
    Eio_File *cp;

    if (argc != 2)
    {
        fprintf(stderr, "eio_ls directory\n");
        return -1;
    }

    ecore_init();
```

```
eio_init();

cp = eio_dir_stat_ls(argv[1],
                    _test_filter_cb,
                    _test_main_cb,
                    _test_done_cb,
                    _test_error_cb,
                    NULL);

ecore_main_loop_begin();

eio_shutdown();
ecore_shutdown();

return 0;
}
```

Chapter 59

eio_file_ls() tutorial

To use `eio_file_ls()`, you just need to define four callbacks:

- The filter callback, which allow or not a file to be seen by the main loop handler. This callback run in a separated thread.
- The main callback, which receive in the main loop all the file that are allowed by the filter. If you are updating a user interface it make sense to delay the insertion a little, so you get a chance to update the canvas for a bunch of file instead of one by one.
- The end callback, which is called in the main loop when the content of the directory has been correctly scanned and all the file notified to the main loop.
- The error callback, which is called if an error occured or if the listing was cancelled during it's run. You can then retrieve the error type as an `errno` error.

Here is a simple example:

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

static Eina_Bool
_test_filter_cb(void *data, Eio_File *handler, const char *file)
{
    fprintf(stderr, "ACCEPTING: %s\n", file);
    return EINA_TRUE;
}

static void
_test_main_cb(void *data, Eio_File *handler, const char *file)
{
    fprintf(stderr, "PROCESS: %s\n", file);
}

static void
_test_done_cb(void *data, Eio_File *handler)
{
    printf("ls done\n");
    ecore_main_loop_quit();
}

static void
_test_error_cb(void *data, Eio_File *handler, int error)
{
    fprintf(stderr, "error: [%s]\n", strerror(error));
    ecore_main_loop_quit();
}

int
main(int argc, char **argv)
{
    Eio_File *cp;

    if (argc != 2)
    {
        fprintf(stderr, "eio_ls directory\n");
        return -1;
    }
}
```

```
    }

    ecore_init();
    eio_init();

    cp = eio_file_ls(argv[1],
                    _test_filter_cb,
                    _test_main_cb,
                    _test_done_cb,
                    _test_error_cb,
                    NULL);

    ecore_main_loop_begin();

    eio_shutdown();
    ecore_shutdown();

    return 0;
}
```


Chapter 60

eio_monitor_add() tutorial

To use `eio_monitor_add()`, you have to define callbacks for events declared by `eio`. Available events are :

- `EIO_MONITOR_FILE_CREATED`
- `EIO_MONITOR_FILE_DELETED`
- `EIO_MONITOR_FILE_MODIFIED`
- `EIO_MONITOR_FILE_CLOSED`
- `EIO_MONITOR_DIRECTORY_CREATED`
- `EIO_MONITOR_DIRECTORY_DELETED`
- `EIO_MONITOR_DIRECTORY_CLOSED`
- `EIO_MONITOR_SELF_RENAME`
- `EIO_MONITOR_SELF_DELETED`

As nothing is worth an example, here it is :

```
#include <Eina.h>
#include <Ecore.h>
#include <Eio.h>

void file_modified(void *data, int type, void *event)
{
    const char *filename = (const char *)data;
    printf("file %s ", filename);
    if( type == EIO_MONITOR_FILE_MODIFIED )
        printf("is being modified");
    else if( type == EIO_MONITOR_FILE_CLOSED )
        printf("is not more being modified");
    else printf("got unexpected changes");
    printf("\n");
}

int main(int argc, char **argv) {
    Eio_Monitor *monitor = NULL,
                *monitor2 = NULL;
    eio_init();
    const char *filename = eina_stringshare_add("/tmp/eio_notify_testfile");

    monitor = eio_monitor_add(filename);
    ecore_event_handler_add(EIO_MONITOR_FILE_MODIFIED, (Ecore_Event_Handler_Cb)file_modified, filename);
    ecore_event_handler_add(EIO_MONITOR_FILE_CLOSED, (Ecore_Event_Handler_Cb)file_modified, filename);

    ecore_main_loop_begin();
    eio_shutdown();
    eina_stringshare_del(filename);
}
```

Build the example doing :

```
gcc -o tutorial_monitor_add tutorial_monitor_add.c `pkg-config --libs --cflags eio ecore ecore-file eina`  
* then create the file /tmp/eio_notify_testfile :  
* touch /tmp/eio_notify_testfile  
* and launch tutorial_monitor_add, and in another terminal, write into /tmp/eio_notify_testfile, doing for example:  
* echo "test" >> /tmp/eio_notify_testfile  
*
```

Chapter 61

eio_dir_direct_ls() tutorial

- The filter callback, which allow or not a file to be seen by the main loop handler. This callback run in a separated thread. It also take care of getting a stat buffer needed by the main callback to display the file size.
- The main callback, which receive in the main loop all the file that are allowed by the filter. If you are updating a user interface it make sense to delay the insertion a little, so you get a chance to update the canvas for a bunch of file instead of one by one.
- The end callback, which is called in the main loop when the content of the directory has been correctly scanned and all the file notified to the main loop.
- The error callback, which is called if an error occured or if the listing was cancelled during it's run. You can then retrieve the error type as an errno error.

Here is a simple example that implement a stupidly simple recursive ls that display file size:

```
#include <Eina.h>
#include <Ecore.h>
#include <Eio.h>

static Eina_Bool
_test_filter_cb(void *data, Eio_File *handler, Eina_File_Direct_Info *info)
{
    Eina_Stat *buffer;
    Eina_Bool isdir;

    isdir = info->type == EINA_FILE_DIR;

    buffer = malloc(sizeof (Eina_Stat));
    if (eina_file_statat(eio_file_container_get(handler), info, buffer))
    {
        free(buffer);
        return EINA_FALSE;
    }

    if (!isdir && info->type == EINA_FILE_DIR)
    {
        struct stat st;
        if (lstat(info->path, &st) == 0)
        {
            if (S_ISLNK(st.st_mode))
                info->type = EINA_FILE_LNK;
        }
    }

    eio_file_associate_direct_add(handler, "stat", buffer, free);
    fprintf(stdout, "ACCEPTING: %s\n", info->path);
    return EINA_TRUE;
}

static void
_test_main_cb(void *data, Eio_File *handler, const Eina_File_Direct_Info *info)
{
    struct stat *buffer;

    buffer = eio_file_associate_find(handler, "stat");
    fprintf(stdout, "PROCESS: %s of size %li\n", info->path, buffer->st_size);
}
```

```
static void
_test_done_cb(void *data, Eio_File *handler)
{
    printf("ls done\n");
    ecore_main_loop_quit();
}

static void
_test_error_cb(void *data, Eio_File *handler, int error)
{
    fprintf(stdout, "error: [%s]\n", strerror(error));
    ecore_main_loop_quit();
}

int
main(int argc, char **argv)
{
    Eio_File *cp;

    if (argc != 2)
    {
        fprintf(stdout, "eio_ls directory\n");
        return -1;
    }

    ecore_init();
    eio_init();

    cp = eio_dir_direct_ls(argv[1],
        _test_filter_cb,
        _test_main_cb,
        _test_done_cb,
        _test_error_cb,
        NULL);

    ecore_main_loop_begin();

    eio_shutdown();
    ecore_shutdown();

    return 0;
}
```

Chapter 62

Eldbus Examples

Examples:

- [banshee.c](#)
- [client.c](#)
- [complex-types.c](#)
- [complex-types-client-eina-value.c](#)
- [complex-types-server.c](#)
- [connman-list-services.c](#)
- [ofono-dial.c](#)
- [server.c](#)
- [simple-signal-emit.c](#)

Chapter 63

Emotion Examples

Here is a page with some Emotion examples explained:

- [Emotion - Basic library usage](#)
- [Emotion signals](#)
- [emotion_test - full API usage](#)

Chapter 64

Emotion - Basic library usage

This example shows how to setup a simple Emotion object, make it start playing and register a callback that tells when the playback started. See [the full code here](#).

We start this example by including some header files that will be necessary to work with Emotion, and to display some debug messages:

Then a callback will be declared, to be called when the object starts its playback:

Some basic setup of our canvas, window and background is necessary before displaying our object on it. This setup also includes reading the file to be opened from the program's argument list. Since this is not directly related to Emotion itself, we are just displaying the code for this without an explanation for it:

Finally, we start the Emotion part. First we have to create the object in this canvas, and initialize it:

Notice that we didn't specify which module will be used, so emotion will use the first module found. There's no guarantee of the order that the modules will be found, so if you need to use one of them specifically, please be explicit in the second argument of the function `emotion_object_init()`.

Now the callback can be registered to this object. It's a normal Evas smart object callback, so we add it with `evas_object_smart_callback_add()`:

The object itself is ready for use, but we need to load a file to it. This is done by the following function:

This object can play audio or video files. For the latter, the image must be displayed in our canvas, and that's why we need to add the object to the canvas. So, like any other Evas object in the canvas, we have to specify its position and size, and explicitly set its visibility. These are the position and dimension where the video will be displayed:

Since the basic steps were done, we can now start playing our file. For this, we can just call the basic playback control function, and then we can go to the main loop and watch the audio/video playing:

The rest of the code doesn't contain anything special:

This code just free the canvas, shutdown the library, and has an entry point for exiting on error.

Chapter 65

Eo Tutorial

65.1 Purpose

The purpose of this document is to explain how to work with Eo, how to port your code to Eo and what are the common pitfalls. It doesn't explain how it works inside.

65.2 Description

Eo is an Object oriented infrastructure for the EFL. It is a API/ABI safe library.

It supports inheritance, mixins, interfaces and composite objects.

Every class can implement functions from every other class.

It supports event signals, function overrides, private/protected/public/etc. variables and functions.

At the creation of the class, a "virtual table" is filled with the needed functions.

The key of this table is a (class id, function id) tuple.

eo_do() is invoked with a list of op ids and their parameters and is in charge to dispatch the relevant functions. Finding the correct function is fast because it is just a lookup table.

65.3 How to use it?

- Creation of an instance of a class

- Old way:

```
object = evas_object_line_add(canvas);
```

- Eo way:

```
object = eo_add(EVAS_OBJ_LINE_CLASS, canvas);
```

- Call to function

- Old way:

```
evas_object_move(obj, 120, 120);  
evas_object_resize(obj, 200, 200);
```

- Eo way:

```
eo_do(obj,  
      evas_obj_move(120, 120),  
      evas_obj_resize(200, 200));
```

- Extract specific data

- Old way:

```
Evas_Object_Line *o = (Evas_Object_Line *) (obj->object_data);
MAGIC_CHECK(o, Evas_Object_Line, MAGIC_OBJ_LINE);
return;
MAGIC_CHECK_END();
```

- Eo way: Two functions can be used to extract object data. The use depends if you want to store the data or not. If you just need to access data in the function (most of the time), just use `eo_data_scope_get`. If you need to store the data (for example in a list of objects data), you have to use `eo_data_ref`. This function references the data. If you don't need the referenced data anymore, call `eo_data_unref`. This reference mechanism will be used in the future to detect bad usage of objects, defragment the memory space used by the objects...

```
Evas_Object_Line *o = eo_data_scope_get(obj, EVAS_OBJ_LINE_CLASS);
if (!o) return;
```

or

```
Evas_Object_Line *o = eo_data_ref(obj, EVAS_OBJ_LINE_CLASS);
if (!o) return;
...
eo_data_unref(obj, o);
```

- Call function of parent

- Old way:

```
ELM_WIDGET_CLASS(_elm_button_parent_sc)->theme(obj));
```

- New way:

```
eo_do_super(obj, elm_wdg_theme(&int_ret));
```

65.4 Important to know

- `eo_do()` is the function used to invoke functions of a specific class on an object.
- `eo_data_scope_get()` and `eo_data_ref()` receives an object and a class and returns the data of the given class for the object. The class must belong to the object class hierarchy.
- `eo_data_unref()` receives an object and the data to unreference. The data **MUST** belong to this object.
- `eo_isa()` indicates if a given object is of a given type.
- `eo_do_super()` is in charge to invoke a function in the next parents that implement it. It is recommended to use `eo_do_super()` only from a function with the same op id.
In addition, there is no way to jump over classes who implement the function. If A inherits from B, B from C and A, B and C implement a virtual function defined in C, the function calls order will be A, then B and finally C. It is impossible to pass over B.
- `eo_do()` returns if the operation succeeded or failed (function found, object deleted...), not the result of the called function. Pay attention to this detail when you call `eo_do()`. The return value needs to be an additional parameter which will hold a return value.
- Don't do this:

```
int w, h;
eo_do(obj,
    evas_obj_size_get(&w, &h),
    evas_obj_size_set(w+10, h+20));
```

w+10 and h+20 are evaluated before the call to size_get. Instead, separate it in two calls to eo_do().

- When creating an object with eo_add(), the reference counter of this one is incremented. If it is called with a parent, two references are on the object. eo_del() removes both of these references.

When there is no more references on an object, this one is deleted and then can be freed. The deletion calls to the destructor (see eo_destructor()). When done, Eo checks if the object can be freed. The free mechanism can be "disabled" through eo_manual_free_set(). If this is the case, it is the responsibility of the developer to call eo_manual_free() on the object in order to free it. This mechanism has been used for example in [Evas](#) on the Evas objects and in [Ecore](#).

- When eo_do() reaches a function of a class, it is the responsibility of the user to extract from the va_list ALL the parameters needed for this function, NO MORE, NO LESS. Otherwise, unknown behavior can occur. eo_do() is called with a list of op id, params, op id, params... A bad extraction of parameters can bring to the parsing of a wrong op id and so in the best case, to an error, in the worst case, to another function not in relation with the actual use case.
- Always pay attention to:
 - the pairing between function id and the function itself. Using the same function for two ids occurs and is hard to debug.
 - the definition of the function macros in the H file and their parameters order/type
 - to extract all the parameters of a function from the va_list
- Enum of the op ids in H file and descriptions in C file must be synchronized, i.e same number of ids and same order. If you change the order by adding, removing or moving ids, you break ABI of your library.
- Avoid exposing your class data to prevent ABI break. Supply access functions instead.

65.5 How to create a class - H side?

- If the object is new, establish the public APIs
- #define \$(CLASS_NAME) \$(class_name)_class_get(): will be used to access data/inherit from this class...
- const Eo_Class *\$(class_name)_class_get(void) EINA_CONST: declaration of the function that will create the class (not the instance), i.e virtual table...
- extern EAPI Eo_Op \$(CLASS_NAME)_BASE_ID: class id that will be essentially used to identify functions set of this class
- enum of the function ids of the class in the form \$(CLASS_NAME)_SUB_ID: used to identify the function inside the class; function id is unique per class but (class id, function id) is unique per system..
- #define \$(CLASS_NAME)_ID(sub_id) (\$(CLASS_NAME)_BASE_ID + sub_id): formula to calculate the system function id
- define of each function consists of:
 - the name of the function that will be used in eo_do
 - parameters without types
 - \$(CLASS_NAME)_ID\$(CLASS_NAME)_SUB_ID_FUNCTION
 - EO_TYPECHECK for each parameter: type and variable name
- And don't forget to document each function
- Example (Evas Object Line):

```

#define EVAS_OBJ_LINE_CLASS evas_object_line_class_get()

const Eo_Class *evas_object_line_class_get(void) EINA_CONST;

extern EAPI Eo_Op EVAS_OBJ_LINE_BASE_ID;

enum
{
    EVAS_OBJ_LINE_SUB_ID_XY_SET,
    EVAS_OBJ_LINE_SUB_ID_XY_GET,
    EVAS_OBJ_LINE_SUB_ID_LAST
}

#define EVAS_OBJ_LINE_ID(sub_id) (EVAS_OBJ_LINE_BASE_ID + sub_id)

/*
    @def evas_obj_line_xy_set
    @since 1.8

    Sets the coordinates of the end points of the given evas line object.

    @param[in] x1
    @param[in] y1
    @param[in] x2
    @param[in] y2

    */
#define evas_obj_line_xy_set(x1, y1, x2, y2) EVAS_OBJ_LINE_ID(EVAS_OBJ_LINE_SUB_ID_XY_SET),
    EO_TYPECHECK(Evas_Coord, x1), EO_TYPECHECK(Evas_Coord, y1), EO_TYPECHECK(Evas_Coord, x2), EO_TYPECHECK(Evas_Coord, y2)

/*
    @def evas_obj_line_xy_get
    @since 1.8

    Retrieves the coordinates of the end points of the given evas line object.

    @param[out] x1
    @param[out] y1
    @param[out] x2
    @param[out] y2

    */
#define evas_obj_line_xy_get(x1, y1, x2, y2) EVAS_OBJ_LINE_ID(EVAS_OBJ_LINE_SUB_ID_XY_GET),
    EO_TYPECHECK(Evas_Coord *, x1), EO_TYPECHECK(Evas_Coord *, y1), EO_TYPECHECK(Evas_Coord *, x2), EO_TYPECHECK(Evas_Coord *, y2)

```

65.6 How to create a class - C side?

Below, the object line as example.

```

#include "Eo.h"
EAPI Eo_Op \$(CLASS_NAME)_BASE_ID = EO_NOOP; // Initialisation of the class id to 0. Will be set
    dynamically by Eo itself.
#define MY_CLASS \$(CLASS_NAME)

...

