A Display Model

Lecture 16

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

Singleton Pattern

- Singleton pattern ensures that there will only be one instance of a particular class
- Instead of creating an object directly (and globally):

```
some_type global(…);
```

- We use a static method:

```
some_type& global = some_type::get_instance();
```

- `some_type::get_instance()` always returns the same instance
- Therefore, it is different from a factory (pattern), which creates new objects

What's Wrong With It?

- Singletons are "anti-functional"
  - And bad for multi-threaded contexts
- They're essentially a fancy global variable
- Global variables are useful sometimes, but usually they're a terrible idea
  - They make refactoring more difficult
  - They make testing much more difficult
  - They make reasoning about system behavior more difficult
  - They create additional issues in multi-threaded environments
- Sometimes they are useful in spite of the above

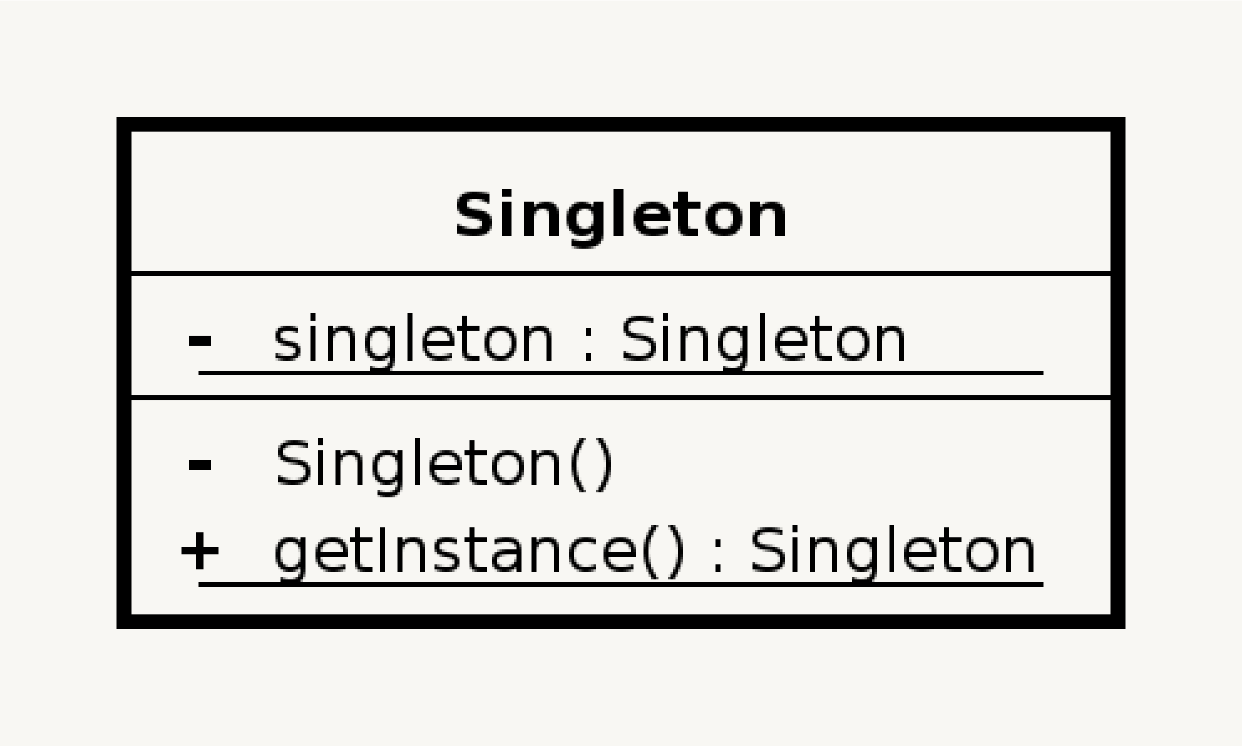
Where you see it

- Singleton is associated with factories:
  - The factory itself can be a singleton (configuration managers)
- It is common when accessing hardware resources:

```
mouse::get_instance();
screen::get_instance();
```

Singleton Class Diagram

- Singleton has the easiest UML class diagram



Singleton

- singleton : Singleton

- Singleton()

+ getInstance() : Singleton

3 Things Worth Noting

- The constructor is private
  - Why?
  - Because if it weren't, the user could create more than one instance, defeating the purpose
- The instance field is a (function-local) `static` (indicated by underline) and private
  - Why?
  - Static because it's an easy way to ensure that there is exactly one, globally accessible field
  - Private to force users to use `get_instance`
- The only way to access the instance is through the `get_instance` method (which is also static)
  - This is where we return a reference to the one instance, or create it if it doesn't exist (on first access)

Example

```
class graphics_manager {
private:
    graphics_manager() { ... }

public:
    static graphics_manager& get_instance()
    {
        static graphics_manager instance; // created on first access
        return instance;
    }

    // ... Other (non-static) functionalities
}
```

Benefits of Function-local static

- Correct order of initialization is forced, so it can’t be wrong
- The instances are also not global, which reduces coupling
- Basically, this is what you should do instead of using a global in almost every circumstance

A Display Model

Abstract

- This lecture presents a display model (the output part of a GUI), giving examples of use and fundamental notions such as screen coordinates, lines, and color. Some examples of shapes are Lines, Polygons, Axis, and Text.

Overview

- Why graphics?
- A graphics model
- Examples

Why bother with graphics and GUI?

- It’s very common
  - If you write conventional PC applications, you’ll have to do it
- It’s useful
  - Instant feedback
  - Graphing functions
  - Displaying results
- It can illustrate some generally useful concepts and techniques

Why bother with graphics and GUI?

- It can only be done well using some pretty neat language features ☺
- Graphics are a good introduction to what is commonly known as ‘OOP’
- Lots of good (small) code examples
- It can be non-trivial to “get” the key concepts
  - So it’s worth teaching
  - If we don’t show how it’s done, you might think it was “magic”
- Graphics is fun!

Why Graphics/GUI?

- WYSIWYG
  - What you see (in your code) is what you get (on your screen)
- Direct correspondence between concepts, code, and output

A Display Model

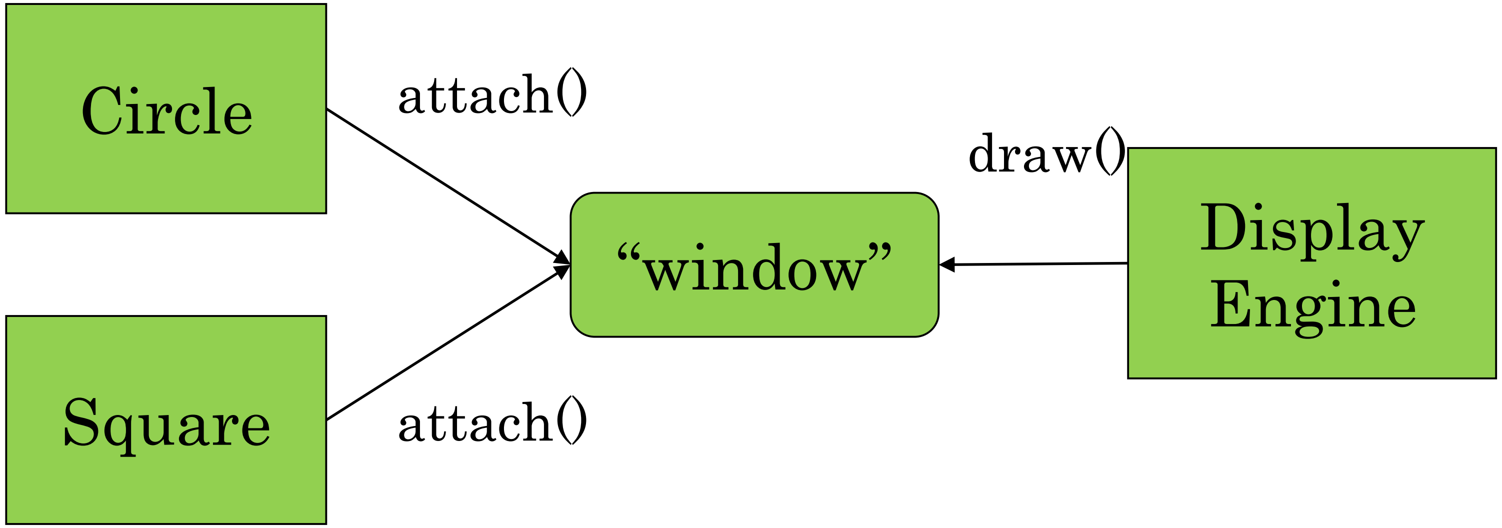
Text vs. Graphics

- So far we have seen text based stream IO
  - Possible to do ‘graphics’
- Simplest HTML:

