ECE297 Quick Start Guide
EZGL

"The purpose of visualization is insight, not pictures."

–Ben Shneiderman

1 Overview

EZGL is a drawing package that is based on its predecessor *EasyGL*. EZGL was created to provide the same *ease of use* as EasyGL but with the flexibility to add custom features more easily. EZGL accomplishes this by providing a thin wrapper around the popular GTK graphics package.

EZGL provides three key features:

- **Ease of use**: The package presents a simple interface for choosing colours, line widths, etc. and drawing primitives. It lets you use any coordinate system you want to draw your graphics, and it handles all zooming in and out of the graphics for you.
- **Extendability**: The thin wrapper around GTK allows you to quickly and easily display simple graphics, while also allowing you to enhance your design using the powerful GTK library.
- **Platform independence**: GTK (and therefore EZGL) generates X-windows and Cairo (Linux, Mac) and Win32 (MS Windows), all from the same graphics calls by your program.

For those who know graphics, EZGL is a 2-dimensional, immediate-mode graphics library. It handles many events (window resizing, zooming in and out, etc.) itself, and you can pass in callback functions that will be used to process other events if you wish. If you don't know graphics, don't worry: you can use EZGL without understanding the terms above!

1.1 Dependencies

The library currently depends on GTK3 and Cairo, both of which are installed on the UG machines.

1.2 Example Files

- `application.{hpp,cpp}`: The main class for the graphics wrapper. This defines the ezgl::application class which wraps a GTK application object.
- `graphics.{hpp,cpp}`: The *renderer* of graphics. It provides drawing primitives to easily draw lines, shapes, etc.
- `main.ui`: This is the user interface (UI) file that describes the layout of your window.
- `other .hpp and .cpp files`: Files that implement EZGL: include these in your project.
- `basic_application.cpp`: An example file showing how to use the graphics.

- `Makefile`: A unix makefile for the example program.

The header and source files are thoroughly commented, so if you wish to learn more about what each file does we suggest you read the files and the comments within. To learn how to use ezgl and what features it supports, the key files are `graphics.hpp`, `application.hpp` and `color.hpp`.

1.3 Compiling

On the *ugXXX* machines simply `cd` to the directory containing the EZGL example, then type:

```
make
./basic_application
```

to build and run the example program.

The ezgl::application looks for the user interface file (i.e. *main.ui*) at runtime. This means that, while your application is running, the UI file cannot be moved or errors may occur.

When adding EZGL to a larger program, any source files which use graphics must `#include "ezgl/application.hpp"` and `#include "ezgl/graphics.hpp"` . You must also add all the .cpp, .h files that come with EZGL, *except basic_application.cpp*, to your makefile or project.

The included makefile is set up for Unix/Linux; you can use this makefile as a model if you are adding EZGL to a larger project. On Linux, EZGL requires a few libraries (GTK and Cairo), so you will need to install their development versions and add them to the link step of compilation. These libraries are already installed on the *ugXXX* machines. Details and tips about what you might need to do on other Unix machines are in the makefile.