```

Example for a developer function called by Eo:

This function receives the Eo object, the data corresponding to this class and the list of parameters.

```

static void
foo(Eo *eo_obj, void *_pd, va_list *list)
{
    int param_1 = va_arg(*list, int);
    Eina_Bool param_2 = va_arg(*list, int);
    Eina_Bool ret = va_arg(*list, Eina_Bool *);
    foo_data obj = _pd;

    if (ret) *ret = EINA_FALSE;
    ...
}

```

You can (not a must) implement a constructor. This constructor MUST call the parent constructor (eo_do_super()). It is the same for the destructor.

See eo_constructor() and eo_destructor().

If you don't have anything to do in constructor (like malloc, variables init...) or in destructor (free), don't implement them.

At the end of the file, you need to describe the class.

First, you need to supply the class constructor that sets the list of the functions that are implemented in this class. It includes functions overridden (constructor, destructor, member_add...) and functions specific to this class. If there is no function implemented in this class, you don't need a class constructor.

Then, you need a list to describe the new functions. For each one, the op id and a description.

Then, the class itself that consists in:

- the version of Eo, used for backward compatibility
- the class name
- the type of the class:
 - regular: an object can be created from this type (mostly used)
 - regular non instant-iable: Same as previous, but objects can't be created from this type (elm_widget)
 - interface: used to extend classes, no implementations for functions and no internal data
 - mixin : interfaces with internal data and pre-made implementations for functions
- the descriptions list described above and the number of ops for this class
- events
- size of the data of the class
- the class constructor, if exists, else NULL
- the class destructor (NULL most of the time)

Finally, we define the class with:

- the function name to give to the function that creates/returns the class
- the class description explained above
- the parent and the interfaces/mixins, finished by NULL

Example (Evas Object Line):

```
#include "Eo.h"

EAPI Eo_Op EVAS_OBJ_LINE_BASE_ID = EO_NOOP;

#define MY_CLASS EVAS_OBJ_LINE_CLASS

...

static void
_line_xy_get(Eo *eo_obj, void *_pd, va_list *list)
{
    const Evas_Object_Line *o = _pd;

    Evas_Coord *x1 = va_arg(*list, Evas_Coord *);
    Evas_Coord *y1 = va_arg(*list, Evas_Coord *);
    Evas_Coord *x2 = va_arg(*list, Evas_Coord *);
    Evas_Coord *y2 = va_arg(*list, Evas_Coord *);

    Evas_Object_Protected_Data *obj = eo_data_scope_get(eo_obj, EVAS_OBJ_CLASS);
    if (x1) *x1 = obj->cur.geometry.x + o->cur.x1;
    if (y1) *y1 = obj->cur.geometry.y + o->cur.y1;
    if (x2) *x2 = obj->cur.geometry.x + o->cur.x2;
    if (y2) *y2 = obj->cur.geometry.y + o->cur.y2;
}

static void
```

```

_constructor(Eo *eo_obj, void *class_data, va_list *list EINA_UNUSED)
{
    eo_do_super(eo_obj, eo_constructor());

    Evas_Object_Protected_Data *obj = eo_data_scope_get(eo_obj, EVAS_OBJ_CLASS);
    evas_object_line_init(eo_obj);
}

...

/* class constructor */
static void
_class_constructor(Eo_Class *klass)
{
    const Eo_Op_Func_Description func_desc[] = {
        /* Virtual functions of parent class implemented in this class */
        EO_OP_FUNC(EO_BASE_ID(EO_BASE_SUB_ID_CONSTRUCTOR), _constructor),
        EO_OP_FUNC(EO_BASE_ID(EO_BASE_SUB_ID_DESTRUCTOR), _destructor),
        /* Specific functions to this class */
        EO_OP_FUNC(EVAS_OBJ_LINE_ID(EVAS_OBJ_LINE_SUB_ID_XY_SET), _line_xy_set),
        EO_OP_FUNC(EVAS_OBJ_LINE_ID(EVAS_OBJ_LINE_SUB_ID_XY_GET), _line_xy_get),
        EO_OP_FUNC_SENTINEL
    };

    eo_class_funcs_set(klass, func_desc);
}

/* Descriptions for the functions specific to this class */
static const Eo_Op_Description op_desc[] = {
    EO_OP_DESCRIPTION(EVAS_OBJ_LINE_SUB_ID_XY_SET, "Sets the coordinates of the end points of the given
        evas line object."),
    EO_OP_DESCRIPTION(EVAS_OBJ_LINE_SUB_ID_XY_GET, "Retrieves the coordinates of the end points of the
        given evas line object."),
    EO_OP_DESCRIPTION_SENTINEL
};

/* Description of the class */
static const Eo_Class_Description class_desc = {
    EO_VERSION,
    "Evas_Object_Line",
    EO_CLASS_TYPE_REGULAR,
    EO_CLASS_DESCRIPTION_OPS(&EVAS_OBJ_LINE_BASE_ID, op_desc, EVAS_OBJ_LINE_SUB_ID_LAST),
    NULL,
    sizeof(Evas_Object_Line),
    _class_constructor,
    NULL
};

/* Definition of the class */
EO_DEFINE_CLASS(evas_object_line_class_get, &class_desc, EVAS_OBJ_CLASS, NULL);

```


Chapter 66

EPhysics Examples

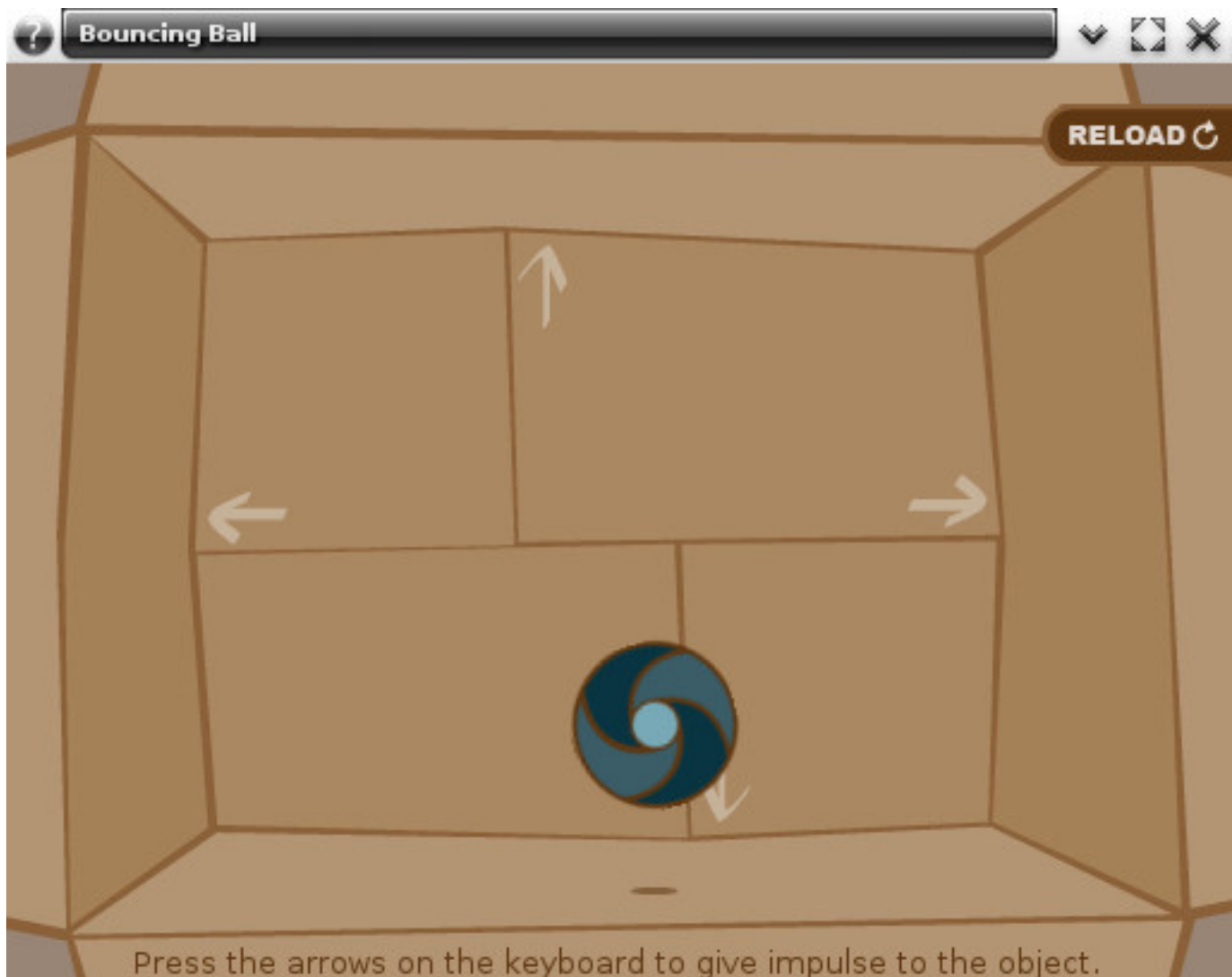
Examples:

- [EPhysics - Bouncing Ball](#)
- [EPhysics - Bouncing Text](#)
- [EPhysics - Camera](#)
- [EPhysics - Camera Track](#)
- [EPhysics - Collision Detection](#)
- [EPhysics - Collision Filter](#)
- [EPhysics - Delete Body](#)
- [EPhysics - Constraint](#)
- [EPhysics - Forces](#)
- [EPhysics - Growing Balls](#)
- [EPhysics - Gravity](#)
- [EPhysics - Logo](#)
- [EPhysics - Rotating Forever](#)
- [EPhysics - Velocity](#)
- [EPhysics - Shapes](#)
- [EPhysics - Sleeping Threshold](#)
- [EPhysics - Slider](#)

Chapter 67

EPhysics - Bouncing Ball

The purpose of this example is to show how to write an simple application - as the name suggests - with a small ball bouncing on the ground and responding to users events by making it jump - applying a central impulse on it.



We'll guide you on defining a EPhysics world, defining its render geometry and the physics limiting boundaries, you'll learn how to add EPhysics bodies and how to associate it to evas objects. We also explain how to change restitution and friction properties. We see how to apply central impulse on a EPhysics_Body by implementing an elementary input event callback and calling the proper function.

67.1 A test struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

67.2 World Initialization

Calling `ephysics_world_new()` will create a new physics world with its collision configuration, constraint solver, broad-phase interface and dispatcher.

The default gravity is set to -9.81. It's possible to stop a running world but its default status is running. Take a look at `ephysics_world_running_set()` for further informations about world running status.

67.3 Render geometry

By setting the render geometry you tell ephysics the dimensions of rendered area to be take on account by default updates.

By default it starts with null x, y, z, width, height and depth. Initially there's no physics limits but - as we'll see later in this example - boundaries can be added by issuing either `ephysics_body_top_boundary_add()`, `ephysics_body_bottom_boundary_add()`, `ephysics_body_left_boundary_add()` and `ephysics_body_right_boundary_add()`.

While setting the worlds render geometry the first parameter is our just created world, the following parameters indicate the x, y, z, width, height and depth of our area of interest.

67.4 Adding boundaries

Boundaries are physics limits added by EPhysics which you can use to limit the area where your objects can move around. Bear in mind that those boundaries are created by EPhysics taking in account the render geometry you have previously defined by calling `ephysics_world_render_geometry_set()`.

In our example we start by adding a bottom boundary. This `EPhysics_Body` represents a physics limit under the world render geometry.

The second line states the restitution factor for that bottom boundary, and the third line its friction. These changes will make our ball to bounce whenever it hits the ground.

Then we add a right boundary limiting the physics world on the left side, we also change its restitution and friction factors but with a smaller value, we don't want to make it bounce as much as it is when hits the ground.

We also add a left boundary taking the same considerations for right boundary.

One of this examples requirements is to make the ball jump after a specific user event, so the ball can suffer an impulse for any direction.

With an upper impulse we don't want our ball to fly all over there, we want to limit its upper movements, it's intended to limit the ball movement within a box, it should not leave the render geometry area, for that purpose we must define a top boundary.

67.5 Adding a ball

Since we have defined the physics limits with our boundaries it's time to add some fun. Here we add a ball as an elementary image widget and tell ephysics about it.

After setting the file that will be used as the image's source of our elm image we move it to the center of render geometry and resize it to 70x70 pixels and show it.

The evas object is just set and we must tell EPhysics about it, creating the EPhysics_Body representing our ball and associating it to the just created evas object.

Once the ball has been moved to the center of render geometry it should start falling after associating it to the EPhysics_Body. By default its mass is initially set to 1 kilo, but it can be changed by calling `ephysics_body_mass_set()`. Bear in mind that if you change its mass to 0 kilos it becomes a static body and will not move at all, the body will remain fixed in the initial position.

In the following code the first line adds a circle body, then we associate the evas object to EPhysics_Body, EPhysics will map every changes on physics object simulation to its evas object. Some restitution and friction factors are added as well.

67.6 Making it jump

The next step is to give us the ability to make our ball to jump - actually apply some impulse whenever a key has been pressed. Then we add a elementary input callback to the window widget.

The jumping callback implementation consists on handling only key up events and discarding any other input event we get. We're interested on keyboard events only. All the operations done in the following lines are done on sphere EPhysics_Body previously created.

We mainly use the `ephysics_body_central_impulse_apply()` function. This function applies an impulse on the center of a body.

Once pressed <Up> key it applies a central impulse of 0 kilos on X axis, 10 kilos on Y and 0 kilos on Z - so the ball is forced up.

If <Down> key has been pressed we apply an impulse of 0 kilos on X axis, -10 kilos on Y and 0 kilos on Z - here the ball is forced down.

In the case of <Right> key pressing it's applied an impulse of 10 kilos on X axis, 0 kilos on Y and 0 kilos on Z - which applies a force to the right side. But if the key being pressed is <Left> the opposite is done, and an impulse of -10 kilos is applied on X, 0 kilos on Y and 0 kilos on Z - and the ball is forced to the left.

Here we finish the very simple bouncing ball example. The full source code can be found at [test_bouncing_ball.c](#).