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>My test page</title>
  </head>
  <body>
    <p>This is my page</p>
  </body>
</html>
```

- Here and in the next lectures we will provide an alternative approach
  - Directly aimed at the screen, drawing lines, rectangles, circles, etc.

Display Model



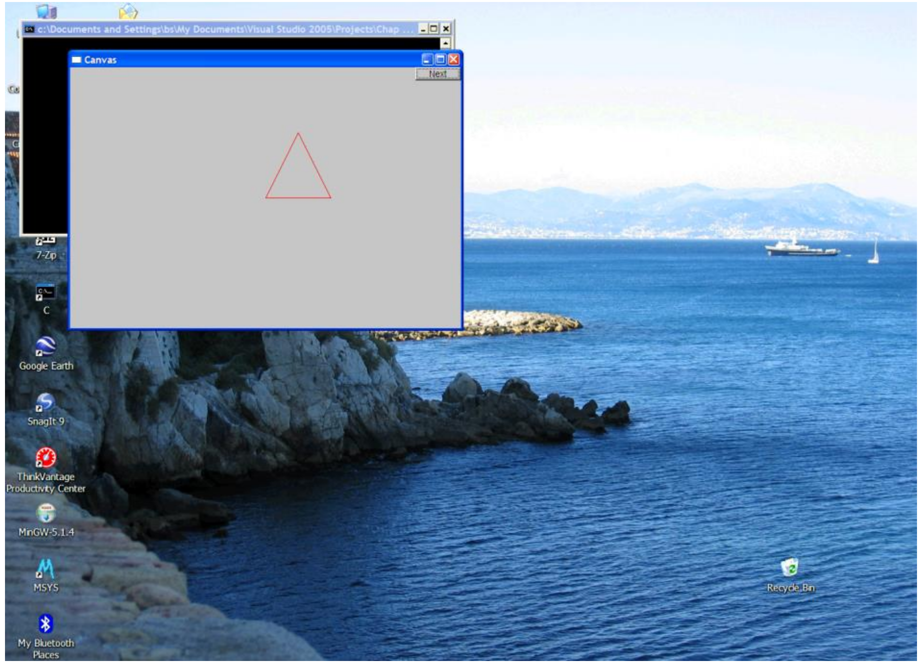
- Objects (such as graphs) are "attached to" a window.
- The "display engine" invokes display commands (such as "draw line from x to y") for the objects in a window
- Objects such as Square contain vectors of lines, text, etc. for the window to draw

Display Model

- An example illustrating the display model

```
int main()
{
    Graph_lib::Point tl(100, 200);     // a point (obviously)

    Simple_window win(tl, 600, 400, "Canvas");     // make a simple window

    Graph_lib::Polygon poly;     // make a shape (a polygon, obviously)

    poly.add(Graph_lib::Point(300, 200));          // add three points
    poly.add(Graph_lib::Point(350, 100));
    poly.add(Graph_lib::Point(400, 200));

    poly.set_color(Graph_lib::Color::red);         // make the polygon red (obviously)

    win.attach(poly);          // connect poly to the window

    win.wait_for_button();     // give control to the display engine
}
```

The Resulting Screen


c:\Documents and Settings\bs\My Documents\Visual Studio 2005\Projects\Chap ...
Canvas
Next
7-Zip
C
Google Earth
SnagIt 9
ThinkVantage Productivity Center
MinGW-5.1.4
MSYS
My Bluetooth Places
Recycle Bin

Display Model

- An example illustrating the display model

```
int main()
{
    Graph_lib::Point tl(100, 200);    // a point (obviously)

    Simple_window win(tl, 600, 400, "Canvas");    // make a simple window

    Graph_lib::Polygon poly;    // make a shape (a polygon, obviously)

    poly.add(Graph_lib::Point(300, 200));         // add three points
    poly.add(Graph_lib::Point(350, 100));
    poly.add(Graph_lib::Point(400, 200));

    poly.set_color(Graph_lib::Color::red);        // make the polygon red (obviously)