2 Interactive (Event-Driven) Graphics

See basic_application.cpp for an example of how to use this package. The basic structure to create a window in which you can draw and that lets the user pan and zoom the graphics is shown below. You call a few setup functions to get the graphics going, then pass control the GTK event loop *run*, which responds to panning and zooming button pushes from the user. To redraw the graphics, the ezgl::application will automatically call a routine you pass into it (a callback function); in the example below this routine is *draw_main_canvas*. Your `draw_main_canvas` function doesn't have to do anything special to enable panning and zooming; all that is handled automatically by the graphics package. If you wish, you can pass in additional functions that will be called when keyboard input or mouse input occurs over the part of the graphics window to which you are drawing. EZGL also supports various UI "widgets", which can be connected to callback functions.

```
1 #include "ezgl/application.hpp"
2 #include "ezgl/graphics.hpp"
3
4 #include <iostream>
5
```

```
/**
 * Draw to the main canvas using the provided graphics object.
 *
 * The graphics object expects that x and y values will be in the main canvas'
 * world coordinate system.
 */
void draw_main_canvas(ezgl::renderer *g);

/**
 * The world drawing coordinates that are passed to add_canvas function.
 *
 * We choose from (xleft,ybottom) = (0,0) to (xright,ytop) = (1100,1150)
 */
static ezgl::rectangle initial_world{{0, 0}, 1100, 1150};

/**
 * The start point of the program.
 *
 * This function initializes an ezgl application and runs it.
 *
 * @param argc The number of arguments provided.
 * @param argv The arguments as an array of c-strings.
 *
 * @return the exit status of the application run.
 */
int main(int argc, char **argv)
{
   ezgl::application::settings settings;

   // Path to the "main.ui" file that contains an XML description of the UI.
   settings.main_ui_resource = "main.ui";

   // Note: the "main.ui" file has a GtkWindow called "MainWindow".
   settings.window_identifier = "MainWindow";

   // Note: the "main.ui" file has a GtkDrawingArea called "MainCanvas".
   settings.canvas_identifier = "MainCanvas";

   // Create our EZGL application.
   ezgl::application application(settings);

   // Set some parameters for the main sub-window, where visualization
   // graphics are draw. Set the callback function that will be called when
   // the main window needs redrawing, and define the (world) coordinate
   // system we want to draw in.
   application.add_canvas("MainCanvas", draw_main_canvas, initial_world);

   // Run the application until the user quits.
   // This hands over all control to the GTK runtime---after this point
   // you will only regain control based on callbacks you have setup.
   // Three callbacks can be provided to handle mouse button presses,
   // mouse movement and keyboard button presses in the graphics area,
   // respectively. Also, an initial_setup function can be passed that will
   // be called before the activation of the application and can be used
```

```
60    // to create additional buttons, initialize the status message, or
61    // connect added widgets to their callback functions.
62    // Those callbacks are optional, so we can pass nullptr if
63    // we don't need to take any action on those events
64    int ret = application.run(initial_setup, act_on_mouse_press, act_on_mouse_move,
      act_on_key_press);
65
66    // Execution returns here when the 'Proceed' button is pressed.
67    // You can do more computation and then re-run the event loop.
68
69    return ret;
70 }
71
72 /**
73  * Function to draw on the main canvas. This function is called by the GTK runtime to
      update the display.
74  *
75  * @param the graphics renderer for drawing.
76  */
77 void draw_main_canvas(ezgl::renderer *g)
78 {
79    const float rectangle_width = 50;
80    const float rectangle_height = rectangle_width;
81    const ezgl::point2d start_point(150, 30);
82    ezgl::rectangle color_rectangle = {start_point, rectangle_width, rectangle_height};
83
84    // Draw a rectangle bordering all the drawn rectangles
85    g->draw_rectangle(start_point, color_rectangle.top_right());
86
87    /*
88     * You can put as much drawing as you like here, call other routines, etc.
89     */
90
91    /* ... */
92 }
```

This screenshot below shows the graphics window created by the example.

