CS 3210, Lab 6, Event-Driven Drawing

1 Introduction

The purpose of this lab is to add event handling capability to our infrastructure and enhance our shapes hierarchy.

2 Outcomes Addressed

- define and use two-dimensional and three-dimensional graphic object representations
- discuss and apply concepts of object-oriented programming, inheritance, polymorphism, and event-driven systems
- be familiar with C++ and STL concepts, including classes and constructors, operator overloading, STL vector class, dynamic memory with new and delete
- apply data structures to the management of computer graphics entities
- compiling and executing C++ programs on the Linux platform

3 Background

3.1.1 Event Handling

At this point we should be able to create individual shape-derived objects, contain them as a logical image, store and retrieve that image, as well as display the image. Our next goal is to be able to interact with our drawing.

One of the issues you should have noticed is that unless you sleep or run an empty while loop, the graphics context goes out of scope, destroys the window and the program finishes. It is in that loop that we need to establish interaction.

What type of interaction? We could accept input from the console if we wanted to using cin and cout, of course. One minor issue here is that the terminal/console will need focus, and once the drawing window pops up, the console may or may not have focus. In addition to console interaction we might also want mouse interaction. cin isn't going to cut it. Another issue that you may not have noticed is that if another window overlaps your drawing window, parts of your drawing may be gone and will not always regenerate. This is known as exposure and when it happens, we must manually force a redraw of our image.

The issue of user interaction and of window exposure (and other system events) can be handled in a unified way. While theoretically we could poll for various events (exposure, keyboard, mouse), for performance reasons it would be better if we blocked until the events we are interested occurred. The general technique would be to register callback functions, and those callback functions would get called when a particular event occurs. In Java Swing, you would create event handlers to accomplish this. We will essentially do the same thing here, but without the benefit of the predefined API, and greatly simplified.

Clearly the actual events will be generated by the windowing system. Since we have tried to keep our application platform independent, we will want to abstract these events as well and allow the actual implementation to coerce its events to fit our expectations. At a minimal level, we need to know about exposure so that we can redraw the image, mouse events, and keyboard events. Waiting for these events needs to happen in some sort of loop. Since the nature of events will be platform dependent, the actual implementation should appear in our platform specific derivative of gcontext, but provide abstracted calls to our drawing. This is where the DrawingBase class comes in.

```
#ifndef DRAWBASE_H
#define DRAWBASE_H

// forward reference
class GraphicsContext;

class DrawingBase
{
    public:
        // prevent warnings
        virtual ~DrawingBase(){}
        virtual void paint(GraphicsContext* gc){}
        virtual void keyDown(GraphicsContext* gc, unsigned int keycode){}
        virtual void keyUp(GraphicsContext* gc, unsigned int keycode){}
        virtual void mouseButtonDown(GraphicsContext* gc, unsigned int button, int x, int y){}
        virtual void mouseButtonUp(GraphicsContext* gc, unsigned int button, int x, int y){}
        virtual void mouseMove(GraphicsContext* gc, int x, int y){}
};
#endif
```

If we provide a DrawingBase reference to our graphics context derivative, our drawbase derived drawing will receive events from our drawing. Now, of course, the graphics context will need to loop to either poll or wait for events from the underlying window system. Thus the point of:

```
virtual void GraphicsContext::runLoop(DrawingBase* drawing) = 0;
```

Of course, this is the pure virtual declaration. Let's take a quick look at the the X11 implementation.

```
void X11Context::runLoop(DrawingBase* drawing)
{
        run = true;

        while(run)
        {
                XEvent e;
                XNextEvent(display, &e);

                // Exposure event - lets not worry about region
                if (e.type == Expose)
                        drawing->paint(this);

                // Key Down
                else if (e.type == KeyPress)
                        drawing->keyDown(this,XLookupKeysym((XKeyEvent*)&e,
                                        (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
                                        (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));

                // Key Up
                else if (e.type == KeyRelease){
                        drawing->keyUp(this,XLookupKeysym((XKeyEvent*)&e,
                                        (((e.xkey.state&0x01)&&!(e.xkey.state&0x02))||
                                        (!(e.xkey.state&0x01)&&(e.xkey.state&0x02)))?1:0));
                                }

                // Mouse Button Down
                else if (e.type == ButtonPress)
                        drawing->mouseButtonDown(this,
                        e.xbutton.button,
                        e.xbutton.x,
                        e.xbutton.y);

                // Mouse Button Up
                else if (e.type == ButtonRelease)
                        drawing->mouseButtonUp(this,
                        e.xbutton.button,
                        e.xbutton.x,
                        e.xbutton.y);

                // Mouse Move
                else if (e.type == MotionNotify)
                        drawing->mouseMove(this,
                        e.xmotion.x,
                        e.xmotion.y);

                // This will respond to the WM_DELETE_WINDOW from the
                // window manager.
                else if (e.type == ClientMessage)
                break;
        }
}
```