    win.attach(poly);         // connect poly to the window

    win.wait_for_button();    // give control to the display engine
}
```

Graphics/GUI Libraries

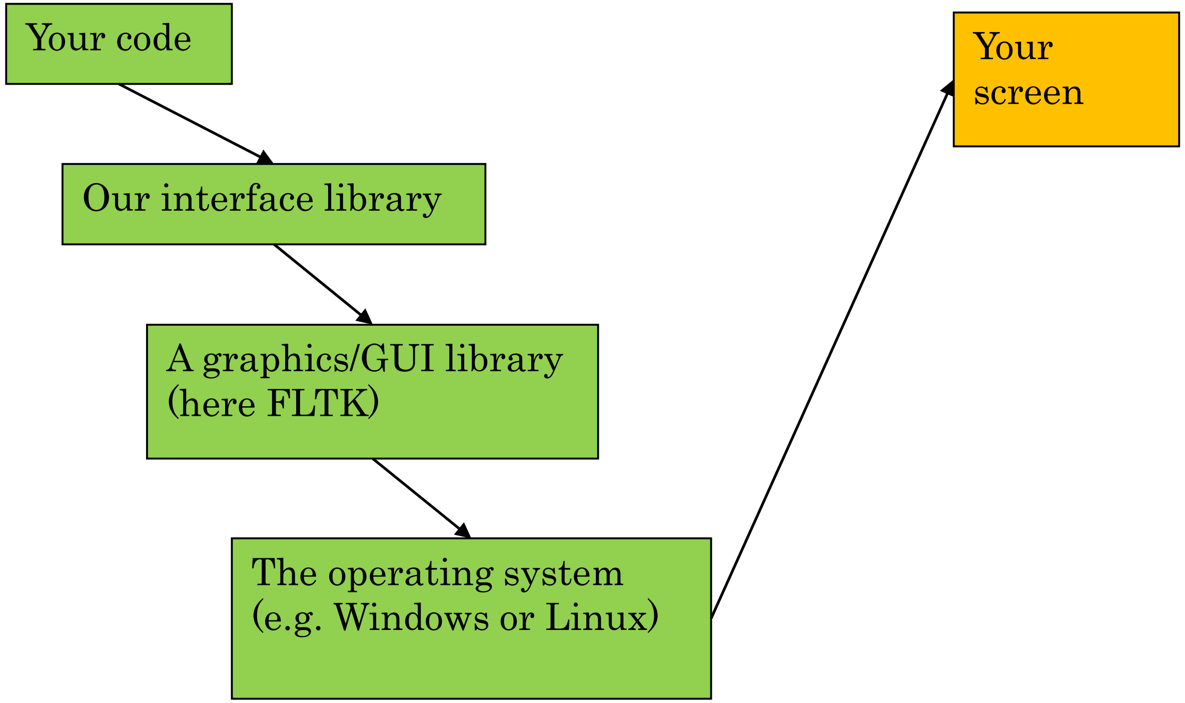
- You’ll be using a few interface classes we wrote
  - Interfacing to a popular GUI toolkit
    - GUI == Graphical User Interface
    - FLTK: www.fltk.org // Fast Light Tool Kit, pronounced ‘FullTick’
  - Installation, etc.
    - FLTK
    - Our GUI and graphics classes
    - Project settings
- This model is far simpler than common toolkit interfaces
  - The FLTK (very terse) documentation is 1000 pages
  - Our interface library is < 20 classes and < 500 lines of code
  - You can write a lot of code with these classes
    - And what you can build on them

Graphics/GUI libraries (cont.)

- The code is portable
  - Windows, Unix, Mac, etc.
- This model extends to most common graphics and GUI uses
- The general ideas can be used with any popular GUI toolkit
  - Once you understand the graphics classes you can easily learn any GUI/graphics library
    - Well, relatively easily – these libraries are huge

Graphics/GUI libraries

- Often called “a layered architecture”



Your code

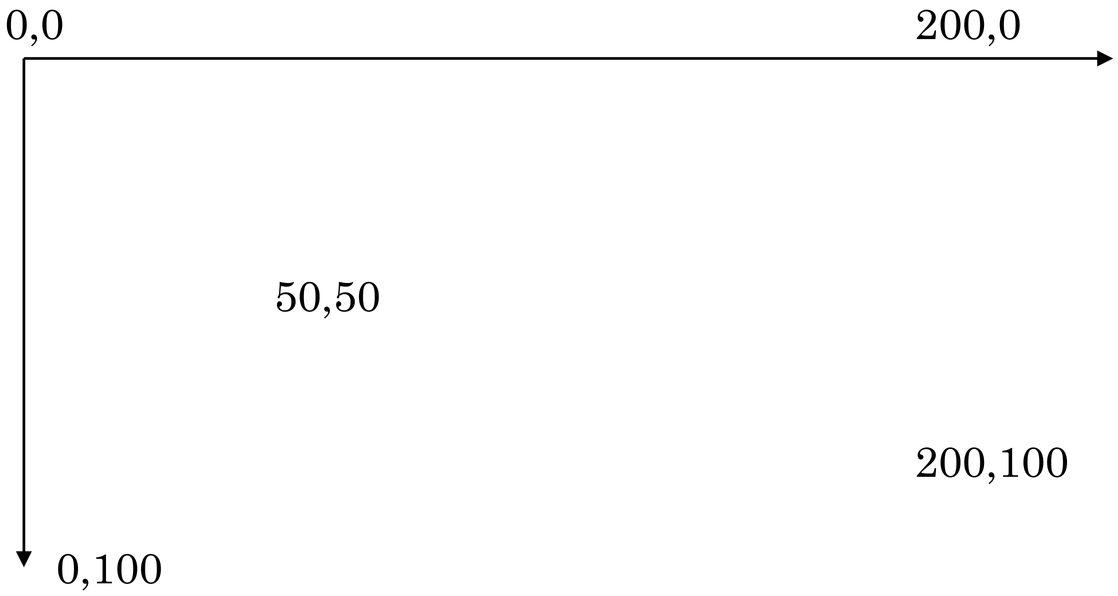
Our interface library

A graphics/GUI library (here FLTK)

The operating system (e.g. Windows or Linux)

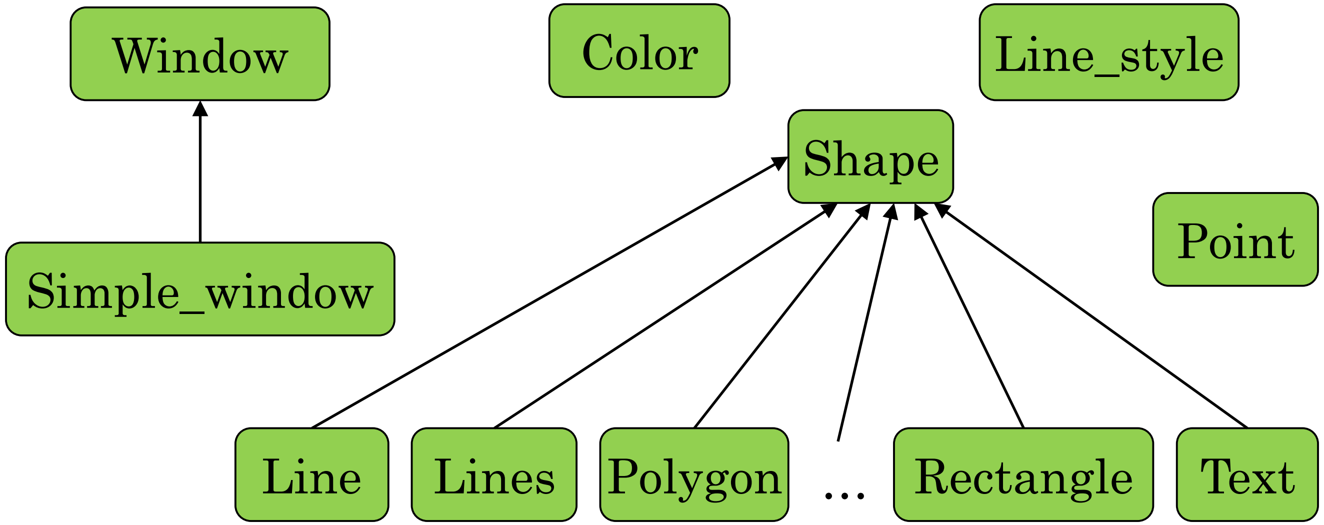
Your screen

Coordinates



- Oddly, y-coordinates “grow downwards” // right, down
- Coordinates identify pixels in the window on the screen
- You can re-size a window (changing x_max() and y_max())

Interface classes



- An arrow means "is a kind of" ⟶
- Color, Line_style, and Point are "utility classes" used by the other classes
- Window is our interface to the GUI library (which is our interface to the screen)

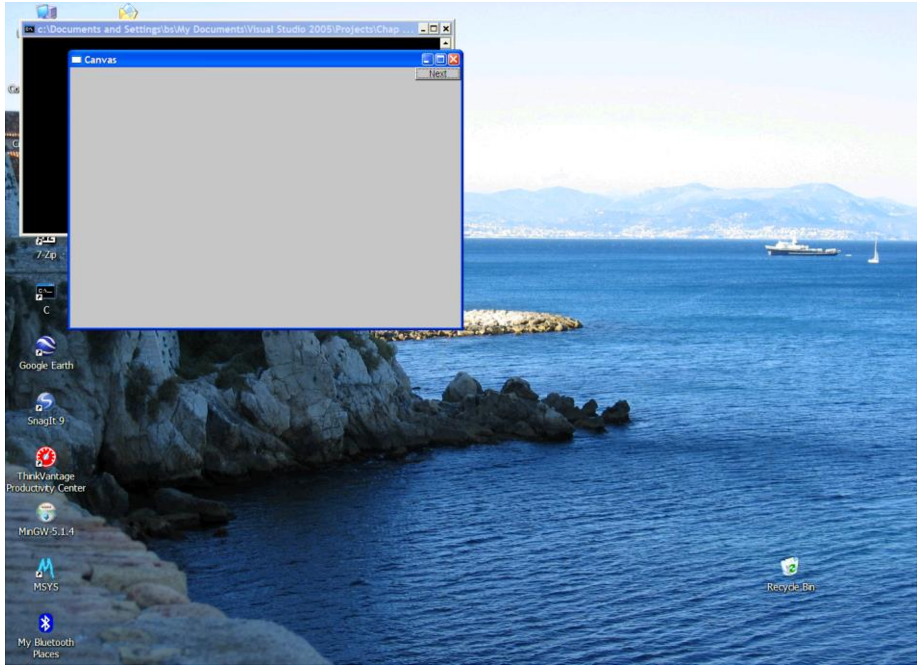
Interface classes

- Current
  - Color, Line_style, Font, Point,
  - Window, Simple_window
  - Shape, Text, Polygon, Line, Lines, Rectangle, …
  - Axis
- Easy to add (for some definition of "easy")
  - Grid, Block_chart, Pie_chart, etc.
- Later, GUI
  - Button, In_box, Out_box, …

Demo code 1

