Lab 4 Bonus
Graphically display the resistor network (Optional, 5 bonus marks)

1. Purpose and Overview

This is an optional addition to lab 4. If completed and fully working, it is worth an **additional 5 lab marks**. The purpose of this part of the lab is to teach you

1. How to read the interface of and use the functions of a library of code you did not write, and
2. How to do more advanced programming that involves graphics and responding to user input from more than just cin.

You will augment your lab4 `rnet` program by implementing an additional command, `draw`. When this command is typed by the user, you should display all the strips and resistors that have been entered by the user so far. To do this, you will use a graphics package created for this course that works on both Linux (X11) and Windows.

The graphics package we will use is called *easygl* and it is an *event-driven* graphics package. That means that when invoked, it will respond to user input (mouse clicks on buttons) by zooming in and out, panning (shifting) the image, and so on. You will need to write code to:

1. Call the appropriate *easygl* functions to set up the window,
2. Determine the coordinates at which you would like to draw each resistor and strip, and
3. Draw each resistor and strip, by using the drawing functions available in the *easygl* graphics package.

An example of how you might choose to draw a resistor network is shown below. In this drawing, we have drawn each node as a vertical strip (like a breadboard) and each resistor has a circle showing where it connects to a node (breadboard strip). There are many ways to draw a resistor network, but the problem of drawing a network so that is looks "nice" (does not have many components overlapping, and looks logical) is surprisingly difficult, so we recommend you use a simple approach such as this "breadboard" picture. The area contained within the red box must be drawn by your code; the buttons and other parts of the window will be automatically drawn by the easygl package, so you do not need to do anything to implement them.

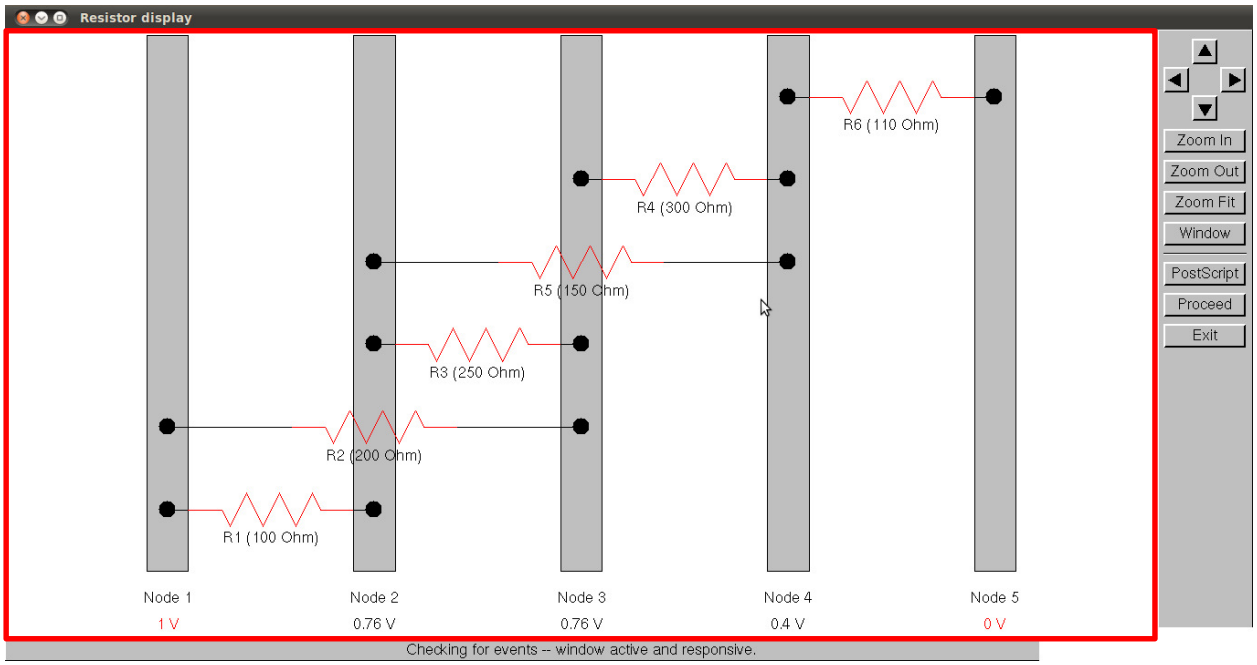


Figure 1: Sample resistor display. The area in the red box will be drawn by your code.

