Upto: Table of Contents of full book "Programming Wayland Clients"

# Chapter 9 EGL

EGL provides a base for graphics programming, common to many systems such as OpenCL, OpenGL, OpenGL ES or OpenVG

skip table of contents

Show table of contents

## 9.1 Resources

- What does EGL do in the Wayland stack by Pekka Paalanen
- eglIntro — introduction to managing client API rendering through the EGL API
- Khronos Native Platform Graphics Interface (EGL Version 1.5)

## 9.2 Overview

From the EGL specification "EGL [is] an interface between rendering APIs such as OpenCL, OpenGL, OpenGL ES or OpenVG (referred to collectively as client APIs) and one or more underlying platforms (typically window systems such as X11)". It is not intended that application programmers write directly to EGL, instead they should use one of the APIs such as OpenGL.

Each underlying platform will have a means of rendering EGL surfaces. In this chapter we need to look at how EGL surfaces are linked to Wayland surfaces and how windows are built. We won't actually draw into any windows, because that is best done using e.g. OpenGL. That will be done in a later chapter.

Why EGL? From the FAQ : " EGL is the only GL binding API that lets us avoid dependencies on existing window systems, in particular X. GLX obviously pulls in X dependencies and only lets us set up GL on X drawables. The alternative is to write a Wayland specific GL binding API, say, WaylandGL. A more subtle point is that libGL.so includes the GLX symbols, so linking to that library will pull in all the X dependencies. This means that we can't link to full GL without pulling in the client side of X, so we're using GLES2 for now. Longer term, we'll need a way to use full GL under Wayland. "

## 9.3 Initialising EGL

EGL has a display that it writes on. The display is built on a native display, and is obtained by the call **eglGetDisplay**. The EGL platform is then initialised using **eglInitialize**.

Typically an EGL display will support a number of configurations. For example, a pixel may be 16 bits (5 red, 5 blue and 6 green), 24 bits (8 red, 8 green and 8 blue) or 32 bits (8 extra bits for alpha transparency). An application will specify certain parameters such as the minimum size of a red pixel, and can then access the array of matching configurations using **eglChooseConfig**. The attributes of a configuration can be queried using **eglGetConfigAttrib**. One configuration should be chosen before proceeding.

Each configuration will support one or more client APIs such as OpenGL. The API is usually requested through the configuration attribute **EGL_RENDERABLE_TYPE** which should have a value such as **EGL_OPENGL_ES2_BIT**.

In addition to a configuration, each application needs one or more *contexts*. Each context defines a level of the API that will be used for rendering. Examples typically use a level of 2, and a context is created using **eglCreateContext**.

Typical code to perform these steps is

```
init_egl() {
   EGLint major, minor, count, n, size;
   EGLConfig *configs;
   int i;
   EGLint config_attribs[] = {
      EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
      EGL_RED_SIZE, 8,
      EGL_GREEN_SIZE, 8,
      EGL_BLUE_SIZE, 8,
      EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
      EGL_NONE
   };

   static const EGLint context_attribs[] = {
      EGL_CONTEXT_CLIENT_VERSION, 2,
      EGL_NONE
   };


   egl_display = eglGetDisplay((EGLNativeDisplayType) display);
   if (egl_display == EGL_NO_DISPLAY) {
      fprintf(stderr, "Can't create egl display\n");
      exit(1);
   } else {
      fprintf(stderr, "Created egl display\n");
   }

   if (eglInitialize(egl_display, &major, &minor) != EGL_TRUE) {
      fprintf(stderr, "Can't initialise egl display\n");
      exit(1);
   }
   printf("EGL major: %d, minor %d\n", major, minor);

   eglGetConfigs(egl_display, NULL, 0, &count);
   printf("EGL has %d configs\n", count);
```

```
    configs = calloc(count, sizeof *configs);

    eglChooseConfig(egl_display, config_attribs,
                    configs, count, &n);

    for (i = 0; i < n; i++) {
      eglGetConfigAttrib(egl_display,
                         configs[i], EGL_BUFFER_SIZE, &size);
      printf("Buffer size for config %d is %d\n", i, size);
      eglGetConfigAttrib(egl_display,
                         configs[i], EGL_RED_SIZE, &size);
      printf("Red size for config %d is %d\n", i, size);

      // just choose the first one
      egl_conf = configs[i];
      break;
    }

    egl_context =
      eglCreateContext(egl_display,
                       egl_conf,
                       EGL_NO_CONTEXT, context_attribs);

}
```