```cpp
// Getting access to the graphics system
#include "Graph.h"              // graphical shapes
#include "Simple_window.h"      // stuff to deal with your system's windows

int main()
{
    // make a simple window

    // screen coordinate (100, 200) top left of window
    // window size(600*400)
    // title: Canvas
    Simple_window win(Graph_lib::Point(100, 100), 600, 400, "Canvas");

    win.wait_for_button();      // give control to the display engine
}
```

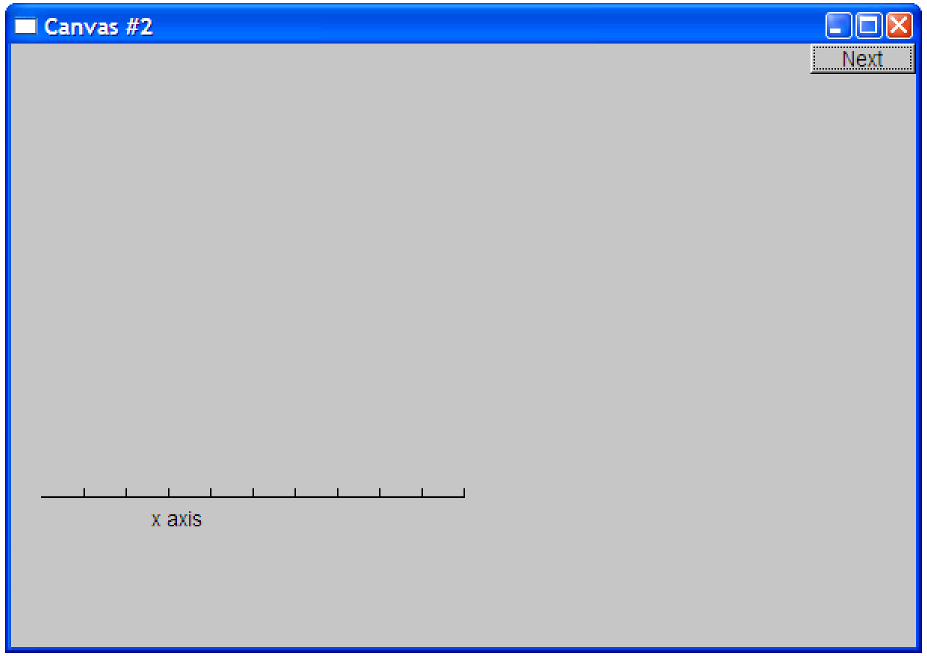
A “blank canvas”


c:\Documents and Settings\bs\My Documents\Visual Studio 2005\Projects\Chap ...
Canvas
Next
7-Zip
C
Google Earth
SnagIt 9
ThinkVantage Productivity Center
MinGW 5.1.4
MSYS
My Bluetooth Places
Recycle Bin

Demo code 2

```
// make an Axis, an axis is a kind of Shape
// Axis::x means horizontal
// starting at (20, 300)
// 280 pixels long
// 10 "notches"
// text "x axis"
Graph_lib::Axis xa(
    Graph_lib::Axis::x, Graph_lib::Point(20, 300), 280, 10, "x axis");

win.set_label("Canvas #2");
win.attach(xa);    // attach axis xa to the window
```

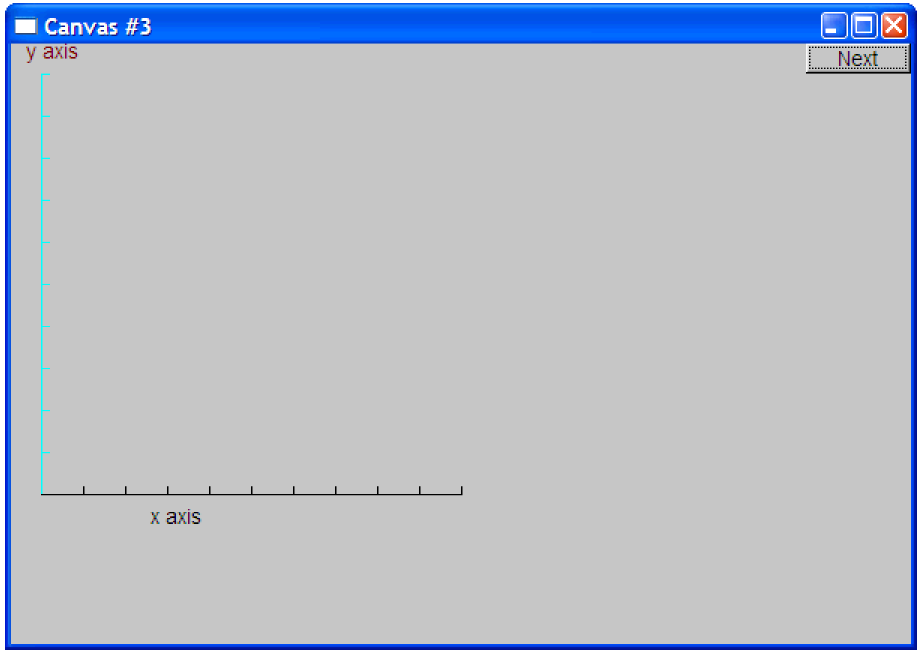
Add an X-axis



Demo code 3

```
win.set_label("Canvas #3");

Graph_lib::Axis ya(
    Graph_lib::Axis::y, Graph_lib::Point(20, 300), 280, 10, "y axis");


ya.set_color(Graph_lib::Color::cyan);           // choose a color for the axis
ya.label.set_color(Graph_lib::Color::dark_red);  // choose a color for the text

win.attach(ya);
```

Add a Y-axis (colored)

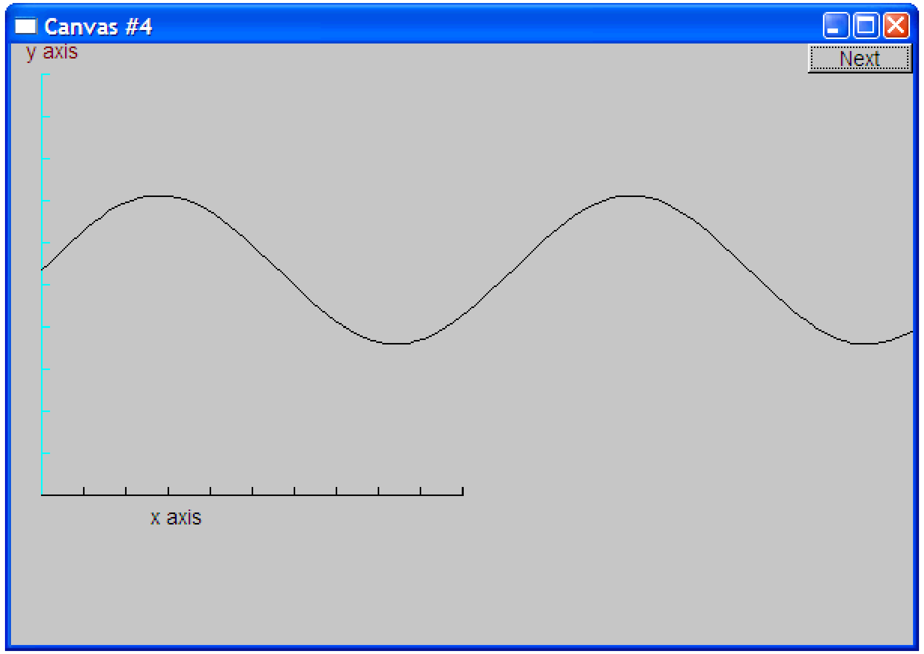


Yes, it’s ugly, but this is a programming course, not a graphics design course

Demo code 4