Chapter 68

test_bouncing_ball.c

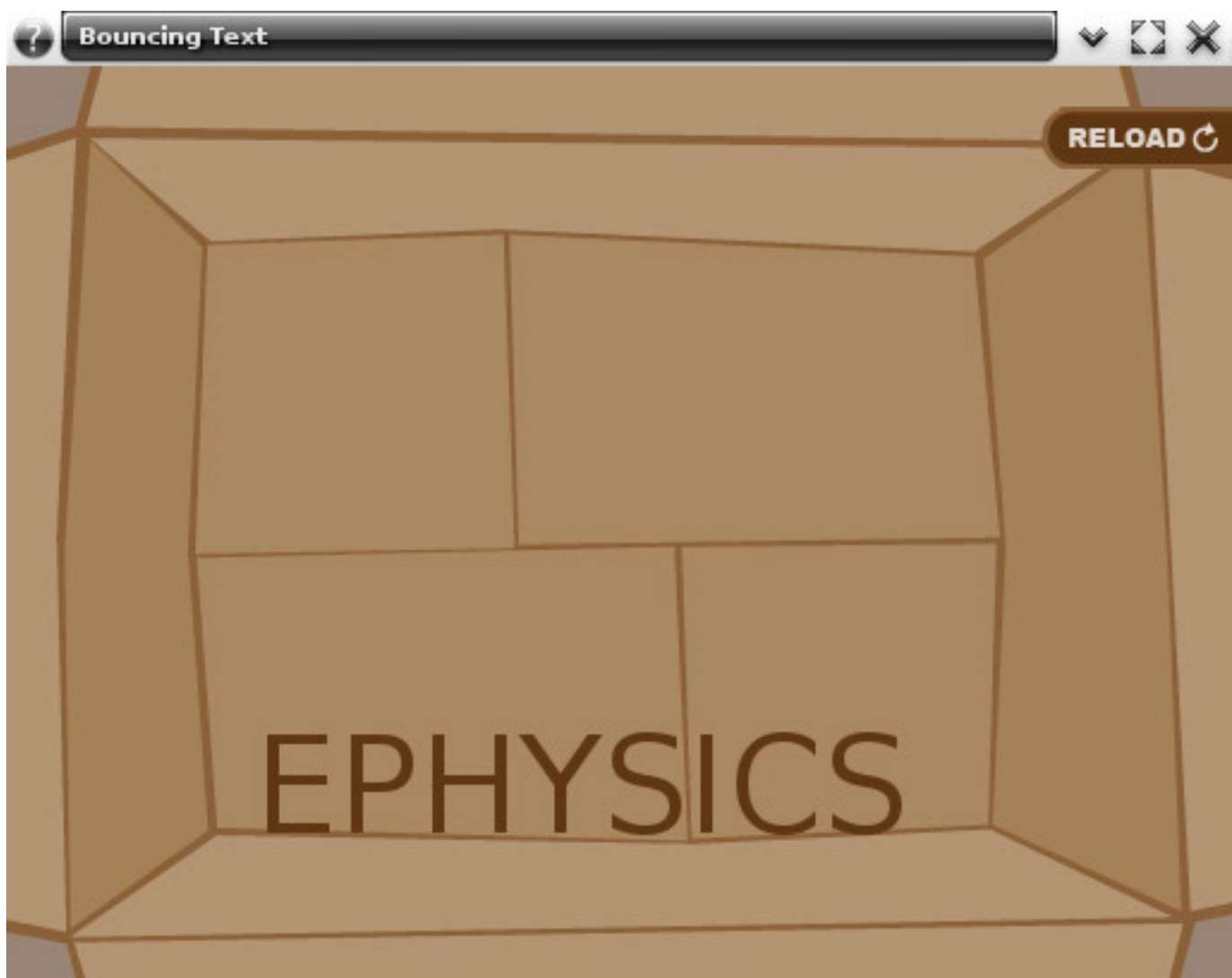
68.1 ephysics_test.h

68.2 test_bouncing_ball.c

Chapter 69

EPhysics - Bouncing Text

The purpose of this example is to demonstrate the EPhysics_Body binding to a text (Evas_Object)



For this example we'll have an EPhysics_World and one basic EPhysics_Body.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, were already covered in [EPhysics - Bouncing Ball](#)

69.1 Creating the text

Create a basic `evas_object_text`.

69.2 Creating the body

Create a simple `EPhysics_Body`.

Note that we use `ephysics_body_geometry_set()` to define its size because the `evas_object` has a different size that we want to represent physically. The text may have accent or letters like `j` and `g`.

69.3 Binding

After creating the body and the text, now we need to bind them.

We set the last parameter as `EINA_FALSE` because in this example we don't want to set the physics body position to match `evas` object position.

Here we finish the example. The full source code can be found at [test_bouncing_text.c](#).

Chapter 70

test_bouncing_text.c

70.1 ephysics_test.h

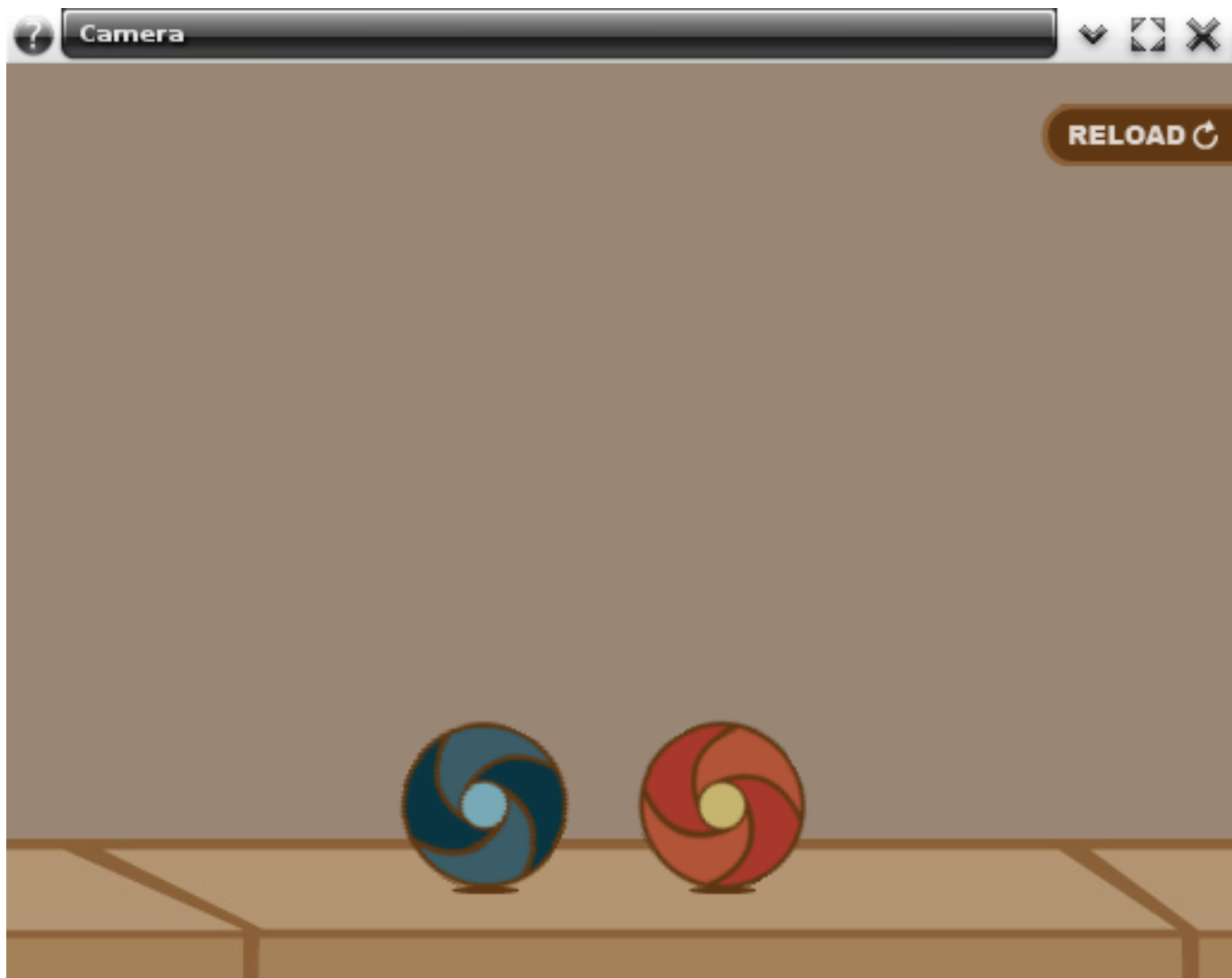
70.2 test_bouncing_text.c

Chapter 71

EPhysics - Camera

The purpose of this example is to demonstrate the EPhysics_Camera usage.

The EPhysics_Camera facilitates the usage of scenarios bigger than the viewport, that's because the EPhysics handles the position of objects which has control.



For this example we'll have an EPhysics_World, two distant EPhysics_Bodys, one with an impulse to collide each other and an EPhysics_Camera that follows the moving body using an animator.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, add an Ephysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

71.1 Camera Data Struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

71.2 Adding a Camera

To move the camera in this example, we'll use an animator.

In the animators function, we'll have to create a specific type of variable: EPhysics_Camera And also get the worlds rendered area width to define a limit to the camera.

Every world has a camera, so here we get this camera used by our EPhysics_World.

Here we get the cameras position to after set the position based on previous.

Here we check if the camera reached the end of scenario (define the limit to the camera) then we stop the animator, else we move the camera + 2 pixel positions to the right.

71.3 Updating the floor

Here we'll use 2 floor images to give the impression of an infinite ground.

Calling ephysics_world_event_callback_add() will register a callback to a type of physics world event.

EPHYSICS_CALLBACK_WORLD_CAMERA_MOVED : called if the camera position changed on physics simulation tick.

In the function, we just get the cameras position to know how much the camera moved and move the same value to the floor passing it as delta_x to the function, note that we use an old_x variable to do this calculation.

Here we get the floors position and plus the delta_x value to move the floor in the same "velocity".

We use 2 floor images because whenever one exits the screen by the left side, another is being shown, when it happens the one which exit the screen is sent to the right side, entering into an infinite loop, giving the impression of an infinite ground image. Its important to note that we need to use the fx to don't gap the images.

Here we finish the example. The full source code can be found at [test_camera.c](#).

Chapter 72

test_camera.c

72.1 ephysics_test.h

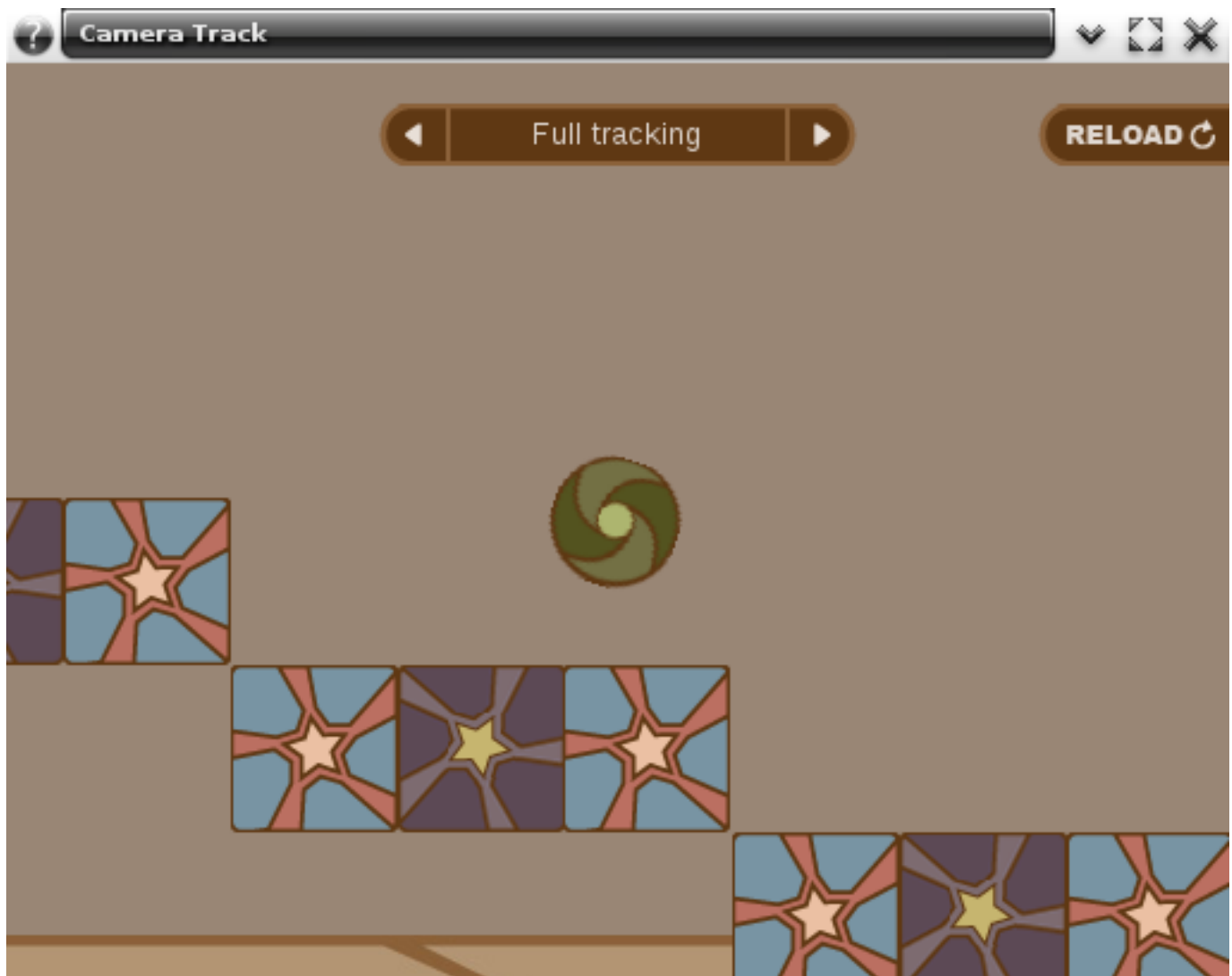
72.2 test_camera.c

Chapter 73

EPhysics - Camera Track

The purpose of this example is to demonstrate the EPhysics_Camera Track usage.

The EPhysics_Camera facilitates the usage of scenarios bigger than the viewport, that's because the EPhysics handles the position of objects which has control.



For this example we'll have an EPhysics_World, one main EPhysics_Body that will be tracked by an EPhysics_Camera on three ways, horizontal, vertical and full tracking. Also nine EPhysics_Bodys with mass 0, that will be

used as scenario in order to our main body change its position on x and y axes when passes through this scenario. The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, add an Ephysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

73.1 Track Data Struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

73.2 Adding a Camera

In this example we'll use 3 kinds of tracking, to change this values we'll have an Elementary spinner widget and handle it on this function.

Every world has a camera, so here we get this camera used by our EPhysics_World.

Here we'll get the elm_spinner value to the tracking base on this value

Here we'll set the camera to track the body, when a body is tracked, the camera will move automatically, following this body. It will keeps the body centralized on rendered area. If it will be centralized horizontally and / or vertically depends if parameters horizontal and vertical are set to EINA_TRUE, in this case we based these values on elm_spinner.

73.3 Updating the floor

Here we'll use 2 floor images to give the impression of an infinite ground.

Calling ephysics_world_event_callback_add() will register a callback to a type of physics world event.

EPHYSICS_CALLBACK_WORLD_CAMERA_MOVED : called if the camera position changed on physics simulation tick.

In the function, we'll get the cameras position to know how much the camera moved and move the same value to the floor passing it as delta_x to the function, note that we use an old_x variable to do this calculation.

We'll get also if the body is being tracked on x and y axes. If the body isn't being tracked on x axis the floors x position won't change, delta_x will be zero.

Here we get the floors position and plus the delta_x value to move the floor in the same "velocity".

We use 2 floor images because whenever one exits the screen by the left side, another is being shown, when it happens the one which exit the screen is sent to the right side, entering into an infinite loop, giving the impression of an infinite ground image. Its important to note that we need to use the fx to don't gap the images.

Note that the fy is being defined considering its offsets, -20 is to the floor image be above the floor, thus having an border above the collision point, +40 is the render area height, to offset the cameras y, basically to draw in the correct position in the canvas.

Here we finish the example. The full source code can be found at [test_camera_track.c](#).

Chapter 74

test_camera_track.c

74.1 ephysics_test.h

74.2 test_camera_track.c

Chapter 75

EPhysics - Collision Detection

The purpose of this example is to demonstrate the EPhysics Collision Detection usage - The code adds two balls, one with impulse and the second with a collision detection callback, to show an effect.



For this example we'll have an EPhysics_World, and two basic EPhysics_Bodys, we'll apply an impulse in one of them and the other will be stopped "waiting" for a collision.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, add an

Ephysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

75.1 Collision Data Struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

75.2 Adding the Callbacks

Calling `ephysics_body_event_callback_add()` will register a callback to a type of physics body event.

`EPHYSICS_CALLBACK_BODY_COLLISION` : called just after the collision has been actually processed by the physics engine. In other words, to be notified about a collision between two physical bodies.