2. Detailed Specification

You can choose to display things differently than in the sample display above, but you should ensure that:

1. You draw all the nodes entered so far by the user which have resistors connected to them.
2. You label each node with its node ID number and current voltage.
3. You draw each resistor entered by the user.
4. You label each resistor with its name and resistance.
5. You show the connections between nodes and resistors.
6. You will need to determine coordinates for each node and resistor so that all the nodes do not draw on top of each other, and the resistors do not draw on top of each other. A simple and effective approach is to draw the first node (lowest ID) at the left side of the window, and move each successive node over some amount to the right. Similarly, you can draw the first resistor at the bottom of the window, and move each successive resistor up some amount to avoid overlap. For complex networks with many nodes and resistors the strips and resistors may become very small and tightly packed – this is fine, as the zoom functions provided by the graphics package will let the user zoom in on areas of interest to make them bigger.

When the user enters the command `draw`, your command parser should recognize the command and

1. Send to cout the text:
   `Draw: control passed to graphics window`
2. Compute drawing coordinates for each node and resistor.

3. Tell easygl the w*orld coordinates* in which you will be drawing, using the *set_world_coordinates* member function. For example, if you wanted to draw in a coordinate system where the bottom-left corner of the window is at (0,0) and the top-right corner of the window is at (1000., 750.), you would call *set_world_coordinates* (0, 0, 1000, 750). Specify a coordinate system that matches the drawing coordinates you computed in step 2, above.
4. Pass control to the easygl graphics package by calling the member function *gl_event_loop ().* The graphics window will now be active, and will respond to mouse clicks on the zoom, window, pan/arrow buttons etc. Your program will not respond to text entry from cin until the user pushes the **Proceed** button in the graphics window.
5. After the **Proceed** button is pushed, the *gl_event_loop ()* member function returns, and the next line of your program will execute. You should print (to cout):
   `Draw: complete; responding to commands again`
6. Your command parser should now continue operating, and read and respond to the next command (insert/print/draw) typed by the user.

An example of session that would result in the network shown in Figure 1 is below.

```
> insertR R1 100 1 2
Inserted: resistor R1 100.00 Ohms 1 -> 2
> insertR R2 200 1 3
Inserted: resistor R2 200.00 Ohms 1 -> 3
> insertR R3 250 2 3
Inserted: resistor R3 250.00 Ohms 2 -> 3
> insertR R4 300 3 4
Inserted: resistor R4 300.00 Ohms 3 -> 4
> insertR R5 150 2 4
Inserted: resistor R5 150.00 Ohms 2 -> 4
> insertR R6 110 4 5
Inserted: resistor R6 110.00 Ohms 4 -> 5
> printNode all
Print:
Connections at node 1: 2 resistor(s)
  R1                     100.00 Ohms 1 -> 2
  R2                     200.00 Ohms 1 -> 3
Connections at node 2: 3 resistor(s)
  R1                     100.00 Ohms 1 -> 2
  R3                     250.00 Ohms 2 -> 3
  R5                     150.00 Ohms 2 -> 4
Connections at node 3: 3 resistor(s)
  R2                     200.00 Ohms 1 -> 3
  R3                     250.00 Ohms 2 -> 3
  R4                     300.00 Ohms 3 -> 4
Connections at node 4: 3 resistor(s)
  R4                     300.00 Ohms 3 -> 4
  R5                     150.00 Ohms 2 -> 4
  R6                     110.00 Ohms 4 -> 5
Connections at node 5: 1 resistor(s)
  R6                     110.00 Ohms 4 -> 5
> setV 1 1
Set: node 1 to 1.00 Volts
> setV 5 0
Set: node 5 to 0.00 Volts
> solve
Solve:
  Node 1: 1.00 V
  Node 2: 0.76 V
```

```
  Node 3: 0.76 V
  Node 4: 0.40 V
  Node 5: 0.00 V
> draw
Draw: control passed to graphics window
Draw: complete; responding to commands again
>
```

Until the **Proceed** button is pressed your program will not take input from cin (between the two draw: output lines).