```
win.set_label("Canvas #4");

// sine curve


// plot sin() in the range [0:100)
// with (0, 0) at (20, 150)
// using 1000 points
// scale x values * 50, scale y values * 50
Graph_lib::Function sine(std::sin, 0, 100, Graph_lib::Point(20, 150), 1000, 50, 50);

win.attach(sine);
```

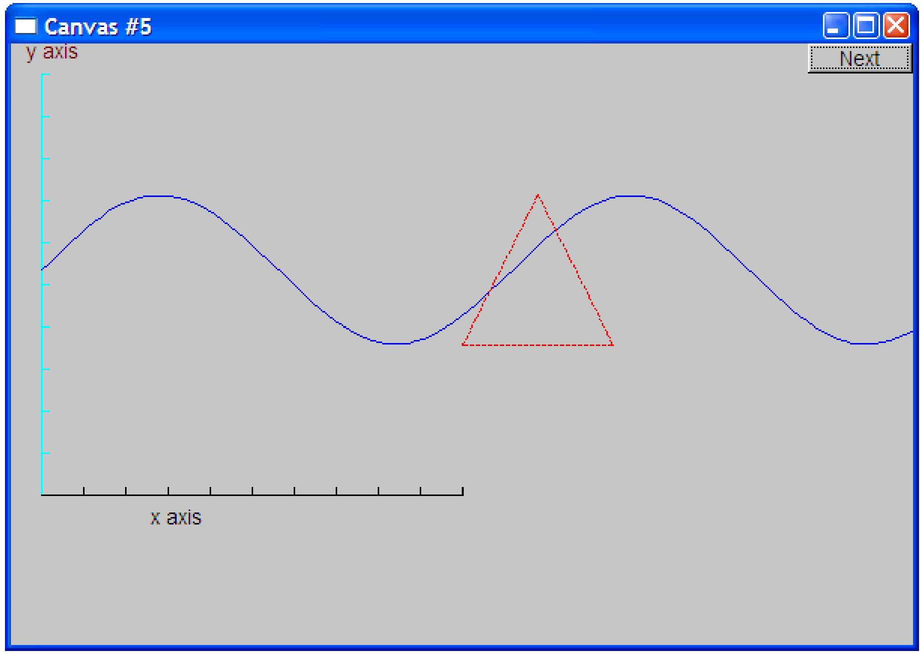
Add a sine curve



Demo code 5

```
win.set_label("Canvas #5");

sine.set_color(
    Graph_lib::Color::blue);    // I changed my mind about sine's color

Graph_lib::Polygon poly;    // a polygon, a Polygon is a kind of Shape
poly.add(Graph_lib::Point(300, 200));    // three points makes a triangle
poly.add(Graph_lib::Point(350, 100));
poly.add(Graph_lib::Point(400, 200));

poly.set_color(Graph_lib::Color::red);          // change the color
poly.set_style(Graph_lib::Line_style::dash);    // change the line style

win.attach(poly);
```

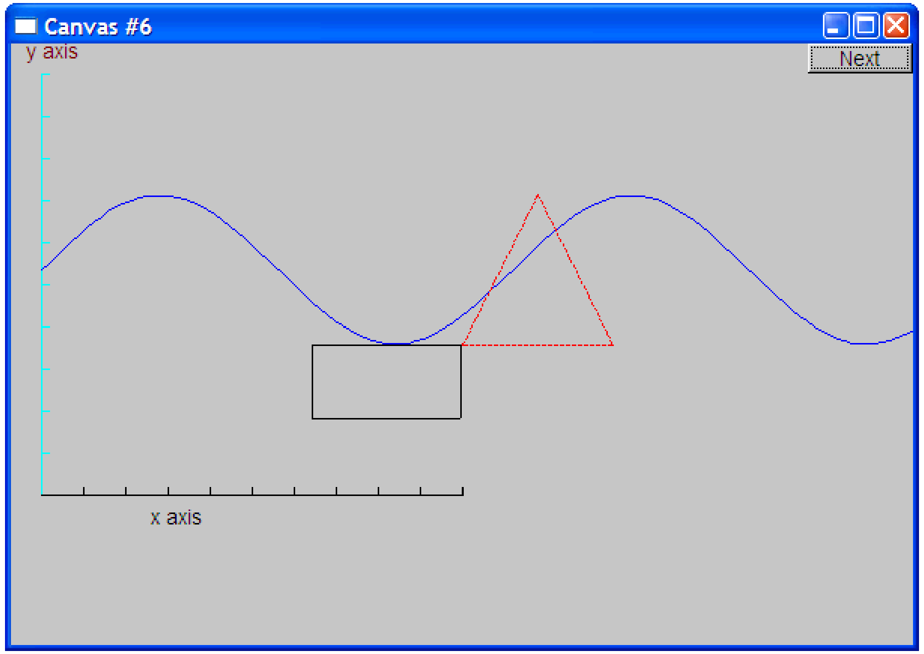
Add a triangle (and color the curve)



Demo code 6

```
// add a rectangular shape

// at position 200, 200
// of size 100*50
Graph_lib::Rectangle r(Graph_lib::Point(200, 200), 100, 50);

win.attach(r);
```

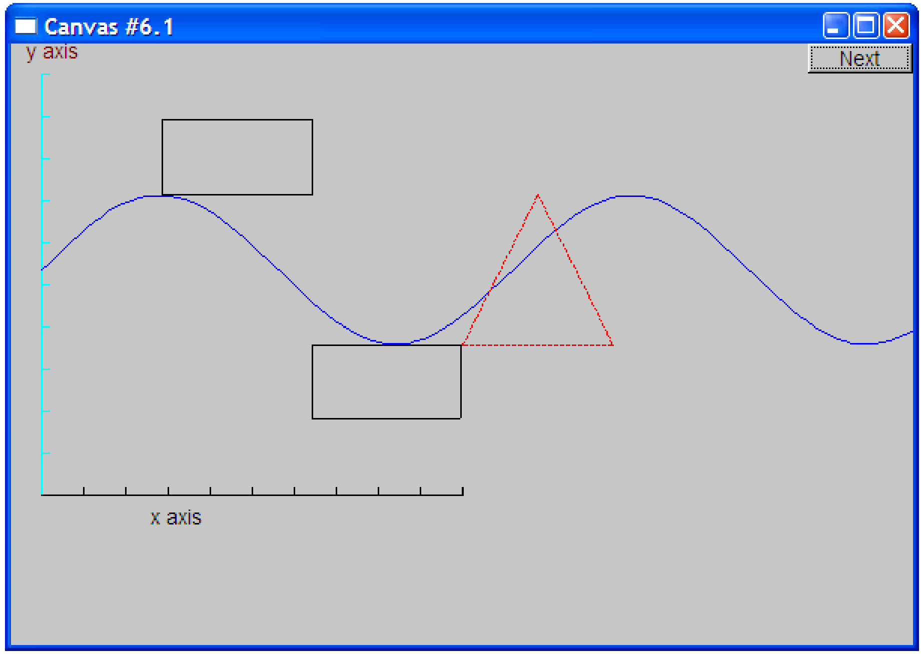
Add a Rectangle



Demo code 6.1

```
// Add a shape that looks like a rectangle
win.set_label("Canvas #6.1");

