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, but this is not necessary for the course.

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.

```
1 #include "ezgl/application.hpp"
2 #include "ezgl/graphics.hpp"
3
4 #include <iostream>
5
6 /**
```

```
 * 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);

   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
   // to create additional buttons, initialize the status message, or
   // connect added widgets to their callback functions.
   // Those callbacks are optional, so we can pass nullptr if
   // we don't need to take any action on those events
   int ret = application.run(initial_setup, act_on_mouse_press, act_on_mouse_move,
```

```
    act_on_key_press);

   // Execution returns here when the 'Proceed' button is pressed.
   // You can do more computation and then re-run the event loop.

   return ret;
}

/**
 * Function to draw on the main canvas. This function is called by the GTK runtime to
    update the display.
 *
 * @param the graphics renderer for drawing.
 */
void draw_main_canvas(ezgl::renderer *g)
{
   const float rectangle_width = 50;
   const float rectangle_height = rectangle_width;
   const ezgl::point2d start_point(150, 30);
   ezgl::rectangle color_rectangle = {start_point, rectangle_width, rectangle_height};

   // Draw a rectangle bordering all the drawn rectangles
   g->draw_rectangle(start_point, color_rectangle.top_right());

   /*
    * You can put as much drawing as you like here, call other routines, etc.
    */

   /* ... */
}
```

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

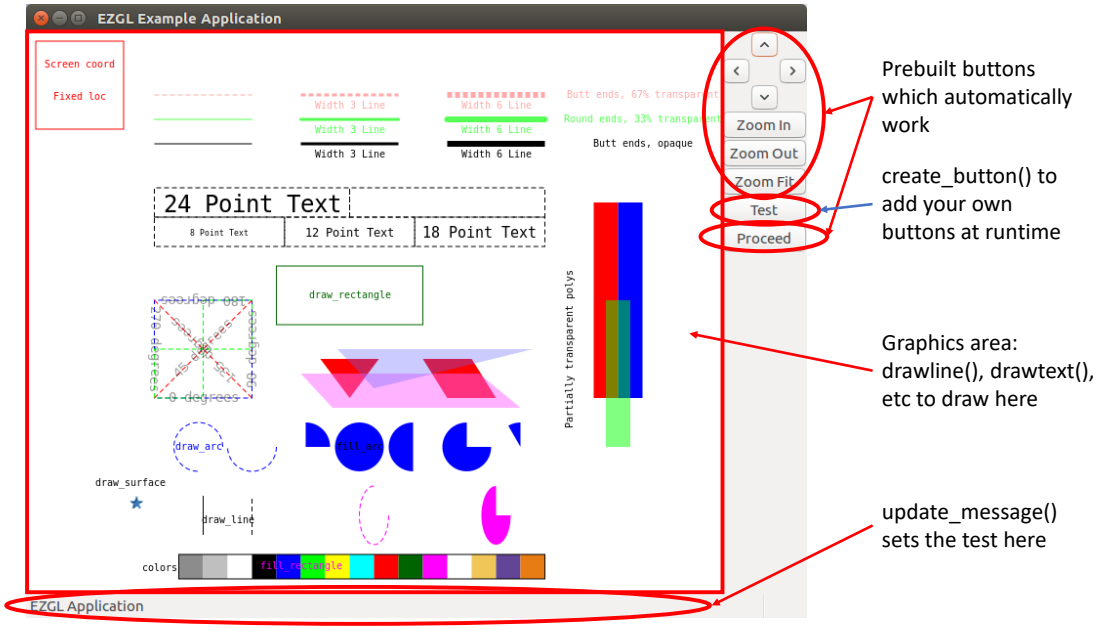

Figure 1: A screenshot of the included example. The area in the red rectangle is where your `draw_main_canvas` function will render graphics with draw_text(), fill_poly(), etc calls. Other functions calls let you add buttons or update the message at the bottom of the screen.

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.

