EasyGL (Version 2.0)

September 25, 2011

An Easy-to-Use Graphics Package for X11, Win32 and PostScript Displays

Vaughn Betz (vaughn@eecg.utoronto.ca)

Overview:

EasyGL is an drawing package that provides a two key features:

- Easy to 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 independent. EasyGL generates X-windows (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!

Files in this Archive:

graphics.cpp: The source code for the graphics package.

graphics.h: The header file for the graphics package; #include this file.

easygl_constants.h: A helper header file, no need to #include.

example.cpp: An example file showing how to use the graphics.

makefile: A unix makefile for the example program.

easygl.vcxproj: A Microsoft Visual Studio project file for the example program.

easygl.sln: A Microsoft Visual Studio solution file for the example program.

manual.pdf: This document.

Compiling the Graphics

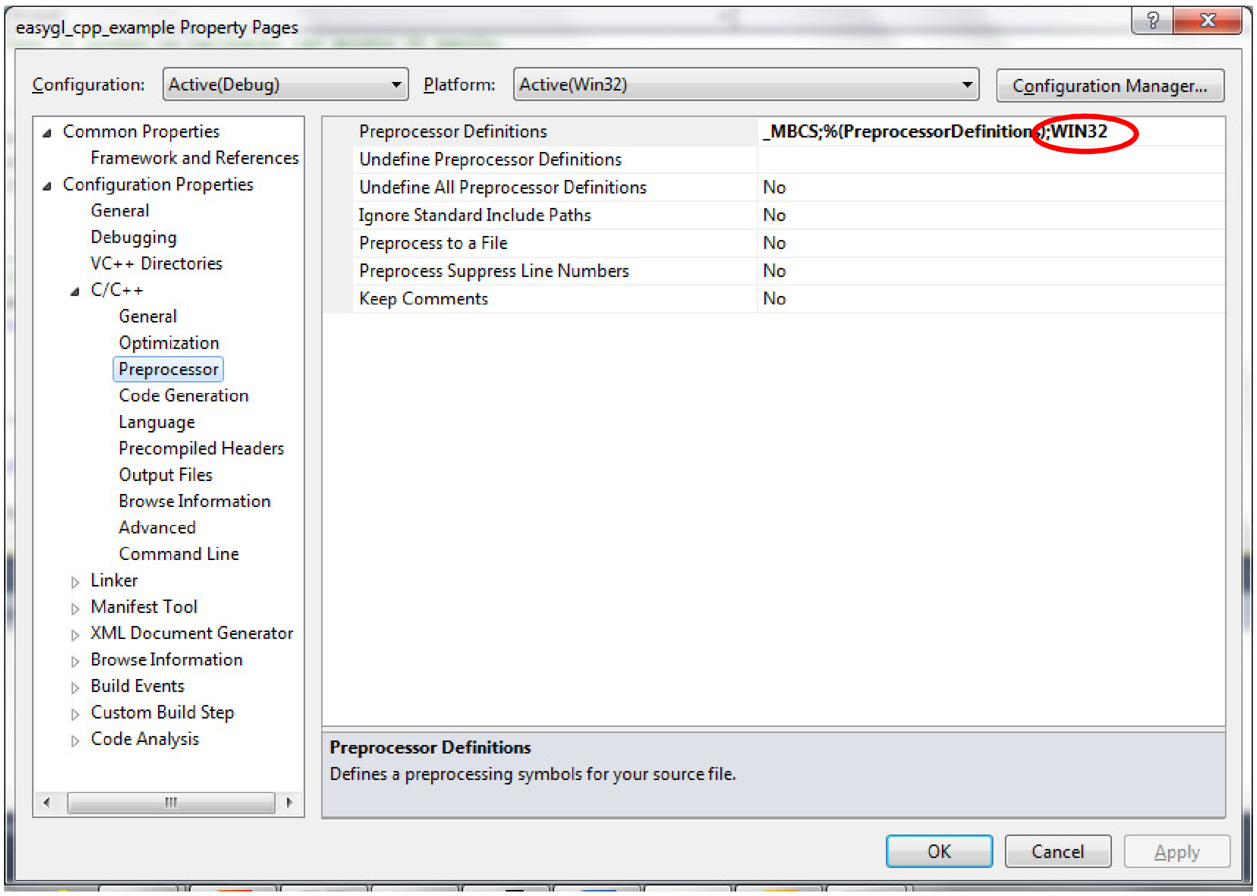
Any source files which use graphics must `#include "graphics.h"` and you must have the graphics.cpp, graphics.h and easygl_constants.h files in your source directory. See the sample makefile for compilation instructions. Compiling the example program should be as easy as selecting the proper platform at the top of the makefile, and then typing make. Alternatively, you can use the Visual Studio Project file to compile if you are working on MS Windows.