See `_EPhysics_Callback_Body_Type` for more event types.

75.3 Collision Function

The callback function will filter the collision to be sure if that body is which we want and then show the effect.

First we need to create a specific variable type to get collision infos: `EPhysics_Body_Collision`

Now we want to know which body collides with and filter it.

We just get the collision position, move the impact effect to this coordinate and send a signal to edge to show it.

Here we finish the example. The full source code can be found at [test_collision_detection.c](#).

Chapter 76

test_collision_detection.c

76.1 ephysics_test.h

76.2 test_collision_detection.c

Chapter 77

EPhysics - Collision Filter

The purpose of this example is to demonstrate the EPhysics Collision Filter usage - The code adds four balls in 2 rows and 2 columns, two on each collision group, the collision only happens when the balls are in the same group (row), to make it easier, balls in the same group has the same color and size.



For this example we'll have an EPhysics_World and four basic EPhysics_Bodys, we'll apply an impulse on then and see what happens when they're in other collision group.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, add an Ephysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

77.1 Adding the Callbacks

We'll use two arrays (color and size) to distinguish the groups.

The balls declaration was placed into a For loop, just to simplify the coding and divide them in two groups.

Note in this part we divide the balls in two groups by color (row).

The impulse will be applied in only 1 ball per group, in this case:

The 1st row 2nd column ball will be applied an impulse to the left ($-300\text{kg} * \text{p/s}$).

The 2nd row 1st column ball will be applied an impulse to the right ($300\text{kg} * \text{p/s}$).

And then saving the body into a list.

Here we finish the example. The full source code can be found at [test_collision_filter.c](#).

Chapter 78

test_collision_filter.c

78.1 ephysics_test.h

78.2 test_collision_filter.c

Chapter 79

EPhysics - Delete Body

The purpose of this example is to demonstrate the EPhysics Callbacks usage - The code adds two balls, one with impulse and the second with a collision detection callback, to delete the body.

For this example we'll have an EPhysics_World and two basic EPhysics_Bodys, we'll apply an impulse in one of them and the other will be stopped "waiting" for a collision.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

79.1 Adding the Callbacks

Calling `ephysics_body_event_callback_add()` registers a callback to a given EPhysics_Body event type.

We'll use two types:

`EPHYSICS_CALLBACK_BODY_DEL` : called when a body deletion has been issued and just before the deletion actually happens. In other words, to know that body has been marked for deletion. Typically to free some data associated with the body.

The callback function will receive the `collision_data` and free some data associated with the body.

`EPHYSICS_CALLBACK_BODY_COLLISION` : called just after the collision has been actually processed by the physics engine. In other words, to be notified about a collision between two physical bodies.

The callback function will get the collision body and check if its body is equal to which we want to delete.

See `_EPhysics_Callback_Body_Type` for more event types.

Here we finish the example. The full source code can be found at [test_delete.c](#).

Chapter 80

test_delete.c

80.1 ephysics_test.h

80.2 test_delete.c

Chapter 81

EPhysics - Constraint

The purpose of this example is to demonstrate the EPhysics Constraint usage - The code apply a constraint between two cubes.

For this example we'll have an EPhysics_World, and two basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

You can use also a slider constraint: [EPhysics - Slider](#)

81.1 Adding a constraint

Constraint is a specific type of variable in EPhysics.

Here we're working with a point-to-point constraint, its purpose is to join two bodies limiting their movements based on specified anchors.

After we create our 2 EPhysics_Bodys, now we'll add a constraint between them and setting an anchor to first body's Y using a p2p constraint (point to point).

Here we finish the example. The full source code can be found at [test_constraint.c](#).

Chapter 82

test_constraint.c

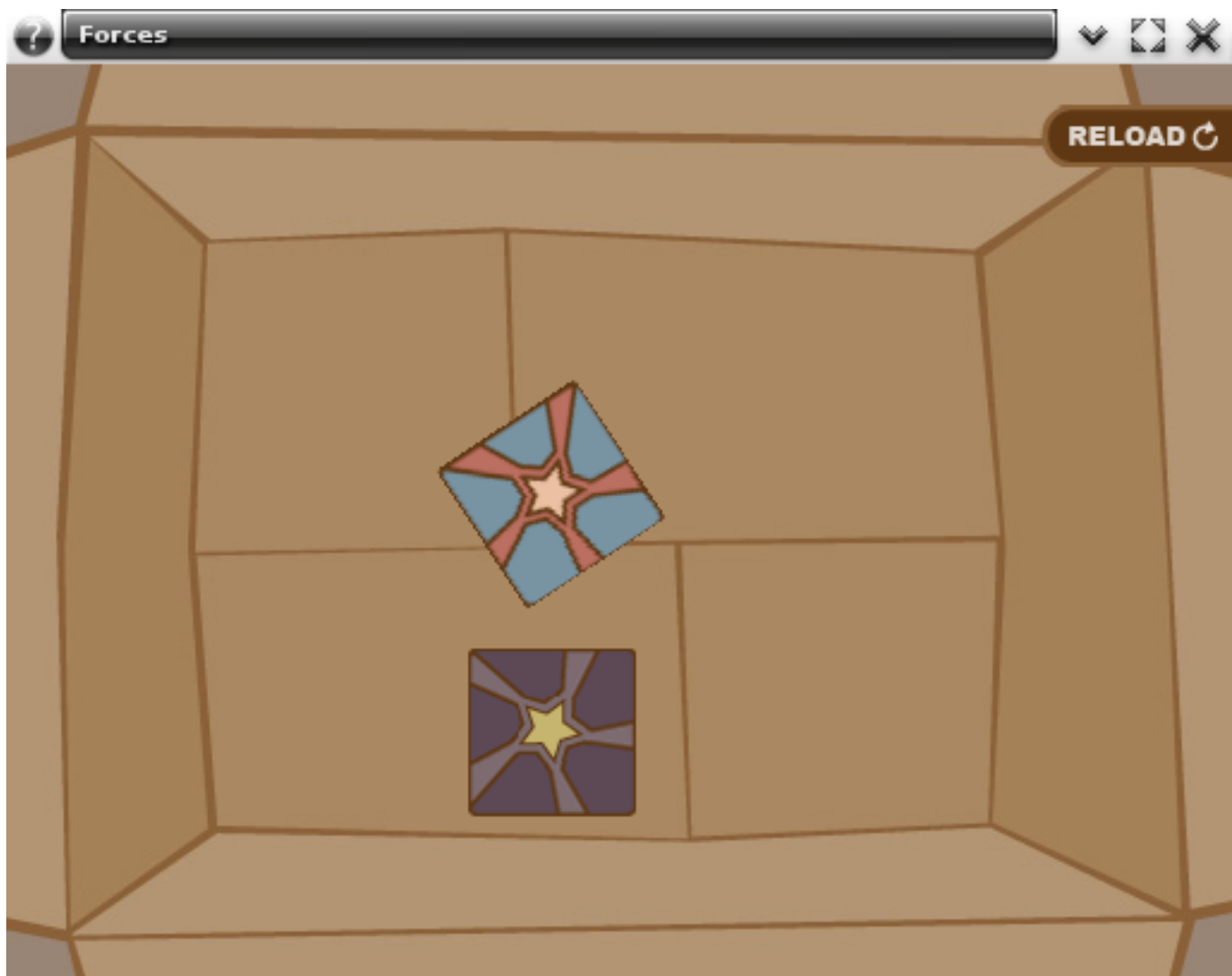
82.1 ephysics_test.h

82.2 test_constraint.c

Chapter 83

EPhysics - Forces

The purpose of this example is to demonstrate the EPhysics Force usage - The code applies force over two cubes.



For this example we'll have an EPhysics_World with gravity setted to zero, and two basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an E-Physics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

83.1 Adding a Force

We apply a force over the first body to change its linear and angular accelerations. Applying a force to a body will lead it to change its velocity gradually.

Note that in this blue cube we use an offset to apply the force, the two last parameters are responsible to set a relative position to apply the force. In other words, the force applied with an offset will make the body rotate. Otherwise (0, 0, 0) the force would be applied on the center of the body, in this case it's recommended to use the `ephysics_body_central_force_apply()`;

Here we apply a central force over the second body avoiding affecting the angular acceleration (rotate).

We can also get all the forces applied over a body, including gravity, but in this case we set it to zero.

Here we finish the example. The full source code can be found at [test_forces.c](#).

Chapter 84

test_forces.c

84.1 ephysics_test.h

84.2 test_forces.c

Chapter 85

EPhysics - Growing Balls

The purpose of this example is to demonstrate the dynamically growing and shrinking of an EPhysics_Body - The code applies the growth of a ball and the shrink of another.



For this example we'll have an EPhysics_World and three EPhysics_Bodys with different sizes associated with an evas_object.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an E-

Physics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

85.1 Adding the growing/shrinking

In this example we'll use a timer to handle the callback function.

In this callback, we'll pass through a list with 3 balls and apply the growth and the shrink between the limit we'll set. Note that the variable *i* receives different values on each iteration (-1, 0, 1). For the first iteration it will decrease the size variable, the second will keep the same value, and the last one will increase the size variable.

Here we finish the example. The full source code can be found at [test_growing_balls.c](#).

Chapter 86

test_growing_balls.c

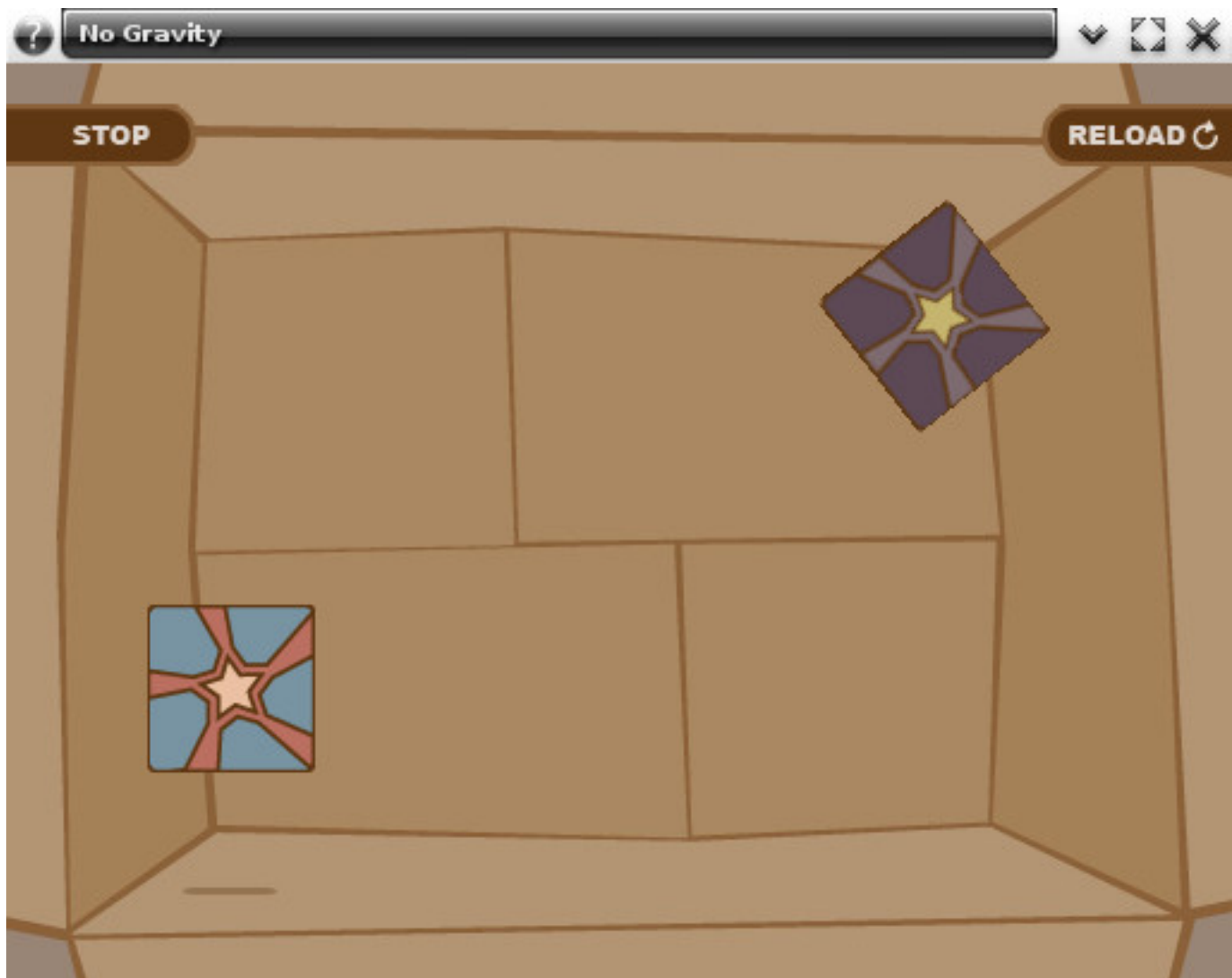
86.1 ephysics_test.h

86.2 test_growing_balls.c

Chapter 87

EPhysics - Gravity

The purpose of this example is to demonstrate the EPhysics Gravity usage - The code apply gravity in an EPhysics_World with two cubes in movement.



For this example we'll have an EPhysics_World, and two basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered

in [EPhysics - Bouncing Ball](#)

Concepts like velocity and sleeping threshold were already covered in:

- [EPhysics - Velocity](#)
- [EPhysics - Sleeping Threshold](#)

87.1 Setting Gravity

Here we set gravity on 3 axes (x, y, z) to (0, 0, 0). Gravity will act over bodies with mass over all the time.

87.2 Stopping a Body

We're using a button to call this function that receives test_data to stop the chosen body.

Stop angular and linear body movement, its equivalent to set linear velocity to 0 on both axis and angular velocity to 0 as well.

Here we finish the example. The full source code can be found at [test_no_gravity.c](#).

Chapter 88

test_no_gravity.c

88.1 ephysics_test.h

88.2 test_no_gravity.c

Chapter 89

EPhysics - Logo

The purpose of this example is to demonstrate the EPhysics_Logo.

For this example we'll have an EPhysics_World.

The basic concepts like - initializing an EPhysics_World, render geometry, physics limiting boundaries, were already covered in [EPhysics - Bouncing Ball](#)

89.1 Logo Data Struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

89.2 Adding the letters

To add the letters we'll use this function that creates the shadow, light and letter images.

In this loop we'll use the function letter_add using the falling_letters declared in logo data struct.

Place image and light on top, above what the viewport can show, to fall later on.

Place shadow below the hit-line: FLOOR_Y, centered at image.

Here we set the letters padding and add letter body using the function below and setting its friction.

Here we call another function that will be common to the circle body as well, note that we add a callback that will be explained later.

This function is used to create the body setting its properties. Note that we disable its angular movement (rotation) on Z axis to this letters don't tilt or recline.

In this callback function that we added to our letter body we'll update its light and shadow.

First we'll update the body, get its image geometry and set the floor distance based on images height.

As long as the letter approaches the floor, its shadow is darker, with bigger y.

And with bigger x – its proportional to x / WIDTH , but varies from 100 to 255

Note that the box shadow is not resized, just moved. And here set also the colors.

As long as the letter approaches the floor, its lighter, with bigger x and y.

89.3 Adding the letter E

Here we'll add the last letter, "E" is a circle that comes rolling on the floor.

First we use the letter_add function, set its shadow color and get its sizes.

Place image and light above the floor and to the left of viewport, to comes rolling later on.

Place the shadow below the hit-line: FLOOR_Y centered at image.

Here we create the body using body_circle function and enable its rotation on Z axis.

Make the "E" logo get into the viewport by applying a horizontal force.

Here we use the letter_body_setup_common to create the body and set its properties, note that we add a callback that will be explained below.

In this callback function that we added to our "E" letter body we'll update its light and shadow.
First we'll update the body and get its image geometry.

As long as the letter approaches the floor, its lighter, with bigger x.

Use the same map from image to the light (rotate it).

As long as the letter approaches the floor, its shadow is darker, with bigger y.

When the letter "E" passes the viewport, we send it to the begin again to collide with the other letters.