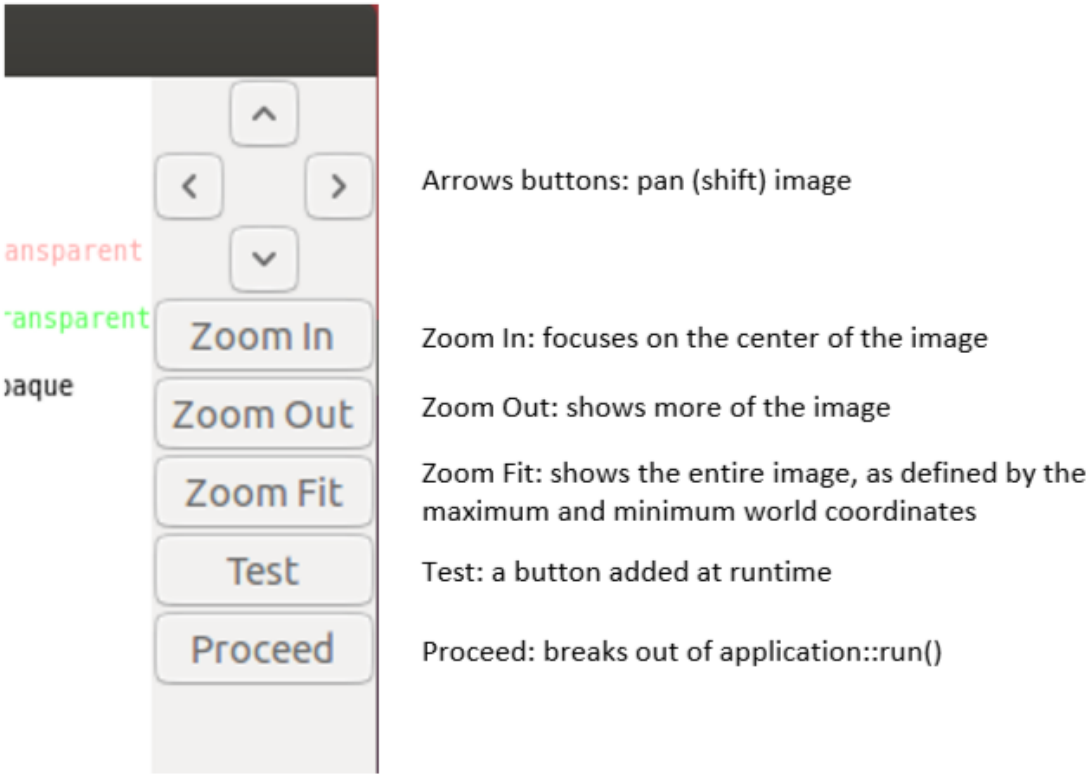

Figure 2: Buttons and their functions.

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 buttons: You can make your own buttons, with a callback function that will be called whenever the button is pressed. You can call these functions only from the `initial_setup` function or any other callback functions (mouse and keyboard callbacks)

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.

4 Known Issues

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

5 Advanced Features

This section describes some more advanced features of GTK that you *may* find useful. You do **not** need read these sections to implement any of the labs, but they are here in case you want to expand your design.

5.1 UI Layout Change

This section will step you through changing the UI layout of the basic example program described in the earlier sections and adding a text entry feature. EZGL through GTK allows you to easily change the UI layout of your application. GTK is used to build most Linux GUI applications such as Chromium and Inkscape. GNOME (the Linux Desktop Environment you are using in the labs) is itself based on GTK. You can easily convert the UI of your mapper application to a layout that resembles your favourite GUI application with features such as a menu bar and popup menus, etc.

Before starting, let's first go through the existing UI layout described in the `main.ui` file. The UI consists of a GtkWindow called MainWindow. A GtkWindow can only have one child, so we need to use a GtkGrid to manage multiple children. Hence the MainWindow has one child called OuterGrid which is a 2x2 GtkGrid. The top left slot of the grid is occupied by a GtkDrawingArea called MainCanvas which we use to draw the graphics. The top right slot is another GtkGrid called InnerGrid that contains the predefined buttons. A status bar is placed in the bottom left slot, while the bottom right slot is empty.

