Toturial for Fltk Impressionist

1 Fltk

FLTK (pronounced "fulltick") is a LGPL'd C++ graphical user interface toolkit for X (UNIX), OpenGL, and WIN32 (Microsoft Windows NT 4.0, 95, or 98). It is currently maintained by a small group of developers across the world. As most GUI toolkits, it is based on event-driven programming paradigm.

1.1 Hello, World!

Let's start from the standard first program, hello.exe, which outputs "Hello, World". Figure 1 shows the snapshot of hello.exe.

Listing 1 - "hello.cxx"

```
1 #include <FL/Fl.H>
2 #include <FL/Fl_Window.H>
3 #include <FL/Fl_Box.H>
4
5 int main(int argc, char **argv) {
6   Fl_Window *window = new Fl_Window(300,180);
7     Fl_Box *box = new Fl_Box(FL_UP_BOX,20,40,260,100,"Hello, World!");
8     box->labelfont(FL_BOLD+FL_ITALIC);
9     box->labelsize(36);
10    box->labeltype(FL_SHADOW_LABEL);
11  window->end();
12  window->show(argc, argv);
13  return Fl::run();
14 }
```

In most of your fltk programs, you need to include at least Fl.H and Fl_Window.H. Besides, you also need to include the header files for the widgets you plan to use in the program. For example, we include <Fl_Box.H> in the line 3 because we will use Fl_Box in the program.

Figure 1: The snapshot of hello.exe

In line 6, we create a Fl_Window, window, which is 300-pixel in the width and 180-pixel in the height. In line 11, we call window->end() to finish the design for window. Note that all the widgets declared between line 6 and line 11 are belonged to window. Here, we only declare a Fl_Box widget with the string "Hello, World!". To know how to set the attribute of the widget, you can look up the widget reference in the Fltk programming manual. Click on Fl_Box in the reference, you will see the description and methods for it as shown in figure 2. However, it is not all. At the top of the reference page, you can find that Fl_Box is a derived class of Fl_Widget. So, it will inherit all the methods from Fl_Widget. Click on Fl_Widget on the top, you can see those methods, such as labelfront, labelsize and labeltype, used in the sample program.

After finishing the design for the window, we call window->show to display it (line 12). Line 13 will have the program enter into the infinite message-handle loop until you close the window.

1.2 A More Complicated Example

In this example, we will add a menu, a slider and a button into the window. We will add some boring callback functions, too. It doesn't do anything useful, but you get the idea. Basically speaking, this example creates a dialog to adjust the width of the main window. Besides, there are some trivial functions implemented for demonstrating how to program in fltk.

FLTK 1.0.4 Programming Manual file:///D|/ta/557/tutorial/Fl_Box.html

Contents Previous Next

---

class Fl_Box

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Box
```

Include Files

```
#include <FL/Fl_Box.H>
```

Description

This widget simply draws its box, and possibly it's label. Putting it before some other widgets and making it big enough to surround them will let you draw a frame around them.

Methods

- Fl_Box
- ~Fl_Box

Fl_Box::Fl_Box(int x, int y, int w, int h, const char * = 0)
Fl_Box::Fl_Box(Fl_Boxtype b, int x, int y, int w, int h, const char *)

The first constructor sets `box()` to `FL_NO_BOX`, which means it is invisible. However such widgets are useful as placeholders or `Fl_Group::resizable()` values. To change the box to something visible, use `box(n)`.

The second form of the constructor sets the box to the specified box type.

Fl_Box::~Fl_Box(void)

The destructor removes the box.

Contents Previous Next

1 of 1 1/17/00 11:51 AM

Figure 2: The description for Fl_Box in the reference manual

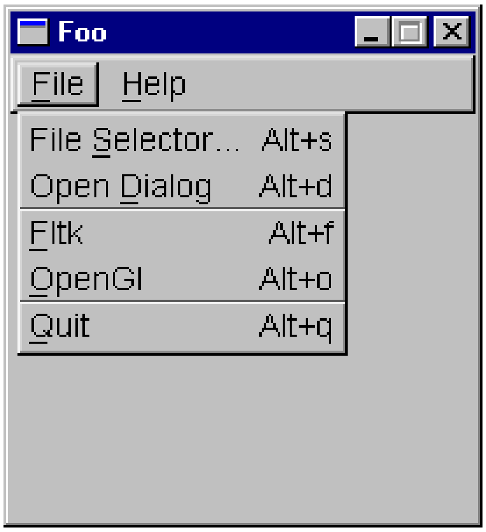
Listing 2 - "fltk_ui.cpp"

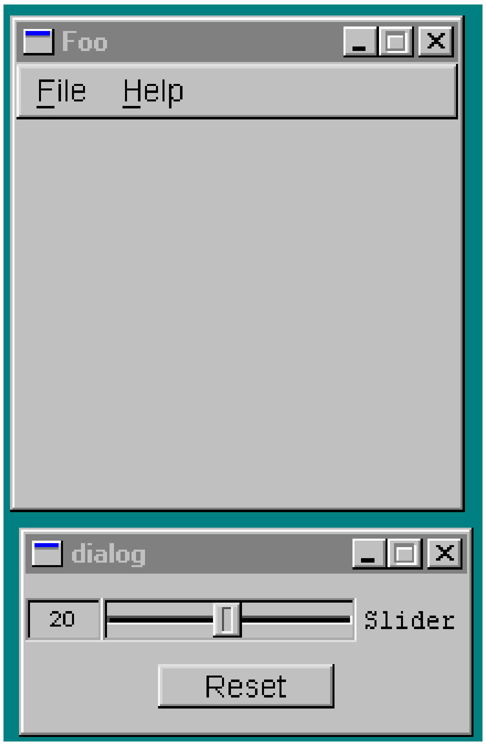
```cpp
#include <stdio.h>