When the *gl_event_loop ()* member function is called, the easygl graphics package responds to user input (clicking on the Zoom In and Zoom Out button, moving the window around, etc.) by making a *callback* to your program when it needs the graphics redrawn. This callback is to a function:

void easygl::drawscreen ()

which you must implement in some .cpp file. This function should call whatever functions are necessary to draw all the strips and resistors, in the coordinate system you specified.

3. Suggested Code Structure

You will need to create an easygl object in order to create a window to which you can draw. You should make this object global so you can draw to the window anywhere in your program. You will also need to make your `NodeList` (or a pointer to it) global so that the `drawscreen` function can access the resistor network and draw it. A convenient place to make these global variables is in `Rparser.cpp`.

Rparser.cpp:

```
#include "easygl.h"

/************** Global variables ***********************************/
// Create a window with name = "Resistor display" and a white background.
// Making it global so we can draw to it from anywhere.
easygl window ("Resistor display", WHITE);

// Make a global scope pointer to the nodelist so your drawing routine can access it.
// Make sure you set this pointer to a valid value early in your program!
NodeList* g_NodeListPtr;

. . .


/*************** parsing and responding to the draw command *********/
// Suggested function structure for a new routine in Rparser.cpp to handle the
// draw command.

void parse_draw (NodeList& nodeList) {

   float xleft, ybottom, xright, ytop;

   // Compute drawing coordinates for each node & resistor. Update the
   // 4 coordinates (passed by reference) to give appropriate coordinates for
   // the window corners.
   nodeList.set_draw_coords (xleft, ybottom, xright, ytop);

   // Tell the graphics package what coordinate system we will draw in.
```

```
   window.set_world_coordinates (xleft, ybottom, xright, ytop);
   g_NodeListPtr = &nodeList;  // Set up global pointer.
   nodeList.draw ();           // Draw the network

   cout << "Draw: control passed to graphics window.\n";

   // This function will not return until the "Proceed" button is pushed.
   window.gl_event_loop ();
   cout << " Draw: complete; responding to commands again";
}


/** Redrawing the screen when the graphics package calls you (drawscreen callback) **/

void easygl::drawscreen () {

   // Clear the area in the red box in Figure 1 (the graphics area).
   // Need to do this to have a blank area before redrawing.
   window.gl_clearscreen ();

   g_NodeListPtr->draw ();    // Call your routine that does all the work of drawing.
                              // Note how we needed the global pointer to nodeList.
}
```

NodeList.cpp:

```
Void NodeList::draw () {
   // Iterate over Nodes, calling appropriate draw functions (probably in
   // class Node).  See easygl.h for a list of drawing functions, like
   // window.gl_drawline (x1, y1, x2, y2).
   // Iterate over resistors, calling appropriate draw functions (probably in
   // class Resistor).
   // These functions will use new member variables that store drawing coordinates
   // for each Node and resistor.
}


void NodeList::set_draw_coords (float& xleft, float& ybottom, float& xleft,
                            float& xright) {
  // Iterate over Nodes and resistors, setting drawing coordinate member variables.
  // Return the minimum and maximum coordinates you will use, so they can be
  // passed to the easygl package via set_world_coordinates().
}
```

4. Graphics Library Reference and Example

When you start your program, a graphics window will be created, but the graphics window will not respond to input until your program calls `easygl::gl_event_loop()`. Until then, the text terminal from which you invoked your program will be active and you can type commands (insertR/printR/draw) as usual. If the graphics window is hiding your terminal window, move it out of the way by clicking and dragging its top (title bar) area. On MS Windows this will take a few tries and a

few seconds, as MS Windows does not move windows that are not checking for events very well. Be persistent and patient.

The graphics library you will use in this lab consists of 5 files. You will need to add all 5 of these files to your netbeans project.

- **easygl.h**: interface to the graphics package
  - #include this file to use the interface
  - Read it to understand the interface.
- **easygl_constants.h**: defines list of colours for graphics package.
  - Read it to see the list of colours. No need to #include.
- **easygl.cpp, graphics.cpp, graphics.h**: Implementation files.
  - No need to ever read them, or #include them.