You can use the makefile and Visual Studio project file from *example* as models to update the makefile of your program or (if you are working on Windows) Visual Studio project file to include graphics.

- **For Linux**: your makefile needs to #define X11 (*-DX11*) when compiling graphics.cpp, and to specify *-lX11* to the link step so it finds the X11 graphics library files. You also need the X11 development headers; if they are not installed on your machine you need to download them. For Ubuntu, the command is *sudo apt-get install libx11-dev.*
- **For Visual Studio on MS Windows**: you need to add graphics.cpp, graphics.h and easygl_constants.h to your project, and *#define WIN32* in your project by setting it in the MS Visual Studio project under *Project | Properties | Preprocessor | Configuration Properties | C/C++ | Preprocessor | Preprocessor Definitions*, as shown below.

Any code you write to work with easygl will work the same way on Linux and Windows. Mac is also supported, through its X11 support.

Note that if you move to some platform that does not support X11 or Win32 output, you can turn off the graphics with no change to your code by selecting the "NO_GRAPHICS" platform.



Preprocessor Definitions	_MBCS;%(PreprocessorDefinitions);WIN32
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	No
Ignore Standard Include Paths	No
Preprocess to a File	No
Preprocess Suppress Line Numbers	No
Keep Comments	No

Preprocessor Definitions
Defines a preprocessing symbols for your source file.

Figure 1: Defining WIN32 in MS Visual Studio.

Interactive (Event-Driven) Graphics:

See example.cpp for an example of how to use this package. The basic structure to make a window that you can draw to, 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 "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.

```
#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,ytop) = (0,0) to
   // (xr,ybot) = (1000,1000)
   init_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 ();
}


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.
   // ...
}
```

The screen shot below shows the graphics window created.

