Week8 : Introduction to Graphics Programming

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 8

Outline

Outline

Why Graphics?

The notes for week 8 are based on chapter 12 of Bjarne Stroustrup, "Programming Principles and Practice Using C++ ," 2nd edition.

- Most problems in science require us to make graphics to understand the concepts better, and to analyze new data or formulae
- Graphics are fun – we get to see the result of our code on the screen in an eye pleasing way
- Graphics have a lot of interesting code that we can learn from, and is a place where the use of object-oriented concepts shines
- Graphics libraries are full of good examples of code design
- A typical graphics library provides a fairly large set of code to read and learn from
- Like learning English, it helps to read lots of good code to learn what good code looks like
- There are a number of non-trivial concepts commonly used in graphics libraries that are useful to learn in a class rather than on your own later

Graphics Libraries

- There are a large number of graphics libraries and tool kits, and picking one was not so simple
- Some popular C++ GUI libraries are `GTK+`, `Qt`, `WxWidgets`, `fltk`...
- We have chosen `fltk` because:
  - It is free and open-source
  - It is lighter weight than GTK+, Qt and wxWidgets
  - It uses some nice features of C++ such as templates, exceptions, and namespaces
  - Its modest size makes it relatively easy to learn
  - It is a cross-platform widget (can be used on Windows, linux and mac)
- One disadvantage of `fltk` is that it does not have the look and feel of the underlying O/S (instead it looks the same on all O/S)

Notes on installing `fltk`

- These slides summarize the instructions at http://t2kwinnipeg.uwinnipeg.ca/~jamieson/courses/scicomp/
- The 2L14 lab computers do not have `fltk` installed, so we need to install it in the directory with the code we want to compile
- To get a copy of the `fltk` library compiled for the 2L14 lab computers unpack the zip file at http://t2kwinnipeg.uwinnipeg.ca/~jamieson/courses/scicomp/project2/fltk.zip into the directory with your source code

Notes on installing `fltk` – at home

One option for Windows systems at home or on a laptop is to get a copy of the tools, with some extras by unpacking the zip file: http://t2kwinnipeg.uwinnipeg.ca/~jamieson/courses/scicomp/usbstick.zip into the location you want the course tools. This zip file comes with the MinGW compiler, fltk libraries, and course notes up to today. Alternatively...

- **If you installed MinGW on your computer at home:**
- make sure you set the system path to also include msys (typically at C: \ MinGW \ msys \ 1.0 \ bin)
- Now install `fltk` from source:
  - Download http://fltk.org/pub/fltk/1.3.3/fltk-1.3.3-source.tar.gz
  - Unpack its contents into C: \fltk-1.3.3 using a tool such as 7-zip (http://www.7-zip.org/a/7z1604-x64.exe)

Notes on installing `fltk` – compiling

- Now compile and install the `fltk` libraries from a windows command line (cmd):

```
C:\> bash
bash-3.1$ cd /c/fltk-1.3.3
bash-3.1$ ./configure --enable-threads
    --enable-localjpeg --enable-localzlib
    --enable-localpng
bash-3.1$ make
bash-3.1$ make install
bash-3.1$ test/demo
bash-3.1$ exit
```

- Note that bash is provided by `msys` in the `MinGW` installation, and is the typical command line in linux type systems
- Also note that the first line `cd /c` is the C: drive
- The line test/demo runs the `fltk` demo program

A first look at using `fltk`

- A short program to open a window, and to draw a box in the window with some text in it is in `fltkTest.cpp`:

```
1  #include <FL/Fl.H>
2  #include <FL/Fl_Box.H>
3  #include <FL/Fl_Window.H>
4
5  int main(){
6    Fl_Window window( 200, 200, "My Fltk Window" );
7    Fl_Box box(0, 0, 200, 200, "Hello World and
         Hello Fltk");
8    window.show();
9    return Fl::run();
10 }
```

- `Fl::run()` starts the GUI's event loop
- Please be careful to put the capitalization in the #include lines as it is there
- While windows may ignore the capitalization, other systems don't

Compiling with a library

- This is our first encounter with using code from a library
- How to link your code with the code provided by the library depends a bit on the library
- The `fltk` library provides a command called `fltk-config` to help get compiler flags
- If you are using MinGW and have installed `fltk` from source you can have a makefile using fltk-config:

```
1 CFLAGS=--std=gnu++11 -Wall -I/usr/local/include
      /usr/local/bin/fltk-config
     --cxxflags
2 LINK=-L/usr/local/lib /usr/local/bin/fltk-config --ldflags
     --use-images
3 all: fltkTest.exe
4 fltkTest.exe: fltkTest.o
5         g++ fltkTest.o ${LINK} -o fltkTest.exe
6 fltkTest.o: fltkTest.cpp
7         g++ ${CFLAGS} -c fltkTest.cpp -o fltkTest.o
```

- The new part in the makefile is to include information to g++ of where to find the libraries, and which files to link to our code

Compiling with a library

- To link to the library if you downloaded the zip file to the same directory as your code (on 2L14 computers) the makefile becomes:

```
1 INCLUDE=-I.
2 LIBS=-L.
3 CFLAGS=${INCLUDE} -std=gnu++11 -Wall -mwindows
      -DWIN32 -DUSE_OPENGL32 -D_LARGEFILE_SOURCE
      -D_LAREFILE64_SOURCE -mconsole
4 LINK=${LIBS} -mwindows -lfltk_images -lfltk_png
      -lfltk_z -lfltk_jpeg -lfltk -lole32 -luuid
      -lcomctl32 -mconsole
5 fltkTest.exe: fltkTest.o
6         g++ fltkTest.o ${LINK} -o fltkTest.exe
7 fltkTest.o: fltkTest.cpp
8         g++ ${CFLAGS} -c fltkTest.cpp -o fltkTest.o
```

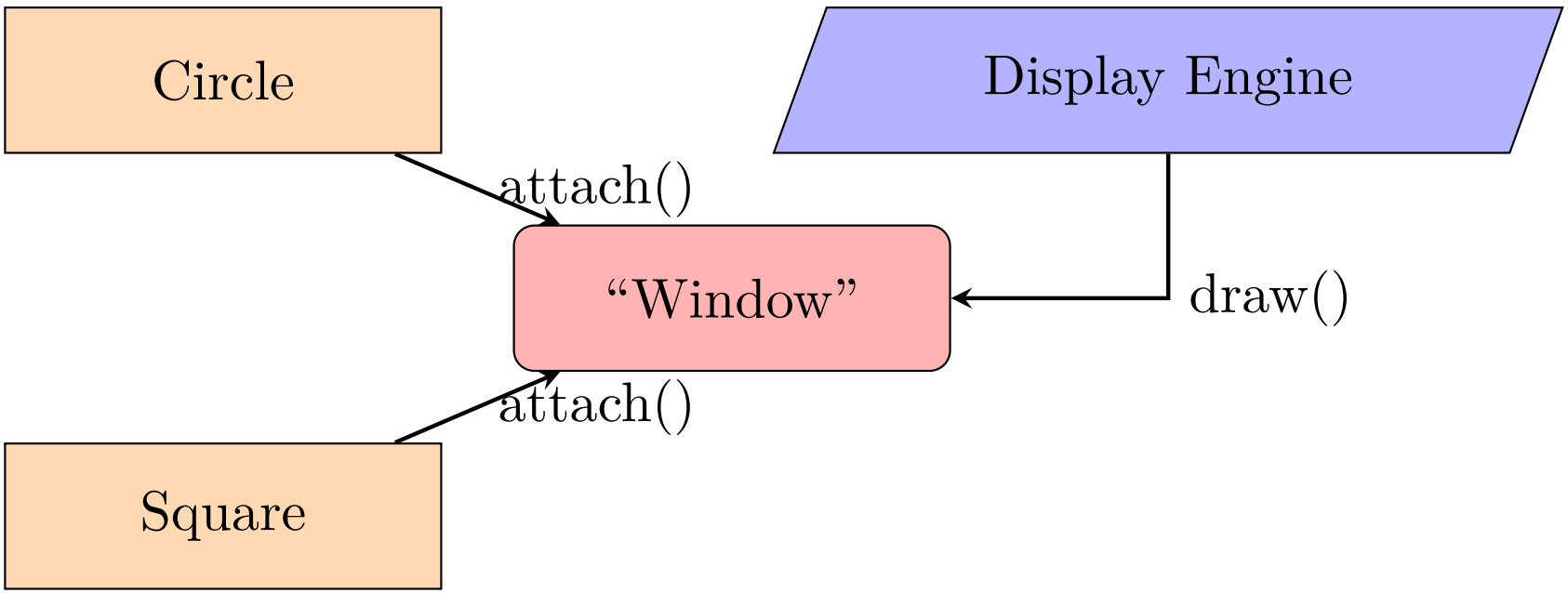
- Here we use . to specify to look in the current directory for the include files and library files (.a files)
- Also we specify the libraries to link with -llibname...

A display model

- This section will present:
  - Screen coordinates
  - Use of `Shapes` such as `Line`, `Lines`, `Polygon`, `Axis` and `Text`.
- We have used `<iostream>` for reading and writing streams of characters to a terminal where positions were character sized
- Instead lets look at graphics that are pixel based, and are in a graphics "window" rather than text window on the screen.
- Basic model – attach shape objects to a "window" then once shapes are attached draw them.

A display model

- Program itself is the display engine / GUI library
- GUI library handles drawing objects we tell it to put on the screen



A second example

Example drawing a triangle

```
1  // get access to window library
2  #include "Simple_window.h"
3  // get access to graphics library
4  #include "Graph.h"
5  int main(){
6    using namespace Graph_lib; // holds graph classes
7    Point tl{100, 100}; // Point for top left corner
8    // Make a simple window
9    Simple_window w{tl,600,400,"My simple window");
10   Polygon tri; // make a shape (polygon)
11   // add three points for vertices of triangle
12   tri.add( Point{300,200} );
13   tri.add( Point{350,100} );
14   tri.add( Point{400,200} );
15   tri.set_color( Color::red );
16   w.attach( tri ); // add triangle to window
17   w.wait_for_button();
18   return 0;
19 }
```

Result of second example

Running the first example will bring up a window 100 pixels down from the top and 100 pixels right of the top left of the screen.