To change the UI layout, you can manually edit the main.ui file; however, it is easier to use a UI designer program such as *Glade* (you can also add UI features by writing code as shown in Section 5.2). Glade is installed on the UG machines. To run it, type:

```
glade /path/to/main.ui
```

Glade will display the current UI layout along with the widgets you can add as shown in Fig. 3. To add a text entry widget at the top of the layout, we have to change the OuterGrid size to 3x2. This is done by clicking on the OuterGrid widget in the top right box of Glade program. This will show the properties of this widget; under the *General* tab, change the rows count to 3 instead of 2. This will add an empty row at the bottom of the UI layout. To move the empty row to the top, you have to change the location of the other widgets. Select the MainCanvas widget, and under *Packing* tab, change the top attachment to 1 (one row below the top) instead of 0. Then, change the top attachment of InnerGrid to 1, and that of StatusBar and GtkSeparator to 2. This moves the empty row to the top. Now drag and drop the *Text Entry* widget to the left cell of that empty row. This will add a GtkEntry to the list of used widgets. Select the GtkEntry widget, and under the *General* tab, write "TextInput" in the *ID*. This name will be used in the EZGL program. After that, save and exit Glade.

To test the functionality of the added widget, we are going to utilize the "Test" button included in the basic example program to read the text entered in the added widget and update the status bar message with that text when it is clicked. To achieve that, add the code below to the *test_button* function (the callback of the "Test" button). For more information on GtkEntry functions, see this.

```
void test_button(GtkWidget *widget, ezgl::application *application)
{
```

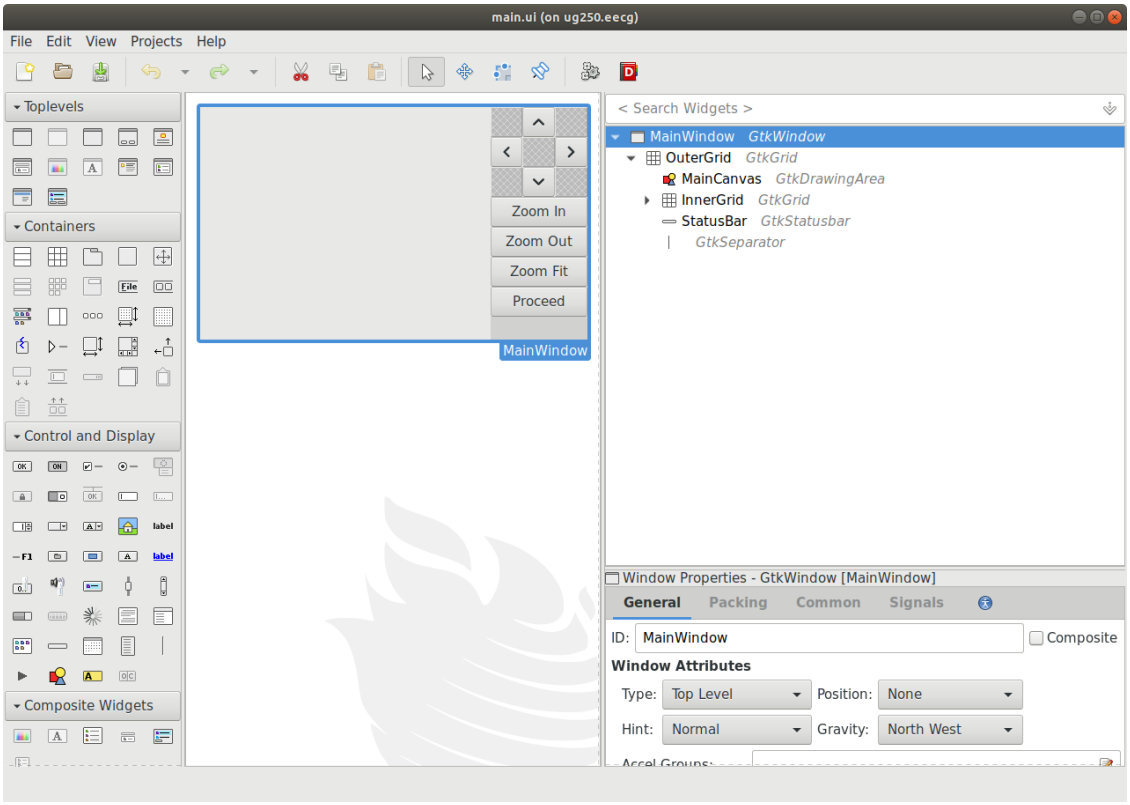

Figure 3: Glade program.

```
  // Get the GtkEntry object
  GtkEntry* text_entry = (GtkEntry *) application->get_object("TextInput");

  // Get the text written in the widget
  const char* text = gtk_entry_get_text(text_entry);

  // Update the status bar message
  application->update_message(text);

  // Redraw the graphics
  application->refresh_drawing();
}
```

That's it! Your application should look like Fig. 4.