Here we finish the example. The full source code can be found at ephysics_logo.c.

Chapter 90

ephysics_logo.c

90.1 ephysics_logo.c

Chapter 91

EPhysics - Rotating Forever

The purpose of this example is to demonstrate the EPhysics Rotate usage - The code applies different ways to rotate an EPhysics_Body, such as torque, torque impulse and rotation set.

For this example we'll have an EPhysics_World with gravity setted to zero, and four basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

91.1 Rotating

For the first body we'll apply a torque impulse to make it rotate around Z axis (rotate on x-y plane). Will make the body rolls on clockwise rotation, if the value is negative, the impulse will be on counter clockwise.

For the second body we'll use an offset to apply the force, the three last parameters are responsible to set a relative position to apply the force. In other words, the force applied with an offset will make the body rotates and move around the other cubes.

For the third body we'll use a timer to rotate the body and a callback to delete it.

In the function we'll get the body rotation on z axis in degrees and handle it increasing 5 degrees on its position on z axis on each tick of the timer.

For the forth body we'll use 2 timers, but before that, we'll apply an initial torque, changing the body angular acceleration and a callback to delete the timers we'll add.

Just the callback function to delete the timers.

As we commented we'll use 2 timers, one to increase the torque and another to stop the torque, cleaning the forces related to the body.

In the increase function we'll apply a torque over the body, changing its angular acceleration, it will leads to a change on angular velocity over time. We're using a timer to increase the angular acceleration on each tick of the timer.

In the stop function we'll clear all the forces applied to the body, setting its linear and angular acceleration to zero. We're using this timer to "control" the body velocity, since we are increasing it by another timer. Note that we set the acceleration to zero not the velocity.

Here we finish the example. The full source code can be found at [test_rotating_forever.c](#).

Chapter 92

test_rotating_forever.c

92.1 ephysics_test.h

92.2 test_rotating_forever.c

Chapter 93

EPhysics - Velocity

The purpose of this example is to demonstrate the EPhysics Velocity usage - The code adds a small bouncing ball on the ground and responding to users events by making it jump - applying a central impulse on it and showing its velocity and acceleration.

We'll see in this example how to get EPhysics_Body Linear and Angular velocity and acceleration.

For this example we'll have an EPhysics_World and one basic EPhysics_Body, we'll apply impulses that follows user events, it were already covered in [EPhysics - Bouncing Ball](#)

93.1 Velocity Data Struct

While in this example we'll be working with a struct to hold some objects in our code. For clarity sake we present you the struct declaration in the following block.

93.2 Adding the Callbacks

Calling `ephysics_body_event_callback_add()` will register a callback to a type of physics body event.

`EPHYSICS_CALLBACK_BODY_UPDATE` : called after every physics iteration. In other words, will be called after each world tick.

`EPHYSICS_CALLBACK_BODY_STOPPED` : called when a body is found to be stopped. In other words, when the body is not moving anymore.

See `_EPhysics_Callback_Body_Type` for more event types.

93.3 Velocity Function

The callback function will be called on every physics iteration to show the linear and angular velocity and acceleration.

Here we're declaring the necessary variables to calculate accelerations and delta time. And checking if its the first time to return before shows informations about the velocity.

Get the delta time to use it soon to calculate the acceleration on every physics iteration.

Note in this part we get the angular and linear velocities.

We need to handle the velocity using delta time to have the acceleration on every tick. Check if its the first time to return before shows informations about the velocity because we don't have the old aceletations and then the calculation of this informations will be wrong.

Here we calculate the aceletarions using this formula:

```
(velocity - old_velocity) / delta_time;
```

Turning data into text, to pass it to edge shows on screen.

Here we finish the example. The full source code can be found at [test_velocity.c](#).

Chapter 94

test_velocity.c

94.1 ephysics_test.h

94.2 test_velocity.c

Chapter 95

EPhysics - Shapes

The purpose of this example is to demonstrate the EPhysics Shapes usage - The code creates two EPhysics_Bodys using a custom shape.



For this example we'll have an EPhysics_World, and two basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered

in [EPhysics - Bouncing Ball](#)

95.1 Adding a Shape

Shapes are used to create bodies with shapes that differ from primitive ones, like box and circle.

A shape consists in a group of points, the vertices of the body to be created later with `ephysics_body_shape_add()`. You can also save and load it from a file.

We'll have to create a specific type of variable: `EPhysics_Shape`

First we add an image we want to add an `EPhysics_Body` to have a reference to after set the points (vertices).

Here we create a new shape, note that the returned shape initially doesn't has points set, so its required to set vertices.

Now we're setting the shape points (vertices) basing on the image that we added, two vertices form a link between them, an edge, so with some vertices is possible to create polygons, in this case a pentagon.

Here we create a new physics body using a custom shape. The center of mass will be the center of the shape. Its collision shape will be the convex shape that has all the points (and edges) we added to this shape before.

Here we just delete the custom shape (not the body) after used to create the wanted bodies, it's required to delete it. It won't be deleted automatically by ephysics at any point, even on shutdown.

In the example we add another shape with the same process we just used, but with different image and points.

Here we finish the example. The full source code can be found at [test_shapes.c](#).

Chapter 96

test_shapes.c

96.1 ephysics_test.h

96.2 test_shapes.c

Chapter 97

EPhysics - Sleeping Threshold

The purpose of this example is to demonstrate the EPhysics Sleeping Threshold usage - The code apply sleeping threshold in two cubes.

For this example we'll have an EPhysics_World, and two basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an E-Physics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

Concept of velocity were already covered in [EPhysics - Velocity](#)

97.1 Adding Max Sleeping Time

Setting the max sleeping time determines how long(in seconds) a rigid body under the linear and angular threshold is supposed to be marked as sleeping.

97.2 Adding a Sleeping Threshold

Here we set EPhysics_Bodys linear and angular sleeping threshold. These factors are used to determine whenever a rigid body is supposed to increment the sleeping time.

After every tick the sleeping time is incremented. After reaching the max sleeping time the body is market to sleep, that means the rigid body is to be deactivated.

We can get the EPhysics_Bodys linear and angular sleeping threshold as well.

97.3 Adding a Damping

Here we set EPhysics_Bodys linear and angular damping values.The damping is a force synchronous with the velocity of the object but in opposite direction - like air resistance.

Here we finish the example. The full source code can be found at [test_sleeping_threshold.c](#).

Chapter 98

test_sleeping_threshold.c

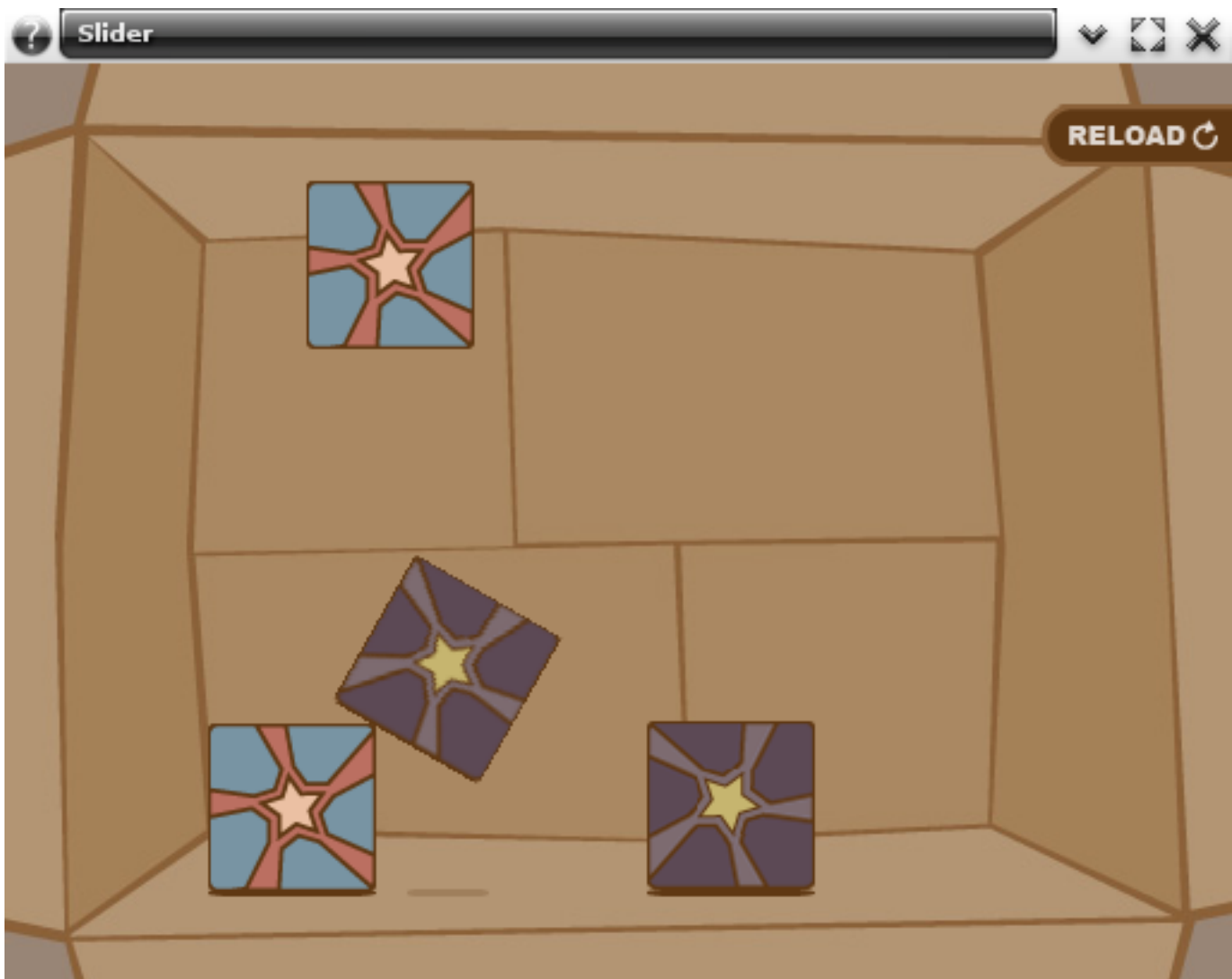
98.1 ephysics_test.h

98.2 test_sleeping_threshold.c

Chapter 99

EPhysics - Slider

The purpose of this example is to demonstrate the EPhysics Slider constraint usage - The code applies slider on three cubes.



For this example we'll have an EPhysics_World, and four basic EPhysics_Bodys.

The basic concepts like - defining an EPhysics_World, render geometry, physics limiting boundaries, add an EPhysics_Body, associate it to evas objects, change restitution, friction and impulse properties, were already covered in [EPhysics - Bouncing Ball](#)

You can use also a P2P (point to point) constraint: [EPhysics - Constraint](#)

99.1 Adding a Slider

Slider is a constraint that will limit the linear and angular moving of a body.

We'll add three sliders on the cubes, starting with the highest purple.

First we need to create a specific variable type to get EPhysics_Body constraint and create a new slider constraint passing the body which we want as parameter.

Here we define the linear moving limits of the slider constraint, in this case we just set moving limit down on Y axis (under), but if we wanted we could set left, right and above also.

Here we set the angular moving limits of the slider constraint. The angular moving limits is defined in degrees and will limit the moving on Z axis, in this case we just set the clockwise direction, but if we wanted we could set the counter clockwise direction also.

When this cube falls by the gravity, the slider constraint will act limiting its linear and angular movings, giving the impression that its hanging.

For the next two cubes is the same process.

Now we set the slider constraint of the highest blue and lowest purple, limiting moving limits to the left on X axis and applying an impulse to the left where the two cubes will be limited by the slider constraint and pushed back.

Here we finish the example. The full source code can be found at [test_slider.c](#).

Chapter 100

test_slider.c

100.1 ephysics_test.h

100.2 test_slider.c

Chapter 101

Evas Examples

Here is a page with examples.

[Simple Evas canvas example](#)

[Evas' init/shutdown routines example](#)

[Evas text object example](#)

[Some image object functions examples](#)

[Some more image object functions examples \(2nd block\)](#)

[Evas events \(canvas and object ones\) and some canvas operations example](#)

[Evas objects basic manipulation example](#)

[Evas aspect hints example](#)

[Evas alignment, minimum size, maximum size, padding and weight hints example](#)

[Evas object stacking functions \(and some event handling\)](#)

[Evas object smart objects](#)

[Evas box example](#) `Evas box`

Chapter 102

Simple Evas canvas example

The canvas will here use the buffer engine.

Chapter 103

Evas' init/shutdown routines example

Chapter 104

Some image object functions examples

In this example, we add two images to a canvas, each one having a quarter of the canvas' size, positioned on the top left and bottom right corners, respectively:

See there is a border image around the top left one, **which is the one that should be displayed**. The other one will (on purpose) fail to load, because we set a wrong file path as image source on it:

This is how one is supposed to test for success when binding source images to image objects: `evas_object_image_load_error_get()`, followed by `evas_load_error_str()`, if one wants to pretty print/log the error. We'll talk about the border image further.

To interact with the program, there's a command line interface. A help string can be asked for with the 'h' key:

The first four commands will change the top left images's **fill** property values, which dictate how the source image (Enlightenment's logo) is to be displayed through the image object's area. Experiment with those switches until you get the idea of `evas_object_fill_set()`.

The 'f' command will toggle that image's "filled" property, which is whether it should track its size and set the fill one to fit the object's boundaries perfectly (stretching). Note that this command and the four above it will conflict: in real usage one would use one or other ways of setting an image object's viewport with regard to its image source.

There are four commands which deal with the border image. This red frame is there to illustrate **image borders**. The image source for the border is a solid red rectangle, with a transparent **rectangular** area in its middle. See how we use it to get a 3 pixel wide frame with `evas_object_image_border_set(d.border, 3, 3, 3, 3)`. To finish the effect of showing it as a border, we issue `evas_object_image_border_center_fill_set(d.border, EVAS_BORDER_FILL_NONE)`.

Use 't' to change the border's thickness. 'b' will change the border image's center region rendering schema: either a hole (no rendering), blending (see the original transparent area, in this case) or solid (the transparent area gets filled). Finally, 'c' will change the border's scaling factor.

While you have the border in 'blending mode', test the command 'm': it will set whether to use or not smooth scaling on the border's source image. Since the image is small originally (30 x 30), we're obviously up-scaling it (except the border pixels, do you remember?). With this last switch, you'll either see the transparent shape in the middle flat (no smoothing) or blurry (smoothed).

The full example follows.

Chapter 105

Some more image object functions examples (2nd block)

In this example, we have three images on the canvas, but one of them is special – we’re using it as a **proxy image object**. It will mirror the contents of the other two images (which are the ones on the top of the canvas), one at a time:

As in other examples, we have a command line interface on it.

The 'p' one will change the source of the proxy image to one of the other two, as seen above.

Note the top right image, the smaller one:

Since we are creating the data for its pixel buffer ourselves, we have to set its size with `evas_object_image_size_set()`, first. We set our data with the function `evas_object_image_data_set()`, where the second argument is a buffer with random data. There’s a last command to print it’s **stride** value. Since its created with one quarter of the canvas’s original width

you can check this value.

The image on the top left also has a subtlety: it is **pre-loaded** on this example.

On real use cases we wouldn’t be just printing something like this

naturally.

The 's' command will save one of the images on the disk, in the png format:

The full example follows.

Chapter 106

Evas events (canvas and object ones) and some canvas operations example

In this example we illustrate how to interact with canvas' (and its objects') events, including the key input ones. We also demonstrate precise point collision on objects and canvas "obscured regions", here.

The example application consists of a window with a white background and an image – the Enlightenment logo. The application begins with this image switching back and forth into two sizes: the exact canvas' size and one quarter of it (when it's placed on the top left quadrant). Thus, we'll have an **animation** going on, with image states set to change each 2 elapsed seconds.

There's a global variable to aid accessing our desired context variables from anywhere in the code:

What interests us there are the `canvas` pointer, our image handle – `img` – and the background one, `bg`.