This routine accepts X11 specific events and translates them to our abstractions and the events are sent polymorphically to our DrawingBase-derived object.

So, on the implementation end, all we have to do is derive a class from drawbase, and pass a reference to that class-type object to an instance of our chosen GraphicsContext implementation (X11context in this case). Note that in class DrawingBase the various event-handling methods are not abstract. Rather, they are virtual, but “empty” implementations. Thus we can choose whether to override or not. If we do not override, no action will occur. If we want to catch the event, we override. From an efficiency perspective, this is not terribly efficient, but it does offer the ability to “subscribe” to events without the complexity other approaches would require. A good compromise for this application.

3.1.2 Drawing with a Mouse

Looking back at class DrawingBase, it appears that we will get a call when a mouse button goes down, when the mouse moves, and when a mouse button is released. How convenient, as this is the typical use case of drawing a line or triangle.

Consider first drawing a line. In the most naive case we would record the mouse coordinates when the mouse button down event occurs, and when the mouse button is released, get the coordinates again, add the line to our image, and then draw the line.

Most drawing programs you are familiar with probably behave in a slightly different way. Most modern drawing programs implement what is known as rubberbanding. Rubberbanding is a technique where, when a line drawing is selected, for example, the first click of the mouse (or click and hold) establishes the start of a line, and as the mouse moves, the prototypical line is continuously undrawn and redrawn until the second click (or mouse release). In order to achieve this functionality, we need a special drawing mode.

The normal drawing mode is usually referred to as copy mode. That is, the rasterized entity is copied into the frame buffer. For rubberbanding, we need to be able to draw our line, but undraw it when the mouse moves. When we undraw it, we do not want to have to redraw the entire drawing, just the line. Consider the logical operation XOR. If, instead of copying the interim line into the framebuffer, we XOR the line in, we draw the line, but a second XOR will undraw the line.

For example, let's say we are drawing a blue line on a red background:

background (red) ⇒ 0xFF0000 line (blue) ⇒ 0x0000FF xor ⇒ 0xFF00FF ⇐ “temp” line drawn as magenta

magenta ⇒ 0xFF00FF line (blue) ⇒ 0x0000FF xor again ⇒ 0xFF0000 ⇐ get original background color again

This will actually recover the original color of whatever was in the framebuffer, whether it be background color, or another shape.

As an example of how your DrawingBase-derived class should work:

Header file for my DrawingBase-derived class:

```
#ifndef MYDRAWING_H
#define MYDRAWING_H

#include "drawbase.h"

// forward reference
class GraphicsContext;

class MyDrawing : public DrawingBase
{
    public:
        MyDrawing();
        // we will override just these
        virtual void paint(GraphicsContext* gc);
        virtual void mouseButtonDown(GraphicsContext* gc, unsigned int button, int x, int y);
        virtual void mouseButtonUp(GraphicsContext* gc, unsigned int button, int x, int y);
        virtual void mouseMove(GraphicsContext* gc, int x, int y);
    private:
        // We will only support one "remembered" line
        // In an actual implementation, we would also have one of our "image"
        // objects here to store all of our drawn shapes.
        int x0;
        int y0;
        int x1;
        int y1;

        bool dragging; // flag to know if we are dragging
};
#endif
```