As you can see from Fig. 3, there are a lot of GTK widgets that you can add to your application other than the GTKEntry widget, such as search entry, popup menu, menu bar and tool bar widgets. The steps to add any of these widgets are similar to the steps shown above for the GTKEntry widget, which can be summarized as:

- Create an empty slot in the OuterGrid
- Drag and drop the widget to that slot

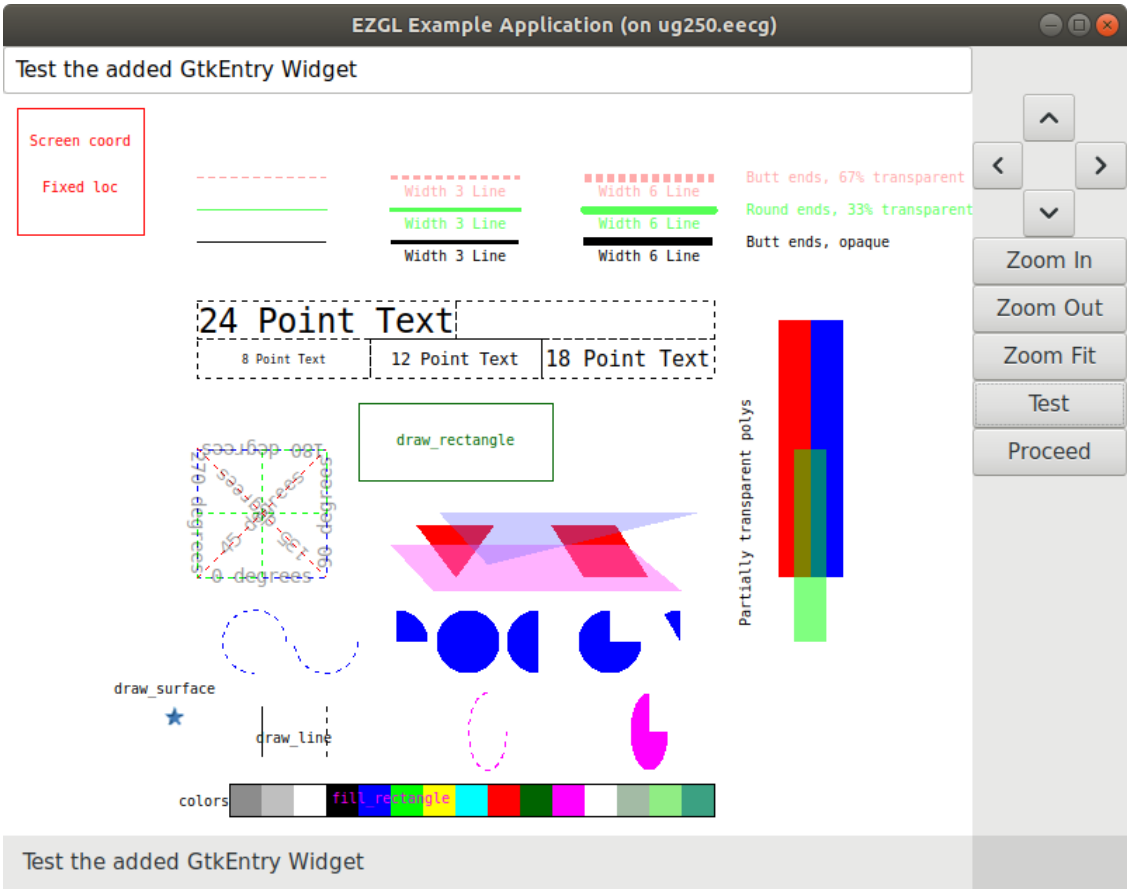

Figure 4: The added Text Entry widget.

- Give the widget an ID
- Learn how to use this widget by looking at available functions and signals (discussed below) for this widget in this link.