## 9.4 Creating an EGL drawing surface

An EGL window needs to be created from a Wayland surface, by the Wayland call **wl_egl_window_create**. This is then turned into an EGL drawing surface by the EGL call **eglCreateWindowSurface**. Drawing into this surface is generally done by an API such as OpenGL, and the choice of this has been set in the **egl_context**. This is set for the drawing surfaces by **eglMakeCurrent**.

Following this, drawing can take place. For now, we don't draw but leave some simple drawing code in place but commented out.

Once drawing is complete, the EGL layer renders the surface by swapping the drawing buffer using **eglSwapBuffers**.

This code looks like

```
create_window() {

    egl_window = wl_egl_window_create(surface,
                         480, 360);
    if (egl_window == EGL_NO_SURFACE) {
      fprintf(stderr, "Can't create egl window\n");
      exit(1);
    } else {
      fprintf(stderr, "Created egl window\n");
    }

    egl_surface =
      eglCreateWindowSurface(egl_display,
                             egl_conf,
                             egl_window, NULL);

    if (eglMakeCurrent(egl_display, egl_surface,
                       egl_surface, egl_context)) {
      fprintf(stderr, "Made current\n");
    } else {
      fprintf(stderr, "Made current failed\n");
    }

    /*
    glClearColor(1.0, 1.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
    */

    if (eglSwapBuffers(egl_display, egl_surface)) {
      fprintf(stderr, "Swapped buffers\n");
    } else {
      fprintf(stderr, "Swapped buffers failed\n");
    }
}
```

## 9.5 Adding an opaque region

In this instance we aren't drawing anything. So we won't see any windows. For this chapter only we can fix that by adding an opaque region to the surface by

```
create_opaque_region() {
    region = wl_compositor_create_region(compositor);
    wl_region_add(region, 0, 0,
                  480,
                  360);
```

```
    wl_surface_set_opaque_region(surface, region);
}
```

With no drawing into this, it will probably show up as a black rectangle.