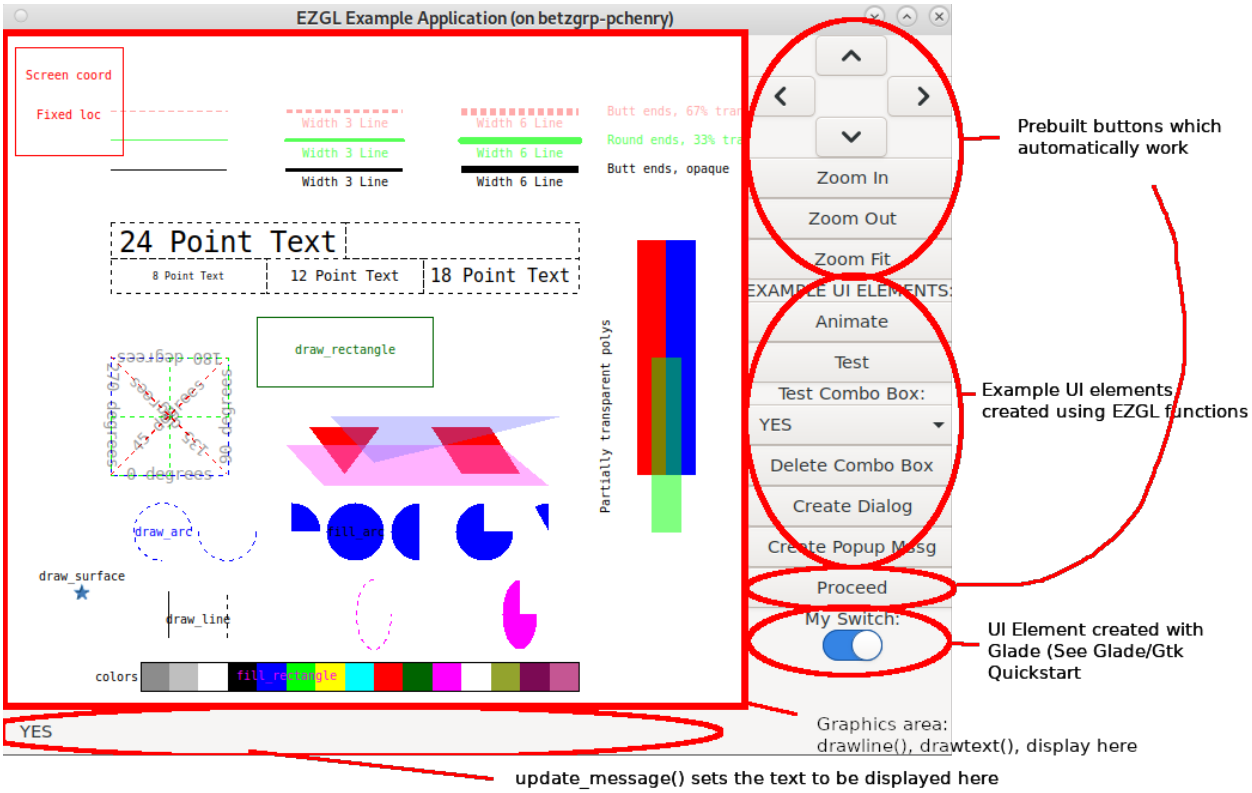

Figure 1: A screenshot of the included example. The area in the red rectangle is where your `draw_main_canvas` function will render graphics with draw_text(), fill_poly(), etc calls. Other functions calls let you add buttons or update the message at the bottom of the screen.

2.1 Built-In Graphics Buttons and Mouse Zoom/Pan

Fig. 2 shows the various buttons that are automatically created in an EZGL window. You can create more buttons if you wish; these ones are created for you and perform the functions shown below. Two mouse functions are also handled automatically for you:

- Spinning the mouse wheel forward and backward zooms in and out, respectively. The center of the zoomed area is wherever your mouse cursor currently is.
- Moving the mouse while holding down the wheel or third button pans (shifts) the graphics as if you were dragging the image.