An example application using easygl has also been provided so you can see what the various drawing commands do. This application is called *example* and is in a subdirectory called *example/*. It provides one file (`example.cpp`) that uses easygl, a `makefile` for Linux, and a project file for Visual Studio (Windows).

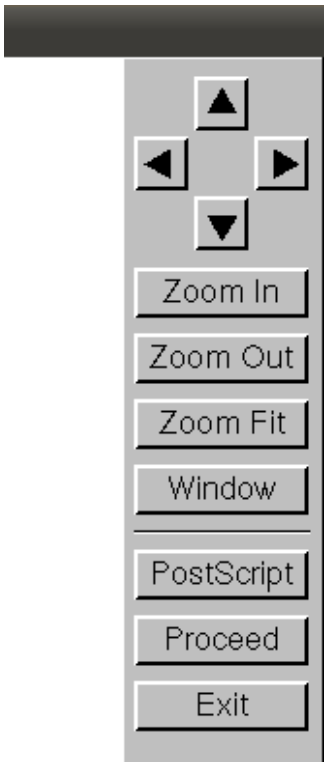
To run the example file on Linux (ECF), open a terminal and type:

```
cd example      // Go to proper directory
make            // Build example program
./example       // Run it. Click on the buttons to see what they do.
```

On Microsoft Windows, you should open the Visual Studio project file, and then build and run from the Visual Studio interface.

The function of the various buttons are indicated in Figure 2. Most buttons adjust the view (Zoom In and Zoom Out, etc.). You can also use your mouse scroll wheel to zoom in and out, and holding down the scroll wheel while you move the mouse will pan (shift) the view. When pressed, the **Proceed** button returns control from the graphics package to your program so you can continue parsing commands.

Any code you write to work with easygl will work the same way on Linux and Windows.



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 gl_event_loop ().

Exit: Ends program.

Figure 2: Function of graphics package buttons.

5. Compiling the Graphics Package into Your Program

Since this program includes graphics, it requires the compiler (specifically the *link* step of compilation) to include the low-level graphics library, which is called the X11 library, in the list of libraries it searches. This requires one extra option to be passed to the compiler.

- **NetBeans on Linux (e.g. ECF):** As shown in **Error! Reference source not found.**, type **–lX11** in the *File | Project Properties | Build | Linker | Additional Options* field.
- **Command line on Linux (e.g. ECF):**
  g++ -g –Wall –lX11 *.cpp –o rnet
- **Apple MacIntosh:** add –lX11 to your linker options, in the same way as for Linux computers.
- **Microsoft Windows with Visual Studio:** X11 is not needed; instead you will tell the easygl graphics package that you are compiling on MS Windows by entering **WIN32** in the *Project | Properties | Preprocessor | Configuration Properties | C/C++ | Preprocessor | Preprocessor Definitions* box in your MS Visual Studio project.
- **Microsoft Windows with NetBeans:**
  1. Tell the easygl graphics package you are compiling for MS Windows by adding **WIN32** to *File | Project Properties | C++ Compiler | Proceprocessor Definitions*.
  2. Tell NetBeans where to find the low-level MS Windows graphics library to which easygl interfaces. Click on *File | Project Properties | Linker | Libraries | Add Library* and select the

path to your gdi32 library. With the default cygwin installation this would be in c:\cygwin\lib\w32api\libgdi32.a.