#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Menu_Bar.H>
#include <FL/Fl_Value_Slider.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_Box.H>

#include <FL/fl_file_chooser.H>      // FLTK file chooser
#include <FL/fl_ask.h>               // FLTK message boxes

Fl_Window* window;
Fl_Window* dlg;
Fl_Menu_Bar* menubar;
Fl_Slider* slider;
Fl_Button* button;

void cb_file_select(Fl_Widget*o, void*v) {
    char msg[256];
    char *newfile=fl_file_chooser("Choose a file", "*.cpp", NULL);

    if (newfile!=NULL) {
        sprintf(msg, "You choose %s", newfile);
        fl_message(msg);
    }
}

void cb_open_dialog(Fl_Widget*o, void*v) {
    dlg->show();
}

void cb_pass(Fl_Widget*o, void*v) {
    fl_message((char *)v);
}

void cb_quit(Fl_Widget*o, void*v) {
    dlg->hide();
    window->hide();
}

void cb_about() {
    fl_message("This is about.");
}

void cb_slides(Fl_Widget*o, void*v) {
    window->resize(window->x(), window->y(),
            10*(int)(((Fl_Slider*)o)->value()), window->h());
    menubar->resize(0, 0, 10*(int)(((Fl_Slider*)o)->value()), 25);
}

void cb_reset(Fl_Widget*o, void*v) {
    slider->value(20);
    cb_slides(slider, (void *)20);
}

```