## 9.6 Final code

The code at the end of this chapter is

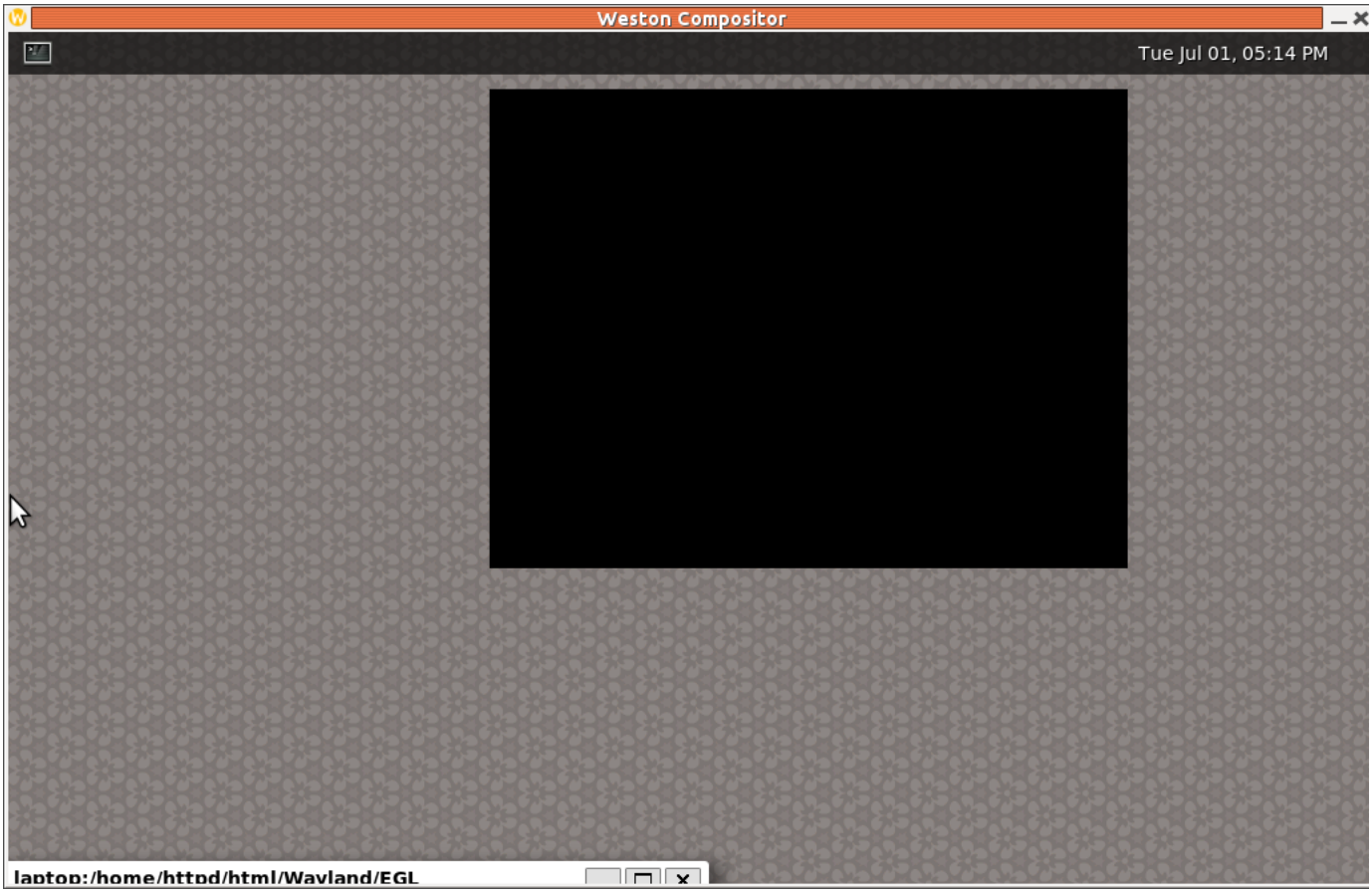
```
    #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wayland-client.h>
#include <wayland-server.h>
#include <wayland-client-protocol.h>
#include <wayland-egl.h>
#include <EGL/egl.h>
#include <GLES2/gl2.h>

struct wl_display *display = NULL;
struct wl_compositor *compositor = NULL;
struct wl_surface *surface;
struct wl_egl_window *egl_window;
struct wl_region *region;
struct wl_shell *shell;
struct wl_shell_surface *shell_surface;

EGLDisplay egl_display;
EGLConfig egl_conf;
EGLSurface egl_surface;
EGLContext egl_context;

static void
global_registry_handler(void *data, struct wl_registry *registry, uint32_t id,
            const char *interface, uint32_t version)
{
    printf("Got a registry event for %s id %d\n", interface, id);
    if (strcmp(interface, "wl_compositor") == 0) {
        compositor = wl_registry_bind(registry,
                                      id,
                                      &wl_compositor_interface,
                                      1);
    } else if (strcmp(interface, "wl_shell") == 0) {
        shell = wl_registry_bind(registry, id,
                                 &wl_shell_interface, 1);

    }
}

static void
global_registry_remover(void *data, struct wl_registry *registry, uint32_t id)
{
    printf("Got a registry losing event for %d\n", id);
}

static const struct wl_registry_listener registry_listener = {
    global_registry_handler,
    global_registry_remover
};

static void
create_opaque_region() {
    region = wl_compositor_create_region(compositor);
    wl_region_add(region, 0, 0,
                  480,
                  360);
    wl_surface_set_opaque_region(surface, region);
}

static void
create_window() {

    egl_window = wl_egl_window_create(surface,
                          480, 360);
    if (egl_window == EGL_NO_SURFACE) {
        fprintf(stderr, "Can't create egl window\n");
        exit(1);
    } else {
        fprintf(stderr, "Created egl window\n");
    }

    egl_surface =
        eglCreateWindowSurface(egl_display,
                               egl_conf,
                               egl_window, NULL);

    if (eglMakeCurrent(egl_display, egl_surface,
                       egl_surface, egl_context)) {
        fprintf(stderr, "Made current\n");
```

```
    } else {
        fprintf(stderr, "Made current failed\n");
    }

    /*
    glClearColor(1.0, 1.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
    */

    if (eglSwapBuffers(egl_display, egl_surface)) {
        fprintf(stderr, "Swapped buffers\n");
    } else {
        fprintf(stderr, "Swapped buffers failed\n");
    }
}

static void
init_egl() {
    EGLint major, minor, count, n, size;
    EGLConfig *configs;
    int i;
    EGLint config_attribs[] = {
        EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
        EGL_RED_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