The first interesting thing on the example is the registration of a callback on each canvas resizing event, where we put our canvas' size and the background rectangle's one in synchrony, so that we don't get bogus content on rendering with canvas resizes:

Then, after grabbing our canvas pointer from the Ecore Evas helper infrastructure, we registrate an event callbacks on it:

It will be called whenever our canvas has to flush its rendering pipeline. In this example, two ways of observing that message which is printed in the cited callback are:

- to resize the example's window (thus resizing the canvas' viewport)
- let the animation run

When one resizes the canvas, there's at least one operation it has to do which will require new calculation for rendering: the resizing of the background rectangle, in a callback we already shown you.

The creation of our background rectangle is so that we give it a **name**, via `evas_object_name_set()` and we give it the canvas **focus**:

Still exemplifying events and callbacks, we register a callback on the canvas event of an object being focused:

In that call, `event_info` is going to be the focused object's handle, in this case our background rectangle. We print its name, so you can check it's the same. We check that pointer is the same reported by Evas' API with regard to the newest focused object. Finally, we check whether that object is really flagged as focused, now using an Evas object API function.

The animation we talked about comes from a timer we register just before we start the example's main loop. As we said, the resizing of the image will also force the canvas to repaint itself, thus flushing the rendering pipeline whenever the timer ticks:

When you start this example, this animation will be running, by default. To interact with the program, there's a command line interface. A help string can be asked for with the 'h' key:

These are the commands the example will accept at any time, except when one triggers the 'f' one. This command will exemplify `evas_event_freeze()`, which interrupts **all** input events processing for the canvas (in the example, just for 3 seconds). Try to issue events for it during that freeze time:

The 'd' command will unregister those two canvas callbacks for you, so you won't see the messages about the focused object and the rendering process anymore:

In this example, we start using a focused object to handle the input events – the background rectangle. We register a callback on an key input event occurring on it, so that we can act on each key stroke:

We do so by examining the `ev->key` string (remember the event information struct for key down events is the `#Evas_Event_Key_Down` one). There's one more trick for grabbing input events on this example – `evas_object_key_grab()`. The 'c' command will, when firstly used, **unfocus** the background rectangle. Unfocused objects on an Evas canvas will **never** receive key events. We grab, then, the keys we're interested at to the object forcefully:

This shows how one can handle input not depending on focus issues – you can grab them globally. Switch back and forth focus and forced key grabbing with the 'c' key, and observe the messages printed about the focused object. Observe, also, that we register two more **object** callbacks, this time on the image object (Enlightenment logo), where we just print messages telling the mouse pointer has entered or exited it area:

Experiment with moving the mouse pointer over the image, letting it enter and exit its area (stop the animation with 'a', for a better experience). When you start the example, Evas will consider this area by being the whole boundary rectangle around the picture. If you issue the 'p' command, though, you get a demonstration of Evas' precise point collision detection on objects. With `evas_object_precise_is_inside_get()`, one can make Evas consider the transparent areas of an object (the middle of the logo's E letter, in the case) as not belonging to it when calculating mouse in/out/up/down events:

To finish the example, try the command bound to Control + 'o', which exemplifies Evas' **obscured regions**. When firstly pressed, you'll get the same contents, in a region in the middle of the canvas, at the time the key was pressed, until you toggle the effect off again (make sure the animation is running on to get the idea better). When you toggle this effect off, we also demonstrate the use of `evas_render_updates()`, which will force immediate updates on the canvas rendering, bringing back the obscured region's contents to normal.

What follows is the complete code for this example.

Chapter 107

Evas objects basic manipulation example

Chapter 108

Evas aspect hints example

Chapter 109

Evas alignment, minimum size, maximum size, padding and weight hints example

In this code, we place a (vertical) box with two rectangles as child elements. It has a command line interface with which to act on those rectangles' **size hints**:

That should be self explanatory. Change those values (possibly resizing the box, which will resize together with the example's window) to get how size hints are honored by a container object, which in this case is the Evas box.

More on this smart object can be found on [Evas box example](#). The full code for this example follows.

Chapter 110

Evas box example

In this example, we demonstrate the use of Evas box objects. We cover changing boxes' layouts (with a custom layout, besides the ones provided by Evas), box padding and alignment influence on the layouts, insertion and removal of box items.

The interesting part of the code starts, naturally, when we add a box object to the canvas. Just after it, we place five rectangles, with random colors, inside of it. Those rectangles get a minimum size hint of 50 pixels on each axis, which will be respected by most of the box's possible layouts:

Just like in other Evas examples, we have a white background on the canvas and a red border around the container object of interest, the box, to mark its boundaries. Resizing of the canvas will keep the box's proportion with regard to the whole canvas', so that you can experiment with different sizes of the box to accommodate its children:

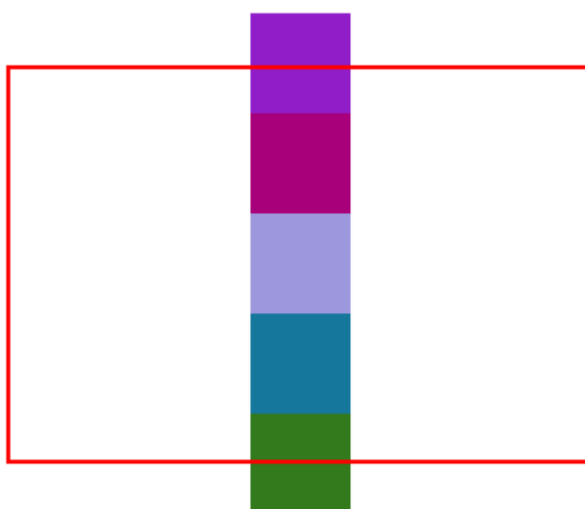
Again, one interacts with this program by means of key commands:

Let's start with the **numeric** ones, each of which will impose a different layout on the box object.

The initial layout the box starts at is the one triggered by the key '1' – the horizontal layout. Thus, the initial appearance of this program, demonstrating this layout, is something like:



The vertical layout (' 2 ' key) is very similar, but just disposing the items vertically:

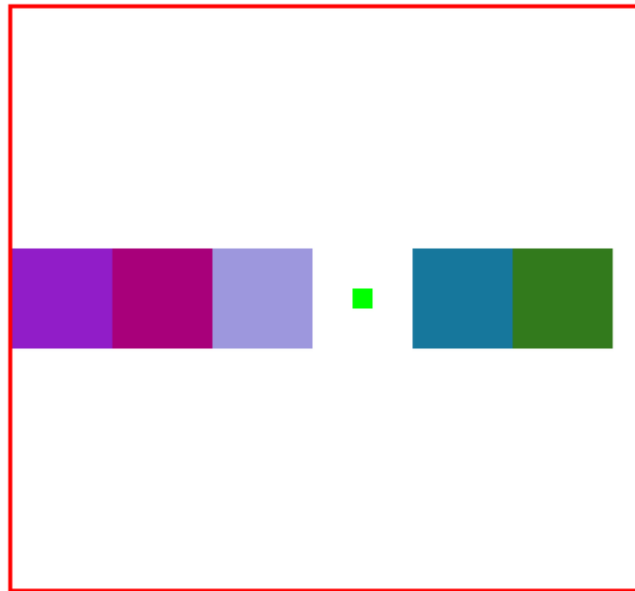


Note the influence of the (default) 0.5 box alignment property, which will let the children line in the middle of the box's area. Also, because the space required by them extrapolates the box's height (we resized it to be smaller), they'll be drawn out of its bounds.

Next, comes the horizontal **homogeneous** layout ('3' key). See how it reserves an equal amount of space for each child to take:



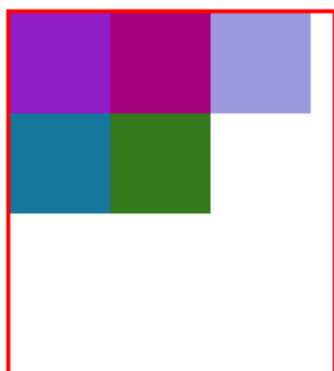
Its vertical equivalent can be triggered by the '4' key. The next different layout of interest is the horizontal maximum size homogeneous ('5' key). It will reserve cells to children sized equally to the dimensions of the child with bigger size (or minimum size hints). For this example, all cells would be just the size of our rectangles' minimum size hints and, to prove that, insert a new (smaller) rectangle at position 3, say, with `Ctrl` and 3 keys together:



The code for the commands inserting and deleting box items is:

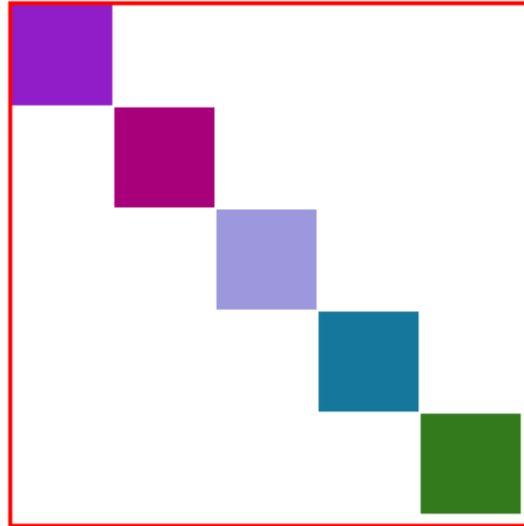
In that code, we exemplify `evas_object_box_children_get()`, to fetch a child element at an exact position. After the element removal from the box (leaving it unparented again), we delete it and free that list. The code inserting a new rectangle, there, is straightforward.

Try the '6' key for the vertical equivalent of the last shown layout. Then, comes the **flow** layout, triggered by the '7' key. We make our box small to demonstrate the effect on the items layouting:



The next two numerical commands are for the vertical equivalent of the last and the stack one, respectively. Try them out to get their looks.

The last numerical key, '0', shows the effect of a **custom** layout on the box. We wrote one that would split the width and height of the box equally and, then, place the items in the cells in the diagonal:



Finally, the `'a'` and `'p'` commands will change the box's alignment and padding property values, respectively. For each of the layouts above, see the effects they make by setting different values on those properties.

The full code for this example follows. For an exercise on **the effect of children box elements' size hints on a box layout**, try the [Evas alignment, minimum size, maximum size, padding and weight hints example](#).

Chapter 111

Evas object stacking functions (and some event handling)

In this example, we illustrate how to stack objects in a custom manner and how to deal with layers.

We have three objects of interest in it – white background, red rectangle, green rectangle and blue rectangle.

Like in other Evas examples, one interacts with it by means of key commands:

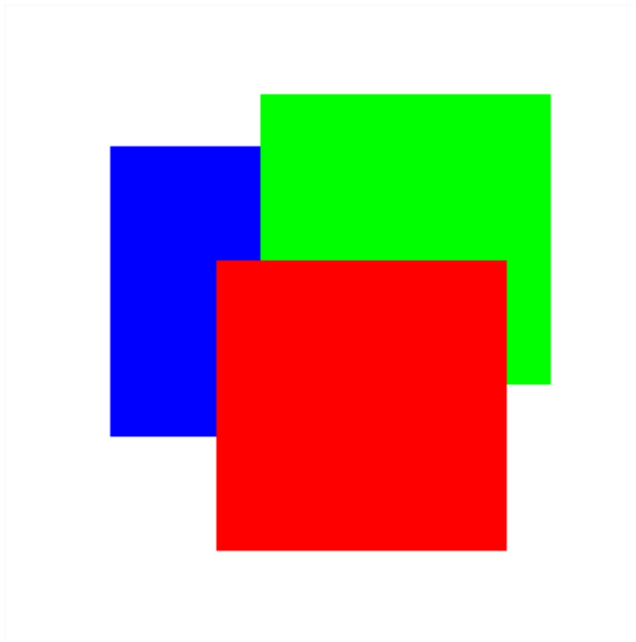
At any given point, like seem above, you'll be operating one rectangle only. You may stacking it below an adjacent object with "b":

"a" will do the opposite:

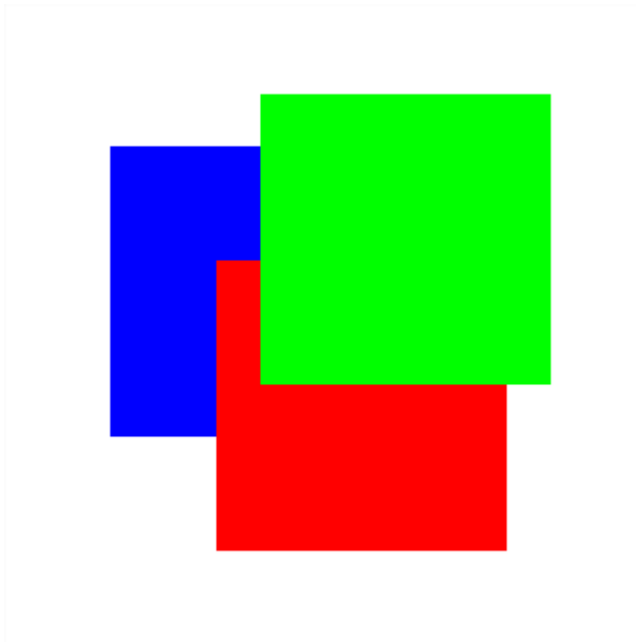
To bring it directly to the top/bottom, use "t"/"m", respectively:

At any time, use the "s" command to see the status of the ordering. It will show the background's ordering too. Note that it also shows the **layer** for this object. It starts at a **different** layer than the others. Use "l" to change its layer (higher layer numbers mean higher layers). If the background is on the same layer as the others (0), you'll see it interact with them on the ordering. If it's in the layer above, no matter what you do, you'll see nothing but the white rectangle: it covers the other layers. For the initial layer (-1), it will never mess nor occlude the others.

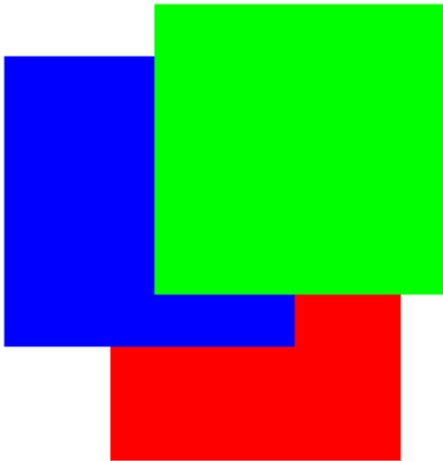
Let's make some tests with those commands. The rectangle which starts selected and which will receive our commands is the **red** one. It starts stacked above all the others, like seem above:



Stack it one level below, with 'b', and you'll get:

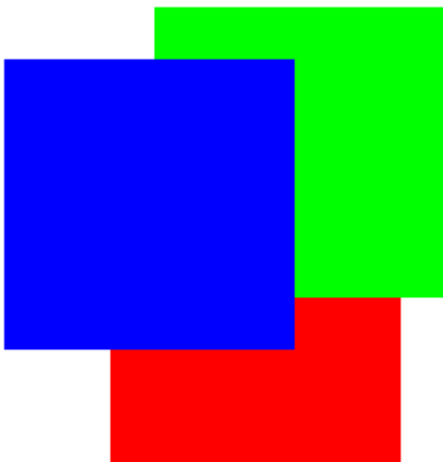


Note how the rectangle which laid above it, the green one, is now on top of it. Now change the rectangle to operate on to the blue one, with two consecutive 'c' commands. Note that it's the lowest one on the stack of rectangles. Issue the 'a' command for it, thus re-stacking it one level above:

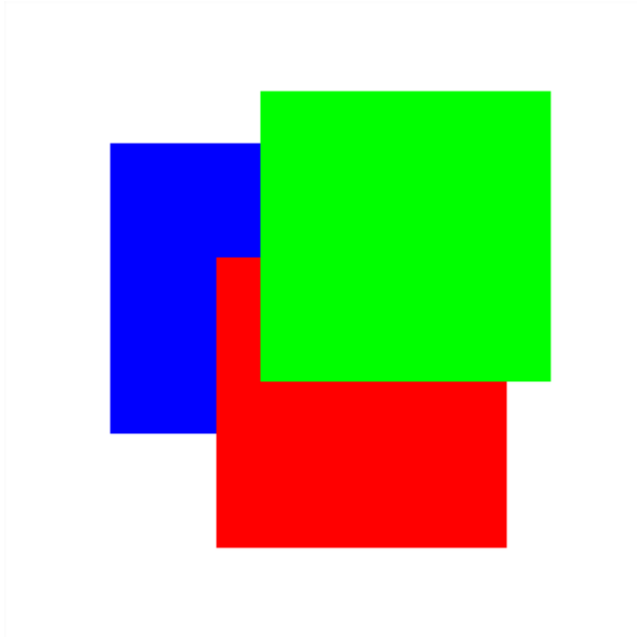


't' command:

You can send it to the top of its layer directly with the



Now put it back to the bottom of that layer with 'm':



Like said above, we have two layers used at the beginning of the example: the default one (0) and the one immediately below it (-1), for the white background. Let's change this setup by issuing the 'l' command, which will change the background's layer to 1, i.e., a layer **above** the one holding the other rectangles:

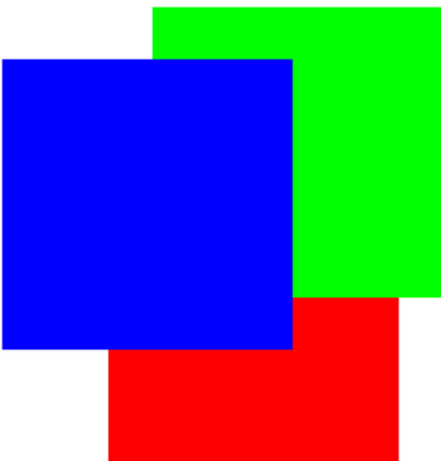


See how it now covers everything else. Press 'l' again, taking it now to layer 0. It's still covering everything because it lands the layer as the highest one on the objects stack. As we have the blue rectangle as the one receiving stacking commands, hit 't' and you'll see it again:



get:

By bringing the background back to layer -1 ('l'), you'll



The last two commands available are "p" and "r", which will make the target rectangle to **pass** (ignore) and **repeat** the mouse events occurring on it (the commands will cycle through on and off states). This is demonstrated with the following #EVAS_CALLBACK_MOUSE_DOWN callback, registered on each of the colored rectangles:

Try to change these properties on the three rectangles while experimenting with mouse clicks on their intersection region.