Graph_lib::Closed_polyline poly_rect;
poly_rect.add(Graph_lib::Point(100, 50));
poly_rect.add(Graph_lib::Point(200, 50));
poly_rect.add(Graph_lib::Point(200, 100));
poly_rect.add(Graph_lib::Point(100, 100));

win.attach(poly_rect);
```

Add a Shape that looks like a Rectangle



But is it a rectangle?

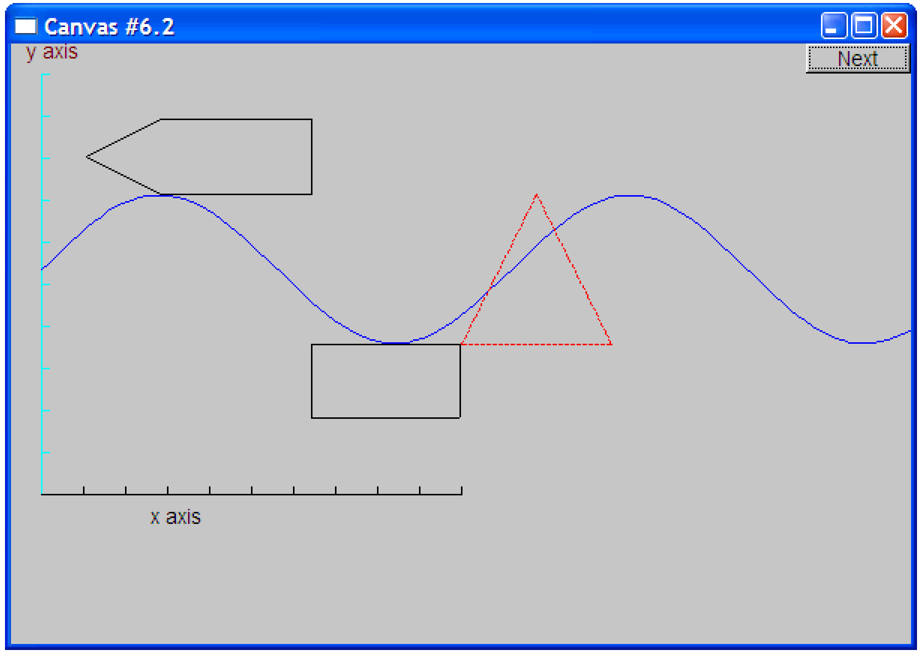
Demo code 6.2

- We can add a point

```
win.set_label("Canvas #6.2");
poly_rect.add(Graph_lib::Point(50, 75));    // now poly_rect has 5 points
```

- “looking like” is not the same as “is”

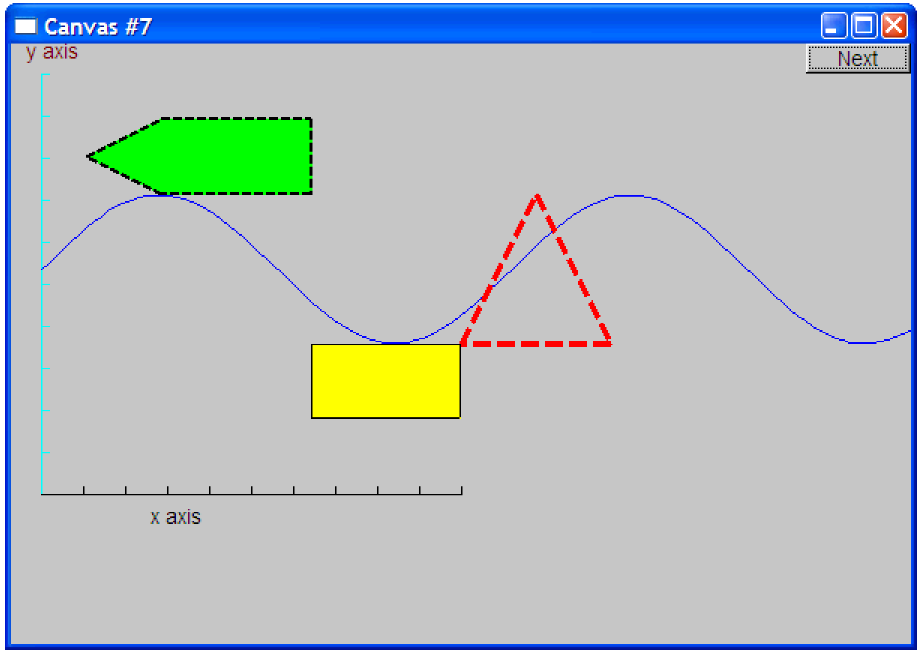
Obviously a polygon



Add fill

```
win.set_label("Canvas #7");

// color the inside of the rectangle
r.set_fill_color(Graph_lib::Color::yellow);

// make the triangle contour fat and dashed
poly.set_style(Graph_lib::Line_style(Graph_lib::Line_style::dash, 4));

poly_rect.set_fill_color(Graph_lib::Color::green);
poly_rect.set_style(Graph_lib::Line_style(Graph_lib::Line_style::dash, 2));
```

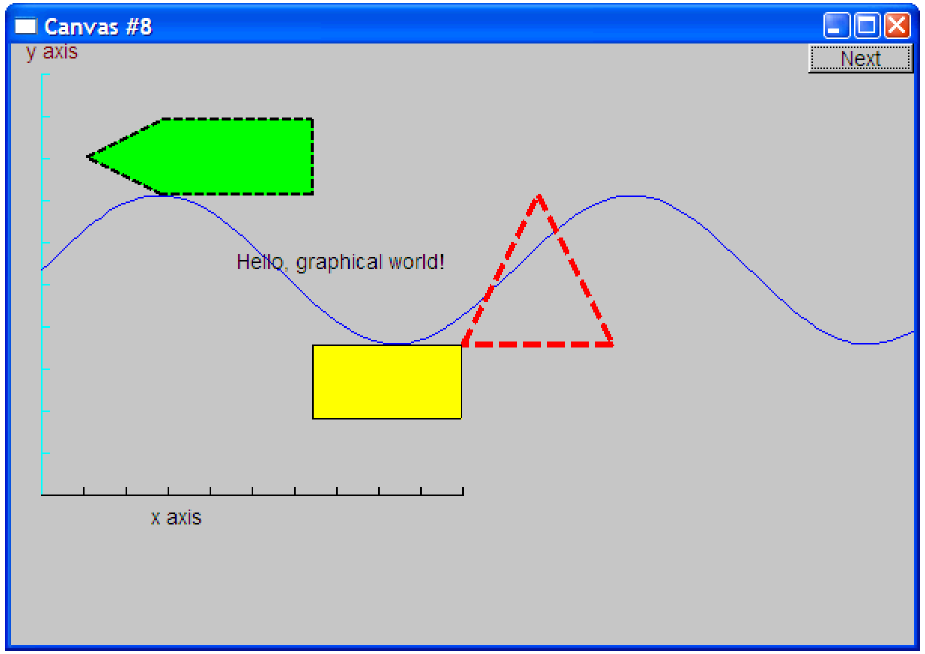
Add fill



Demo Code 8

```
win.set_label("Canvas #8");