        EGL_NONE
    };

    static const EGLint context_attribs[] = {
        EGL_CONTEXT_CLIENT_VERSION, 2,
        EGL_NONE
    };


    egl_display = eglGetDisplay((EGLNativeDisplayType) display);
    if (egl_display == EGL_NO_DISPLAY) {
        fprintf(stderr, "Can't create egl display\n");
        exit(1);
    } else {
        fprintf(stderr, "Created egl display\n");
    }

    if (eglInitialize(egl_display, &major, &minor) != EGL_TRUE) {
        fprintf(stderr, "Can't initialise egl display\n");
        exit(1);
    }
    printf("EGL major: %d, minor %d\n", major, minor);

    eglGetConfigs(egl_display, NULL, 0, &count);
    printf("EGL has %d configs\n", count);

    configs = calloc(count, sizeof *configs);

    eglChooseConfig(egl_display, config_attribs,
                    configs, count, &n);

    for (i = 0; i < n; i++) {
        eglGetConfigAttrib(egl_display,
                           configs[i], EGL_BUFFER_SIZE, &size);
        printf("Buffer size for config %d is %d\n", i, size);
        eglGetConfigAttrib(egl_display,
                           configs[i], EGL_RED_SIZE, &size);
        printf("Red size for config %d is %d\n", i, size);

        // just choose the first one
        egl_conf = configs[i];
        break;
    }

    egl_context =
        eglCreateContext(egl_display,
                         egl_conf,
                         EGL_NO_CONTEXT, context_attribs);

}

static void
get_server_references(void) {

    display = wl_display_connect(NULL);
    if (display == NULL) {
        fprintf(stderr, "Can't connect to display\n");
        exit(1);
    }
    printf("connected to display\n");

    struct wl_registry *registry = wl_display_get_registry(display);
    wl_registry_add_listener(registry, &registry_listener, NULL);
```

```
    wl_display_dispatch(display);
    wl_display_roundtrip(display);

    if (compositor == NULL || shell == NULL) {
        fprintf(stderr, "Can't find compositor or shell\n");
        exit(1);
    } else {
        fprintf(stderr, "Found compositor and shell\n");
    }
}

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

    get_server_references();

    surface = wl_compositor_create_surface(compositor);
    if (surface == NULL) {
        fprintf(stderr, "Can't create surface\n");
        exit(1);
    } else {
        fprintf(stderr, "Created surface\n");
    }

    shell_surface = wl_shell_get_shell_surface(shell, surface);
    wl_shell_surface_set_toplevel(shell_surface);

    create_opaque_region();
    init_egl();
    create_window();

    while (wl_display_dispatch(display) != -1) {
        ;
    }

    wl_display_disconnect(display);
    printf("disconnected from display\n");

    exit(0);
}
```

An image of it running is an unexciting black rectangle:

Black window




If you want to make it more colourful, uncomment out the GL calls.

Copyright © Jan Newmarch, jan@newmarch.name

If you like this book, please contribute using Flattr
or donate using PayPal