The full example follows.

Chapter 112

Evas Map - Overview

Down to the very bottom, Map is simple: it takes an object and transforms the way it will be shown on screen. But using it properly can be a bit troublesome.

For the most common operations there are utility functions that help in setting up the map to achieve the desired effects. Now we'll go through an overview of the map API and some of the things that can be done with it.

The full code can be found [here](#).

To show how some functions work, this example listens to keys pressed to toggle several options.

In this program, we divide the window in four quadrants, each holding an object that will have different map configurations applied to them in each call to an animator function.

Let's first create a map and set some of our options to it. Only four points maps are supported, so we'll stick to that magic number. We can set a color for each vertex or apply one for all of them at once

For the first object, we'll have a plain rectangle. At its creation, this rectangle was set to be semi-transparent, but whether its own alpha is used will be defined by the map's alpha setting. If the map's alpha is disabled, then the object will be completely opaque. The map's own color, however, will use any alpha set to it.

So we get our object, initialize our map geometry to match the rectangle and make it rotate around its own center, then apply the map to the object so it takes effect.

The second object is an image. Here we don't have any color set for the object, but the image itself contains an alpha channel that will not be affected by the map settings, so even with alpha set to be off, the image will still be transparent. Color applied to the map will tint it though. Since setting a map copies it into the object, we can reuse the same one we created before. We initialize it to the new object while all other options are kept the same. Notice that no rotation will be done here, as that's just an utility function that takes the coordinates set for each point of the map and transforms it accordingly.

This time the object is a bit farther into the screen, by using a *z* value higher than 0 to init the map. We also need to map the image used by the object, so Evas knows how to transform it properly. For this we use the `evas_map_point_image_uv_set()` to tell the map what coordinate within the image corresponds to each point of the map.

This object will also be rotated, but in all three axis and around some other point, not its center, chosen mostly at random. If enabled, lighting will be applied to, from a light source at the center of the window.

For the third object we are doing, once more, a 3D rotation, but this time perspective will be applied to our map to make it look more realistic. The lighting source also follows the mouse cursor and it's possible to toggle backface culling, so that the object is hidden whenever we are not seeing its front face.

And we free this map, since since we messed too much with it and for the last object we want something cleaner.

The last object is actually two. One image, with an image set to it, and one image proxying the first one with `evas_object_image_source_set()`. This way, the second object will show whatever content its source has. This time we'll be using a map more manually to simulate a simple reflection of the original image.

We know that the reflection object is placed just like the original, so we take a shortcut by just getting the geometry of our to-be-mapped object. We also need to get the image size of the source.

For this we'll create a map shaped so that it begins at the base of our image and it expands horizontally as it grows (downwards) in height.

Since the reflection should show the image inverted, we need to map it this way. The first point of the map (top-left) will be mapped to the mapped to the first pixel of the last row. There's no horizontal reflection and we want the full width of the image, but as we map its upper side `ww` will only take two thirds of the image.

Finally, to fade out our reflection we set the colors for each point in the map. The two at the top need to be visible, but we'll tone them down a bit and make them a bit translucent. The other two will go straight to full transparency. Evas interpolates the colors from one point to the next, so this will make them fade out.

Close up by freeing the map and do some other things needed to keep stuff moving in our animations and we are done.

The rest of the program is setup and listening to key events. Nothing that matters within the scope of this example, so we are going to skip it. Refer to it [here](#) however to see how everything fits together.

Chapter 113

Evas object smart objects

In this example, we illustrate how to create and handle Evas smart objects.

A smart object is one that provides custom functions to handle clipping, hiding, moving, resizing, color setting and more on **child** elements, automatically, for the smart object's user. They could be as simple as a group of objects that move together (see `Evas_Smart_Object_Clipped`) or implementations of whole complex UI widgets, providing some intelligence (thus the name) and extension to simple Evas objects.

Here, we create one as an example. What it does is to control (at maximum) 2 child objects, with regard to their geometries and colors. There can be a "left" child and a "right" one. The former will always occupy the top left quadrant of the smart object's area, while the latter will occupy the bottom right. The smart object will also contain an **internal** decorative border object, which will also be controlled by it, naturally.

Here is where we add it to the canvas:

The magic starts to happen in the `evas_smart_example_add()` function, which is one in the example smart object's defined **interface**. These should be the functions you would export to the users of your smart object. We made three for this one:

- `evas_smart_example_add()`: add a new instance of the example smart object to a canvas
- `evas_smart_example_remove()`: remove a given child of the smart object from it
- `evas_smart_example_set_left()`: set the left child of the smart object
- `evas_smart_example_set_right()`: set the right child of the smart object

The object's creation takes place as:

Smart objects are defined by **smart classes**, which are structs defining their interfaces, or **smart functions** (see `#Evas_Smart_Class`, the base class for any smart object). As you see, one has to use the `evas_object_smart_add()` function to instantiate smart objects. Its second parameter is what matters – an `#Evas_Smart` struct, which contains all the smart class definitions (smart functions, smart callbacks, and the like). Note, however, that `_evas_smart_example_smart_class_new()` seems not to be defined in our example's code. That's because it came from a very handy **helper macro**:

What it does is to **subclass** a given existing smart class, thus specializing it. This is very common and useful in Evas. There is a built-in smart object, the "clipped smart object", which implements a behavior mostly desired by many other smart object implementors: it will clip its children to its area and move them along with it, on `evas_object_move()` calls. Then, our example smart object will get that behavior for free.

The first argument to the macro,

will define the new smart class' name. The second tells the macro what is the **prefix** of the function it will be declaring with a `_smart_set_user()` suffix. On this function, we may override/extend any desired method from our parent smart class:

The first function pointer's code will take place at an example smart object's **creation** time:

The `#EVAS_SMART_DATA_ALLOC` macro will take care of allocating our smart object data, which will be available on other contexts for us (mainly in our interface functions):

See that, as we're inheriting from the clipped smart object's class, we **must** have their data struct as our first member. Other data of interest for us is a child members array and the border object's handle. The latter is what is created in the last mentioned function. Note how to tell Evas the border will be managed by our smart object from that time on: `evas_object_smart_member_add(priv->border, o);`. The counterpart of this function is exemplified on the smart object's interface function to remove children:

At the end of that function we make use of an constant defined by the `#EVAS_SMART_SUBCLASS_NEW`: `__evas_smart_example_parent_sc`. It has the same prefix we passed to the macro, as you can see, and it holds a pointer to our **parent** smart class. Then, we can call the specialized method, itself, after our code. The `del`, `hide`, `show` and `resize` specializations are straightforward, we let the reader take a look at them below to check their behavior. What's interesting is the `calculate` one:

This code will take place whenever the smart object itself is flagged "dirty", i.e., must be recalculated for rendering (that could come from changes on its clipper, resizing, moving, etc). There, we make sure the decorative border lies on the edges of the smart object and the children, if any, lie on their respective quadrants.

After instantiating our smart object, we do some checks to exemplify some of the API on smart objects:

The `evas_object_smart_type_check()` one will assure we have the string naming our smart class really set to the live object. The `evas_object_smart_clipped_clipper_get()` exemplifies usage of "static clippers" – clipped smart objects have their global clippers flagged static.

Other important things we also exemplify here are **smart callbacks** and smart callback **introspection**:

Here we declare our array of smart callback descriptions, which has one element only, in this case. That callback will take place, as the name indicates, whenever the number of member objects in our smart object example instance changes. That global array variable must be the last argument to `#EVAS_SMART_SUBCLASS_NEW`, so that it's registered as the **smart class's** callbacks description.

After we instantiate the smart object, we take a look on those descriptions and register a callback on that unique smart event:

The code of the callback will just print how many member objects we have, which is an integer argument of the callback itself, as flagged by its description:

One of the points at which we issue that callback is inside the `evas_smart_example_remove()`, code that was already shown.

As in other examples, to interact with this one there's a command line interface. A help string can be asked for with the 'h' key:

Use 'l' and 'r' keys, to create new rectangles and place them on the left (`evas_smart_example_set_left()`) or right (`evas_smart_example_set_right()`) spots of our smart object, respectively. The 'w' command will remove all member objects from the smart object and delete them. The keyboard arrows will move the smart object along the canvas. See how it takes any child objects with it during its movement. The 'd' and 'i' keys will increase or decrease the smart object's size – see how it affects the children's sizes, too. Finally, 'c' will change the color of the smart object's clipper (which is the exact internal clipper coming from a clipped smart object):

"Real life" examples of smart objects are Edge and Emotion objects: they both have independent libraries implementing their behavior. The full example follows.

Chapter 114

Evas object smart interfaces

In this example, we illustrate how to create and handle Evas smart **interfaces**. Note that we use the same code base of the [Evas object smart objects](#) example, here. We just augment it with an interfaces demonstration.

A smart interface is just a functions interface a given smart object is declaring to support and or use. In Evas, interfaces are very simple: no interface inheritance, no interface overriding. Their purpose is to extend an object's capabilities and behavior beyond the sub-classing schema.

Here, together with a custom smart object, we create and declare the object as using an Evas interface. It'll have a custom function, too, besides the `add()` and `del()` obligatory ones. To demonstrate interface data, which is bound to object **instances**, we'll have a string as this data.

Here is where we declare our interface:

Note that there's error checking for interfaces creation, by means of the `add()` method's return value (`_iface1_add()`, here).

Now note that here we are filling in the interface's fields dynamically. Let's move on to that code region:

As important as setting the function pointers, is declaring the `private_size` as to match exactly the size of the data blob we want to have allocated for us by Evas. This will happen automatically inside `evas_smart_example_add()`. Later, on this code, we deal exactly with that data blob, more specifically writing on it (as it's not done inside `_iface1_add()`, here:

Before accessing the interface data, we exercise the interface fetching call `evas_object_smart_interface_get()`, with the name string we used to be interface's name. With that handle in hands, we issue `evas_object_smart_interface_data_get()` and write the string we want as data on that memory region. That will make up for the string you get on `_iface1_del()`.

The full example follows.

Chapter 115

Evas text object example

In this example, we illustrate how to use text objects in various manners.

We place, in the canvas, a text object along with a border image to delimit its geometry. After we instantiate the text object, we set lots of properties on it to the initial ones from a preset list, which has the following declaration:

Then, we set the text string itself, on it, with `evas_object_text_text_set()`. We set an explicit size of 30 points for our font, as you could see, what we check back with the getter `evas_object_text_font_get()`.

Look at how it translates to code:

Like in other Evas examples, one interacts with it by means of key commands:

Use the 't' key to exercise the `evas_object_text_style_set()` function on the text – it will cycle through all styles on `#Evas_Text_Style_Type` (note we start on `#EVAS_TEXT_STYLE_PLAIN`, thus with no effects on it) and, with other keys, you'll be able to set properties applicable to individual styles on the text object.

The 'z' key will change the text's **size**, keeping the font family for it. Use 'f' to change the font, keeping the last size set. There are three font families the example will cycle through:

The 'b' command shows us that `evas_object_color_set()`, on a given text object, will change the text's **base** color. Experiment with it, which will cycle through the colors in the `.text` list in `init_data`.

The 's', 'o', 'w' and 'g' keys will make the text object to cycle to the preset values on colors for shadow, outline, glow and 'glow 2' effects, respectively. Naturally, they will only take effect on the text styles which resemble them.

The full example follows.

Chapter 116

Table Smart Object example

This example will arrange rectangles of different sizes (and colors) in a table. While it's possible to create the same layout we are doing here by positioning each rectangle independently, using a table makes it a lot easier, since the table will control layout of all the objects, allowing you to move, resize or hide the entire table.

We'll start with creating the table, setting it to `EVAS_OBJECT_TABLE_HOMOGENEOUS_NONE` to have maximum flexibility and setting its padding to 0:

We then create each rectangle and add it to the table:

Note

Each rectangle has a different minimum size based on how many rows and columns it will occupy.