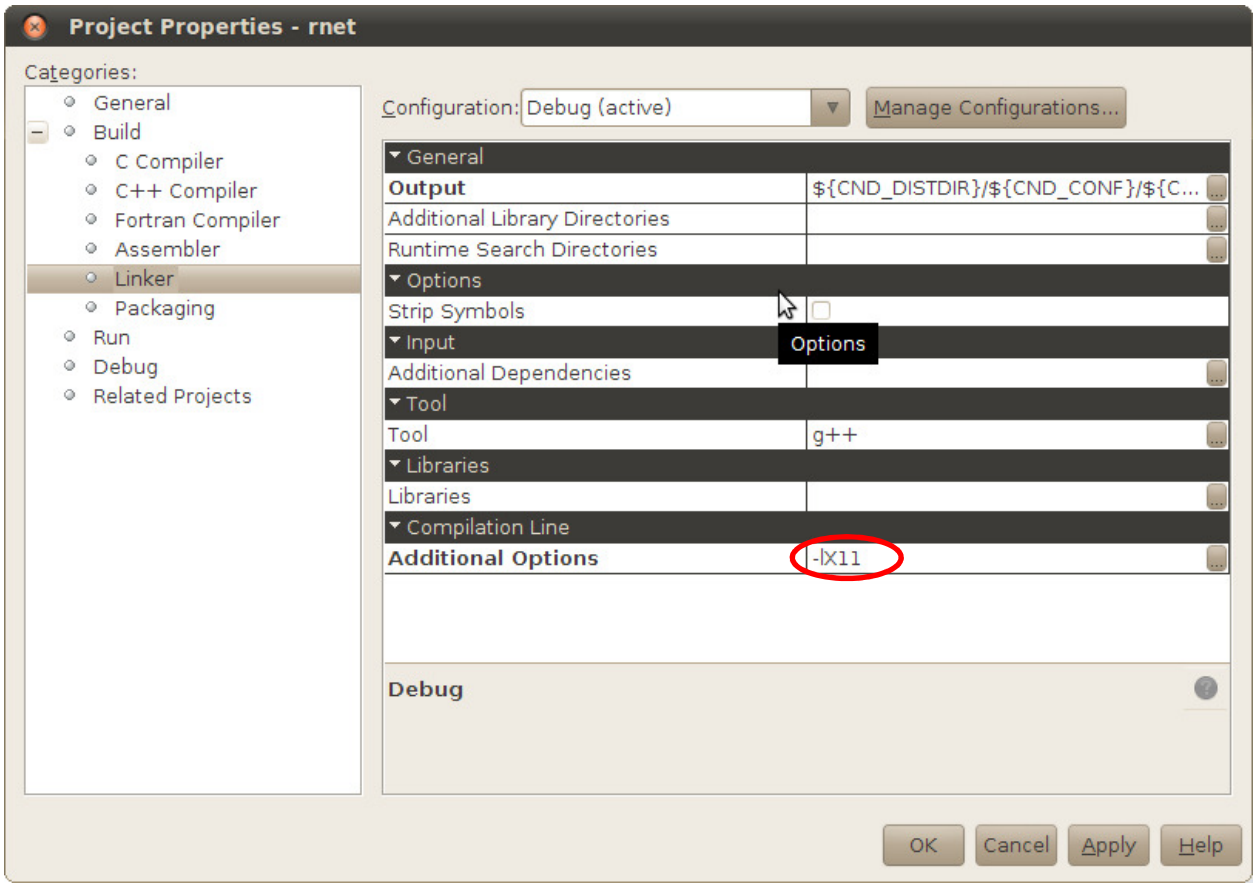


Figure 3: Compiling with graphics in NetBeans on a Linux system: add the circled *–lX11* setting.

6. Procedure and Deliverables

We strongly suggest you attempt this bonus lab only once your lab4 is fully working, debugged and submitted.

You will submit this assignment as Lab 7. Start by making a new directory, and copying all your lab4 files into it.

```
mkdir lab7
cp lab4/* lab7
```

Add the five graphics library files you have been given to this directory, and then create a new NetBeans project that builds a program called rnet, and add all the source files to it. You can use:

```
~ece244i/public/exercise 7 rnet
```

To run some test cases through your program. Remember that you will have to press **Proceed** after each `draw` command to continue the testing by exercise.

Your program should consist of these files:

- Main.cpp
- Rparser.cpp and Rparser.h
- ResistorList.h and ResistorList.cpp
- Resistor.h and Resistor.cpp
- Node.h and Node.cpp
- NodeList.h and NodeList.cpp
- easygl.h, easygl_constants.h, graphics.h, easygl.cpp and graphics.cpp

You will not need to make any changes to the 5 easygl files. Submit your program using

```
~ece244i/public/submit 7
```

When your assignment is graded, an Xserver (graphical display) will be active, so the graphics window will open correctly.