ECE297 Quick Start Guide

EasyGL

“The purpose of visualization is insight, not pictures.”

–Ben Shneiderman

1 Overview

EasyGL is an drawing package that provides two 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. It can also print (to PostScript) any graphics you draw.

• Platform independence. EasyGL generates X-windows and Cairo (Linux, Mac), Win32 (MS Windows) and PostScript (printer) output, all from the same graphics calls by your program.

For those who know graphics, EasyGL 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 EasyGL without understanding the terms above!

1.1 Example Files

• graphics.h: The header file for the graphics package; #include this file.
• example.cpp: An example file showing how to use the graphics.
• other .h and .cpp files: Files that implement Easgyl: include these in your project.
• makefile: A unix makefile for the example program.

1.2 Compiling

On the ugaXXX machines simply cd to the directory containing the easyGL example, then type:

```
make
./example
```

to build and run the example program.
When adding easyGL to a larger program, any source files which use graphics must include "graphics.h". You must also add all the .cpp, .h files that come with easygl, except example.cpp, to your makefile or project.

The included makefile is set up for Unix/Linux and tells easyGL that X11 is your machine's low-level graphics API; you can use this makefile as a model if you are adding easyGL to a larger project. On Linux, easyGL requires a few libraries (specifically X11 graphics, the Xft text renderer, fontconfig 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 /textitugXXX 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 example.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 to the event_loop, which responds to panning and zooming button pushes from the user. To redraw the graphics, event_loop will automatically call a routine you pass into it (a callback function); in the example below this routine is drawscreen. Your drawscreen 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.

```c
#include "graphics.h"

int main () {
  // Create a window with name and background colour as specified
  init_graphics("Some Example Graphics", WHITE);

  // Set-up drawing coordinates. We choose from (xl,ybottom) = (0,0) to (xr,ytop) = (1000,1000)
  set_visible_world (0.,0.,1000.,1000.);

  // This message will show up at the bottom of the window.
  update_message("Interactive graphics example.");

  // Pass control to the window handling routine. It will watch for user input and
  // redraw the screen / pan / zoom / etc. the graphics in response to user input or
  // windows being moved around the screen. This is done by calling the four callbacks
  // below. You can turn mouse button presses, mouse movement, and keyboard input (events
  // off for your graphics, and you can send NULL for those routines if you do. You MUST
  // provide the drawscreen function, since it is what actually draws your graphics in
  // response to the user zooming in and out, etc
  event_loop(act_on_button_press,act_on_mouse_move,act_on_key_press, drawscreen);

  // The program will stay in event loop until the user clicks "Proceed" or "Exit". When
  // you don't want any more graphics, close them down.
  close_graphics ();
}
```
```c
void drawscreen (void) {
     // redrawing routine for still pictures. The graphics package calls this routine to do
     // redrawing after the user changes the window in any way. You can always draw things
     // the same size; the graphics package will handle panning and zooming itself.
     clearscreen(); /* Should precede drawing for all drawscreens */
     setfontsize (10);
     setcolor (BLACK);
     drawtext (110, 55, "colors", 150) // draw centered at x = 100, y = 55
     // 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 drawscreen function will render graphics with drawtext(), fillpoly(), etc calls. Other functions calls let you add buttons or update the message at the bottom of the screen.](image)
2.1 Built-In Graphics Buttons and Mouse Zoom/Pan

Fig. 2 shows the various buttons that are automatically created in an EasyGL 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.

![Arrow buttons: pan (shift) image](image)

- Zoom In: focuses on center of image
- Zoom Out: shows more image
- Zoom Fit: shows entire image, as defined by the maximum and minimum world coordinates in the last set_visible_world()
- Window: lets you click on the diagonally-opposite corners of a box in which to zoom
- Postscript: writes graphics area image to pic1.ps (first click), pic2.ps (2nd click) etc.
- Proceed: returns from event_loop()
- Exit: ends the program

Figure 2: Buttons and their functions.
3 Noninteractive Graphics

To animate graphics, you want to draw primitives and have them appear on the screen without waiting for events (the user clicking on Zoom In and Zoom Out etc.). This can also be done with easyGL, and an example use is shown in the example.cpp file. The basic structure is shown below. The main difference from the prior code is that instead of calling event_loop(), you draw whatever you want and then call flushinput() to make it appear on the screen.

In this mode, the buttons in the easyGL window are greyed out, since you can’t pan and zoom the graphics.

```
#include "graphics.h"

int main () {
    // Create a window with name and background colour as specified
    init_graphics("Non-interactive graphics", WHITE);

    // Set-up drawing coordinates. We choose from (xl,ybottom) = (0,0) to (xr,ytop) =
    // (1000,1000)
    set_visible_world (0.,0.,1000.,1000.);

    // This message will show up at the bottom of the window.
    update_message("Non-interactive graphics example.");

    // Draw whatever you want.
    my_drawing_routine (1); // Assume 1 means first frame
    flushinput (); // Need this to make graphics appear
    my_delay (); // Wait a while so the user can actually see the display
    my_drawing_routine (2); // Draw the 2nd frame (shifted picture or something)
    ...

    // When you don’t want any more graphics, close them down.
    close_graphics ();
}
```

4 Subroutine Reference

Read graphics.h for a list of functions you can call, and easygl_constants.h for a list of constants you can call them with (colour names, etc.). They are well commented so they’re better documentation than a manual. The 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, choose the coordinate system in which you’d like to draw, close down the graphics, and automatically respond to pan and zoom events. If you wish, you can also control what events (user input) you’d like passed on to callback functions you write – these callback functions let you choose exactly what you’d like to do when the user clicks a mouse button or presses a key when the cursor is in the area where you are drawing your graphics (the red outlined area in Figure 1).

**Set graphics attributes:** colour, linewidth, linestyle, text size and text rotation, etc. These
attributes are sticky; they affect all subsequent drawing until you change them again. Colours 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.

**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_from_file`, and can then draw this surface wherever you would like using `draw_surface()`.

**Create new buttons:** you can make your own buttons, with a callback function that will be called whenever the button is pressed.

**Level of Detail routines:** These routines let you ask how much of the coordinate system you’ve specified is visible on the screen. You can use this information to decide if the user is zoomed way out and you should draw only major features, or if the user is zoomed way in and hence it would be appropriate to draw lots of little details.

**Coordinate systems:** You choose your own world coordinate system with `set_visible_world()` and by default you draw in this coordinate system. Easygl 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 `drawscreen()` callback and the graphics will pan and zoom as the user requests. 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(GL_SCREEN)`. You can determine what the limits of the graphics area are in pixels (screen coordinates) with the `get_visible_screen()` function.

**Double buffering:** If your graphics take some time to draw, you may notice that the image flickers as the user pans and zooms rapidly. You can avoid this flickering by using **double buffering** in which you draw to an offscreen buffer, and then rapidly copy the completed drawing to the screen. To draw with double buffering call `set_drawing_buffer(OFF_SCREEN)`. The last line in your `drawscreen` callback should then be `copy_off_screen_buffer_to_screen()`.

5 Known Issues

- With VNC or with MS Windows, closing the graphics with `close_graphics()` and then re-opening them with `init_graphics()` may not work. You should simply leave the graphics window open for the lifetime of the program.

- Postscript does not support transparency or bitmap drawing. Transparent graphics will show as opaque and `draw_surface()` calls will not display in the PostScript output.

- Many updates have been made to this package in 2017 and they have not been tested yet on Microsoft Windows; this testing is planned for the future.