The full source for this example follows:

Chapter 117

Authors

Author

Adam Simpkins <adam@adamsimpkins.net>
Aharon Hillel <a.hillel@samsung.com>
Albin "Lutin" Tonnerre <albin.tonnerre@gmail.com>
Andre Dieb <andre.dieb@gmail.com>
Andrew Elcock <andy@elcock.org>
Arnaud de Turckheim 'quarium' <quarium@gmail.com>
Bernhard Nemec <Bernhard.Nemec@viasyshc.com>
billiob (Boris Faure) <billiob@gmail.com>
Bluezery <ohpowel@gmail.com>
Boris "billiob" Faure <billiob@gmail.com>
Brett Nash <nash@nash.id.au>
Brian 'rephorm' Mattern <rephorm@rephorm.com>
Brian Mattern <rephorm@rephorm.com>
Bruno Dilly <bdilly@profusion.mobi>
Burra <burra@colorado.edu>
Carsten Haitzler <raster@rasterman.com>
Cedric BAIL <cedric.bail@free.fr>
Chidambar Zinnoury <illogict@online.fr>
Chris Ross <chris@darkrock.co.uk>
Christophe Dumez <christophe.dumez@intel.com>
Christopher 'devilhorns' Michael <cpmichael1@comcast.net>
ChunEon Park (Hermet) <hermet@hermet.pe.kr>
Corey "atmos" Donohoe <atmos@atmos.org>
dan sinclair <zero@everburning.com>
Daniel Juyung Seo <seojuyung2@gmail.com> <juyung.seo@samsung.com>
Daniel Willmann <d.willmann@samsung.com>
Daniel Zaoui <daniel.zaoui@yahoo.com>
David 'onefang' Seikel <onefang@gmail.com>
David Goodlad <dgoodlad@gmail.com>
Davide Andreoli <dave@gurumeditation.it>
Doyoun Kang <wayofmine@gmail.com> <doyoun.kang@samsung.com>
Fabiano Fidêncio <fidencio@profusion.mobi>
Flavio Ceolin <flavio.ceolin@profusion.mobi>
Govindaraju SM <govi.sm@samsung.com> <govism@gmail.com>
Guilherme Silveira <xguiga@gmail.com>
Guillaume Friloux <guillaume.friloux@asp64.com>
Gustavo Chaves <glima@profusion.mobi>
Gustavo Lima Chaves <glima@profusion.mobi>
Gustavo Sverzut Barbieri <barbieri@profusion.mobi>
Gwanglim Lee <gl77.lee@samsung.com> <gwanglim@gmail.com>
Haifeng Deng <haifeng.deng@samsung.com>

Hisham 'CodeWarrior' Mardam Bey <hisham@hisham.cc>
 Howell Tam <pigeon@pigeond.net>
 Hugo Camboulive <hugo.camboulive@zodiacaerospace.com>
 Hyoyoung Chang <hyoyoung@gmail.com>
 Ibukun Olumuyiwa <ibukun@computer.org>
 Iván Briano <ivan@profusion.mobi>
 Jaehwan Kim <jae.hwan.kim@samsung.com>
 Jihoon Kim <jihoon48.kim@samsung.com> <imfine98@gmail.com>
 Jorge Luis Zapata Muga <jorgeluis.zapata@gmail.com>
 Jose O Gonzalez <jose_ogp@juno.com>
 José Roberto de Souza <zehortigoza@profusion.mobi>
 Jérémy Zurcher <jeremy@asynk.ch>
 Jérôme Pinot <ngc891@gmail.com>
 Kim Shinwoo <kimcinoo.efl@gmail.com>
 Kim Woelders <kim@woelders.dk>
 Kim Yunhan <spbear@gmail.com>
 Lars Munch <lars@segv.dk>
 Leandro Dorileo <dorileo@profusion.mobi>
 Leandro Pereira <leandro@profusion.mobi>
 Leif Middelschulte <leif.middelschulte@gmail.com>
 Lucas De Marchi <lucas.demarchi@profusion.mobi>
 Luis Felipe Strano Moraes <lfelipe@profusion.mobi>
 Mathieu Taillefumier <mathieu.taillefumier@free.fr>
 Matt Barclay <mbarclay@gmail.com>
 Michael 'Mickey' Lauer <mickey@tm.informatik.uni-frankfurt.de>
 Michael Bouchaud (yoz) <michael.bouchaud@gmail.com>
 Mikael Sans <sans.mikael@gmail.com>
 Mike Blumenkrantz <michael.blumenkrantz@gmail.com>
 Mike McCormack <mj.mccormack@samsung.com>
 Myoungwoon Roy Kim (roy_kim) <myoungwoon.kim@samsung.com> <myoungwoon@gmail.com>
 Myungjae Lee <mjae.lee@samsung.com>
 Nathan 'RbdPngn' Ingersoll <ningerso@d.umn.edu>
 Nicholas 'Mekius' Hughart
 Nicholas Curran <quasar@bigblue.net.au>
 Nicolas Aguirre <aguirre.nicolas@gmail.com>
 Peter Wehrfritz <peter.wehrfritz@web.de>
 Pierre Le Magourou <pierre.lemagourou@openwide.fr>
 PnB <Poor.NewBie@gmail.com>
 Prince Kumar Dubey <prince.dubey@samsung.com> <prince.dubey@gmail.com>
 Rafael Antognolli <antognolli@profusion.mobi>
 Rafal Krypa <r.krypa@samsung.com>
 Rajeev Ranjan (Rajeev) <rajeev.r@samsung.com> <rajeev.jnnce@gmail.com>
 Raphael Kubo da Costa <kubo@profusion.mobi>
 Ricardo de Almeida Gonzaga <ricardo@profusion.mobi>
 Robert David <robert.david.public@gmail.com>
 Rui Miguel Silva Seabra <rms@1407.org>
 Samsung Electronics
 Samsung SAIT
 Sangho Park <gouache95@gmail.com>
 Sebastian Dransfeld <sd@tango.flipp.net>
 Seungsoo Woo <om101.woo@samsung.com>
 Shilpa Singh <shilpa.singh@samsung.com> <shilpasingh.o@gmail.com>
 Simon Poole <simon.armlinux@themalago.net>
 Sohyun Kim <anna1014.kim@samsung.com>
 Steve Ireland <sireland@pobox.com>
 Sung W. Park <sungwoo@gmail.com>
 Term <term@twistedpath.org>
 Thierry el Borgi <thierry@substantiel.fr>

Tiago Falcão <tiago@profusion.mobi>
Till Adam <till@adam-lilienthal.de>
Tilman Sauerbeck <tilman@code-monkey.de>
Tim Horton <hortont424@gmail.com>
Tom Gilbert <tom@linuxbrit.co.uk>
Tom Hacoen <tom@stosb.com>
Tristan <blunderer@gmail.com>
Vikram Narayanan <vikram186@gmail.com>
Vincent Torri <vincent.torri@gmail.com>
Willem Monsuwe <willem@stack.nl>
WooHyun Jung (woohyun) <woohyun0705@gmail.com>
xlopez@igalia.com
Youness Alaoui <kakaroto@kakaroto.homelinux.net>
Yuri <da2001@hotmail.ru>
Yuri Hudobin <glassy_ape@users.sourceforge.net>
ZigsMcKenzie <zigsmckenzie@gmail.com>

Please contact <enlightenment-devel@lists.sourceforge.net> to get in contact with the developers and maintainers.

Chapter 118

pkgconfig

118.1 Introduction

pkg-config (<http://pkgconfig.freedesktop.org/wiki/>) is a helper tool used when compiling applications and libraries. It helps you insert the correct compiler options on the command line based on installed software, instead of hard-coded values.

118.2 Usage

Using pkg-config it is as simple as:

```
# compile:
gcc -c -o main.o main.c `pkg-config --cflags PKGNAME`

# link:
gcc -o my_application main.o `pkg-config --libs PKGNAME`

# compile + link in a single step:
gcc -o my_application main.c `pkg-config --cflags --libs PKGNAME`
```

Where **PKGNAME** is your module, such as eina, eet, evas, ecore, ecore-x, eio and so on.

One can do some queries such as the module version, other variables:

```
pkg-config --modversion PKGNAME
pkg-config --variable=prefix PKGNAME
```

118.3 Troubleshooting

Make sure pkg-config command is in your \$PATH, otherwise you'll end with:

```
pkg-config: command not found
```

The **PKGNAME** it searched using pkg-config's build location, usually /usr/lib/pkgconfig. This can be overwritten with \$PKG_CONFIG_LIBDIR (usually for cross compile) or extended with \$PKG_CONFIG_PATH. If you installed EFL to /opt/efl, then use:

```
export PKG_CONFIG_PATH="$PKG_CONFIG_PATH:/opt/efl/lib/pkgconfig"
pkg-config --cflags --libs PKGNAME
```

Otherwise you'll end with:

```
Package PKGNAME was not found in the pkg-config search path.  
Perhaps you should add the directory containing 'PKGNAME.pc'  
to the PKG_CONFIG_PATH environment variable  
No package 'PKGNAME' found
```


Chapter 119

Module Index

119.1 Modules

Here is a list of all modules:

Eo	277
Evas	278
Eet	279
Eina	280
Embryo	281
Evil	282
Escape	283
Ecore	284
Eio	285
Eldbus	286
Efreet	287
Eeze	288
Edje	289
Emotion	290
Ethumb	291

Chapter 120

Module Documentation

120.1 Eo

Generic object system.

Generic object system.

120.2 Evas

Drawing canvas.

Drawing canvas.

120.3 Eet

Binary data parser and serializer.

Binary data parser and serializer.

120.4 Eina

Data types and low-level/basic abstractions.

Data types and low-level/basic abstractions.

120.5 Embryo

Embedded script language.

Embedded script language.

120.6 Evil

Microsoft Windows portability layer.

Microsoft Windows portability layer.

120.7 Escape

PlayStation3 portability layer.

PlayStation3 portability layer.

120.8 Ecore

Operating System Abstraction and Integration.

Operating System Abstraction and Integration.

120.9 Eio

Asynchronous input/output and file manipulation.

Asynchronous input/output and file manipulation.

120.10 Eldbus

D-Bus integration with EFL (Ecore).

D-Bus integration with EFL (Ecore).

120.11 Efreet

FreeDesktop.Org (XDG) menu and desktop integration.

FreeDesktop.Org (XDG) menu and desktop integration.

120.12 Eeze

Hardware device manipulation and notification, wraps UDev and similar.

Hardware device manipulation and notification, wraps UDev and similar.

120.13 Edje

Layout and theme library with super-powers.

Layout and theme library with super-powers.

120.14 Emotion

Plays music and videos.

Plays music and videos.

120.15 Ethumb

Generates thumbnail images of files.

Generates thumbnail images of files.

Chapter 121

Example Documentation

121.1 banshee.c

Access Banshee music player and send commands to it.

121.2 client.c

Client to test various call message types against a provided server ([server.c](#))

121.3 complex-types-client-eina-value.c

Client to test complex types (arrays, structs, dicts) against a provided server ([complex-types-server.c](#)) returning them as Eina_Value.

121.4 complex-types-server.c

Server to test complex types (arrays, structs, dicts).

121.5 complex-types.c

Client to test complex types (arrays, structs, dicts) against a provided server ([complex-types-server.c](#))

121.6 connman-list-services.c

Client to list networks/services from connman.

121.7 `ecore_animator_example.c`

121.8 `ecore_con_client_simple_example.c`

Shows how to setup a simple client that connects to a server and sends a "hello" string to it. See the complete example description at [Ecore_Con - Creating a client](#)

121.9 `ecore_con_lookup_example.c`

Shows how to make a simple DNS lookup. See the complete example description at [Ecore_Con - DNS lookup](#)

121.10 `ecore_con_server_simple_example.c`

Shows how to setup a simple server that accepts client connections and sends a "hello" string to them. See the complete example description at [Ecore_Con - Creating a server](#)

121.11 `ecore_con_url_cookies_example.c`

Shows how to manage cookies on a `Ecore_Con_Url` object. See the complete example description at [Ecore_Con_Url - Managing cookies](#).

121.12 `ecore_con_url_download_example.c`

Shows how to download a file using an `Ecore_Con_Url` object. See the complete example description at [Ecore_Con_Url - downloading a file](#)

121.13 `ecore_con_url_headers_example.c`

Shows how to make GET or POST requests using an `Ecore_Con_Url` object, and make use of most of its API. See the complete example description at [Ecore_Con_Url - customizing a request](#)

121.14 `ecore_evas_basics_example.c`

121.15 ecore_evas_buffer_example_01.c**121.16 ecore_evas_buffer_example_02.c****121.17 ecore_evas_callbacks.c****121.18 ecore_evas_object_example.c****121.19 ecore_evas_window_sizes_example.c****121.20 ecore_event_example_01.c**

This example shows how to create an event handler. Explanation: [Handling events example](#)

121.21 ecore_event_example_02.c

This example shows how to setup, change, and delete event handlers. See [the explanation here](#).

121.22 ecore_exe_example.c

This is a process that will send messages to a child and it will stop when it receives "quit". Check the [Full tutorial](#)

121.23 ecore_exe_example_child.c

This is a child process used to receive messages and send it back to its father. Check the [Full tutorial](#)

121.24 ecore_fd_handler_example.c

This example shows how to setup and use an fd_handler. See [the explanation here](#).

121.25 ecore_fd_handler_gnutls_example.c

Shows how to use fd handlers.

121.26 ecore_idler_example.c

This example shows when Ecore_Idler, Ecore_Idle_Enterer and Ecore_Idle_Exiters are called. See [the explanation here](#).

121.27 ecore_job_example.c

This example shows how to use an Ecore_Job. See [the explanation here](#).

121.28 ecore_pipe_gstreamer_example.c

121.29 ecore_pipe_simple_example.c

121.30 ecore_poller_example.c

This example shows how to setup and use a poller. See [the explanation here](#).

121.31 ecore_thread_example.c

121.32 ecore_time_functions_example.c

Shows the difference between the three time functions. See [the example explained](#).

121.33 ecore_timer_example.c

This example shows how to use timers to have timed events inside ecore. See [the example explained](#).

121.34 edje-basic.c

121.35 edje-box.c

121.36 edje-box2.c

121.37 edje-color-class.c

121.38 edje-drag.c

121.39 edje-perspective.c

121.40 edje-signals-messages.c

121.41 edje-swallow.c

121.42 edje-table.c

121.43 edje-text.c

121.44 eet-basic.c

121.45 eet-data-cipher_decipher.c

121.46 eet-data-file_descriptor_01.c

121.47 eet-data-file_descriptor_02.c

121.48 eet-data-nested.c

121.49 eet-data-simple.c

121.50 eet-file.c

121.51 eina_accessor_01.c

121.52 eina_array_01.c

121.53 eina_array_02.c

121.54 eina_error_01.c

121.55 eina_file_01.c

121.56 eina_hash_01.c

121.57 eina_hash_02.c

121.58 eina_hash_03.c

121.59 eina_hash_04.c

121.60 eina_hash_05.c

121.61 eina_hash_06.c

121.62 eina_hash_07.c

121.63 eina_hash_08.c

121.64 eina_inarray_01.c

121.65 eina_inarray_02.c

121.66 eina_inlist_01.c

121.67 eina_inlist_02.c

121.68 eina_inlist_03.c

121.69 eina_iterator_01.c

121.70 eina_list_01.c

121.71 eina_list_02.c

121.72 eina_list_03.c

121.73 eina_list_04.c

121.74 eina_log_01.c

121.75 eina_log_02.c

121.76 eina_log_03.c

121.77 eina_magic_01.c

121.78 eina_model_01.c

121.79 eina_model_02.c

121.80 eina_model_03.c

121.81 eina_model_04_animal.c

121.82 eina_model_04_child.c

121.83 eina_model_04_human.c

121.84 eina_model_04_main.c

121.85 eina_model_04_parrot.c

121.86 eina_model_04_whistler.c

121.87 eina_simple_xml_parser_01.c

121.88 `eina_str_01.c`

121.89 `eina_strbuf_01.c`

121.90 `eina_stringshare_01.c`

121.91 `eina_tiler_01.c`

121.92 `eina_value_01.c`

121.93 `eina_value_02.c`

121.94 `eina_value_03.c`

121.95 `eio_file_ls.c`

121.96 `emotion_basic_example.c`

This example shows how to create and play an Emotion object. See [the explanation here](#).

121.97 `emotion_signals_example.c`

This example shows that some of the information available from the emotion object, like the media file play length, aspect ratio, etc. can be not available just after setting the file to the emotion object.

One callback is declared for each of the following signals, and some of the info about the file is displayed. Also notice that the order that these signals are emitted can change depending on the module being used. Following is the full source code of this example:

121.98 emotion_test_main.c

This example covers the entire emotion API. Use it as a reference.

121.99 ephysics_logo.c**121.100 evas-aspect-hints.c****121.101 evas-box.c****121.102 evas-buffer-simple.c****121.103 evas-events.c****121.104 evas-hints.c****121.105 evas-images.c****121.106 evas-images2.c****121.107 evas-init-shutdown.c**

121.108 **evas-map-utils.c**

121.109 **evas-object-manipulation.c**

121.110 **evas-smart-interface.c**

121.111 **evas-smart-object.c**

121.112 **evas-stacking.c**

121.113 **evas-table.c**

121.114 **evas-text.c**

121.115 **ofono-dial.c**

Client to ask oFono to dial.

121.116 **server.c**

Server to reply to [client.c](#) requests.

121.117 **simple-signal-emit.c**

Server that registers a service interface and emits simple signals.

121.118 test_bouncing_ball.c

121.119 test_bouncing_text.c

121.120 test_camera.c

121.121 test_camera_track.c

121.122 test_collision_detection.c

121.123 test_collision_filter.c

121.124 test_constraint.c

121.125 test_delete.c

121.126 test_forces.c

121.127 test_growing_balls.c

121.128 test_no_gravity.c

121.129 test_rotating_forever.c

121.130 test_shapes.c

121.131 test_sleeping_threshold.c

121.132 test_slider.c

121.133 test_velocity.c

Index

Ecore, [284](#)
Edge, [289](#)
Eet, [279](#)
Eeze, [288](#)
Efreet, [287](#)
Eina, [280](#)
Eio, [285](#)
Eldbus, [286](#)
Embryo, [281](#)
Emotion, [290](#)
Eo, [277](#)
Escape, [283](#)
Ethumb, [291](#)
Evas, [278](#)
Evil, [282](#)