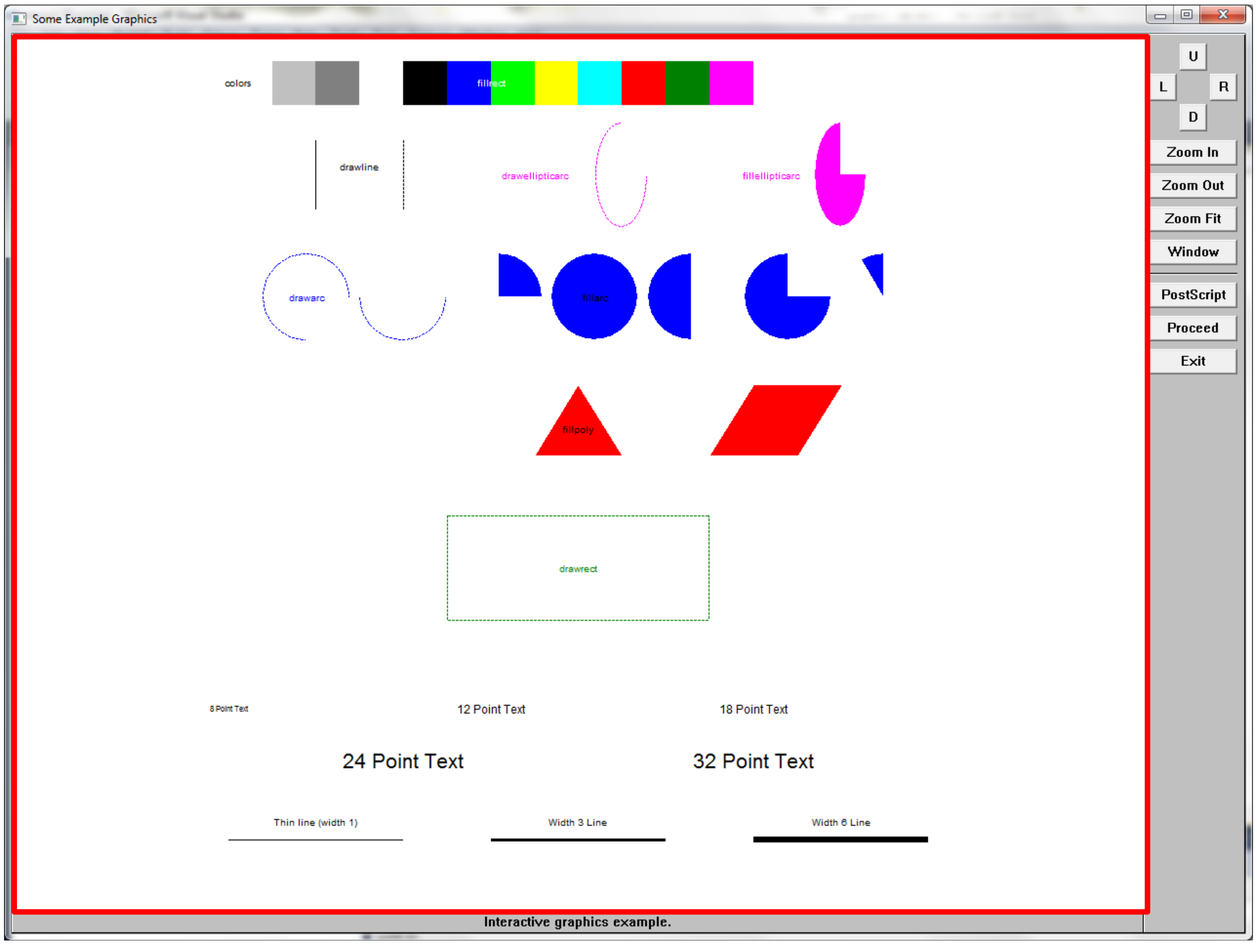
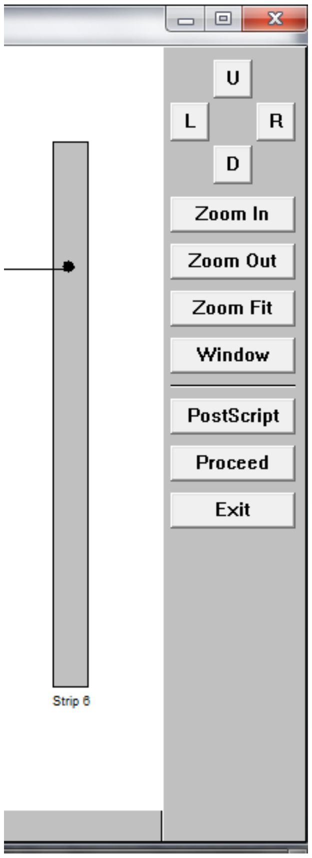


Figure 2: Example graphics window. The area in red is drawn by your code (drawscreen in this case).

Graphics button functions:



Arrow Buttons: pan (shift) 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_world_coordinates).

Window: Click on the diagonally-opposite corners of a box in which to zoom.

Postscript: writes image to pic1.ps (first click), pic2.ps (2nd click) etc.

Proceed: returns from event_loop ().

Exit: Ends program.

Non-Interactive 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,ytop) = (0,0) to
   // (xr,ybot) = (1000,1000)
   init_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 ();
}
```

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. At a high-level, functions exist to:

Set graphics attributes: colour, linewidth, linestyle, etc. These attributes are “sticky”: they affect all subsequent drawing until you change them again.

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

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

Setup and control routines: You’ve already seen most of these. They allow you to set up the graphics, close down the graphics, and automatically respond to pan and zoom events, etc. If you wish, you can also control what “events” (user input) you’d like passed on to callback functions you write.

Known Issues:

You should be able to call init_graphics(), then close_graphics() to close the graphics window, then init_graphics() to open it again and so on. This doesn’t work though; the second init_graphics() call fails. *Workaround*: don’t close the graphics down until your program is done or you are sure you won’t need anymore graphics. I will fix this when I have time ...