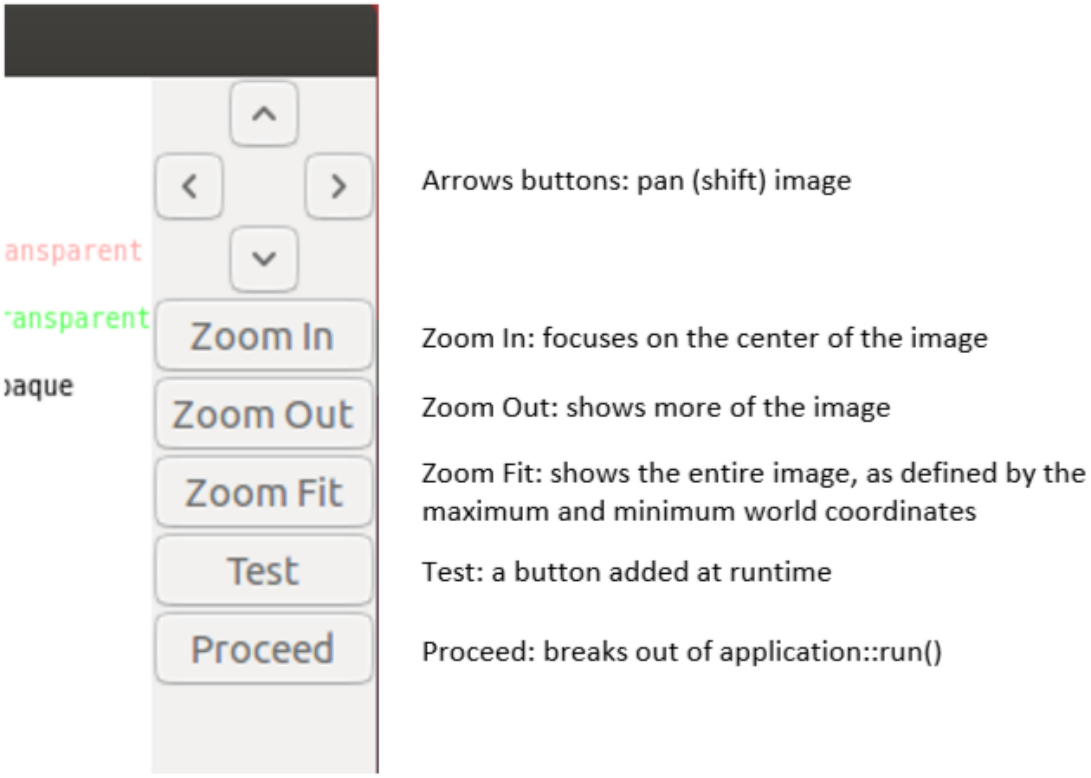

Figure 2: Buttons and their functions.

2.2 Callback Functions

EZGL lets you easily create interactive user interface elements, including buttons, combo boxes, and dialog windows. You can connect a special type of function, called a callback function, to these widgets through passing the function as a parameter to the EZGL widget creation functions. The function comments in `application.hpp` will tell you the function prototype you need to follow, and `basic_application.cpp` will have example uses of these functions.

3 Subroutine Reference

Read `graphics.hpp` for a list of drawing functions you can call and `color.hpp` for the `color` class which is stored in *RGBA* format and contains some predefined colors. These files are well commented so they're better documentation than a manual. The functions fall into the categories below:

Set graphics attributes: color, linewidth, linestyle, text size, text rotation and text justification, etc. These attributes are sticky; they affect all subsequent drawing until you change them again. Colors are red, green and blue values from 0 to 255 (8-bits each), and can optionally also include alpha transparency from 0 which is transparent to 255 which is completely opaque. Drawing with partial transparency (i.e. alpha != 255) is slower than drawing with opaque colours as the graphics hardware has more work to do. You can also change the text font. For example, if you want to display Chinese characters, you can switch to a Chinese font such as *Noto Sans CJK SC* (for more info about Noto fonts, see this) by using the `format_font()` function. To list all installed fonts that are available on the UG machines, just run *fc-list*.

Draw graphics primitives: text, lines, arcs, elliptical arcs, filled rectangles, filled arcs/circles/ellipses, filled polygons.

Draw bitmap images: You can create a surface (bitmap) from a .png file using `load_png`, and can then draw this surface wherever you would like using `draw_surface()`.

Coordinate systems: You choose your own *world* coordinate system when you create your applications main canvas (the `add_canvas()` function) and by default you draw in this coordinate system. EZGL automatically adjusts how this coordinate system transforms (is mapped) to the screen as the user pans and zooms the graphics, so you can draw your scene the same way in your `draw_main_canvas()` callback and the graphics will pan and zoom as the user requests. You can also determine what the current visible bounds of the world with the `get_visible_world()` function. If you want some graphics to stay in a fixed location on the screen no matter how the user pans and zooms, you should draw them in screen (pixel) coordinates using `set_coordinate_system(ezgl::SCREEN)`.

Read `application.hpp` for a list of functions you can call to control and update your application (GUI window). These functions fall into the categories below:

Setup and control routines: You've already seen most of these. They allow you to set up the graphics, and run the application.

Create and destroy interactive UI elements: You can make various widgets (buttons, combo boxes, labels), with a callback function that will be called whenever the widget is used. You can also create simple popup windows. Call these functions in `initial_setup` for widgets that will always be present, and in callback functions for widgets created by user interaction.

Update messages: You can update the message written in the created status bar. This function can be called only from the `initial_setup` function or any other callback functions.

Create Dialog windows and messages You can create popup messages or dialog windows.

4 Known Issues

- Windows support is assumed but has not been tested. Use at your own risk.

5 GTK and Glade

EZGL is a wrapper for the much larger GTK graphics/UI library. You do not need to use GTK/Glade to complete any of the labs. However, they can be used to create more complex UI features.

For an overview of how GTK works that will help you understand these examples, see the GTK/Glade quickstart guide.