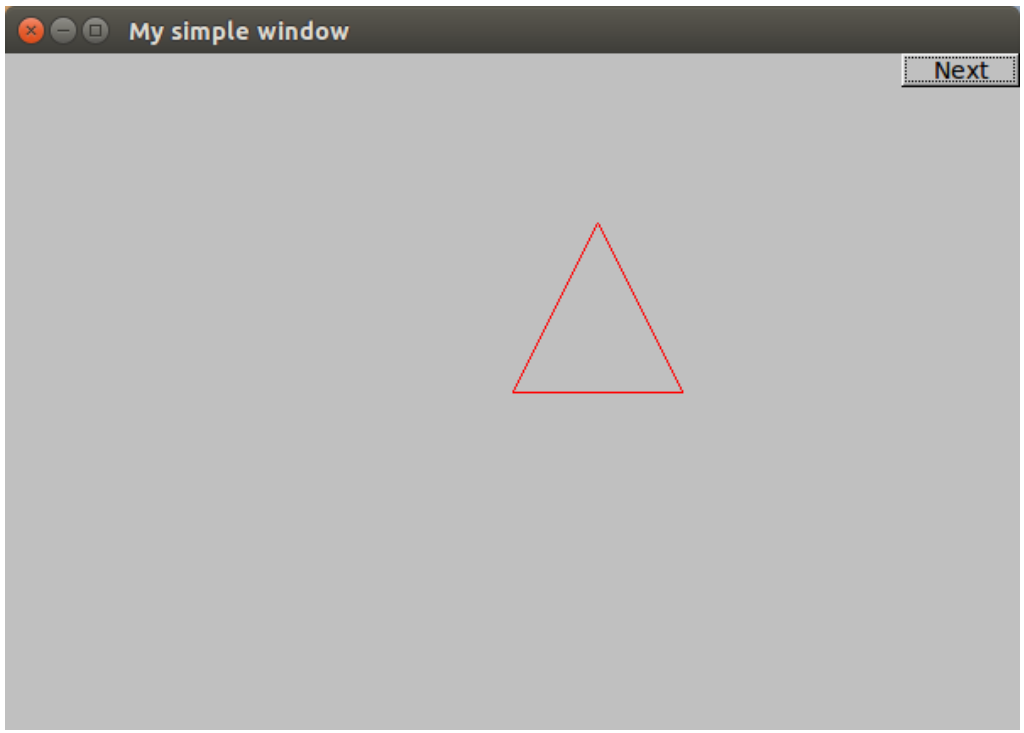


Figure : Result of first example

A few notes on second example

- Rather than directly accessing the interfaces provided by `fltk` we are using some simplified interface
- The simplified interface is in the include files `Simple_window.h` and `Graph.h`
- All of the classes and definitions in the simplified interface are in the namespace `Graph_lib`
- To define a window we pass it the arguments:

```
1 Point tl{100,100}
2 Simple_window w( tl, 600, 400, "My simple window"
      );
```

  - the point on the screen of the top-left of the window
  - the window width in pixels (600)
  - the window height in pixels (400)
  - the title on the window
- The variable `w` is an object of type `Simple_window`

Notes on second example continued

- Next we put a triangle on the window by first defining a `Polygon` with three points

```
Polygon tri;
// add first corner of triangle
tri.add( Point{300,200} );
// add second corner of triangle
tri.add( Point{350,100} );
// add third corner of triangle
tri.add( Point{400,200} );
```

- The first line makes an instance of a Polygon variable `tri` that starts out empty
- `Polygon` begins empty and we can add as many points to it as we like to define a shape
- We added three x, y points that are coordinates within the window

Notes on second example continued

- The color of the polygon was changed to red by calling the method `set_color` of the class `Polygon`

```
1 poly.set_color( Color::red );
```

- Then the triangle is "attached" to the window:

```
1 w.attach( tri ); // connect triangle to window
```

- Up to this point nothing is displayed on the screen
  - We created the window (of `Simple_window` type)
  - We created a red `Polygon` with the shape of a triangle
  - We attached the triangle to the window
  - But we haven't asked for it to be displayed on the screen

Giving control to the graphics system

- The final line of the program:

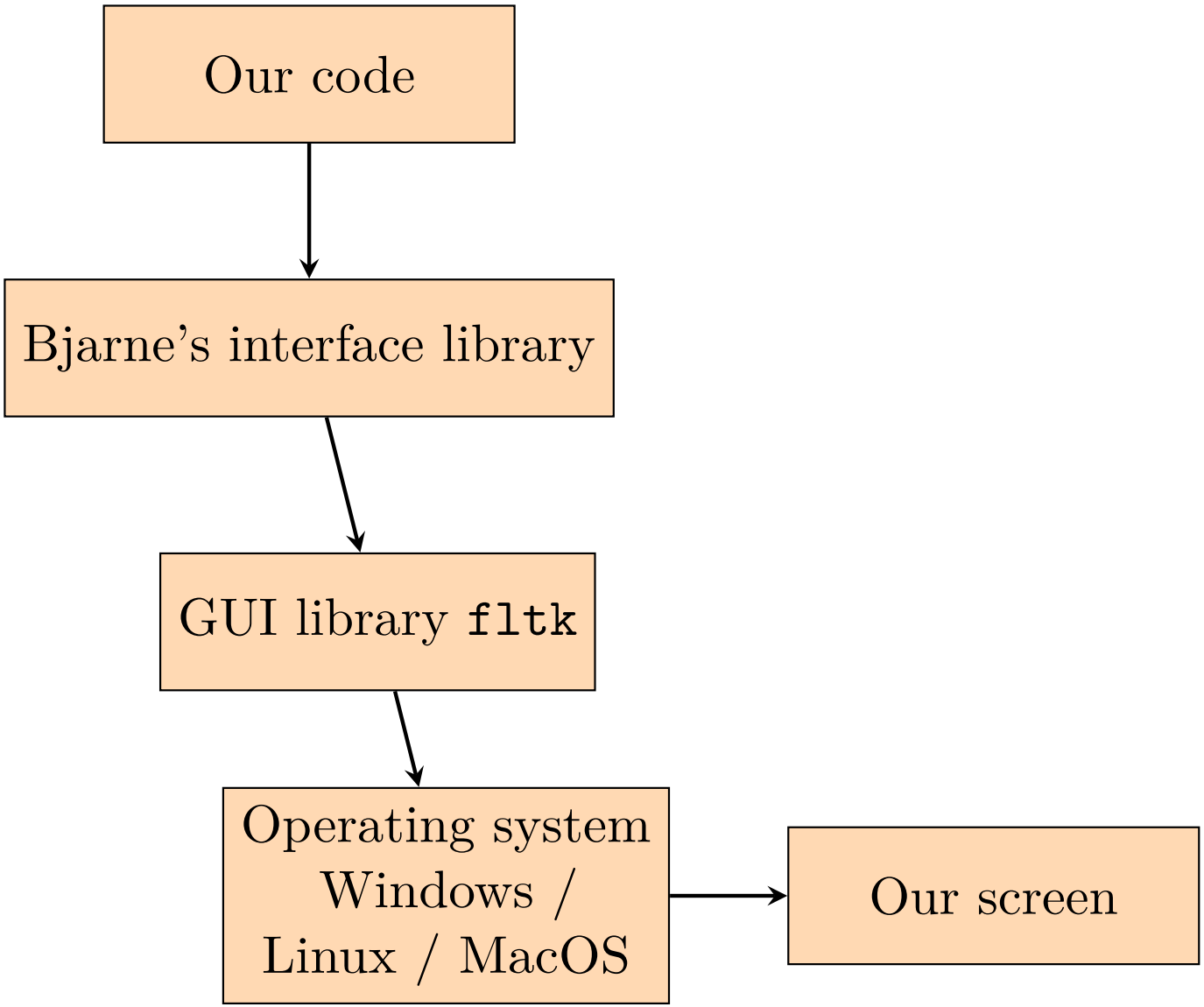
```
1 w.wait_for_button(); //gives control to display
      engine
```

- For GUI systems to display to the screen give it control of the system
- The `wait_for_button()` gives control to the system, and only returns to the `main` program after you click the "Next" button
- Note that the "Next" button is built into the `Simple_window` class
- We can use the "Next" button in next examples to move from one display to the next
- Notice that in the image of the window, I ran in Ubuntu which provided the buttons on the top left of the window – that part of the window is system dependent.

Using the GUI library

- By using a GUI library, rather than operating system's built in facilities:
  - We don't have to deal directly with a lot of messy details
  - We aren't limited to running on a single operating system
- Note that C++ doesn't have a standard GUI, so we need not be tied to a particular GUI
- We are using the GUI interface classes from Bjarne Stroustrup that wrap the `fltk` GUI library
- Bjarne's interface library is about 600 lines of code
- Method of making simplified interfaces can be applied to any library of code

Interface library code layering approach

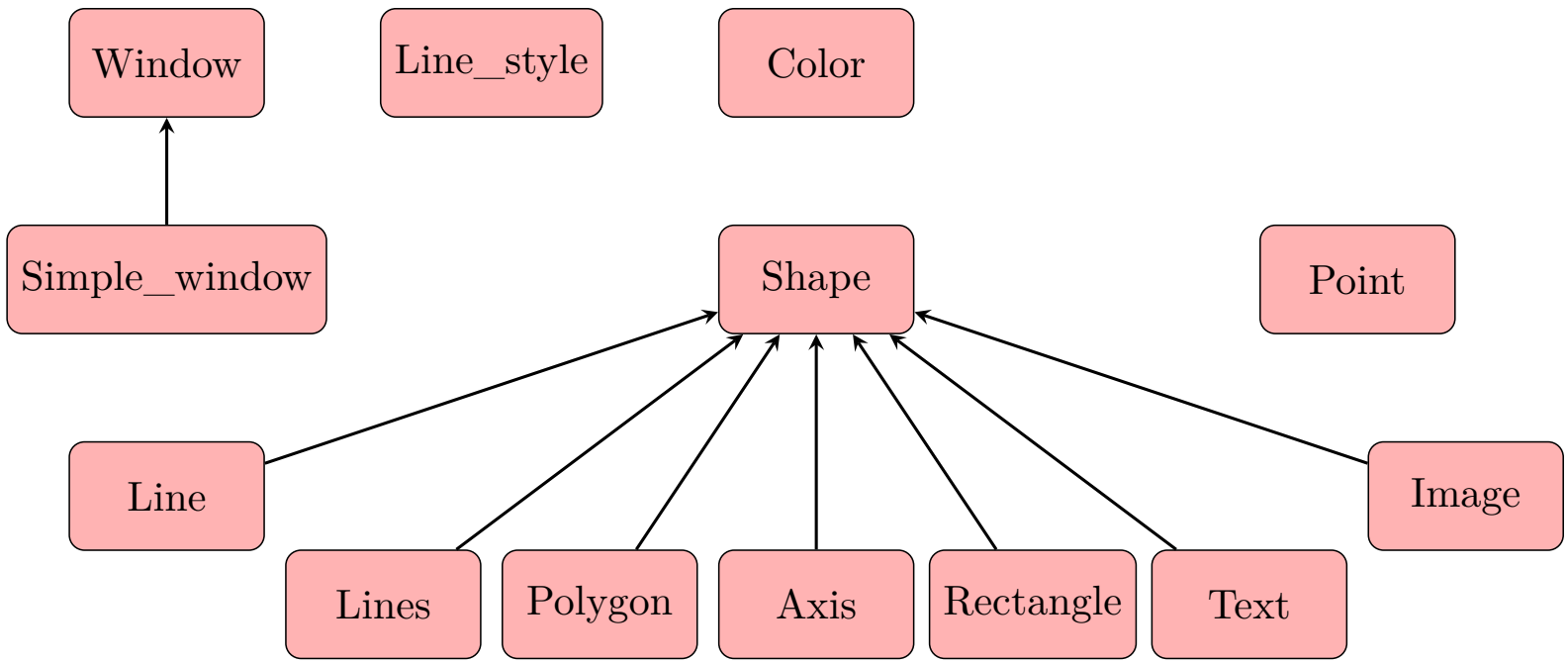


Screen coordinates

- Computer screen is rectangular and has dimensions measured in pixels
- The top left of the screen is at the coordinates $x, y = 0, 0$
- The x-axis values increase going to the right on the screen and the y-axis values increase going down the screen
- The y coordinate growing downwards is a bit awkward compared to our usual notion of y-axes on graphs in math and physics
- The size of the screen in pixels depends on the monitor being used
- Typical screen sizes are 1024×768, 1280×1024, 1400×1050, and 1600×1200
- The `Simple_window` we defined is 600×400 of the overall screen, and is addressed 0-599 (left to right) and 0-399 (top to bottom)
- Note that the window frame is not counted in the pixel count – that increases the size of the window on the screen

Shapes

- The set of shapes available for drawing is shown in the figure below, where the arrow indicates that the pointing class can be used wherever the class pointed to is required
- Later we will add interaction classes `Button`, `In_box`, `Menu`, etc.
- Could easily add many more classes, but interface is kept minimal



Using `Shape` type objects

- First step in using the `Shape` objects is to include the relevant include files
- Include "Window.h" for a plain window, or "Simple_window.h" for a window with a "Next" button
- Include "Graph.h" which has facilites for drawing shapes in windows
- Next the `main()` function contains the code and deals with exceptions, for example:

```
int main(){
  try {
    // put code here to make window(s)
    // and add shapes etc to it
  } catch ( exception &e ){
    // some error reporting
    return 1;
  } catch (...) {
    // more error reporting
    return 2;
  }
  return 0;
```

Using `Shape` type objects

- To compile the code on the previous slide, need to have `exception` defined (which is included if we use `std_lib_facilities.h`, or could directly use <stdexcept>
- Now we can make a blank window for example inside the try block:

```
Point tl{100,100};// near top left
Simple_window w{tl, 600, 400, "Empty window"};
// above window has its top-left corner 100,100
// below and to right of top-left of screen
// window is 600x400 pixels with title "Empty
    window"
w.wait_for_button(); // display window
```

- Above example included `Simple_window.h` and used `Point` which is used to hold two integers as x, y coordinate point

Resulting empty window

Similar to previous example, here we make an empty window, 100 pixels down from the top and 100 pixels right of the top left of the screen. It has a “Next” button to move on to whatever happens next in the program.

Figure : Resulting empty window

Adding an `Axis`

- A graph without an axis is not very useful
- The `Axis` shape takes an axis orientation, a starting point on the screen, the length of the axis in pixels, and the number of divisions (notches on the axis), and a label

```
Axis xax { Axis::x,          // horizontal axis
           Point{20,300}, // start at 20,300
           280,              // 280 pixels long
           10,               // 10 notches
           "x axis" };       // axis label
w.attach( xax ); // attach xax to window w
w.set_label("Window with axis"); // relabel window
w.wait_for_button();
```

- Axis label will appear just below axis
- Should place x-axis near bottom of screen
- Instead of hard coded point, should have symbolic constants such as `y_max_bottom_margin` and `x_max_left_margin`

Resulting x-axis in window

We can see that the x-axis produced has the required 10 notches, and the label “x-axis” as we requested.

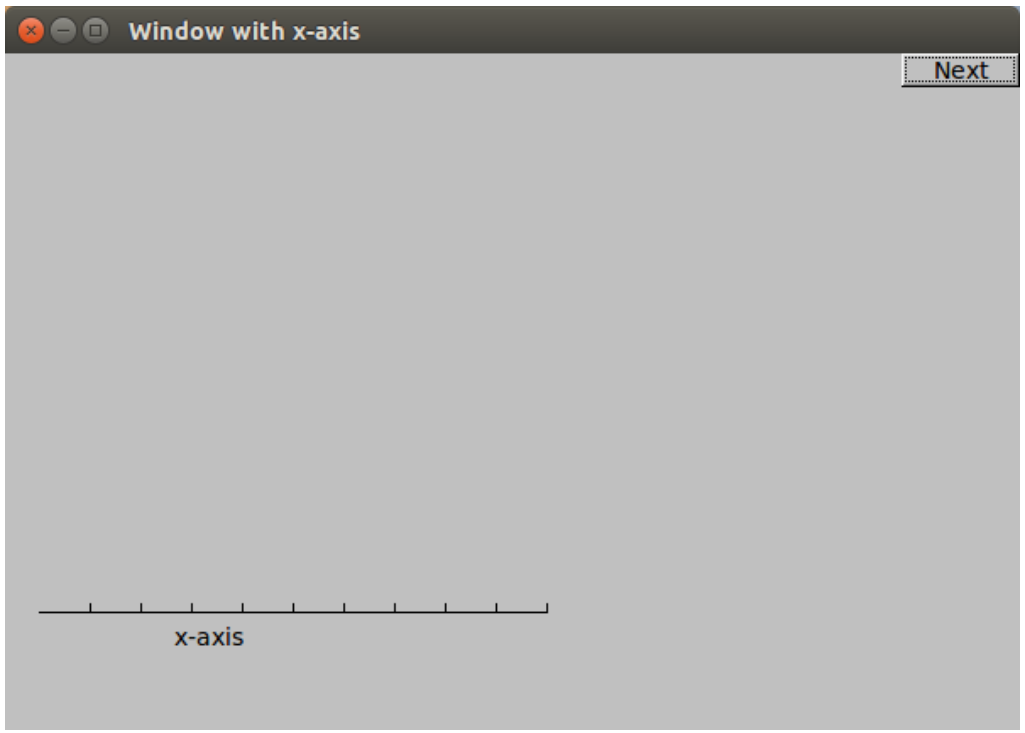


Figure : Resulting window with an x-axis

Adding a y-Axis

- Lets define a y-axis and set some of its properties

```
1 Axis yax{ Axis::y, Point{ 20, 300 }, 280,
2           10, "y-axis label" };
3 yax.set_color( Color::cyan );
4 yax.label.set_color( Color::dark_red );
5 w.attach( yax ); // add axis to window
6 w.set_label( "Window with axes" );
7 w.wait_for_button(); // display!
```

A window with x and y-axes

Its not recommended to have different coloured axes, but this just shows how to set the color of a shape. Also note the American spelling of color is used for the interface classes.