```
57 char fltk[]="fltk"; char opengl[]="opengl";
58
59 Fl_Menu_Item menuitems[] = {
60     { "&File",        0, 0, 0, FL_SUBMENU },
61         { "File &Selector...", FL_ALT + 's', (Fl_Callback *)cb_file_select },
62         { "Open &Dialog",      FL_ALT + 'd', (Fl_Callback *)cb_open_dialog,
63                                                     0, FL_MENU_DIVIDER },
64         { "&Fltk",             FL_ALT + 'f', (Fl_Callback *)cb_pass, (void *)fltk },
65         { "&OpenGl",           FL_ALT + 'o', (Fl_Callback *)cb_pass, (void *)opengl,
66                                                     FL_MENU_DIVIDER },
67         { "&Quit",             FL_ALT + 'q', (Fl_Callback *)cb_quit },
68         { 0 },
69
70     { "&Help",        0, 0, 0, FL_SUBMENU },
71         { "&About", FL_ALT + 'a', (Fl_Callback *)cb_about },
72         { 0 },
73
74     { 0 }
75 };
76
77 int main(int argc, char** argv) {
78     window = new Fl_Window(300, 300, 200, 200, "Foo");
79         // install menu bar
80         menubar = new Fl_Menu_Bar(0, 0, 200, 25);
81         menubar->menu(menuitems);
82
83         window->callback(cb_quit);
84         window->when(FL_HIDE);
85     window->end();
86
87     dlg = new Fl_Window(300, 530, 200, 70, "dialog");
88         // install slider size
89         slider = new Fl_Value_Slider(0, 10, 150, 20, "Slider");
90         slider->type(FL_HOR_NICE_SLIDER);
91         slider->labelfont(FL_COURIER);
92         slider->labelsize(12);
93         slider->minimum(1);
94         slider->maximum(40);
95         slider->step(1);
96         slider->value(20);  // set its value
97         slider->align(FL_ALIGN_RIGHT);
98         slider->callback(cb_slides);
99
100        Fl_Button* button = new Fl_Button(60, 40, 80, 20, "Reset");
101        button->callback(cb_reset);
102
103    dlg->end();
104
105    window->show(argc, argv);
106
107    return Fl::run();
108 }
```

In line 3-7, we include all the header files for the widgets used in the example. In line 10 and 11, we include the other two include files, fl_file_chooser.h and fl_ask.h. These two files define some global functions. The first one is for file filter and selector and the second one is for the popup message box.

The design of an event-driven program is usually divided into two steps: to design the user interface and then to add in the callback functions. The user interface of the sample program looks like:





Corresponding to the design of the menubar, we have the menuitems array defined in line 63-79. For each menu entry, we declare its caption, shortcut and callback function sequentially. When there is an event occurring on the widget, the registered callback function of the widget for that event will be called. The prototype for Fl_Callback in Fl_Widget.H is

typedef void (Fl_Callback)(Fl_Widget*, void*);

Now, let's look at these calback functions for menu entries one by one.

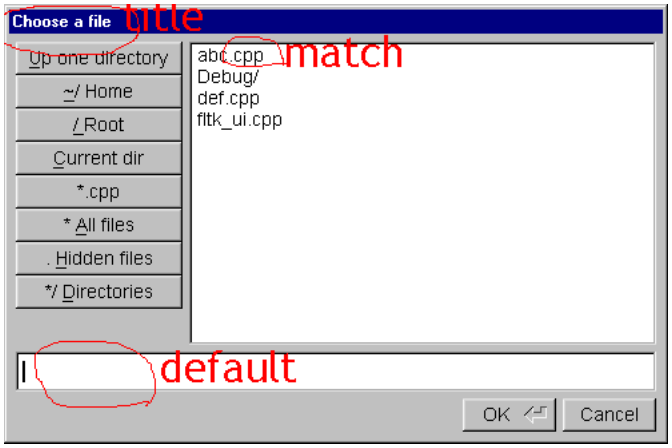


Figure 3: The snapshot of file selection dialog

For "File Selector", the callback function is cb_file_select. It will call fl_file_Chooser(char *title, char *match, char *default) to popup a file selection dialog. It will return the file name chosen or NULL if it is cancelled. If the user do choose some file, the program will pop up a message window by calling fl_message(char *msg) to display the chosen file name.

For "fltk" and "opengl" menu entries, I just demonstrate how to pass parameters into the callback function. They share the same callback function, cb_pass, but with the different parameters. The message box will prompt the passing parameters appropriately.

For "Quit", I just call window->hide() to destroy the window. We destroy the dialog by calling hide() as well. Since all windows are destroyed, the Fl:run() will return and it will terminate the program properly.

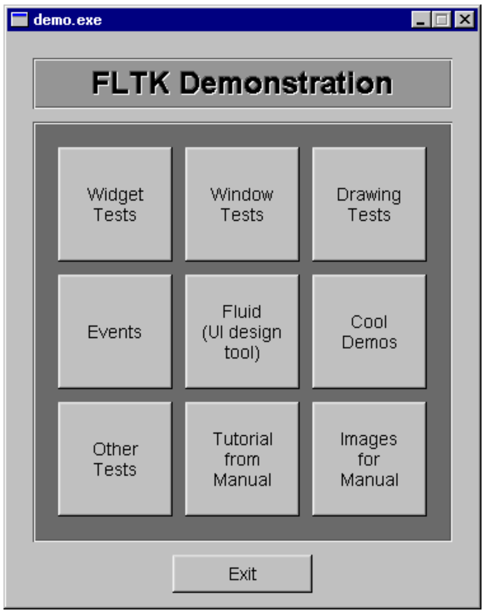
Now, let's come back to the layout of the windows. In this example, we have two windows, the main window and the dialog window. They are the same for fltk and we just distinguish them conceptually. In line 78-85, we set the layout for the main window. We first install a menubar in line 80-81. In line 83-84, we install a callback function for the event FL_HIDE. Or, if the user terminates the program by pressing ESC, then the dialog won't be terminated since no callback function for that event is registered. We do this to force the conceptual child window to terminate when its parent window terminates.

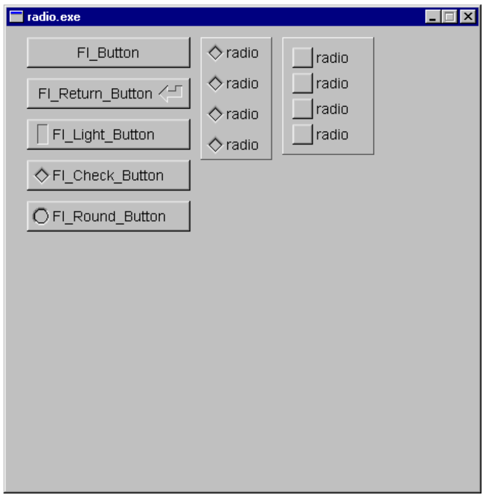
In the dialog, we have a slider and a button. The setting for them should be quite obvious. When the position of the slider changes, the callback function, cb_slide, will be called. In this function, we will resize the window and the menubar such that the width is 10 times of the value in the slider. The member function, x(), y(), w() and h(), will return the current position and dimension for that widget. resize() will change the size and position of the widget.

For the reset button, we reset the value of the slider back to 20 and call cb_slide to change the size of the window consistently. Note that overloaded function, value(), is used both to set or read the value of the slider.

1.3 How to Learn More About The Widgets

The best way to learn what you can use and how to use them is to look at the demo examples included in fltk distribution. Run 'demo' and go through the demos. When you find the widget you might want, go to its source. The source is usually a single file with less than 300 lines of code. You can also read the reference manual to know more about the widgets.





2 OpenGL

OpenGL is a 2D/3D Graphics Library developed at Silicon Graphics Inc. It has been widely used in computer graphics industry. It is a must to work in this field. The best book for learning OpenGL is the OpenGL Programming Guide, Second Edition. We have several Copies of this book available in Sieg 228. You can also look at http://www.opengl.org for more information. If you are working on NT, MSDN also includes the reference to OpenGL functions[1].

OpenGL is a state machine. It will keep various states, such as the drawing color, point size, line width, buffer to write and so on. Those states will keep the same until you change them explicitly. Many state variables are switched on or off by glEnable() or glDisable() commands. For example, you can turn off depth test by issuing glDisable(GL_DEPTH_TEST). Besides, you will need to turn on the blend function by calling glEnable(GL_BLEND) to have alpha blending effect.

2.1 OpenGL Conventions

All the functions in the OpenGL library have names beginning with "gl". Defined constants have names beginning with "GL_".

OpenGL has its own definitions for variable types. They are simply redefinitions of the basic types; GLint is simply an int. It is better to use these definitions when you program in OpenGL.

To overcome the lack of overload functions, OpenGL use the following convention for a family of procedures with the same function but different arguments. The ends of the names for these functions work as the tags to indicate the type of arguments. For instance, glColor*() refers to any of the 32 functions available within OpenGL for setting the current color, for example,

- glColor3f(GLfloat, GLfloat, GLfloat) -Takes 3 floats.
- glColor4d(GLdouble, GLdouble, GLdouble, GLdouble) - Takes 4 doubles (the fourth is the alpha value)

[1] the material in this section is adapted from the previous tutorials for CSE457

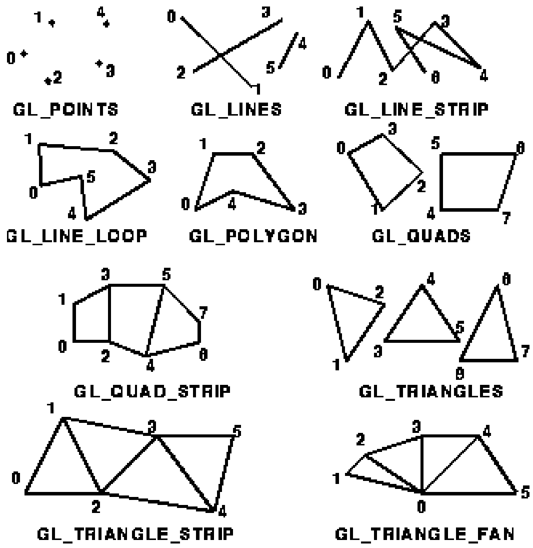


Figure 4: OpenGL primitives

- glColor3ubv(GLubyte*) - Takes a vector (or array) containing 3 unsigned bytes.

2.2 OpenGL Primitives

We will use OpenGL in the Impressionist program to draw the various brush strokes. Figure 4 nicely illustrates the primitives and the arrangement for the vertices in OpenGL.

The typical calling sequence for drawing a primitive looks like:

```
glColor4f( red, green, blue, alpha);
glBegin( GL_LINE_STRIP );
    glVertex2i( Ax, Ay );
    glVertex2i( Bx, By );
    glVertex2i( Cx, Cy );
    glVertex2i( Ax, Ay );
glEnd();
glFlush()
```

glColor*() specifies the color in which to draw primitive(s). glBegin() and glEnd() delimit the vertices of a primitive. The assignment of the vertices to the primitives is illustrated in figure 4 . glVertex*() specifies the coordinate of the vertices. Finally, glFlush() tells OpenGL to draw the primitives now.

2.3 Basic OpenGL Transformations

Recall that OpenGL is basically a state machine. For many aspects of it, you set up certain parameters, and until you change them, GL will use those parameters for everything it draws.

You have probably already seen how this is used for things like object color (via glColor) and drawing mode (via glBegin / glEnd). However, there are also state variables for things like position (accomplished via "translation", or shifting) and direction (accomplished via rotation).

For example, you can call glRotate* to set the rotation state. If you tell OpenGL to rotate 45 degrees around the z axis (with glRotate3f(45, 0.0, 0.0, 1.0)), then everything you draw will be rotated 45 degrees before it's drawn to the screen. This is a quick and easy way of changing the location and orientation of an object.

So how does this apply to Impressionist? Recall that you'll be drawing various brush strokes on a digital canvas, each at a different position and in a different direction. By setting the GL state variables for position and orientation before drawing your brush, you can use the same code regardless of the brush's position or orientation. The stroke will automatically be drawn at the correct position and in the correct direction! Of course, you can do this by yourself. And, sometimes, you have to do it by yourself, for example, to clip lines by edges.

Here are the OpenGL calls needed to do some simple image transformations.

1. Choosing the right matrix: There are several matrices in OpenGL. The projection matrix is used to control the camera position, and the modelview matrix is used to control drawing. We want to use the modelview matrix, so we need to explicitly tell that to OpenGL with a call to glMatrixMode:

   glMatrixMode(GL_MODELVIEW);

2. Pushing/Popping Matrices: Image transformations are accomplished using matrices. A series of matrices are multiplied to produce a given

image transformation. Without going into too much detail, let's just say that you'll want to save your current transformation matrix (”push” it onto a matrix stack) before doing your brush-specific translation/rotation, and restore the original matrix (”pop” it off the matrix stack) when you're done with that brush stroke. If you are not sure what already happened in the matrix stack, you can call glLoadIdentity() to clear the currently modifiable matrix for future transformation.

Here are the calls you need to use.