To allow user interaction, most widgets emit signals when the user interacts with them. For example when you click on a button, the widget emits the *clicked* signal. If you want your application to do something when a signal is emitted, you will have to connect this signal to a callback function; whenever the signal is emitted, your callback function is called. To do that, you need to use *g_signal_connect* function as listed below.

```
g_signal_connect(
  G_OBJECT(my_widget_id), // The ID of your widget (the id value you set in glade)
  "signal_name", // Signal name (from the widget documentation) e.g. "clicked"
  G_CALLBACK(my_func), // name of callback function (you write this function)
  NULL // pointer to (arbitrary) data to pass to callback my_func or NULL for no data
);
```

If you have added a UI element that should always be visible in the main window, this *g_signal_connect* call should be done in the *initial_setup* routine passed to *application.run*.

5.2 Creating a Popup Dialog Box

This section will step you through creating a popup dialog box for the basic example program described in the earlier sections. We add this in a somewhat different way than the UI elements of the prior section, as we want this pop up to appear and disappear so it is only on screen when appropriate, instead of always being part of the main window. We will utilize the "Test" button included in the sample program to trigger the popup dialog. To start, locate the callback function for the test button `void test_button(GtkWidget *widget, ezgl::application *application)` in the basic_application.cpp file. Inside this function you will see two lines of code that simply update the status message and refresh the drawing. We will add most of the code inside this function.

First we will declare some variables that will be used for the dialog box. Add the following code to the `test_button` function:

```
GObject *window;         // the parent window over which to add the dialog
GtkWidget *content_area; // the content area of the dialog
GtkWidget *label;        // the label we will create to display a message in the content
    area
GtkWidget *dialog;       // the dialog box we will create
```

Now that all of the variables are declared, we can start making the dialog. We need to get a pointer to the main UI window (the one in fig 1). To do this, add the following code to get a pointer to the main window from the EZGL application:

```
// get a pointer to the main application window
window = application->get_object(application->get_main_window_id().c_str());
```

To create the dialog we will use the `gtk_dialog_new_with_buttons()`. Below is the GTK definition of the function.

```
GtkWidget * gtk_dialog_new_with_buttons(
  const gchar *title,
  GtkWindow *parent,
  GtkDialogFlags flags,
  const gchar *first_button_text,
  ...
);
```

- title: The title of the dialog box which is placed in the top bar.
- parent: The parent window of the dialog box.
- flags: The type of dialog box (GTK_DIALOG_MODAL, GTK_DIALOG_DESTROY_WITH_PARENT, GTK_DIALOG_USE_HEADER_BAR. See this webiste for more details on these options).

After the `flags` argument, button text/response ID pairs can be added with a `NULL` pointer at the end. We will create a simple dialog box with an *OK* and *CANCEL* button. Note that creating the dialog box does not automatically show it, we will do this later. Add the following code to create the dialog:

```
// Create the dialog window.
// Modal windows prevent interaction with other windows in the same application
dialog = gtk_dialog_new_with_buttons(
  "Test Dialog",
  (GtkWindow*) window,
  GTK_DIALOG_MODAL,
  ("OK"),
  GTK_RESPONSE_ACCEPT,
  ("CANCEL"),
  GTK_RESPONSE_REJECT,
  NULL
);
```

Note the button text/response ID pairs in the code above. The *OK* button sends the predefined *GTK_RESPONSE_ACCEPT* response ID while the *CANCEL* button sends the *GTK_RESPONSE_REJECT* ID. You can add as many buttons as you like and use the predefined responses described here or define your own response IDs (integers).

Next we will add a label to the dialog box we just created using the following code:

```
// Create a label and attach it to the content area of the dialog
content_area = gtk_dialog_get_content_area(GTK_DIALOG(dialog));
label = gtk_label_new("Simple Dialog With a Label. Press OK or CANCEL too close.");
gtk_container_add(GTK_CONTAINER(content_area), label);
```

The first line creates the content area for the dialog we created. The second line creates a new GTK label with the given text. The third line adds the label to the content area. Now that we have created the dialog box and added a label to its content area, we are ready to show it. This is done with a single line:

```
// The main purpose of this is to show dialog's child widget, label
gtk_widget_show_all(dialog);
```