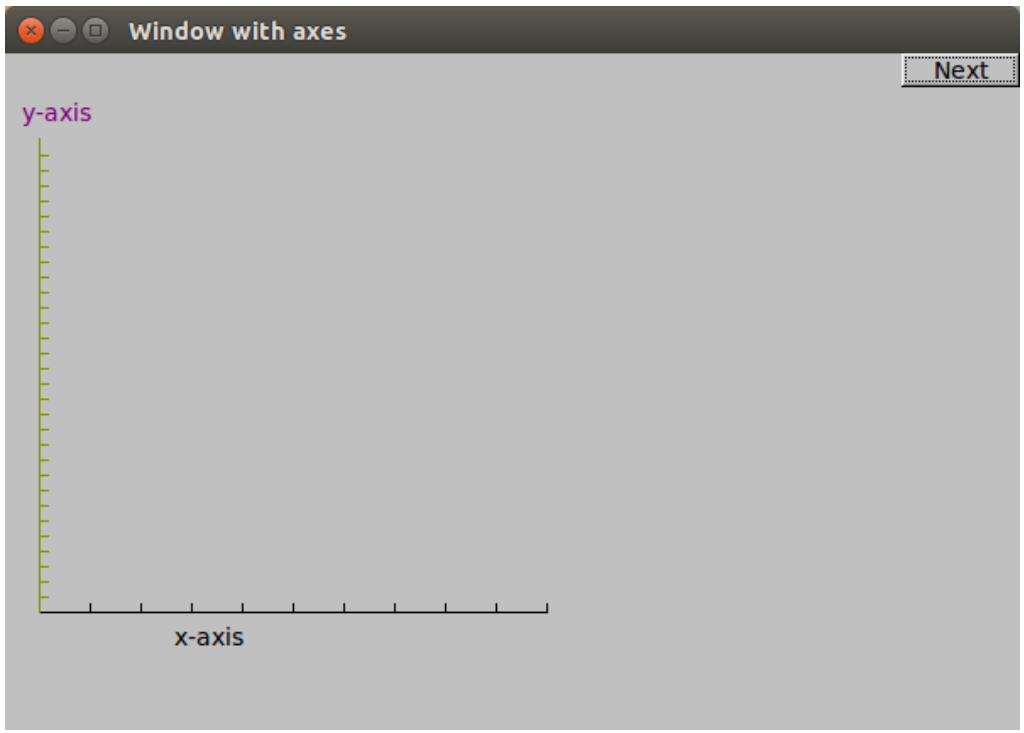


Figure : Resulting window with an x-axis and y-axis

Graphing a function

- Graphics interface also defines a `Function` class that can be used to graph a function:

```
Function f { sin,                    // function
             0,                      // x-min of range
             100,                    // x-max of range
             Point{ 20, 150 }, // Origin of
                 function coords
             1000,                   // Using 1000 points
             50,                     // scale x values
             50 };                   // scale y values
w.attach( f );
w.set_label("sin(x)+c vs x");
w.wait_for_button();
```

- For `Function` to appear right size for screen we apply scaling in x and y directions

Window with function drawn

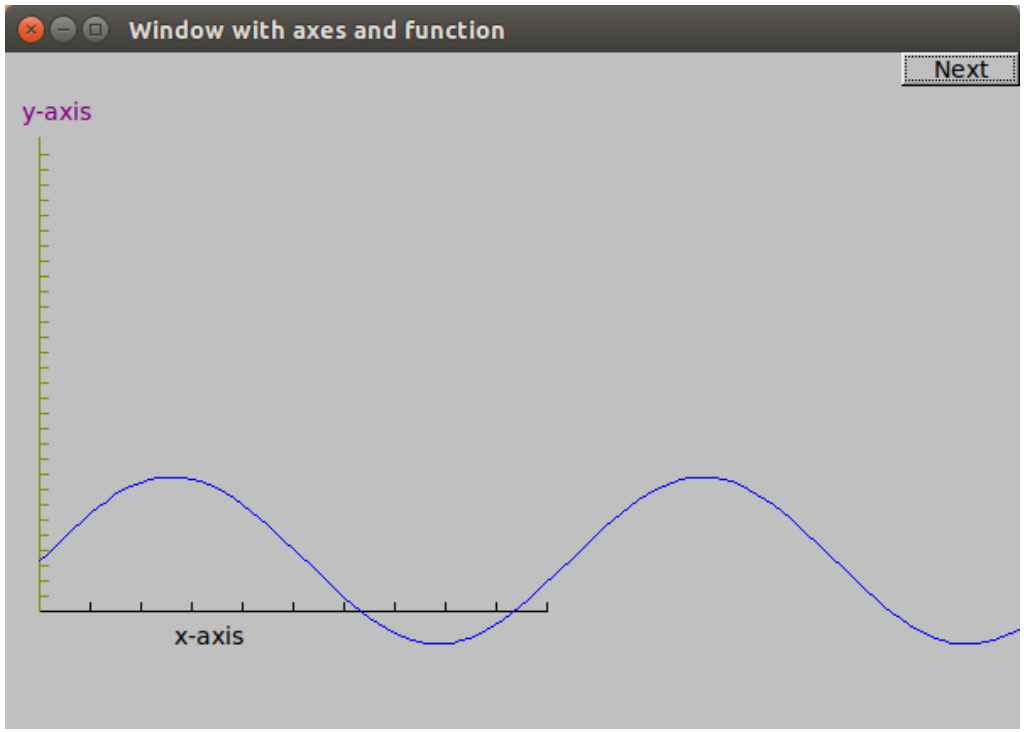


Figure : Resulting window with an x-axis, y-axis and function

- Note how function stops drawing at edge of window
- points drawn outside our window are ignored by the GUI

Polygon

- We can represent a geometric shape with the `Polygon` class
- Represented by a sequence of points connected by lines
- Order of connecting lines follows order of adding points
- For example lets draw a diamond-like shape:

```
1  Polygon poly;
2  poly.add(Point(300,200));
3  poly.add(Point(275,175));
4  poly.add(Point(350,100));
5  poly.add(Point(425,175));
6  poly.add(Point(400,200));
7  poly.set_color(Color::dark_blue);
8  poly.set_style(Line_style::dot);
9  w.attach( poly );
10 w.set_label("Now there is a diamond");
11 w.wait_for_button();
```

Resulting window with diamond shape added

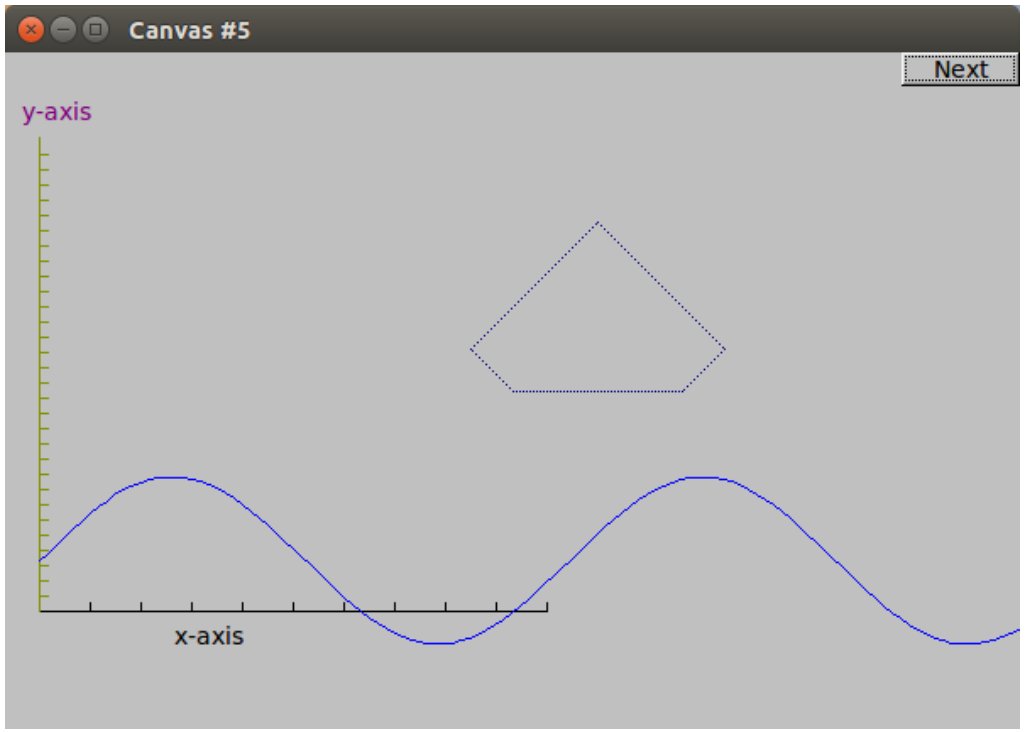


Figure : Resulting window with diamond shape added

- Note `Line_style::dot` line style
- By default line style is solid – here we have made the `Polygon` lines dotted

Rectangle

- Rectangles are special since screen is rectangular, and is simplest shape to deal with
- In this interface the rectangle is defined by its top-left corner, a width and a height in pixels.
- For example lets draw a rectangle:

```
1 Graph_lib::Rectangle r(Point(200,200),100,30);
2 win.attach(r);
3 win.set_label("Added closed polygon rectangle");
4 w.wait_for_button();
```