Source file for my DrawingBase-derived class:

```
#include "mydrawing.h"
#include "gcontext.h"

// Constructor
MyDrawing::MyDrawing()
{
	dragging = false;
	x0 = x1 = y0 = y1 = 0;
	return;
}

void MyDrawing::paint(GraphicsContext* gc)
{
	// for fun, let's draw a "fixed" shape in the middle of the screen
	// it will only show up after an exposure
	int middlex = gc->getWindowWidth()/2;
	int middley = gc->getWindowHeight()/2;

	gc->setColor(GraphicsContext::MAGENTA);

	for (int yi=middley-50;yi<=middley+50;yi++)
	{
		gc->drawLine(middlex-50,yi,middlex+50,yi);
	}

	gc->setColor(GraphicsContext::GREEN);

	// redraw the line if requested
	gc->drawLine(x0,y0,x1,y1);

	return;
}

void MyDrawing::mouseButtonDown(GraphicsContext* gc, unsigned int button, int x, int y)
{
	// mouse button pushed down
	// - clear context
	// - set origin of new line
	// - set XOR mode for rubber-banding
	// - draw new line in XOR mode.  Note, at this point, the line is
	//   degenerate (0 length), but need to do it for consistency
	x0 = x1 = x;
	y0 = y1 = y;

	gc->setMode(GraphicsContext::MODE_XOR);
	gc->drawLine(x0,y0,x1,y1);
	dragging = true;
	return;
}

void MyDrawing::mouseButtonUp(GraphicsContext* gc, unsigned int button, int x, int y)
{
	if(dragging)
	{
		// undraw old line
		gc->drawLine(x0,y0,x1,y1);

		// just in x and y here do not match x and y of last mouse move
		x1 = x;
		y1 = y;

		// go back to COPY mode
		gc->setMode(GraphicsContext::MODE_NORMAL);

		// new line drawn in copy mode
		gc->drawLine(x0,y0,x1,y1);

		// clear flag
		dragging = false;
	}
	return;
}
```

```
void MyDrawing::mouseMove(GraphicsContext* gc, int x, int y)
{
        if(dragging)
        {
                // mouse moved - "undraw" old line, then re-draw in new position
                // will already be in XOR mode if dragging

                // old line undrawn
                gc->drawLine(x0,y0,x1,y1);

                // update
                x1 = x;
                y1 = y;

                // new line drawn
                gc->drawLine(x0,y0,x1,y1);
        }
        return;
}
```

How it is used:

```
#include "x11context.h"
#include <unistd.h>
#include <iostream>
#include "mydrawing.h"

int main(void)
{
        GraphicsContext* gc = new X11Context(800,600,GraphicsContext::BLACK);

        gc->setColor(GraphicsContext::GREEN);

        // make a drawing
        MyDrawing md;

        // start event loop - this function will return when X is clicked
        // on window
        gc->runLoop(&md);

        delete gc;

        return 0;
}
```

3.2 LAB ACTIVITIES

1. Derive a new class from DrawingBase. This class should compose (contain) an instance of your image class along with variables to remember the state of drawing between events.

2. Establish and implement a protocol for drawing lines and triangles via a combination of keyboard commands and mouse movements. Use the DrawingBase event handlers.

   a. Use the keyboard to set the mode (l for line, t for triangle, etc.).

   b. Once the mode is set, use mouse down, move, and up to draw the entity. Use rubberbanding for all shapes.

   c. Be able to control the color of the drawn entity (via keyboard - perhaps 1 = black, 2 = yellow, etc.).

3. Be able to save and load images. You can use single keypresses with hard-coded files names, or, although inconvenient, you can use std::cin with the console.

4. Add user instructions to your myDrawing constructor that print to console ("press t to select triangle…").

5. You must add at least one of the following (it is up to you which one).

   - Add ability to undo last added shape
   - Add ability to use arrow keys to move the last added shape
   - Add ability to add a filled shape

4 GRADING CHECKLIST

- Submission/Demo received on time
- Check items that will apply to all assignments
  - Submission complies with guidelines (color pdf, no wrapped lines, line numbers, proper ordering, zip file with buildable code)
  - Source code presents as professionally written (comments, indentation, variable naming, structured)
  - Compiles with no errors or warnings (using -Wall)
  - Runs to completion with no crashes or memory leaks (using valgrind)
  - Makes exclusive use of C++ standard library features - no libc unless necessary
- Check items that are particular to this assignment
  - Custom DrawingBase-derived class implemented.
  - Can draw lines and triangles with rubberbanding.
  - Drawn shapes are added to shapes container.
  - Image can be saved and loaded.
  - Window refreshes when resized/overwritten/exposed.