This line does **not** block execution. It shows the dialog box in a new thread. To capture the events of the dialog first define a new function shown below.

```
void on_dialog_response(GtkDialog *dialog, gint response_id, gpointer user_data)
{
  // For demonstration purposes, this will show the enum name and int value of the button
     that was pressed
  std::cout << "response is ";
  switch(response_id) {
    case GTK_RESPONSE_ACCEPT:
      std::cout << "GTK_RESPONSE_ACCEPT ";
      break;
    case GTK_RESPONSE_DELETE_EVENT:
      std::cout << "GTK_RESPONSE_DELETE_EVENT (i.e. 'X' button) ";
      break;
```

```
    case GTK_RESPONSE_REJECT:
      std::cout << "GTK_RESPONSE_REJECT ";
      break;
    default:
       std::cout << "UNKNOWN ";
        break;
  }
  std::cout << "(" << response_id << ")\n";

  // This will cause the dialog to be destroyed and close
  // without this line the dialog remains open unless the
  // response_id is GTK_RESPONSE_DELETE_EVENT which
  // automatically closes the dialog without the following line.
  gtk_widget_destroy(GTK_WIDGET (dialog));
}
```

Now, back in the `test_button` function add the following after the `gtk_widget_show_all()` call:

```
// Connecting the "response" signal from the user to the associated callback function
g_signal_connect(
  GTK_DIALOG(dialog),
  "response",
  G_CALLBACK(on_dialog_response),
  NULL
);
```

This code connects the *response* signal to the `on_dialog_response()` function we just created.

Alternatively, you could create a dialog box that blocks the execution of the main thread (therefore you wouldn't have to create the `on_dialog_response` callback function and connect the `response` signal). This GTK page shows you how to do this using the `gtk_dialog_run()` function.

Thats it! For more details on the various dialog options you are encouraged to read the GTK documents. All of the code added to basic_application.cpp is show below.

```
/** OTHER CODE IN BASIC_APPLICATION.CPP */

/**
 * A callback function for responding to the dialog (pressing buttons or the X button).
 */
void on_dialog_response(GtkDialog *dialog, gint response_id, gpointer user_data)
{
  // For demonstration purposes, this will show the int value of the response type
  std::cout << "response is ";
  switch(response_id) {
    case GTK_RESPONSE_ACCEPT:
      std::cout << "GTK_RESPONSE_ACCEPT ";
      break;
    case GTK_RESPONSE_DELETE_EVENT:
      std::cout << "GTK_RESPONSE_DELETE_EVENT (i.e. 'X' button) ";
      break;
```

```
    case GTK_RESPONSE_REJECT:
      std::cout << "GTK_RESPONSE_REJECT ";
      break;
    default:
      std::cout << "UNKNOWN ";
      break;
  }
  std::cout << "(" << response_id << ")\n";

  /*This will cause the dialog to be destroyed*/
  gtk_widget_destroy(GTK_WIDGET (dialog));
}

/**
 * A callback function to test the Test button
 */
void test_button(GtkWidget *widget, ezgl::application *application)
{
  // BEGIN: EXISTING EXAMPLE CODE
  application->update_message("Test Button Pressed");
  application->refresh_drawing();
  // END: EXISTING SAMPLE CODE

  // BEGIN: CODE FOR SHOWING DIALOG
  GObject *window;            // the parent window over which to add the dialog
  GtkWidget *content_area;    // the content area of the dialog (i.e. where to put stuff
    in the dialog)
  GtkWidget *label;           // the label we will create to display a message in the
    content area
  GtkWidget *dialog;          // the dialog box we will create

  // get a pointer to the main application window
  window = application->get_object(application->get_main_window_id().c_str());

  // Create the dialog window. Modal windows prevent interaction with other windows in
    the same application
  dialog = gtk_dialog_new_with_buttons(
    "Test Dialog",
    (GtkWindow*) window,
    GTK_DIALOG_MODAL,
    ("OK"),
    GTK_RESPONSE_ACCEPT,
    ("CANCEL"),
    GTK_RESPONSE_REJECT,
    NULL
  );

  // Create a label and attach it to the content area of the dialog
  content_area = gtk_dialog_get_content_area(GTK_DIALOG(dialog));
  label = gtk_label_new("Simple Dialog With a Label. Press OK or CANCEL too close.");
  gtk_container_add(GTK_CONTAINER(content_area), label);

  // The main purpose of this is to show dialog's child widget, label
  gtk_widget_show_all(dialog);
```

```
  // Connecting the "response" signal from the user to the associated callback function
  g_signal_connect(
    GTK_DIALOG(dialog),
    "response",
    G_CALLBACK(on_dialog_response),
    NULL
  );
  // END: CODE FOR SHOWING DIALOG
}
```