Resulting window with rectangle added

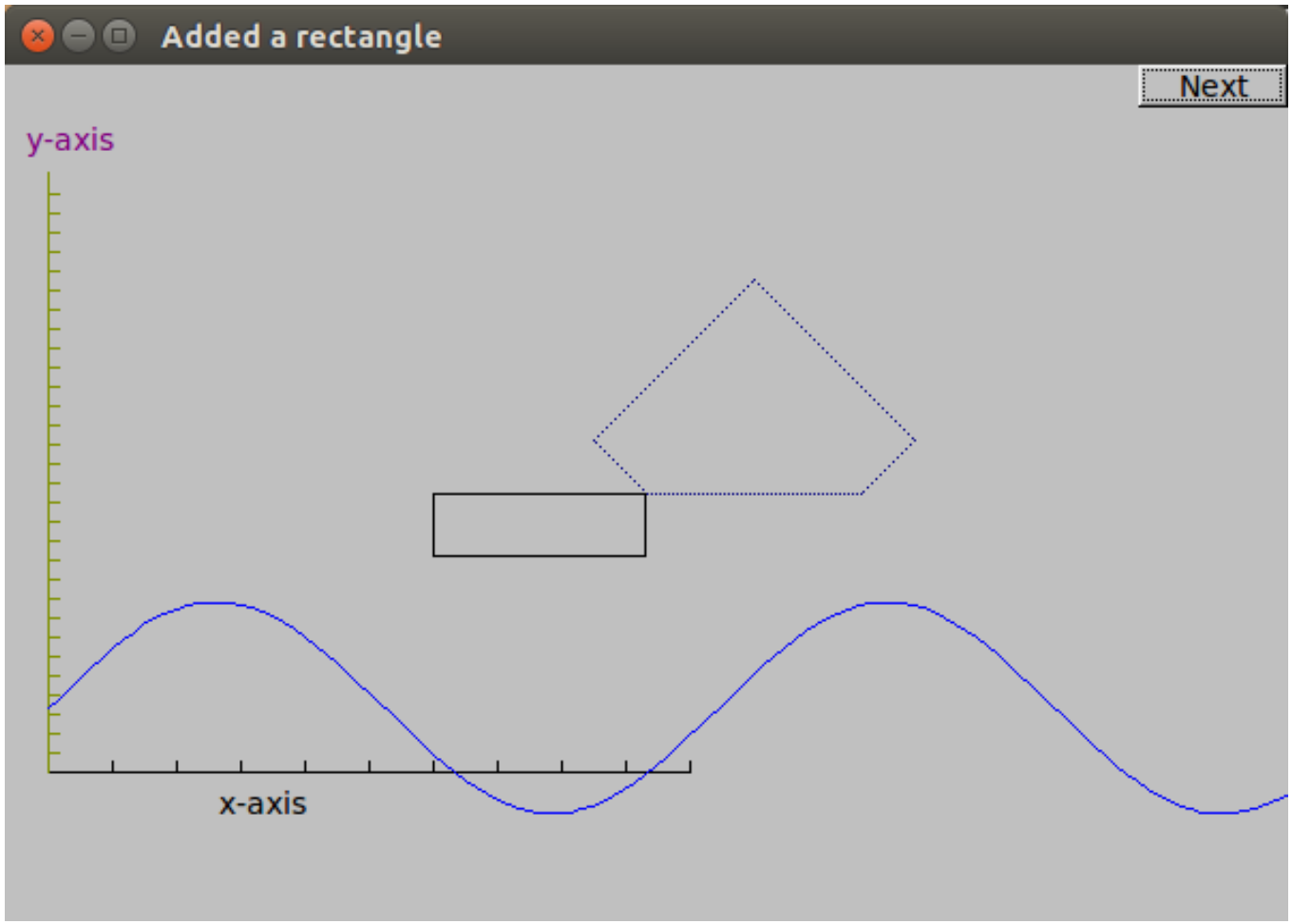


Figure : Resulting window with rectangle added

`Colsed_polyline` as a rectangle?

- It seems we can draw a rectangle using a `Closed_polyline` class
- However it is easy to make a mistake in the coordinates as shown in this example:

```
1 Closed_polyline poly_rect;
2 poly_rect.add(Point(100,50));
3 poly_rect.add(Point(200,50));
4 poly_rect.add(Point(200,120));
5 poly_rect.add(Point(100,100));
6 w.attach(poly_rect);
7 w.set_label("Added a four sided polygon");
8 w.wait_for_button();
```

Added a four sided polygon

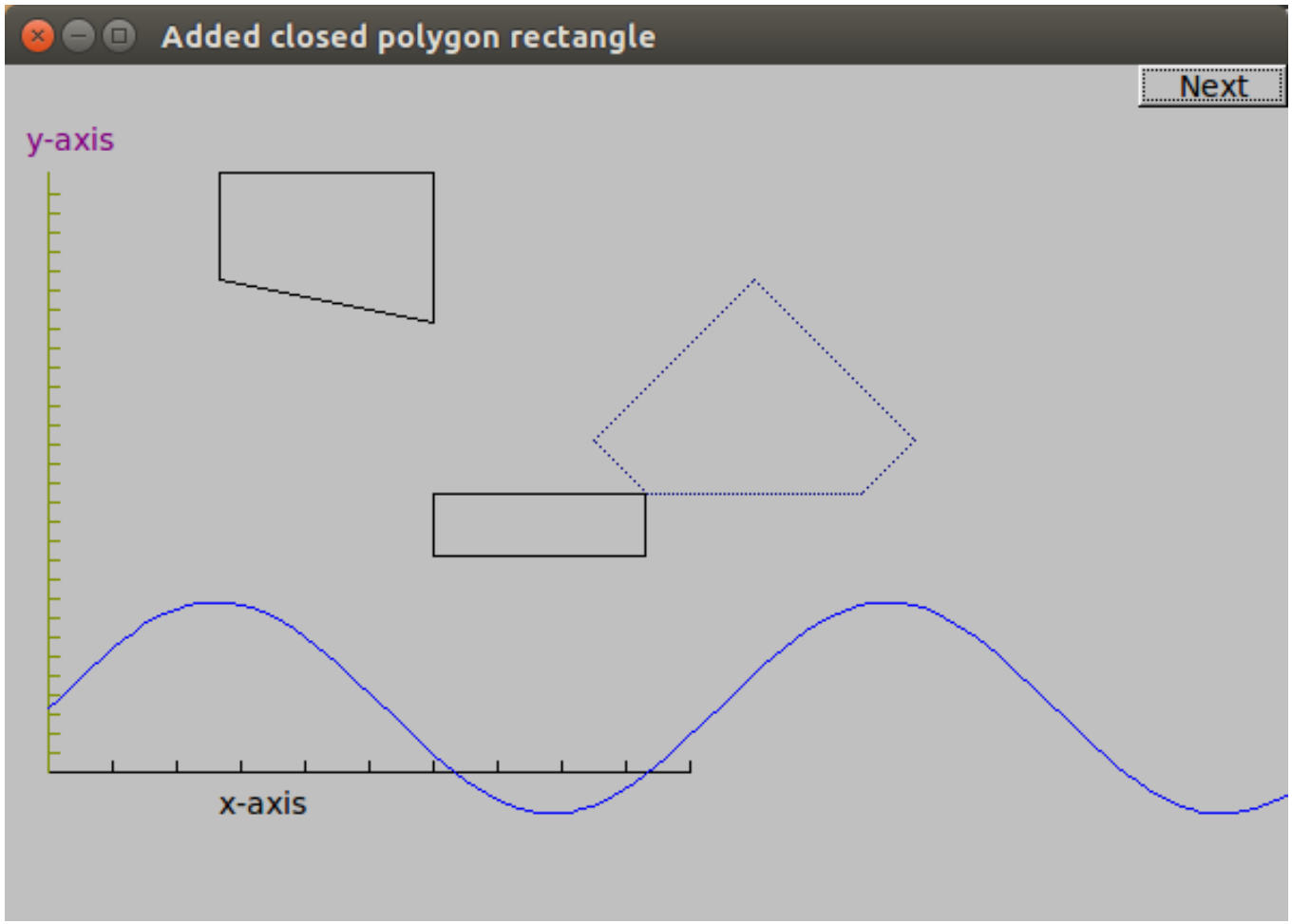
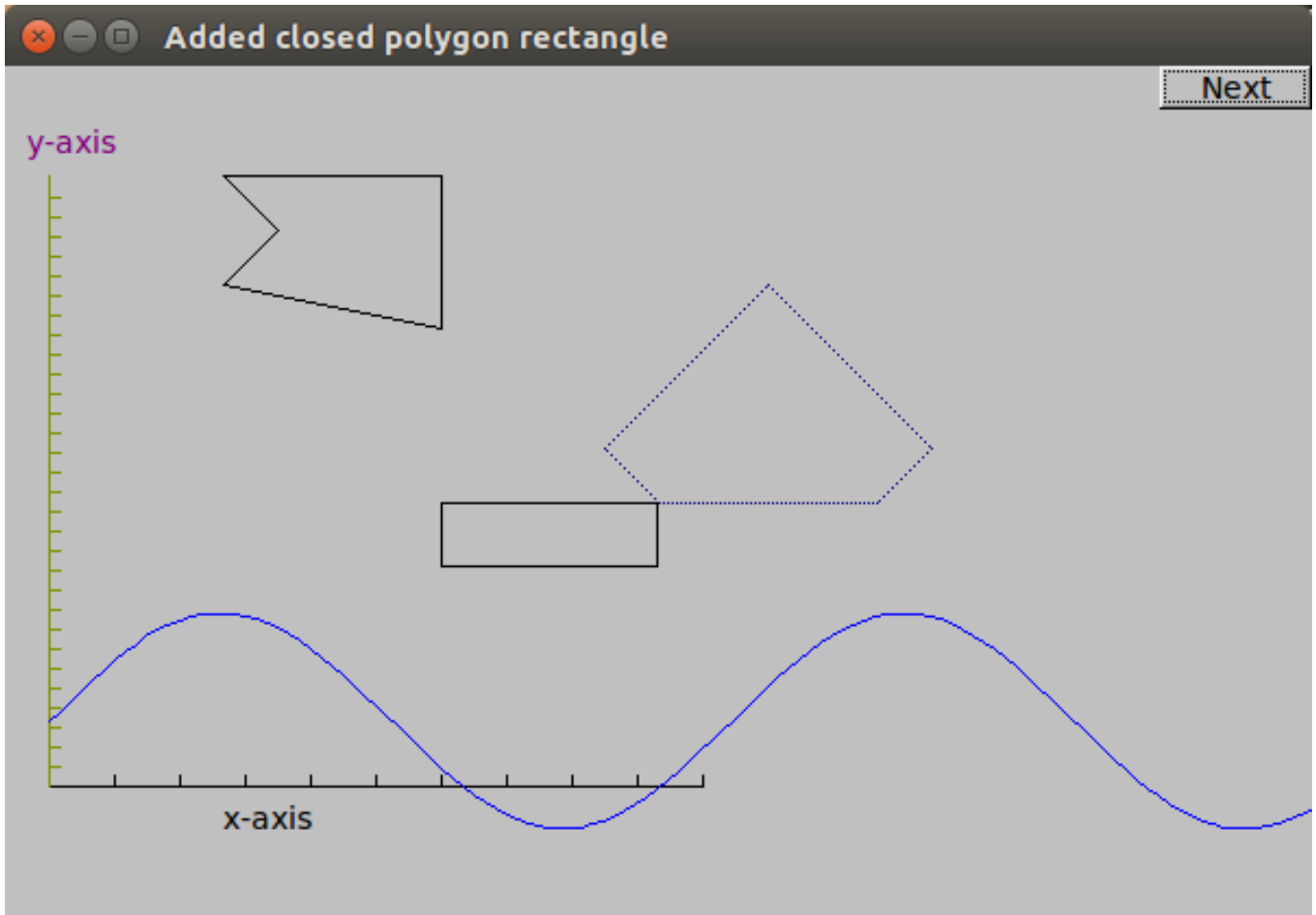


Figure : Resulting window with four sided polygon added

Oops, can still add more points to `Closed_polyline`

- In addition to the possibility of making a mistake in defining rectangular shape using `Closed_polyline` class – can still add more points to the line:

```
1 poly_rect.add(Point(125,70));
2 w.wait_for_button();
```



Fill color for shapes

- We can fill closed shapes with color, and change line styles, for example:

```
1 // r is Rectangle, poly_rect is Closed_polyline
2 r.set_fill_color(Color::yellow);
3 poly_rect.set_style(Line_style(Line_style::dashdotdot,2));
4 poly_rect.set_style(Line_style(Line_style::dashdot,4));
5 poly_rect.set_fill_color(Color::green);
6 w.set_label("Set some styles for shapes");
7 w.wait_for_button();
```

- Note second argument of `Line_style` is line width

Result of adding style to shapes

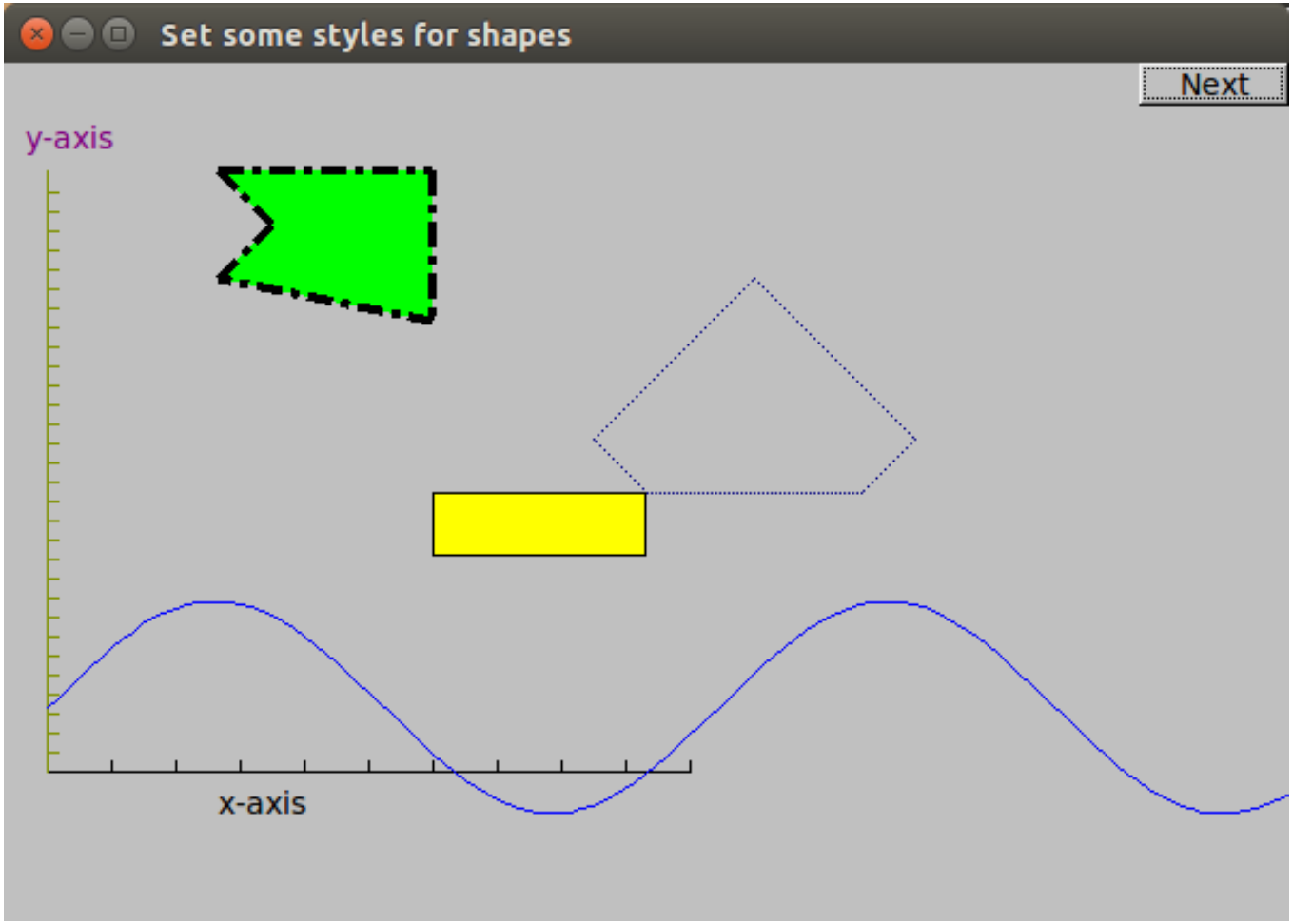


Figure : Result of adding style to shapes

Text in graphics window

- Of course we may want to add text to our window at an arbitrary location that we specify

```
1 Text t(Point(150,150),"Hello, customised world!");
2 w.attach(t);
3 w.set_label("Add some text");
4 w.wait_for_button();
```

Text added to window

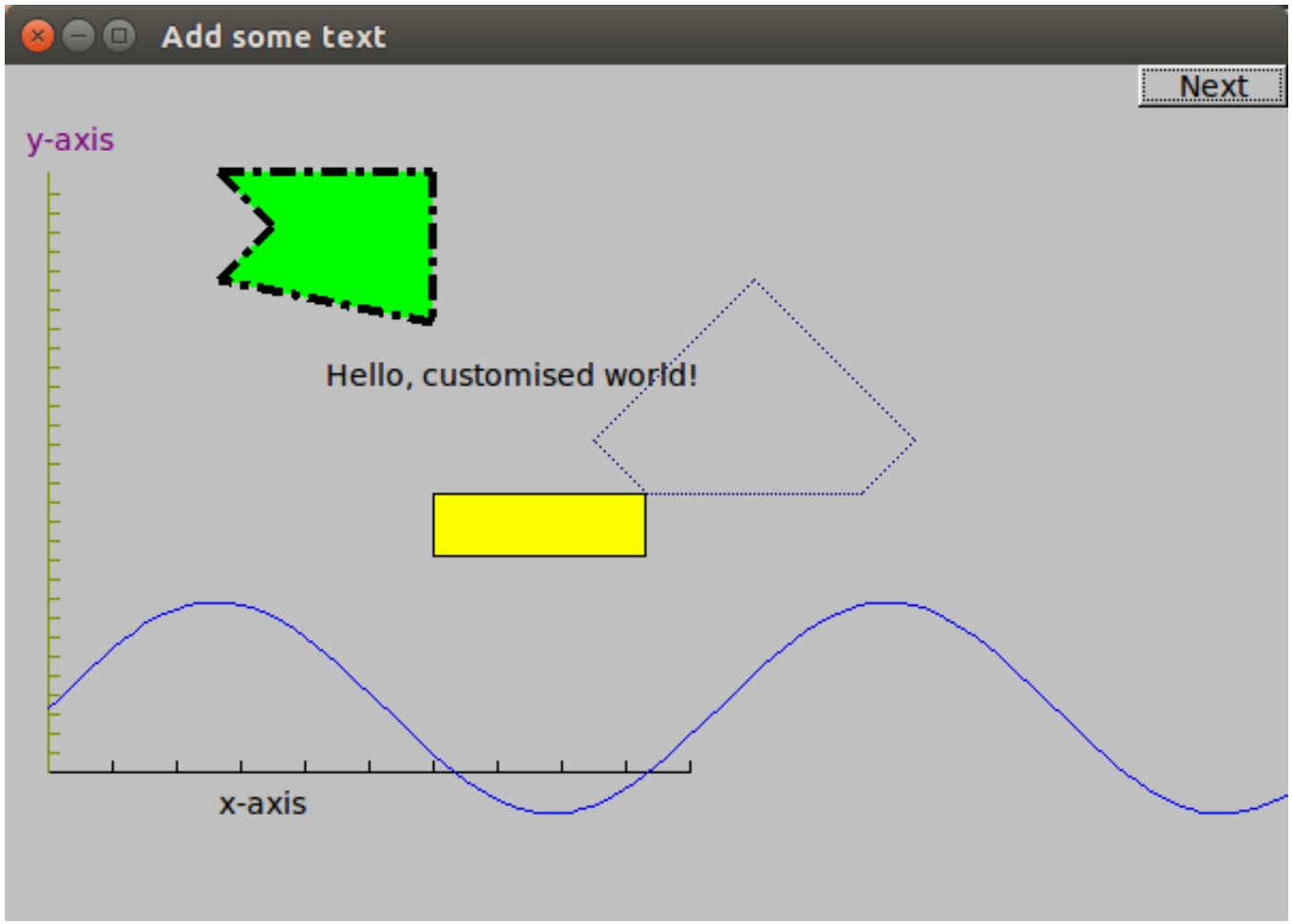
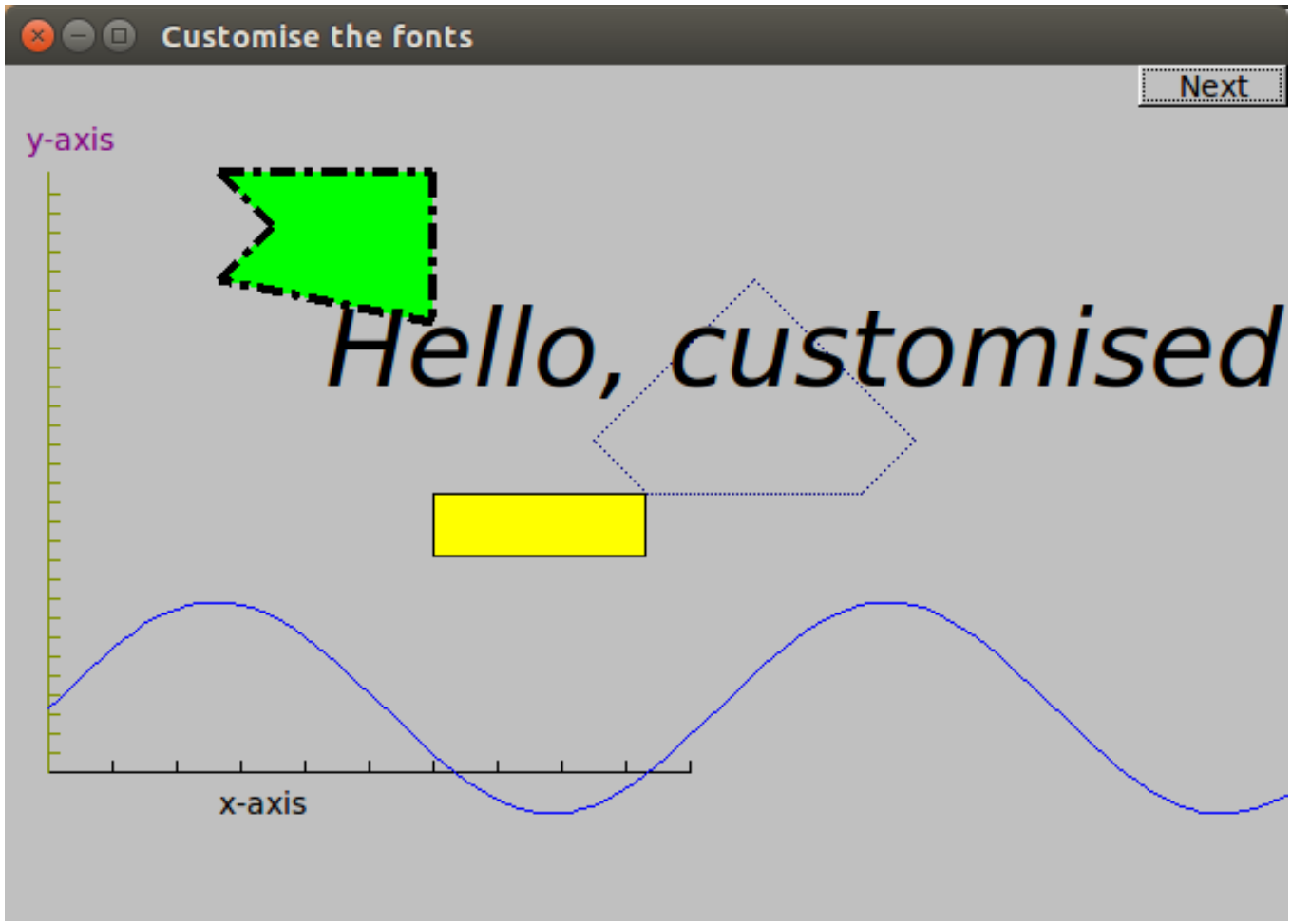


Figure : Text added to window

Make fonts bigger!

- We can choose font size and type

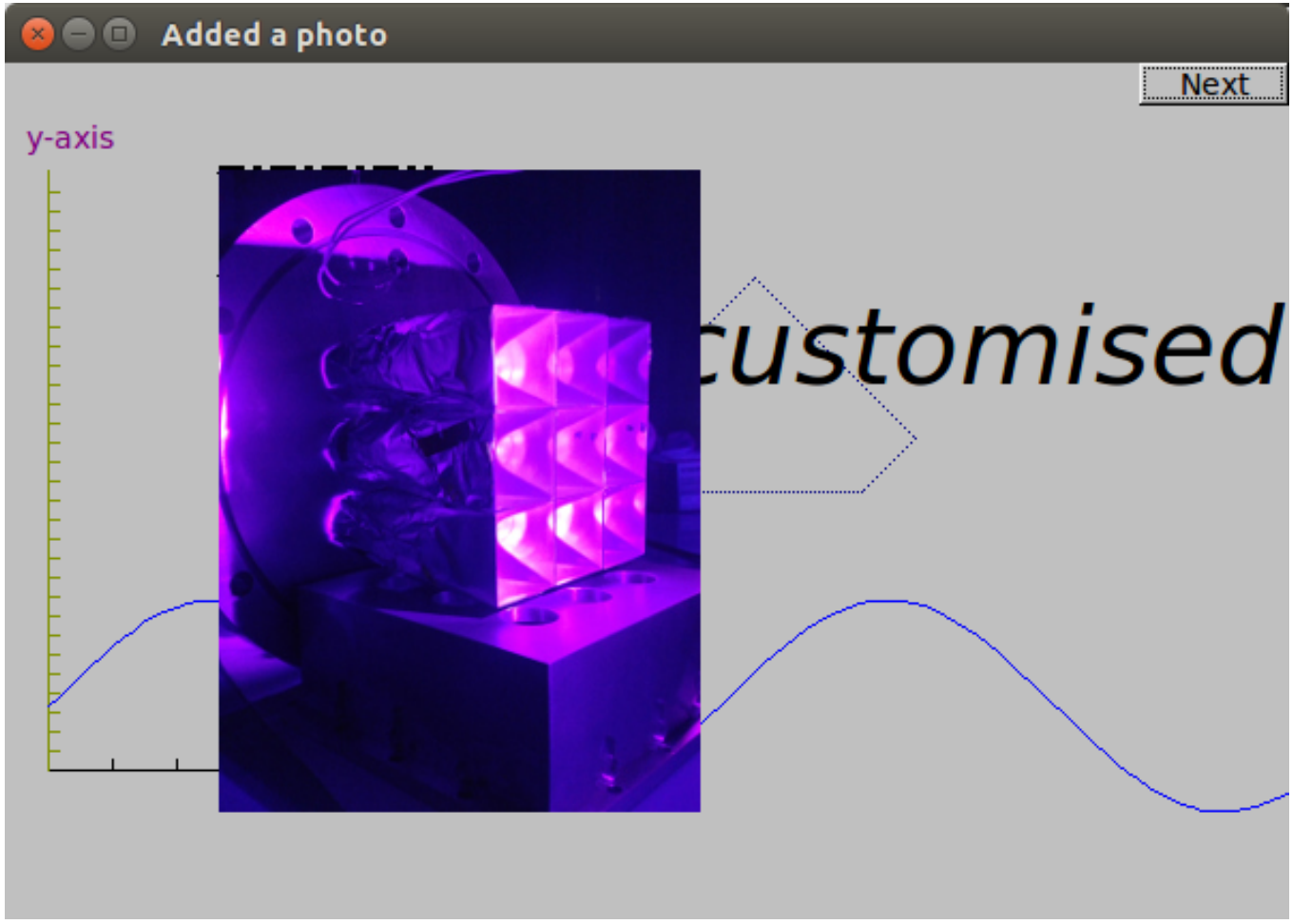
```
1 // t is Text type object
2 t.set_font( Graph_lib::Font::helvetica_italic );
3 t.set_font_size( 50 );
```



Images

- We can display images from a file:

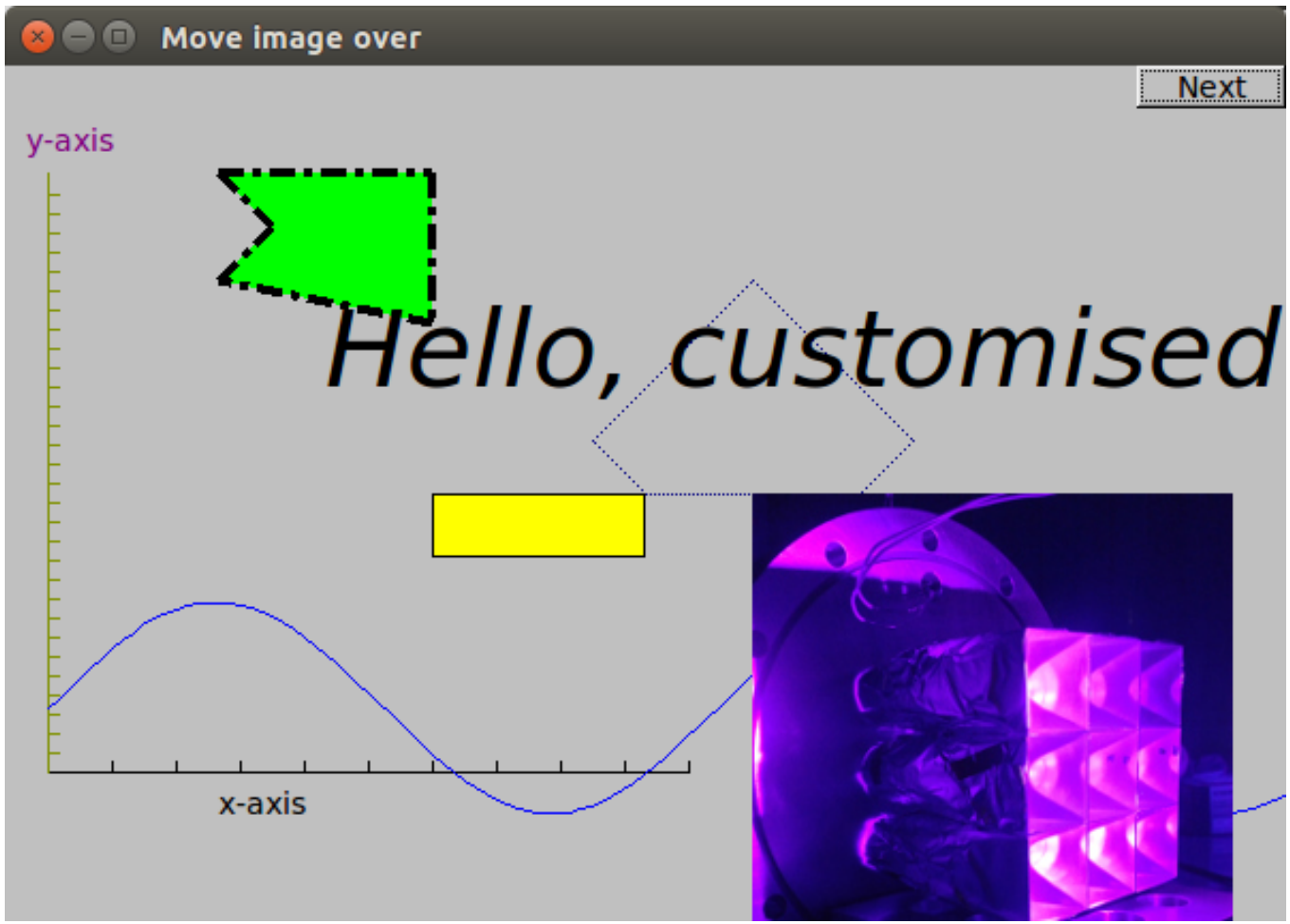
```
1 Image ii( Point{100, 50}, "LiGlassPhoto.jpg");
2 win.attach( ii );
3 w.set_label("Added a photo");
4 w.wait_for_button();
```



Move the image after placing it

- We can display images from a file:

```
1 ii.move(250,150);
2 w.set_label("Move image over");
3 w.wait_for_button();
```

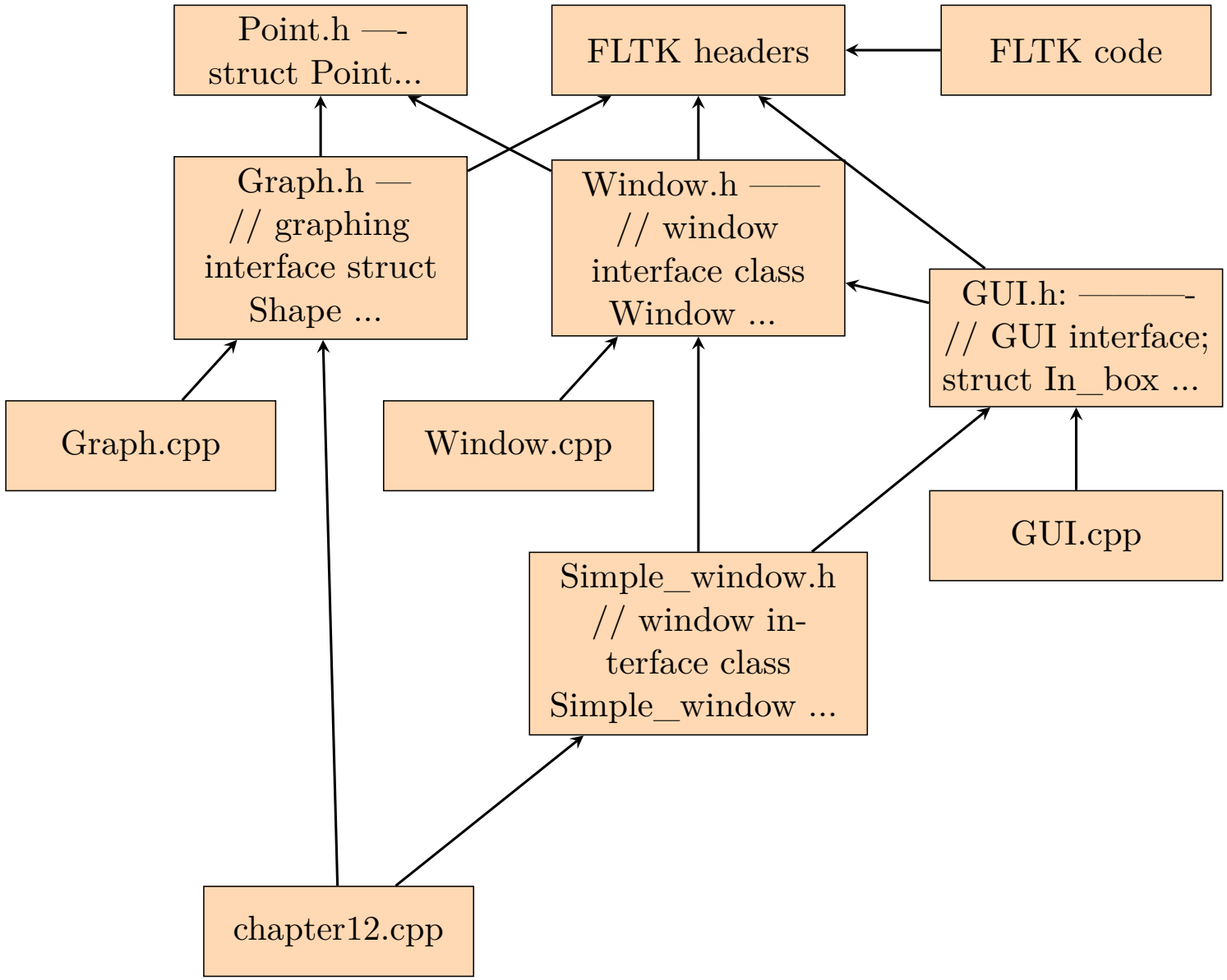


And so on...

Guess what this does?

```
Circle c(Point(100,200),50);
Graph_lib::Ellipse e(Point(100,200),75,25);
e.set_color(Color::dark_red);
Mark m(Point(100,200),'X');
ostringstream oss;
oss << "screen size: " << x_max() << "*" << y_max()
    << "; window size: " << win.x_max() << "*"
    << win.y_max();
Text sizes(Point(100,20),oss.str());
Image cal(Point{415,25},"hyperklogo.jpg");
cal.set_mask( Point{40,40},200,150);
win.attach(c);
win.attach(m);
win.attach(e);
win.attach(sizes);
win.attach(cal);
win.set_label("A very busy window!");
win.wait_for_button();
```

Summary of graphics interface



Add a bunch more!

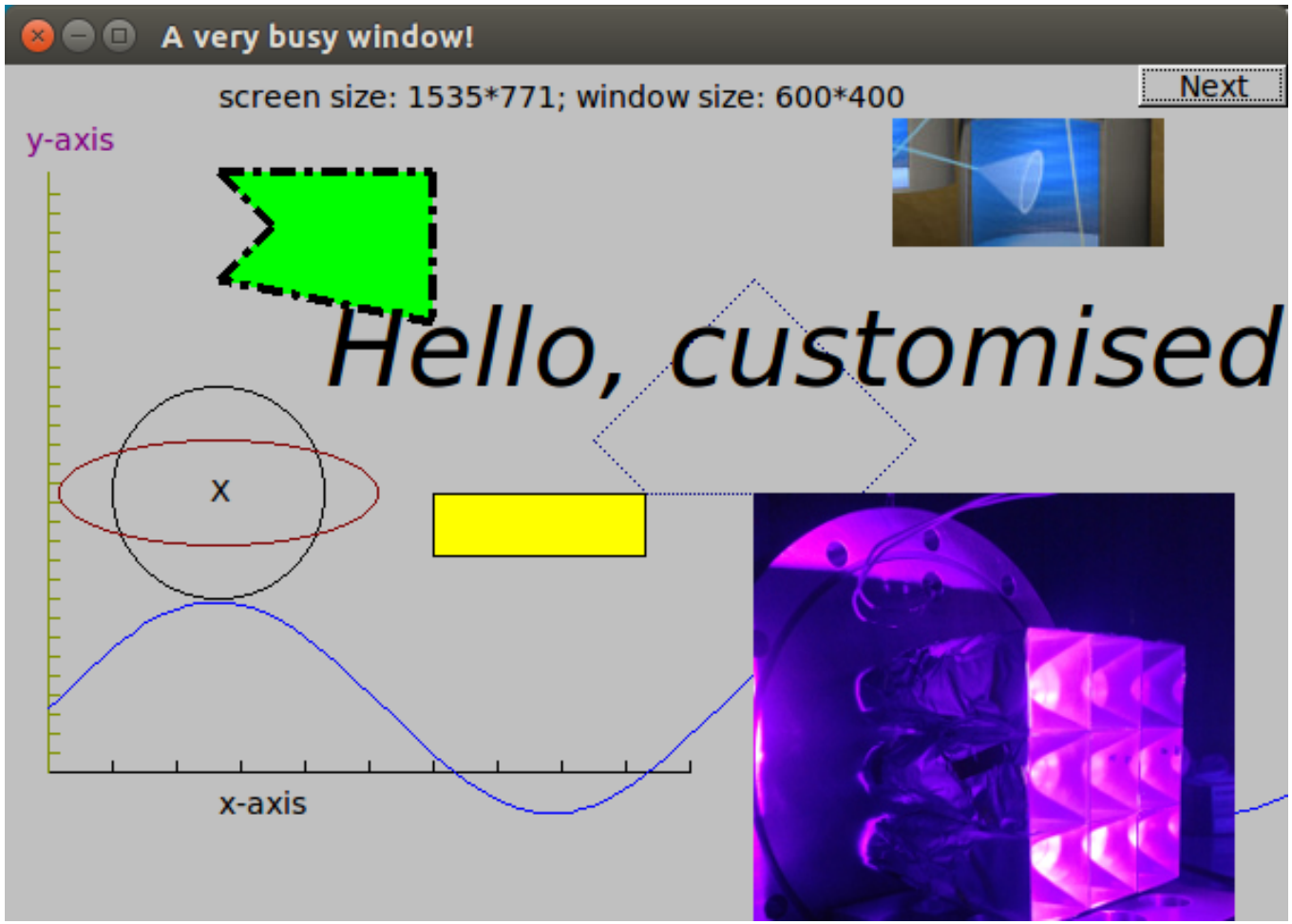


Figure : Final window after all is added

Week8 Done!

- Homework 6 given out today is due Nov. 11 (17:00)
- Reminder: final project is due Nov. 29 (17:00)