// add text
Graph_lib::Text t(Graph_lib::Point(100, 100), "Hello, graphical world!");
win.attach(t);
```

Add text

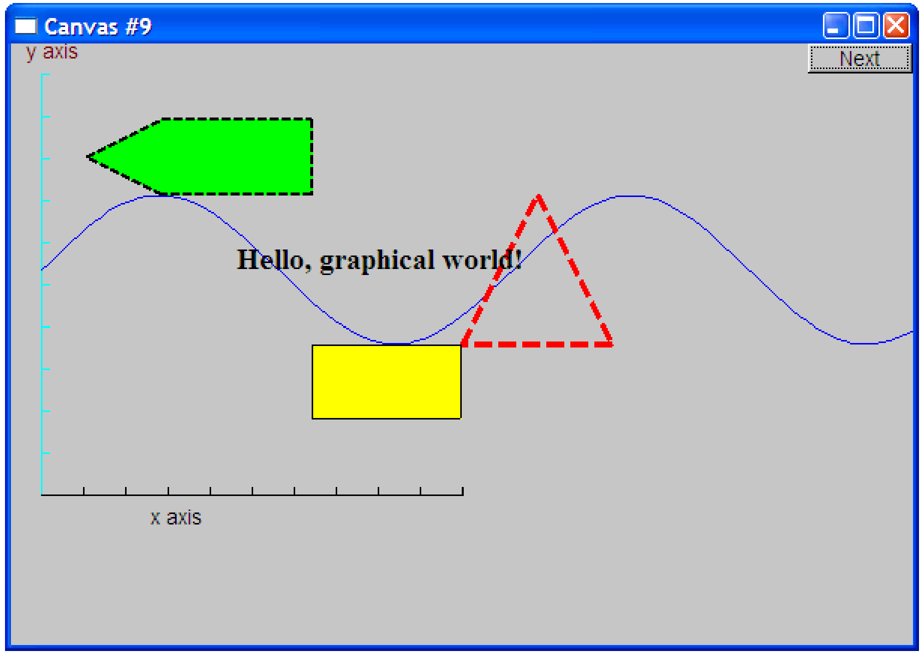


Demo Code 9

- Modify text font and size

```
t.set_font(Graph_lib::Font::times_bold);
t.set_font_size(20);
```

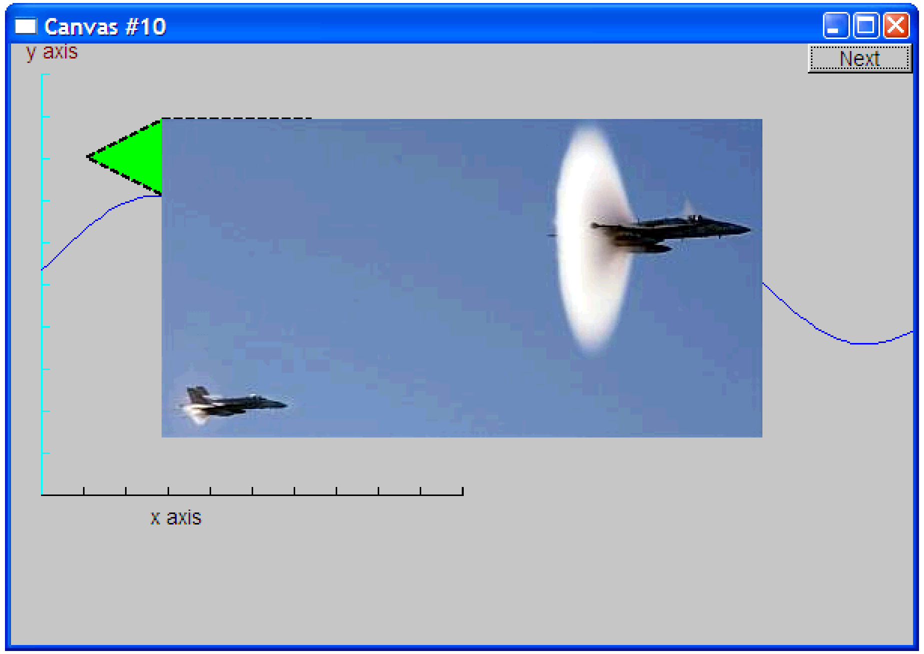
Text font and size



Add an Image

```
win.set_label("Canvas #10");

// open an image file
Graph_lib::Image ii(Graph_lib::Point(100, 50), "image.gif");
win.attach(ii);
```

Add an image

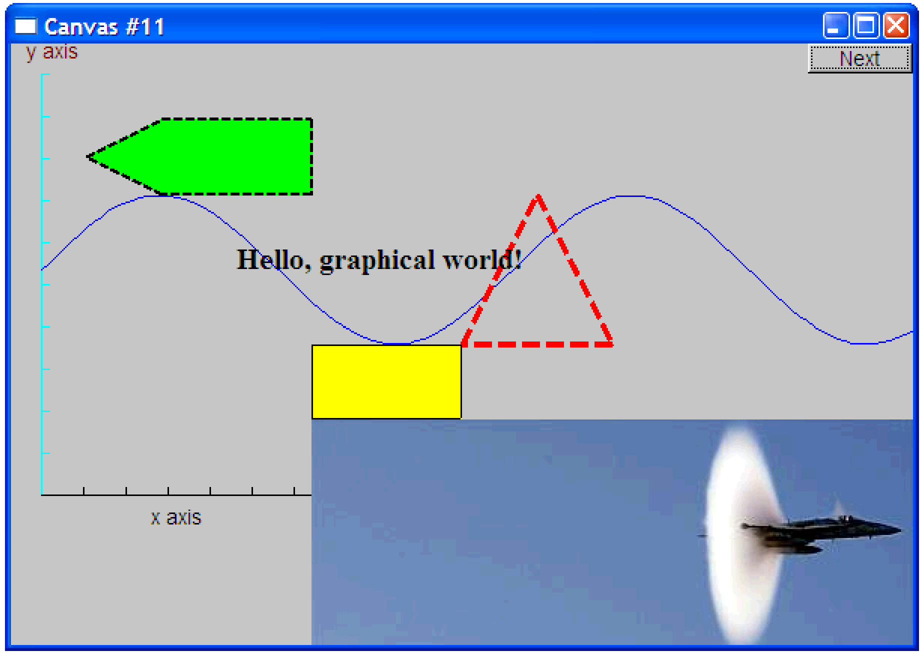


Oops!

- The image obscures the other shapes
  - Move it a bit out of the way

```
win.set_label("Canvas #11");
ii.move(100, 200);
```

Move the Image

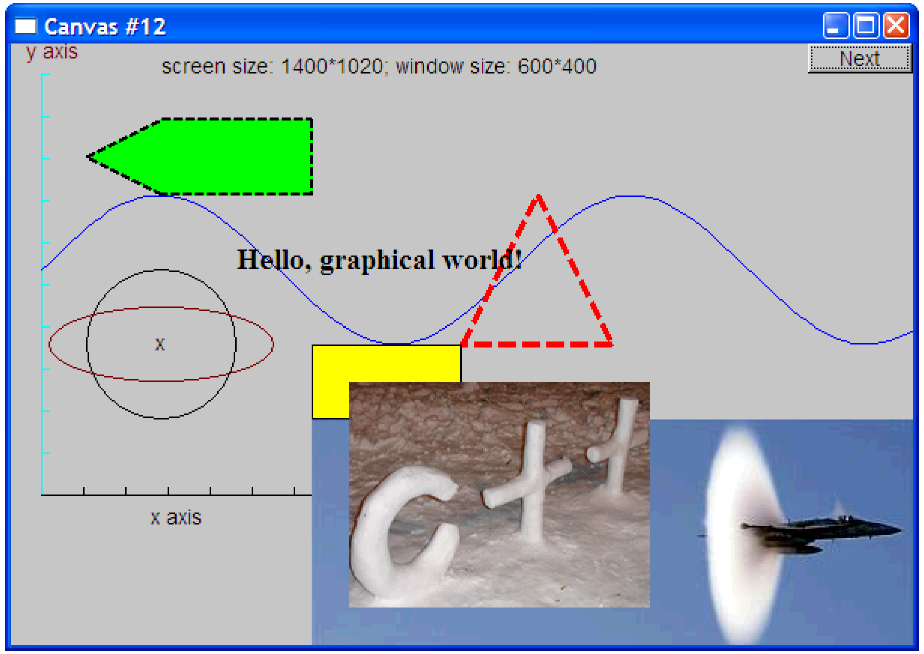


Note how the parts of a shape that don’t fit in the window are “clipped” away

Demo Code 12