```
glPushMatrix();

<&&ltDo the translation, rotation>>
<&&ltDraw the brush stroke>>

glPopMatrix();
```

For translation, if want to ”translate” your origin to that position. Here is the call you use in OpenGL to do 2D translation:

glTranslate*(startX, startY, 0.0);

(Note: the * is replaced by a letter that depends on the type of the parameters you pass.)

Here is the OpenGL call you'll want to use for rotation:

glRotate*(angle, 0.0, 0.0, 1.0);

(Note: we use 0.0, 0.0, 1.0 because we want to rotate around the z-axis.)

2.4 Manipulation of Frame Buffer

To fast save and restore the canvas, we use some frame buffer manipulation functions provided by OpenGL. They are used to fast read out/write in the frame buffer to/from a block of memory.

glReadPixels() and glWritePixels() are used read/write a block of pixels from/to the frame buffer. Before you call them, you need to specify several things:

- glReadBuffer/glWriteBuffer: to specify which buffer you will work on.
- glPixelStore*: to specify how pixels are arranged
- glRasterPos*: set the start point for pixel write operations

3 Fltk+OpenGL

In this section, we will introduce how to use OpenGL in fltk. The most convenient way is to subclass Fl_Gl_Window. The only restriction is that you can only invoke OpenGL drawing functions in draw(). Whenever you want to update the display, call redraw() and fltk will call draw() later. Since all the drawing functions must be called within draw(), you need to have some mechanism to inform draw() what it should do. In the skeleton code, we set some variables, isAnEvent and eventToDo, for this purpose. The better way may be to maintain a message queue for it?

3.1 Make a subclass of Fl_Gl_window

To make a subclass of Fl_Gl_Window, you must provide [2]:

- A class definition
- A draw() method
- A handle() method

3.1.1 Defining the Subclass

To define the subclass you just subclass the Fl_Gl_Window class:

```
class MyGLWindow : public Fl_Gl_Window {
  void draw();
  int handle(int);
public:
  MyGLWindow(int X, int Y, int W, int H, const char *L)
    : Fl_Gl_Window(X, Y, W, H, L) {}
};
```

3.1.2 The draw() Method

```
void MyGLWindow::draw() {
  if (!valid()) {
    // whenever the window size is changed, valid will be turned off
    // and turned on after the first call to draw()
    // you need to set up projection, viewpoint ...
    // get the dimension of the window from w(), h()
  }
  // put your drawing operations here
}
```

[2]the material in this section comes from chap. 9 in *Fltk Programming manual*

3.1.3 The handle() Method

The handle() method is used to handle mouse and keyboard events for the window. Note that you can't call any drawing functions within handle() since the OpenGL context is not set up yet! Call redraw() and let draw() do the work.

```
int MyGLWindow::handle(int event) {
  switch (event) {
    case FL_PUSH:
        coord.x = Fl::event_x();
        coord.y = Fl::event_y();
        if (Fl::event_button()>1)
            eventToDo=RIGHT_MOUSE_DOWN;
        else
            eventToDo=LEFT_MOUSE_DOWN;
        isAnEvent=1;
        redraw();
      return 1;
    case FL_DRAG:
        coord.x = Fl::event_x();
        coord.y = Fl::event_y();
        if (Fl::event_button()>1)
            eventToDo=RIGHT_MOUSE_DRAG;
        else
            eventToDo=LEFT_MOUSE_DRAG;
        isAnEvent=1;
        redraw();
      return 1;
    case FL_RELEASE:
        coord.x = Fl::event_x();
        coord.y = Fl::event_y();
        if (Fl::event_button()>1)
            eventToDo=RIGHT_MOUSE_UP;
        else
            eventToDo=LEFT_MOUSE_UP;
        isAnEvent=1;
      return 1;
    default:
      // tell FLTK that I don't understand other events
      return 0;
  }
}
```

3.2 An Example

In this section, we will present a extremely simplified impressionist program. This program has a canvas of the size 300x300. When user click the left mouse button, it will draw a 20x20 square at the clicked position. Clicking right mouse button will pick up the drawing color among red, green and blue sequentially. If the left button is clicked outside the canvas, the square

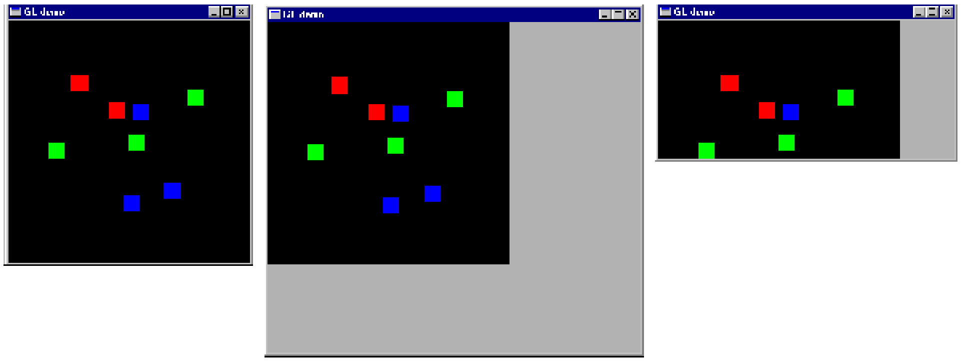


Figure 5: The snapshot of fltk GL example

will still be drawn but they will disappear when the window is resized. Any drawing within the canvas is guaranteed to be kept. Anything outside is not. Figure 5 is the snapshot for the sample program.

In line 6, we first create a Fl_Gl_Window as described in the previous section. We call window->resizable(window) in line 8 such that the window can be resized by the user. You should be able to understand the content of MyGLWindow.h now. In class MyGLWindow, we maintain the variables, windowWidth and windowHeight, for the dimension of the window. The variables, drawWidth and drawHeight, are for the dimension of the canvas. Figure 6 illustrates the relationship between those variables and the dimension we used to save and restore. The other things should be quite obvious to you now.

For the mysterious fltk draw() function, I still don't quite know how it is implemented. However, here is the way I think it works:

- It will set GL content for you before draw() is called.
- The default buffer it draw is the back buffer.
- It will automatically swap the buffers and make both buffers the same as the back buffer.
- When window is resized, that is, the frame buffers are destroyed and recreated, all the content in the frame buffers is gone. It is why we need to have a buffer in the memory to keep the content of the canvas.

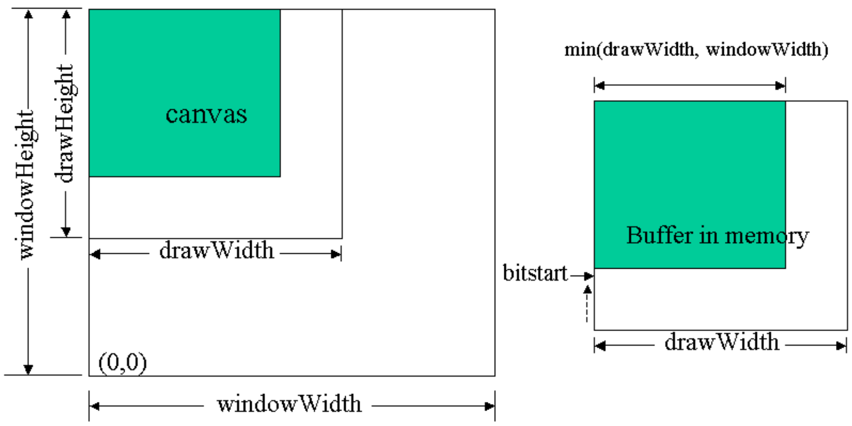


Figure 6: Illustration for drawWidth and windowWidth

Listing 3 - "main.cpp"

```
#include "MyGLWindow.h"

MyGLWindow *window;

int main(int argc, char** argv) {
    window = new MyGLWindow(200, 200, 300, 300, "GL_demo");

    window->resizable(window);

    window->show();

    return Fl::run();
}
```

Listing 4 - "MyGLWindow.h"

```
#ifndef _MY_GL_WINDOW_H_
#define _MY_GL_WINDOW_H_

#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

class MyGLWindow : public Fl_Gl_Window {
    void draw();
    int handle(int);
public:
    MyGLWindow(int X, int Y, int W, int H, const char *L);

    void SaveCurrentContent();
```

```
    void RestoreContent();

    int windowWidth, windowHeight;
    int drawWidth, drawHeight;

    int mx, my;
    int isAnEvent;
    int eventToDo;

    int curColor;

    unsigned char *buf, *bitstart;
};

#endif
```

Listing 5 - "MyGLWindow.cpp"

```
#include "MyGLWindow.h"

#define LEFT_MOUSE_DOWN          1
#define RIGHT_MOUSE_DOWN         2

#ifndef WIN32
#define min(a, b)  ( ((a)<(b)) ? (a) : (b) )
#endif

MyGLWindow::MyGLWindow(int X, int Y, int W, int H, const char *L)
        : Fl_Gl_Window(X, Y, W, H, L)
{
    windowWidth = drawWidth = W;
    windowHeight = drawHeight = H;

    buf = new unsigned char [W*H*3];
    memset( buf, 0, W*H*3 );

    curColor = 0;
}

static GLubyte drawColor[3][3]={{255,0,0}, {0,255,0}, {0,0,255}};

void MyGLWindow::draw() {
    if (!valid()) {
        glClearColor(0.7f, 0.7f, 0.7f, 1.0);

        glDisable( GL_DEPTH_TEST );

        ortho();

        glClear( GL_COLOR_BUFFER_BIT );      // clear the window

        windowWidth = w();
        windowHeight = h();

        int startrow = drawHeight - min( drawHeight, windowHeight );
        if ( startrow < 0 ) startrow = 0;
```

```
        bitstart = buf + 3 * ((drawWidth * startrow));

        RestoreContent();
    }

    if ( !isAnEvent) {
        RestoreContent();
    } else {
        isAnEvent=0;    // clear it after processing

        switch (eventToDo) {
            case LEFT_MOUSE_DOWN:

                glColor3ubv(drawColor[curColor]);
                glBegin( GL_QUADS );
                    glVertex2d(mx,      (windowHeight-my));
                    glVertex2d(mx,      (windowHeight-my)-20);
                    glVertex2d(mx+20,   (windowHeight-my)-20);
                    glVertex2d(mx+20,   (windowHeight-my));
                glEnd();

                SaveCurrentContent();

                break;
            case RIGHT_MOUSE_DOWN:
                curColor=(curColor+1)%3;
                break;
        }
    }
}

int MyGLWindow::handle(int event) {
    switch (event) {
        case FL_PUSH:
            mx = Fl::event_x();
            my = Fl::event_y();
            if (Fl::event_button()>1)
                eventToDo=RIGHT_MOUSE_DOWN;
            else
                eventToDo=LEFT_MOUSE_DOWN;
            isAnEvent=1;
            redraw();
        return 1;

        default:
            // tell FLTK that I don't understand other events
        return 0;
    }

    return 0;
}

void MyGLWindow::SaveCurrentContent()
{

    glReadBuffer(GL_BACK);

```

```
    glPixelStorei( GL_PACK_ALIGNMENT, 1 );
    glPixelStorei( GL_PACK_ROW_LENGTH, drawWidth );

    glReadPixels( 0, windowHeight - min(drawHeight, windowHeight) ,
                  min(drawWidth, windowWidth), min(drawHeight, windowHeight),
                  GL_RGB, GL_UNSIGNED_BYTE, bitstart );
}

void MyGLWindow::RestoreContent()
{
    glDrawBuffer(GL_BACK);

    glRasterPos2i( 0, windowHeight - min(drawHeight, windowHeight) );
    glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
    glPixelStorei( GL_UNPACK_ROW_LENGTH, drawWidth );
    glDrawPixels( min(drawWidth, windowWidth), min(drawHeight, windowHeight),
                  GL_RGB, GL_UNSIGNED_BYTE, bitstart );
}
```

4 Impressionist

4.1 The Structure of Impressionist

The ancestor of fltk impressionist is MFC impressionist. Inherited from it, this version also adapts the Document-View architecture. Figure 7 illustrates the relationship among all these classes.

4.2 Some Tips

To enable alpha, you need to enable GL_BLEND and set the blend function properly using glBlendFunc().

To calculate the gradient of an image, you can follow the three steps:

1. Convert to gray-level image: use the formula, I=0.299R + 0.587G + 0.114B to convert RGB tuple into gray level.

2. Blur the gray-level image: use some filter to remove the effect of the noise and the unnecessary details.

3. Calculate gradient: you can use the following kernels to calculate the gradient for x-direction and y-direction.

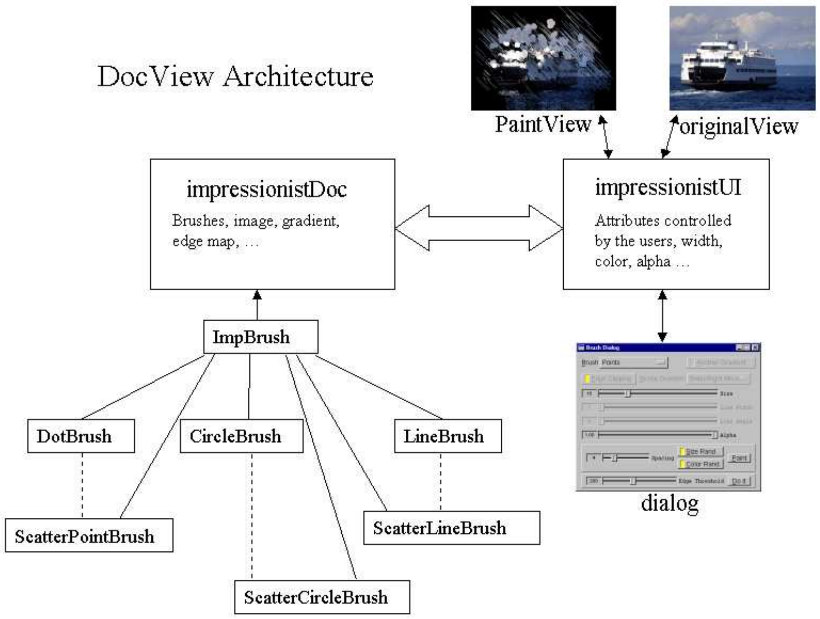


Figure 7: The Structure of Impressionist

1	0	-1		1	2	1
2	0	-2		0	0	0
1	0	-1		-1	-2	-1

or simply,

-1	1		-1
			1

Note that the result is an un-normalized vector. You may want to normalize it before you use it. Besides, the first pair of the kernel is called Sobel filter, you can use it to determine if a pixel is an edge pixel. When the magnitude of the gradient is larger than the specified threshold, you can say that the pixel is an edge pixel.