GUI支持代码分析报告

王晨 PE20060014

## 一、需求说明

程序使用者需要在程序使用过程中，通过图示化显示以便使用者即使不知道底层实现方式也能够操作，及时得到反馈并显示结果。（所见即所得）

## 二、图形化显示的代码分层结构

注：Simplewindow.h严格来说不在GUI层次中

Simplewindow.h

GUI.h

Driver

Graph.h

Window.h

point.h

User’s code

OS/System call

Our screen

GUI library/interface

FLTK Library

## 二、概要设计和实现描述

## 1-point.h中的主要接口类

point.h中只定义了坐标点（x,y）的结构体，以及重载运算符==和！=用于判断两点坐标是否相同。

struct Point {

int x, y;

Point(int xx, int yy) : x(xx), y(yy) { }

Point() :x(0), y(0) { }

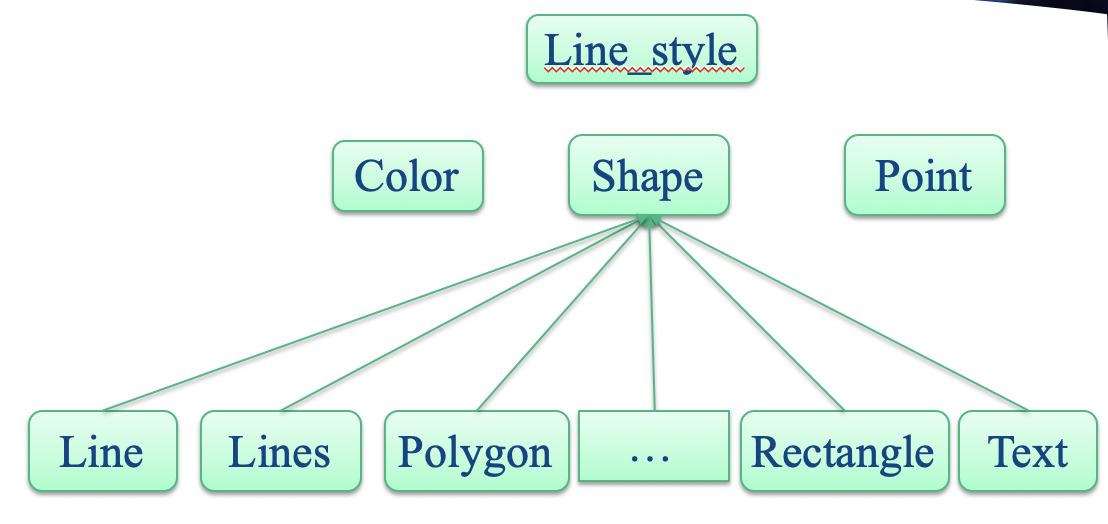
};

inline bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }

inline bool operator!=(Point a, Point b) { return !(a==b); }

Point.h在被上层的window.h和graph.h调用。

## 2-Graph.h/cpp中的主要接口类



Graph.h中定义了一些诸如颜色color，线条形式Line\_sytle，字体Font等辅助类，以及Shape基类，Shape基类是所有图形所共有的东西，因此shape是不能被实例化的，之后graph中的图形都通过直接或间接继承shape而来。

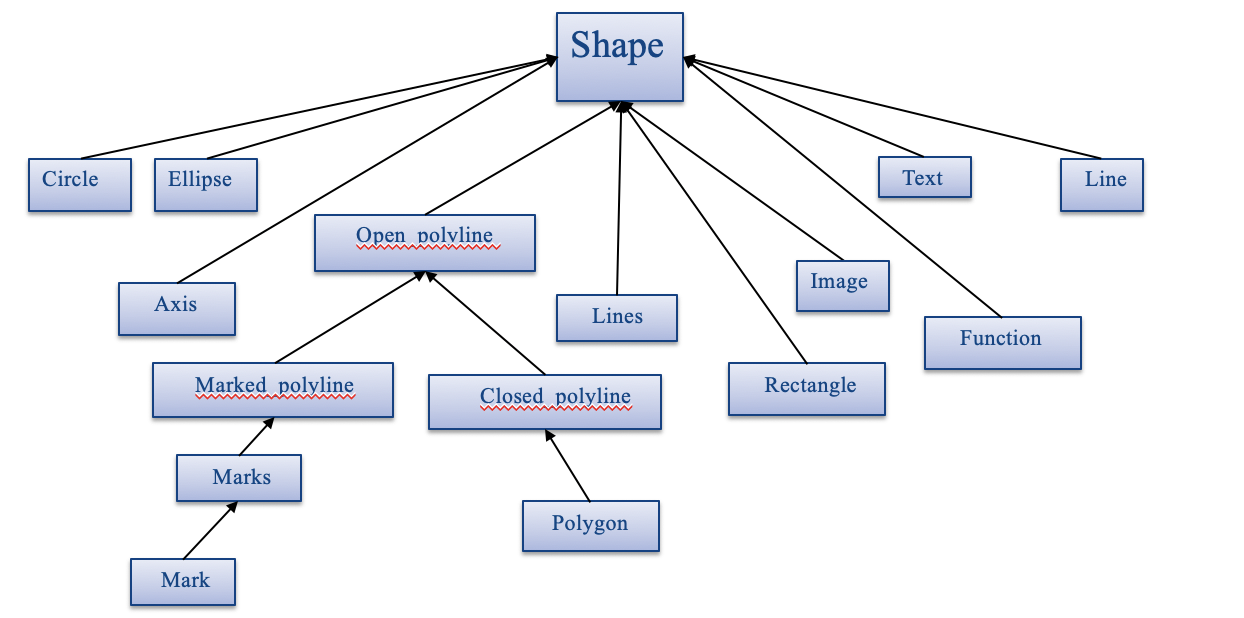
例如：

struct Line : Shape //由两个point组成的shape

struct Open\_polyline : Shape //就是由多个点集组成的开shape

struct Circle : Shape //定义了圆心点和半径的shape

等等。



还有一些图形由继承自shape的图形继承而来以增加特性或者实现特例化，例如：

struct Closed\_polyline 🡨 struct Open\_polyline 🡨 Shape

Marked\_polyline 🡨 Open\_polyline 存储 string mark (13.14) —— 增加功能

Marks 🡨 Marked\_polyline 设置 Color::invisible (13.15) ——特例化

Mark 🡨 Marks 设置 string mark = string(1,c) (13.16) —— 特例化

在graph.cpp中定义了诸如shape类中的基本方法，例如：

Shape::add(Point p) //添加一个点

void Shape::draw\_lines() const //绘制线段，实现方式是调用fltk中的基本绘制函数fl\_line

void Shape::move(int dx, int dy) // 图形的移动

继承自shape的图形子类在实现他们各自方法时会继承shape类的基本方法再加拓展，例如

Line::Line(Point p1, Point p2) //构建两点间的线段

{

Shape::add(p1);

Shape::add(p2);

}

void Rectangle::draw\_lines() const //绘制三角形，fl\_rectf是fltk中三角形的基本绘制函数

{

if (fill\_color().visibility()) { // fill

fl\_color(fill\_color().as\_int());

fl\_rectf(point(0).x,point(0).y,w,h);

}

if (color().visibility()) { // lines on top of fill

fl\_color(color().as\_int());

fl\_rect(point(0).x,point(0).y,w,h);

}

}

可以看到在Graph中类的方法中，最终都是调用fltk的基本函数实现的，例如：

fl\_line();

fl\_color();

fl\_begin\_complex\_polygon();

fl\_draw();

fl\_font();

等等

## 3-window.h/cpp中的主要接口类

Window是我们对GUI库的主要接口 (对计算机屏幕的接口)。在文件中有class Widget声明和 class Window的定义。其中window类由fltk中的Fl\_Window基类继承而来：

class Window : public Fl\_Window {

public:

// let the system pick the location:

Window(int w, int h, const string& title);

// top left corner in xy

Window(Point xy, int w, int h, const string& title);

virtual ~Window() { }

int x\_max() const { return w; }

int y\_max() const { return h; }

void resize(int ww, int hh) { w=ww, h=hh; size(ww,hh); }

void set\_label(const string& s) { copy\_label(s.c\_str()); }

void attach(Shape& s) { shapes.push\_back(&s); }

void attach(Widget&);

void detach(Shape& s); // remove s from shapes

void detach(Widget& w); // remove w from window (deactivates callbacks)

void put\_on\_top(Shape& p); // put p on top of other shapes

protected:

void draw();

private:

vector<Shape\*> shapes; // shapes attached to window

int w,h; // window size

void init();

};

Window类中包含以下几个方法，用来实现窗口，以及将shape和widget显示在窗口上：

void Window::init()

void Window::draw()

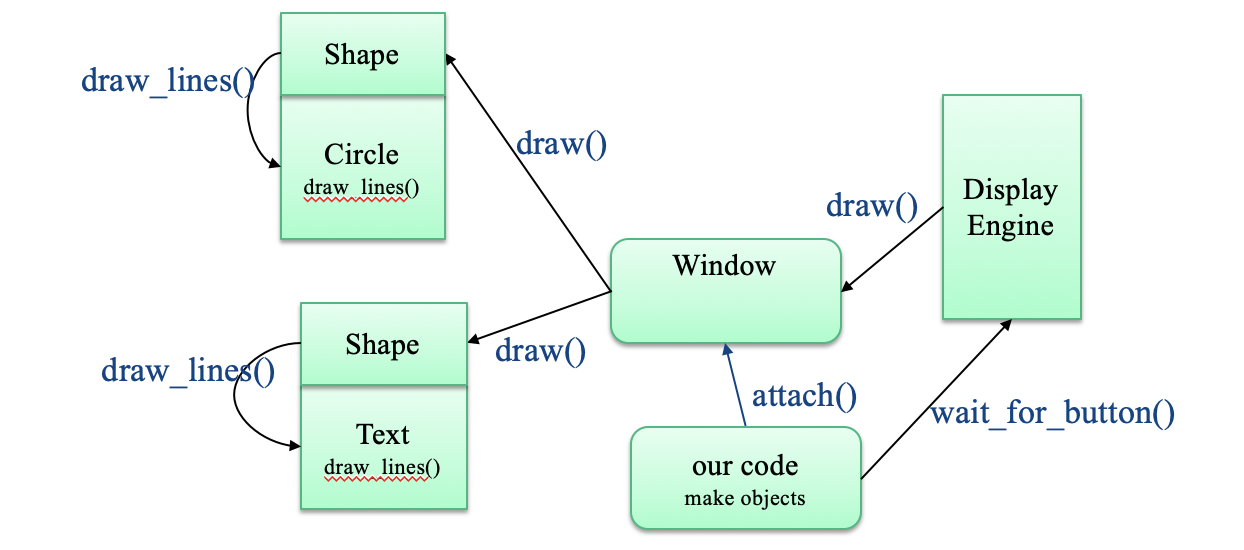
void Window::attach()

void Window::detach()

void Window::put\_on\_top()

同样的，这些方法最终也是通过调用fltk的基本函数实现的。

完整的显示模型如下：



## 4-GUI.h/cpp中的主要接口类

GUI.h严格来说是GUI Library层次中最上层的接口函数，Simplewindow.h/cpp严格来说不在GUI层次中。GUI的任务是让用户函数和I/O接口分离，提供一些基本的类例如Button, In\_box, Out\_box给用户使用，触发事件动作(回调函数)。

在GUI.h中定义了Widget类，(Widget) 是窗口中具有关联动作的控件。用于派生诸如Button,Inbox,Outbox，Menu等的派生类。

class Widget {

// Widget is a handle to an Fl\_widget - it is \*not\* an Fl\_widget

// We try to keep our interface classes at arm's length from FLTK

public:

Widget(Point xy, int w, int h, const string& s, Callback cb)

: loc(xy), width(w), height(h), label(s), do\_it(cb)

{}

virtual void move(int dx,int dy) { hide(); pw->position(loc.x+=dx, loc.y+=dy); show(); }

virtual void hide() { pw->hide(); }

virtual void show() { pw->show(); }

virtual void attach(Window&) = 0;

Point loc;

int width;

int height;

string label;

Callback do\_it;

virtual ~Widget() { }

protected:

Window\* own; // every Widget belongs to a Window

Fl\_Widget\* pw; // connection to the FLTK Widget

private:

Widget& operator=(const Widget&); // don't copy Widgets

Widget(const Widget&);

};

从上面Widget的代码中可以看出，widget本质上和window是一致的，可以把widget看成一个小的window。不同的是widget类中声明了Callback do\_it回调函数，用于触发实现事件动作。

struct Button : Widget {

Button(Point xy, int w, int h, const string& label, Callback cb)

: Widget(xy,w,h,label,cb)

{}

void attach(Window&);

};

可以看到button定义中表明，当用户按下一个button后，相关的回调函数会被触发。而回调函数是由用户自己定义的。

## 5-Simplewindow.h/cpp —— 一个简单图形化窗口实现

struct Simple\_window : Window {

Simple\_window(Point xy, int w, int h, const string& title );

bool wait\_for\_button(); // simple event loop

private:

Button next\_button; // the "next" button

bool button\_pushed; // implementation detail

static void cb\_next(Address, Address); // callback for next\_button

void next(); // action to be done when next\_button is pressed

};

Simple\_window::Simple\_window(Point xy, int w, int h, const string& title) :

Window(xy,w,h,title),

next\_button(Point(x\_max()-70,0), 70, 20, "Next", cb\_next),

button\_pushed(false)

{

attach(next\_button);

}

bool Simple\_window::wait\_for\_button()

// modified event loop:

// handle all events (as per default), quit when button\_pushed becomes true

// this allows graphics without control inversion

{

show();

button\_pushed = false;

#if 1

// Simpler handler

while (!button\_pushed) Fl::wait();

Fl::redraw();

#else

// To handle the case where the user presses the X button in the window frame

// to kill the application, change the condition to 0 to enable this branch.

Fl::run();

#endif

return button\_pushed;

}

void Simple\_window::cb\_next(Address, Address pw)

// call Simple\_window::next() for the window located at pw

{

reference\_to<Simple\_window>(pw).next();

}

void Simple\_window::next()

{

button\_pushed = true;

hide();

}

next()函数是button触发后的回调函数，这里可以根据程序的实际需求或功能由开发者自己定义，例如在绘图程序中，在输入两个坐标点后点击next\_button连线，那么next（）可以定义为：

void Lines\_window::next()

{

int x = next\_x.get\_int();

int y = next\_y.get\_int();

lines.add(Point(x,y)); // *update current position readout:*

stringstream ss;

ss << '(' << x << ',' << y << ')';

xy\_out.put(ss.str());

redraw(); // *now redraw the screen*

}