```
win.set_label("Canvas #12");

Graph_lib::Circle c(Graph_lib::Point(100, 200), 50);

Graph_lib::Ellipse e(Graph_lib::Point(100, 200), 75, 25);
e.set_color(Graph_lib::Color::dark_red);

Graph_lib::Mark m(Graph_lib::Point(100, 200), 'x');

std::ostringstream oss;
oss << "screen size: " << Graph_lib::x_max() << "*" << Graph_lib::y_max()
    << "; window size: " << win.x_max() << "*" << win.y_max();
Graph_lib::Text sizes(Graph_lib::Point(100, 20), oss.str());

Graph_lib::Image cal(Graph_lib::Point(225, 225), "snow_cpp.gif"); // 320*240 pixel gif
cal.set_mask(Graph_lib::Point(40, 40), 200, 150);                 // display center of image

win.wait_for_button();
```

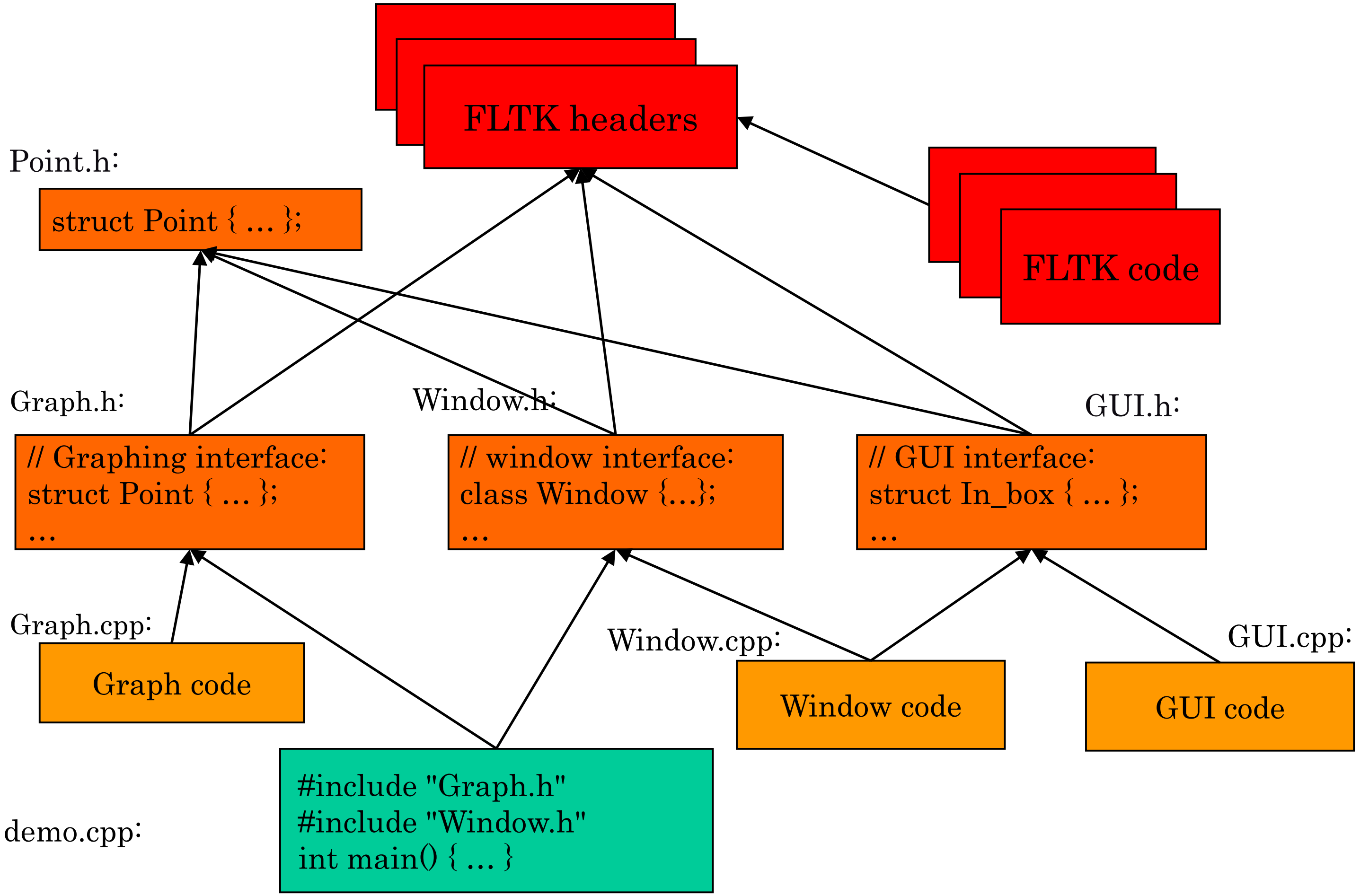
Add shapes, more text



Boiler Plate

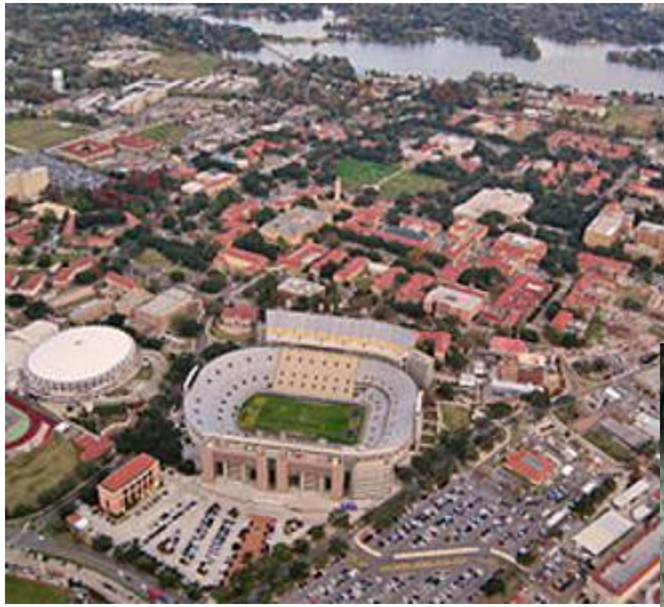
```
#include "Graph.h"            // graphical shapes
#include "Simple_window.h"    // stuff to deal with your system's windows

int main()
try {
    demo();    // the  main part of your code
    return 0;
}
catch (std::exception const& e) {
    std::cerr << "exception: " << e.what() << '\n';
    return 1;
}
catch (...) {
    std::cerr << "Some exception\n";
    return 2;
}
```

Code Organization

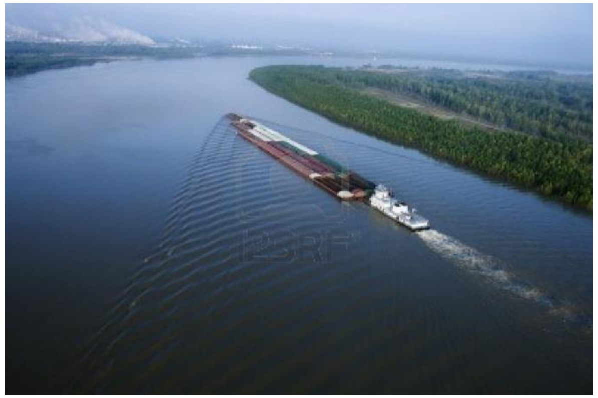


Primitives and Algorithms

- The demo shows the use of library primitives
  - Just the primitives
  - Just the use
- Typically what we display is the result of
  - an algorithm
  - reading data
- Next lectures
  - Graphics Classes
  - Graphics Class Design
  - Graphing Functions and Data
  - Graphical User Interfaces



LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP