

第一章 FLTK 编程模型

1.1 FLTK 功能简介

1.2 搭建 FLTK 开发环境

1.21 windows 下搭建 FLTK 开发环境

1.22 ubuntu 下搭建 FLTK 开发环境

1.3FLTK 构件简介

1.4FLTK 事件处理

1.5FLTK 消息处理

1.6OpenGL 编程

第二章 常用的控件和属性

2.1 按钮

2.2 文本

2.3 颜色

2.4 Box 类型

第三章 FLTK 的画图函数

3.1 何时可以画图

3.2 FLTK 的画图函数

3.3 剪切

3.4 颜色

3.5 设置线条的属性

3.6 画一般的图形函数

3.7 画封闭的线，一次连接个顶点

3.8 画三边形或四边形，并填充内部

3.9 复杂图形函数

3.10 字体

3.11 覆盖画图函数

第四章 在 FLTK 中自定义控件

4.1 定制图形控件说明

4.2 如何开发一个控件的子类

4.3 处理事件

4.4 画控件

第一章：FLTK 编程模型

FLTK(Fast Light Tool Kit 发音为 fulltick) 是一种使用 C++开发的 GUI 工具包，它可以应用于 Unix,Linux,MS-Windows95/98/NT/2000 和 MacOS 操作系统平台，相对于其它的许多图形接口开发工具包（如 MFC、GTK、QT 等），它具有体积很小、速度比较快，且有着更好的移植性。FLTK(Fast Light Tool Kit 发音为 fulltick) 是一种使用 C++开发的 GUI 工具包，它可以应用于 Unix，Linux，MS-Windows95/98/NT/2000 和 MacOS 操作系统平台，相对于其它的许多图形接口开发工具包（如 MFC、GTK、QT 等），它具有体积很小、速度比较快，

且有着更好的移植性。本文就 FLTK 编程的一些基本方法进行介绍。

1.1 FLTK 功能简介

(1)提供丰富的跨平台的 GUI 构件(Widget)。有按钮，菜单，窗口等，近六十个。

(2)支持 OpenGL，提供 Fl_GL_Window，支持 OpenGL 相关的操作。

(3)提供界面设计工具 FLUID，非常方便进行界面的设计。

(4)良好的跨平台移植性。

(5)支持多种 C++编译器，Gcc，BC，VC 等等。

(6)灵活性。FLTK 本身可以定制，以满足不同的需要。这使得 FLTK 在嵌入式开发上有着极大的竞争力，这正是我要推荐使用 FLTK 的原因。

本文就 FLTK 编程的一些基本方法进行介绍。

1.2 搭建 FLTK 开发环境

安装 FLTK 很简单，我们只需要下载它的源文件，解压缩到目录下，在 Linux 下我们只需要输入 make，编译完成然后 make install 就头文件安装到/usr/include/FL 目录下。库文件就在 /usr/lib 下，也可以自己编译之后把这些文件复制到这些目录，或者不需要复制，只在编译连接的时候指定路径。在 window 下可以使用 VC,BC 打开相应目录下的工程文件编译即可。

1.2.1 windows 下搭建 FLTK 开发环境

第一步:下载 FLTK 源码包

FLTK 官网:<http://www.fltk.org/>

下载后解压缩到 C 盘根目录下,命名为 FLTK

进到 C:\FLTK\ide\visualc 目录下,找到 fltk.dsw

用 Visual C++6.0 打开项目，然后选择【组建】-->全部重建，就开始编译了

编译完成后关闭 Visual C++6.0

第二步:添加 FLTK 库文件

(1)重新打开 Visual C++6.0, 新建一个 Win32 Application 项目,命名为 FLTK,然后再新建一个 hello.cxx 文件

(2)建立好之后选择【工程】->【设置】-->选择“连接”选项卡-->“分类”下拉框选择输入，在对象/

库模块添加：

fltkd.lib fltkgld.lib comctl32.lib wsock32.lib opengl32.lib glu32.lib

还要在忽略库中添加：LIBCD libcd.lib

(3)之后选择“C/C++”选项卡,"分类"下拉框选择：code generation, 在“use run-time library”

中选择” Multi-threaded DLL “ 最后确定。

(4)选择【工具】-->【选项】--> "目录"选项卡

在” 目录 “下拉框中选择"Include Files" 然后新增一项 C:\FLTK(导入头文件)

(5)在 “目录 “下拉框中选择"Library Files” 然后新增一项 C:\FLTK (导入类库)

(6)编译运行 hello.cxx

1.22 ubuntu 下搭建 FLTK 开发环境

第一步：配置基础开发环境 GCC

```
xhy@xhy-desktop:~$sudo apt-get install build-essential
```

第二步：安装 QT 开发环境

```
xhy@xhy-desktop:~$sudo apt-get install qt4-dev-tools qt4-doc qt4-qconfig qt4-demos qt4-designer
```

第三步：下载 FLTK 源码包

FLTK 官网:<http://www.fltk.org/>

下载后解压缩：

```
xhy@xhy-desktop:~$sudo tar zxvf FLTK.tar.gz
```

第四步：编译安装 FLTK

```
xhy@xhy-desktop:~$cd FLTK
```

```
xhy@xhy-desktop:~$make
```

```
xhy@xhy-desktop:~$sudo make install
```

第五步：测试环境

写一个简单的 FLTK 程序

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

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

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

```
int main(int argc, char **argv)
```

```
{
```

```
    Fl_Window *window = new Fl_Window(300,180);
```

```
    Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
```

```
    box->box(FL_UP_BOX);
```

```
    box->labelsize(36);
```

```
    box->labelfont(FL_BOLD+FL_ITALIC);
```

```
    box->labeltype(FL_SHADOW_LABEL);
```

```
    window->end();
```

```
    window->show(argc, argv);
```

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

```
}
```

包含了需要的头文件后，该程序创建了一个窗口

```
Fl_Window *window = new Fl_Window(300,180);
```

还创建了一个 box 类，标签是 “Hello World!”

```
Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
```

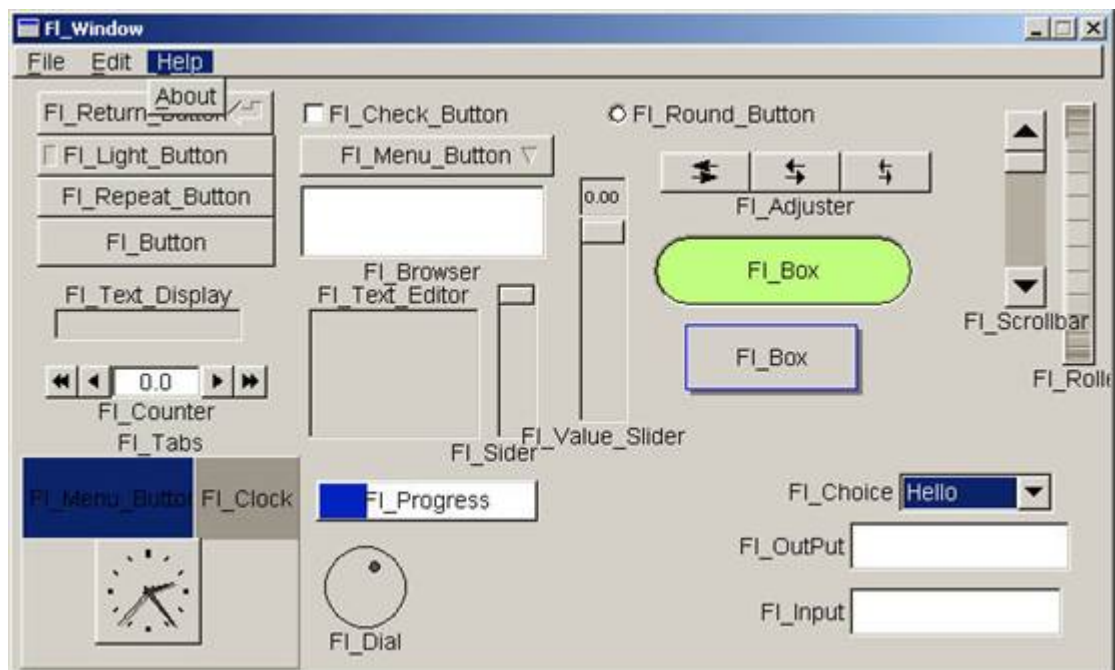
下一步，我们设置了 box 的类型，大小，字体和标签的类型

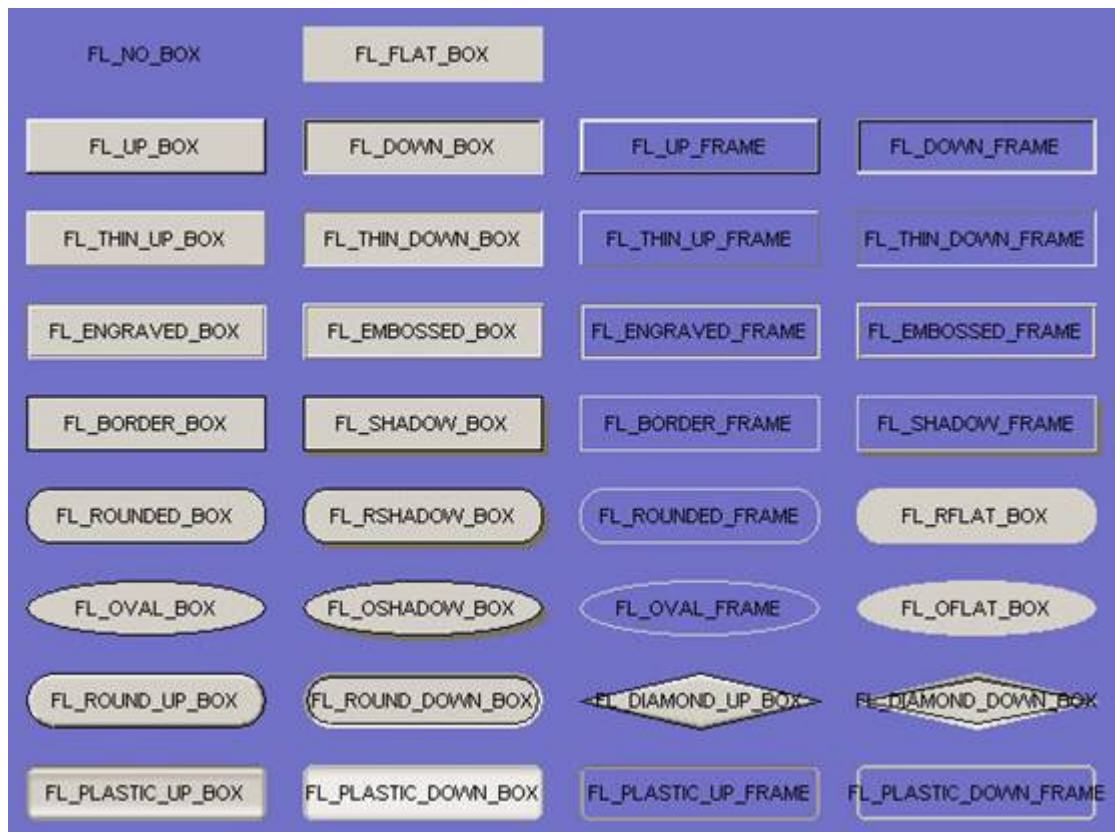
```
box->box(FL_UP_BOX);
```

```
box->labelsize(36);
```

1.3FLTK 构件简介

FLTK 作为 GUI 开发包，包含了常用的图形用户接口需要的一些构件，视觉表现非常丰富，如下两图所示：





常用按钮构件

按钮名称	头文件	按钮名称	头文件
FI_Button	FI_Button.H	FI_Check_Button	FI_Check_Button.H
FI_Light_Button	FI_Light_Button.H	FI_Repeat_Button	FI_Repeat_Button.H
FI_Return_Button	FI_Return_Button.H	FI_Round_Button	FI_Round_Button.H

对于具有 FI_Check_Button、FI_Loght_Button 和 FI_Round_Button 当状态为 off 时 value() =0，On 时 value() 返回 1。

处理按钮时间可以使用回调(callback)函数,参见后面的事件处理。

文本处理构件

构件名称	头文件	构件名称	头文件
FI_Input	FI_Input.H	FI_Output	FI_Output.H
FI_Multiline_Input	FI_Multiline_Input.H	FI_Multiline_output	FI_Multiline_output.H

设置和取得文本内容使用 value();

如:

```
(new FI_Input(x, y, width, height, "Label"))->value("Hello World!");
```

其他构件参见FLTK.org的 [文档](#)说明。

写一个简单的 FLTK 程序

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

```

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

int main(int argc, char **argv)
{
    Fl_Window *window = new Fl_Window(300,180);
    Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
    box->box(FL_UP_BOX);
    box->labelsize(36);
    box->labelfont(FL_BOLD+FL_ITALIC);
    box->labeltype(FL_SHADOW_LABEL);
    window->end();
    window->show(argc, argv);
    return Fl::run();
}

```

包含了需要的头文件后，该程序创建了一个窗口

```
Fl_Window *window = new Fl_Window(300,180);
```

还创建了一个 box 类，标签是 “Hello World!”

```
Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
```

下一步，我们设置了 box 的类型，大小，字体和标签的类型

```
box->box(FL_UP_BOX);
```

```
box->labelsize(36);
```

```
box->labelfont(FL_BOLD+FL_ITALIC);
```

```
box->labeltype(FL_SHADOW_LABEL);
```

最后，我们显示该窗口并进入 FLTK 事件循环

```
window->end();
```

```
window->show(argc, argv);
```

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

运行该程序得到的界面如下，你能直接关闭该窗口退出，也可以按 ESC 键退出



1.4FLTK 事件处理

对于一般构件的如按钮，菜单等常用事件的处理一般可以使用回调函数实现，回调函数的原

型是:

```
void XXX_callback( Fl_Widget *w,void *data )
{
    //添加自己处理的内容
}
```

使用 Fl_Widget->callback(XXX_callback, data) 注册回调函数

```
/******
```

按钮事件例子

```
*****/
```

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_ask.H>
void Btn01_cb(Fl_Widget *w, void *data)
{
    ((Fl_Button *)w)->label((char *)data);
    fl_alert("Hello");
}
int main(int argc, char **argv)
{
    char *buff = "Hello";
    Fl_Window* w = new Fl_Window(272, 144);
    Fl_Button* Btn01 = new Fl_Button(85, 50, 105, 25, "&Test callback");
    Btn01->shortcut(FL_ALT+'t'); //定义按钮的快捷键
    Btn01->callback((Fl_Callback*)Btn01_cb,buff); //调用处理函数 buff 作为参数
    w->end();
    w->show(argc, argv);
    return Fl::run();
}
```

编译运行程序，鼠标点击按钮，按钮标签会发生改变，并且会弹出提示框。

通常的 callback 是当构件的 value 改变时调用，可以使用 when()改变为其他事件发生调用回调函数，主要事件有以下事件

事件	说明
FL_WHEN_NEVER	从不调用回调函数
FL_WHEN_CHANGED	当构件值改变时调用
FL_WHEN_RELEASE	当释放按键或者鼠标并且构件值改变
FL_WHEN_RELEASE_ALWAYS	当释放按键或者鼠标，即使构件值没有改变
FL_WHEN_ENTER_KEY	按下 Enter 键并且构件值改变
FL_WHEN_ENTER_KEY_ALWAYS	按下 Enter 键，即使构件值没有改

通过使用 `Fl_Widget->when(FL_WHEN_XXXX)` 来改变回调事件。

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>
#include <FL/Fl_ask.H>
void Btn01_cb(Fl_Widget *w, void *data)
{
    fl_alert("Hello");
}
int main(int argc, char **argv)
{
    char *buff = "Hello";
    Fl_Window* w = new Fl_Window(272, 144);
    Fl_Button* Btn01 = new Fl_Button(85, 50, 105, 25, "&Test callback");
    Btn01->shortcut(FL_ALT + 't');
    Btn01->callback((Fl_Callback*) Btn01_cb, buff);
    Btn01->when(FL_WHEN_RELEASE_ALWAYS);
    w->end();
    w->show(argc, argv);
    return Fl::run();
}
```

编译运行程序，在按钮上按下鼠标左键，移动到按钮外，松开鼠标按键，仍然会弹出对话框，对比上面的两程序，看看有什么不同。

1.5FLTK 消息处理

在 FLTK 中是通过 `Fl_Widget::handle()` 虚拟函数来处理系统的消息。我们可以查看 Fltk 的源代码来分析系统是怎样处理一些系统消息的，如按钮的消息处理

/******

Fl_Button 中处理消息的代码，省略了具体的处理代码

*****/

```
int Fl_Button::handle(int event) {
    switch (event)
    {
        case FL_ENTER:
        case FL_LEAVE:      return 1;
        case FL_PUSH:       .....
        case FL_DRAG:       .....
        case FL_RELEASE:    .....
        case FL_SHORTCUT:   .....
        case FL_FOCUS :     .....
        case FL_UNFOCUS :   .....
        case FL_KEYBOARD :  .....
```

```

        default: return 0;
    }
}

```

可以看出，系统的一些消息，都是在构件的 `handle()` 中处理的。
系统的主要消息有以下

鼠标事件消息	焦点事件消息
FL_PUSH	FL_ENTER
FL_DRAG	FL_LEAVE
FL_RELEASE	FL_FOCUS
FL_MOVE	FL_UNFOCUS
键盘事件消息	剪贴板事件消息
FL_KEYBOARD	FL_PASTE
FL_SHORTCUT	FL_SELECTIONCLEAR
	构件事件消息
FL_DEACTIVATE	FL_ACTIVE
FL_HIDE	FL_SHOW

通过重载 `handle` 函数我们可扩充标准构件，下面是一个鼠标移动到上面就改变颜色的按钮的实现源代码。

```

#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Button.H>
#include <FL/fl_ask.H>
class EnterButton : public Fl_Button
{
    int handle(int e)
    {
        switch (e)
        {
            case FL_ENTER:
                color(FL_GREEN);
                labelsize(18);
                redraw();
                return 1;
            case FL_LEAVE:
                color(FL_GRAY);
                labelsize(18);
                redraw();
                return 1;
            default: return Fl_Button::handle(e);
        }
    }
}

```

```

public:
    EnterButton(int x, int y, int w, int h, const char *l) : Fl_Button(x,y,w,h,l) {}
};
static void cb(Fl_Widget* s, void *data)
{
    fl_alert( "Hello World!" );
}
int main(int argc, char **argv)
{
    Fl_Window* w = new Fl_Window(130, 50);
    EnterButton *eBtn = new EnterButton(25,50,120,25,"HelloWorld");
    eBtn->callback((Fl_Callback*)cb);
    w->end();
    w->show(argc, argv);
    return Fl::run();
}

```

运行显示效果如图:



Linux 下演示[截屏时鼠标没有取到]



1.6OpenGL 编程

在 FLTK 中很容易使用 OpenGL 进行图形编程的，我们只需要使用它的 `Fl_Gl_Window` 构件，重新定义一个派生于 `Fl_Gl_Window` 的类，重载 `draw()`和 `handle()`就可以。所需要的代码和步骤如下：

1.包含以下头文件

```

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

```

2.定义一个子类，如下代码所示

```

class MYGLWindow : public Fl_Gl_Window
{
    void draw(); //作图操作
    void handle( int ); //消息事件处理
public :
    MYGLWindow(int x,int y,int w,int h,const char *L) : Fl_Gl_Window(x,y,w,h,L){};
};

```

3.实现 draw()事件

void MYGLWindow::draw() //作图

```
{
    if(!valid())
    {
        //设置 viewport 窗口大小等等 例如
        /*****
        valid(1);
        glLoadIdentity();
        glViewport(0,0,w(),h());
        *****/
    }
    //添加使用 OPENGGL 作图操作
};
```

4.事件处理实现

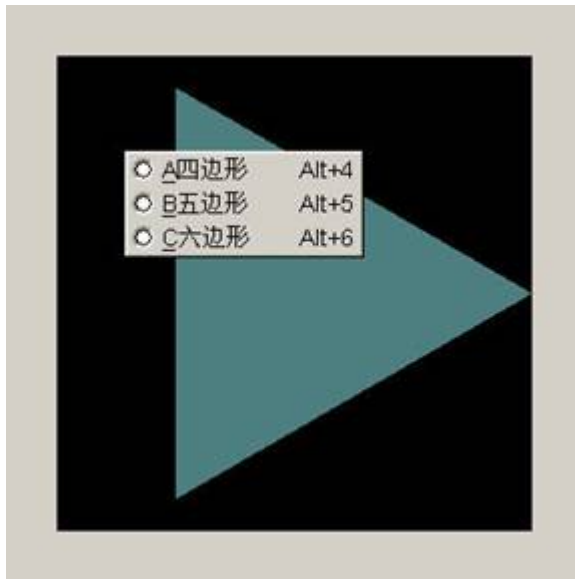
void MYGLWindow::handle(int event) //事件处理

```
{
    switch (event)
    {
        case FL_PUSH : //操作等
            return 1;
        case .....
    }
}
```

注意

1. 编译时需要包含 `openGL32` 的库文件，名字在不同的平台名字稍微不同。
2. 使用 `<FL/gl.h>` 代替 `<GL/gl.h>` 头文件，不要使用后者的头文件。

例子程序比较长，附在参考中。运行显示，弹出菜单后如图所示：



第二章：常用的控件和属性

这章将描述 FLTK 提供的控件，并介绍如何得到和设置控件的标准属性。

2.1 按钮

FLTK 提供了很多类型的按钮

Fl_Button	普通按钮
Fl_Check_Button	带有选择框的按钮
Fl_Light_Button	带有指示灯的按钮
Fl_Repeat_Button	
Fl_Return_Button	能被 Enter 激活的按钮
Fl_Round_Button	带有圆形选择框的按钮

每一个按钮都需要相应的<FL/Fl_xyz_Button.H>头文件。

构造函数包含了控件的位置，大小和可选的标签

```
Fl_Button *button = new Fl_Button(x, y, width, height, "label");
```

```
Fl_Light_Button *lbutton = new Fl_Light_Button(x, y, width, height);
```

```
Fl_Round_Button *rbutton = new Fl_Round_Button(x, y, width, height, "label");
```

每一个按钮可以设置自己的类型用 type(),通过这个设置，可以让一个按钮为 push button, toggle button, or radio button:

```
button->type(FL_NORMAL_BUTTON);lbutton->type(FL_TOGGLE_BUTTON);rbutton->type(FL_RADIO_BUTTON);
```

对于 toggle 和 radio 按钮，value()函数返回当前的状态，开/关（0 代表关，1 代表开），set()和 clear()分别用来设置和清除 togglebutton 的状态。Radio Button 可以用 setonly()打开，同组中的其他 Radio button 按钮将关闭。

2.2 文本

FLTK 提供了几种文本控件来显示和接收文本信息

Fl_Input	输入单行的文本
Fl_Output	输出单行的文本
Fl_Multiline_Input	多行文本输入框
Fl_Multiline_Output	多行文本输出框
Fl_Text_Display	显示多行文本控件
Fl_Text_Editor	多行文本编辑控件
Fl_Help_View	显示 HTML 文本控件

Fl_Output and Fl_Multiline_Output 控件允许互相 copy，但是不能改变 Value()函数用来设置和得到显示的字符串

```
Fl_Input *input = new Fl_Input(x, y, width, height, "label");
```

```
input->value("Now is the time for all good men...");
```

这个字符串将被拷贝到该控件的存储空间内，当用 value()设置后

Fl_Text_Display and Fl_Text_Editor 用 Fl_Text_Buffer 来设置他的值，而不是一个简单的字符串。

Valuators

Valuators	用来显示数字轨迹信息
Fl_Counter	带有箭头按钮的控件显示当前值
Fl_Dial	圆形手柄
Fl_Roller	
Fl_Scrollbar	滚动条控件
Fl_Slider	带有手柄的滑块
Fl_Value_Slider	显示当前值的滑块

Figure 3-2: FLTK valuator widgets

value()函数得到和设置控件的当前值，minimum()和 maximum()设置了控件的范围

群 Groups

Fl_Group 控件被用来做一般的容器控件。除了单选按钮群以外，还被用来形成 windows,tabs,scrolled windows 等控件。一下是 FLTK 提供的群类。

Fl_Double_Window	一个双缓冲的窗口
Fl_Gl_Window	一个 OpenGL 的窗口类
Fl_Group	容器类的基类。能被用来包含所有的控件
Fl_Pack	将控件收集到一个群区域中
Fl_Scroll	滚动窗口区域
Fl_Tabs	
Fl_Tile	
Fl_Window	

设置控件的位置和大小

控件的位置和大小在你创建的时候就已经设置了，你可以通过 x(),y(),w(),h(),来得到。

改变大小和位置用 position(),resize(),size()函数。

```
button->position(x, y);
```

```
group->resize(x, y, width, height);
```

```
window->size(width, height);
```

2.3 颜色

FLTK 用一个 32 位的无符号整形存储颜色。它可能是 256 种颜色一个索引，也可能是一个 24 位的 RGB 颜色。调色板不是 X 或 WIN32 的 colormap,它是有对应固定内容的调色板
以下是一些常用的颜色的符号定义：

FL_BLACK

FL_RED

FL_GREEN

FL_YELLOW

FL_BLUE

FL_MAGENTA

FL_CYAN

FL_WHITE

这些符号是 FLTK 控件默认的颜色，详细情况请参考 Enumerations

FL_FOREGROUND_COLOR

FL_BACKGROUND_COLOR

FL_INACTIVE_COLOR

FL_SELECTION_COLOR

RGB 颜色可以用 fl_rgb_color()函数设置。

Fl_Color c = fl_rgb_color(80,170,255);

控件的颜色用 color()函数设置

button->color(FL_RED);

类似的，标签的颜色用 labelcolor()函数设置

button->labelcolor(FL_WHITE);

2.4Box 类型

Fl_Boxtype 的类型在<Enumeration.H>中定义，可以用 Fl_Widget::box()设置和得到。

FL_NO_BOX 意思是任何东西都不要画，但仍然是留在窗口上。FL_..._FRAME 类型只是画边框，中间不做任何改变。如图中蓝色的部分。

制作你自己的 Boxtypes

你可以自己制作个性风格的 boxtype.通过一个小函数，并将其加到 boxtypes 的列表中画图函数

画图函数传递的参数控件的是 box 的边界和背景颜色

void xyz_draw(int x, int y, int w, int h, Fl_Color c)

```
{  
}
```

如一个简单的画图函数填充一个矩形，给定颜色并画一个黑色的外框

void xyz_draw(int x, int y, int w, int h, Fl_Color c)

```
{  
    fl_color(c);  
    fl_rectf(x, y, w, h);  
}
```

```

    fl_color(FL_BLACK);
    fl_rect(x, y, w, h);
}

```

加入自定义的 box 类型

Fl::set_boxtype 函数添加或取代特定的 box 类型

```
#define XYZ_BOX FL_FREE_BOXTYPE Fl::set_boxtype(XYZ_BOX, xyz_draw, 1, 1, 2, 2);
```

最后 4 个参数是偏移量，当画该 box 时，x,y,w,h 会减去相应的偏移量

第三章：FLTK 的画图函数

这章涵盖了 FLTK 提供的所有画图函数

3.1 何时可以画图

什么时候可以画图，只有在几个地方可以执行画图代码。在其他地方调用该函数会出现未定义该行为的错误：

1. 最常出现的地方是在虚拟函数 Fl_Widget::draw() 中。你的类需要继承一个 Fl_Widget 类，然后在自己的类中写 draw() 函数。
2. 在写 boxtype 和 labeltype 函数中用到。
3. 你可以调用 Fl_Window::make_current() 来增加控件的更新。用 Fl_Widget::window() 找到要更新的窗口

3.2 FLTK 的画图函数

调用这些画图函数之前，要包含头文件 <FL/fl_draw.H>

FLTK 提供以下画图函数：

Boxes

Clipping

Colors

Line dashes and thickness

Fase Shapes

Complex Shapes

Text

Images

Overlay

Boxes

FLTK 提供了三个函数来画 box，主要用于画按钮和其他的 UI 控件。每一个函数都提供了 box 的左上角，宽，高等参数。

```
void fl_draw_box(Fl_Boxtype b, int x, int y, int w, int h, Fl_Color c);
```

该函数画了一个标准的 box, box 类行为 b, 颜色是 c

```
void fl_frame(const char *s, int x, int y, int w, int h);
```

该函数画了一个边框，s 是 4 个字母，A 代表黑色，X 代表白色，顺序是上，左，下，右。

```
void fl_frame2(const char *s, int x, int y, int w, int h);
```

与 fl_frame 不同时 s 代表的颜色的顺序，分别是下，右，上，左。

3.3 剪切

你可以限制你的画图行为在一个矩形之内，应用 `fl_push_clip(x,y,w,h)`，释放用 `fl_pop_clip`。该矩形用像素为单位，不会受变换矩阵的影响

另外，系统会提供更新窗口的剪切域，但是比一个简单的矩形要复杂的多

`void fl_clip(int x, int y, int w, int h)`

`void fl_push_clip(int x, int y, int w, int h)`

用一个矩形剪切一个区域，并把这个区域压入堆栈。`Fl_clip()`不提倡，并将在以后的版本中去除该函数

`void fl_push_no_clip()`

压入一个空的剪切域到堆栈

`void fl_pop_clip()`

恢复剪切域，画图范围不再受矩形限制，`fl_push_clip()`一定要调用该函数。

`int fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H)`

新的剪切域与旧的剪切域相交，相交的矩形位置保存在 `X,Y,W,H`，如果完全没有相交，则 `W,H` 为 0；

3.4 颜色

FLTK 将颜色处理为 32 位的整形。0-255 分别代表不同的颜色。`Fl_color` 枚举类型定义了前 256 个基本的颜色。

颜色值大于 255 的被认为是 24 位的 RGB 值。显示的是最接近该值的颜色。

`void fl_color(Fl_Color)` 设置当前使用的颜色

`Fl_Color fl_color()` 返回最后设定的颜色

`void fl_color(uchar r, uchar g, uchar b)` 设置 rgb 颜色。

3.5 设置线条的属性

FLTK 支持设定线条的宽度和类型。

`void fl_line_style(int style, int width=0, char* dashes=0)`

`style` 是以下几种类型之一，默认的是 `FL_SOLID`。

`FL_SOLID` -----

`FL_DASH` - - - -

`FL_DOT`

`FL_DASHDOT` - . - .

`FL_DASHDOTDOT` - .. -

`FL_CAP_FLAT`

`FL_CAP_ROUND`

`FL_CAP_SQUARE` (extends past end point 1/2 line width)

`FL_JOIN_MITER` (pointed)

`FL_JOIN_ROUND`

`FL_JOIN_BEVEL` (flat)

宽度是以像素值为单位，默认的 0

3.6 画一般的图形函数

下面的函数几乎可以用来画所有的控件，这些函数画图非常精确，也非常快。他们可以在任何支持 FLTK 的平台上使用。

Void fl_point(int x,int y) 画点函数

Void fl_rectf(int x,int y,int w,int h)画一个矩形并填充内部

Void fl_rectf(int x,int y,int w,int h,uchar r,uchar g,uchar b)自定义颜色填充矩形

Void fl_line(int x, int y, int x1,int y1)画一条直线，起点为 x,y,终点为 x1,y1

Void fl_line(int x,int y,int x1,int y1,int x2,int y2) 画两条直线

void fl_loop(int x, int y, int x1, int y1, int x2, int y2)

void fl_loop(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Outline a 3 or 4-sided polygon with lines.

3.7 画封闭的线，一次连接个顶点

void fl_polygon(int x, int y, int x1, int y1, int x2, int y2)

void fl_polygon(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Fill a 3 or 4-sided polygon. The polygon must be convex.

3.8 画三边形或四边形，并填充内部

void fl_xyline(int x, int y, int x1)

void fl_xyline(int x, int y, int x1, int y2)

void fl_xyline(int x, int y, int x1, int y2, int x3)

先画一条水平的线，再画一条垂直的线条，最后画一条水平线

void fl_yxline(int x, int y, int y1)

void fl_yxline(int x, int y, int y1, int x2)

void fl_yxline(int x, int y, int y1, int x2, int y3)

首先画垂直线条，接着是水平线，最后是垂直线

void fl_arc(int x, int y, int w, int h, double a1, double a2)

void fl_pie(int x, int y, int w, int h, double a1, double a2)

画弧形线，两个角度是以三点处为 0 度，逆时针旋转，a2 必须大于或等于 a1

fl_pie()填充弧形内部

3.9 复杂图形函数

复杂的画图函数利用 2-D 线性转换能让你画出任意图形。这个功能与 Adobe? PostScript 语言实现的功能很相似，在 X 和 Win32 上，在画线段之前所有的转换顶点都是用整数表示，这就限制了画图的精确性。如果要画比较精确的图形，最好用 OpenGL 来画。

void fl_push_matrix()

void fl_pop_matrix()

保存和恢复当前的转换，堆栈的最大深度为 4

void fl_scale(float x, float y)

void fl_scale(float x)

void fl_translate(float x, float y)

void fl_rotate(float d)

void fl_mult_matrix(float a, float b, float c, float d, float x, float y)

在当前的转换基础上连接另外一个转换。旋转角度是度数不是弧度，逆时针旋转。

void fl_begin_line()

void fl_end_line()

开始和结束画线

void fl_begin_loop()

void fl_end_loop()

开始和结束画一系列封闭的线

void fl_begin_polygon()

void fl_end_polygon()

开始和结束画多边形并填充

void fl_begin_complex_polygon()

void fl_gap()

void fl_end_complex_polygon()

开始和结束画一个复杂的多边形并填充。这个多边形可以是凹凸不同的，不连贯的，甚至中间有空心的。调用 fl_gap() 分开路径。不必也是有害的如果在第一个顶点之前或最后一个顶点之后调用 fl_gap() 函数，在一行中多次调用也是不行的。

Fl_gap() 只能用在 fl_begin_complex_polygon() 和 fl_end_complex_polygon() 之间。画多边形的轮廓，使用 fl_begin_loop 并用 fl_end_loop 和 fl_begin_loop 代替 fl_gap();

void fl_vertex(float x, float y)

在当前路径中增加一个顶点

void fl_curve(float x, float y, float x1, float y1, float x2, float y2, float x3, float y3)

在路径中增加一系列的点画 Bezier 曲线。该曲线的末端是 x,y 和 x3,y3。

void fl_arc(float x, float y, float r, float start, float end)

增加一系列的点在当前圆环的弧线上。在调用 fl_arc() 之前应用 scale 和 rotate 可以得到椭圆的路径。X,y 是圆的中心，r 是半径。Fl_arc() 从 start 角度画弧直到 end, 按逆时针旋转。如果 end 大于 start 则它是按照顺时针转

void fl_circle(float x, float y, float r)

fl_circle 等于 fl_arc(...,0,360), 但是更快, 如果你在画多边形的时候用到 圆, 则必须用 fl_arc(). 文本的画法

所有的文本都字体都是适用当前字体。现在还不明确在转换情况下，位置或字符是否会改变

void fl_draw(const char *, int x, int y)

void fl_draw(const char *, int n, int x, int y)

在窗口中画出字符串，位置是靠左，接近底线

void fl_draw(const char *, int x, int y, int w, int h, Fl_Align align, Fl_Image *img = 0, int draw_symbols = 1)

void fl_measure(const char *, int &w, int &h, int draw_symbols = 1)

int fl_height()

得到当前字体的高度

int fl_descent()

float fl_width(const char*)

```
float fl_width(const char*,int n)
float fl_width(uchar)
const char *fl_shortcut_label(ulong)
返回按钮或菜单的快捷键字符串
```

3.10 字体

FLTK 支持很多标准的字体，比如 Times, Helvetica/Arial, Courier, and Symbol typefaces，用户也可以自定义字体，每一个字体都有自己的索引列表
初始化只安装了 16 种字体，他们的名字是 FL_HELVETICA, FL_TIMES, FL_COURIER,另外有二个修饰体 FL_BOLD,FL_ITALIC。加上 FL_SYMBOL,FL_ZAPF_DINGBATS
不能超过 255 种字体，因为 Fl_Widget 是以一个字节来存储的。

```
Void fl_font(int face , int size)设置字体和大小
int fl_font()
int fl_size()
得到字体和大小
```

3.11 覆盖画图函数

```
void fl_overlay_rect(int x, int y, int w, int h);
void fl_overlay_clear();
```

前者与先前颜色异或操作，后者清楚异或操作
使用该函数非常的巧妙，你应该在控件中有 handle()和 draw()函数，draw()应该调用 fl_overlay_clear()在做任何事情之前。Handle()函数应该调用 window()->make_current()然后在 FL_DRAG 事件中调用 fl_overlay_rect()，在 FL_RELEASE 事件中调用 fl_overlay_clear()。
绘制图片

第四章：在 FLTK 中自定义控件

4.1 定制图形控件说明

新控件的创建是通过继承已经存在的控件来得到的,一般控件继承 Fl_Widget 得到,组合控件继承 Fl_Group 得到
一个普通控件一般通过接收和显示一个值来与用户交互
一个组合控件包含一组子控件并处理子控件的移动，改变大小，显示或隐藏事件。Fl_Group 是所有组合控件的基类,其他组合控件比如 Fl_Pack, Fl_Scroll, Fl_Tabs , Fl_Tile, Fl_Window 都是他的子类
你也可以通过继承其他的已存在控件来得到你要的控件，通过提供不同的外观和接口。比如 Button 控件都是 Fl_Button 类的子类。他们的共同点是都是通过鼠标点击事件与用户交互。唯一不同的是按钮的外观。

4.2 如何开发一个控件的子类

你的子类可以直接继承 Fl_Widget 类，也可以继承任何 Fl_Widget 类的子类。Fl_Widget 只有四个虚拟函数，子类必须重载所有的或部分的这些函数。

构造函数

构造函数应该有以下参数

```
MyClass(int x, int y, int w, int h, const char *label = 0);
```

这就允许该类能很好的应用于 FLUID 中

这个构造函数必须调用基类的构造函数并传递相同的参数

```
MyClass::MyClass(int x, int y, int w, int h, const char *label): Fl_Widget(x, y, w, h, label) { // do initialization stuff... } Fl_Widget 的保护构造函数通过传递的参数 x,y,w,h,label 分别设置 x(),y(),w(),h()和 label()并初始化其他的属性如:
```

```
type(0);
box(FL_NO_BOX);
color(FL_BACKGROUND_COLOR);
selection_color(FL_BACKGROUND_COLOR);
labeltype(FL_NORMAL_LABEL);
labelstyle(FL_NORMAL_STYLE);
labelsize(FL_NORMAL_SIZE);
labelcolor(FL_FOREGROUND_COLOR);
align(FL_ALIGN_CENTER);
callback(default_callback,0);
flags(ACTIVE|VISIBLE);
image(0);
deimage(0);
```

Fl_Widget 的保护成员函数

以下的成员函数是 Fl_Widget 提供给子类的:

Fl_Widget::clear_visible

Fl_Widget::damage

Fl_Widget::draw_box

Fl_Widget::draw_focus

Fl_Widget::draw_label

Fl_Widget::set_flag

Fl_Widget::set_visible

Fl_Widget::test_shortcut

Fl_Widget::type

void Fl_Widget::damage(uchar mask)

void Fl_Widget::damage(uchar mask, int x, int y, int w, int h)

uchar Fl_Widget::damage()

第一个函数是指对象的部分需要更新。参数 mask 中的位设置传递给 damage().draw()函数能根据该值得到哪些需要重画。公共成员函数 Fl_Widget::redraw() 只是简单的做 Fl_Widget::damage(FL_DAMAGE_ALL),即所有的都重画,但是你的控件真正执行的时候会调用私有成员函数 damage(n).

第二个函数指某个区域无效,需要重画。

第三个函数返回所有 damage(n)的调用所产生的位。

当重新画一个控件时，你应该先看看无效位，再决定你的控件的哪部分需要重新画。**Handle()**函数能够设置单独的无效位限制需要重画的数量。

```
MyClass::handle(int event)
```

```
{
    if (change_to_part1) damage(1);
    if (change_to_part2) damage(2);
    if (change_to_part3) damage(4);
}
```

```
MyClass::draw()
```

```
{

    if(damage() & FL_DAMAGE_ALL)
    {

        ... draw frame/box and other static stuff ...

    }
    if (damage() & (FL_DAMAGE_ALL | 1)) draw_part1();
    if (damage() & (FL_DAMAGE_ALL | 2)) draw_part2();
    if (damage() & (FL_DAMAGE_ALL | 4)) draw_part3();

}
```

```
void Fl_Widget::draw_box() const
```

第一个函数根据该控件的尺度画他的 box().第二个函数根据 box 的类型 b ,颜色 c 画 box.

```
void Fl_Widget::draw_focus() const
```

```
void Fl_Widget::draw_focus(Fl_Boxtype b, int x, int y, int w, int h) const
```

在一个空间的限制 box 中画出焦点筐。第二个函数允许指定另一个不同 box 来画焦点

```
Fl_Widget::draw_label() const
```

```
void Fl_Widget::draw_label(int x, int y, int w, int h) const
```

```
void Fl_Widget::draw_label(int x, int y, int w, int h, Fl_Align align) const
```

Draw()函数调用该函数来画一个控件的 label,如果标签出了该控件的 box 范围，将不会被画出。

第二种形式自定义一个 box 来画标签，比如用于移动的滑块

第三种形式可以将标签画在任意的地方

```
void Fl_Widget::set_visible()
```

```
void Fl_Widget::clear_visible()
```

与 Fl_Widget::show() Fl_Widget::hide()作用相同，但不发送 FL_SHOW,FL_HIDE 事件。

```
int Fl_Widget::test_shortcut() const
```

```
static int Fl_Widget::test_shortcut(const char *s)
```

```
uchar Fl_Widget::type() const
```

```
void Fl_Widget::type(uchar t)
```

返回一个 8 位的标示符，用于与 Forms 兼容，你也可以同于其他任何目的，设置的值应该小于 100，以免与系统的保留值冲突

4.3 处理事件

虚拟函数 `int handle(int event)` 被用来处理任何发送给控件的事件。他能：

改变控件的状态

调用 `Fl_Widget::redraw()` 如果该控件需要重新显示

调用 `Fl_Widget::damage(n)` 当控件需要部分更新时(假如你在 `Fl_Widget::draw()` 函数中提供了对该函数的支持)

调用 `Fl_Widget::do_callback()` 如果一个回调函数产生时。

调用 `Fl_Widget::handle()` 对子控件

事件用一个整数来标识。最近事件产生的其他消息静态存储在本地，调用 `Fl::event_*` 可以得到。

以下是一个利用 `handle()` 处理事件的例子，该控件的行为类似按钮同时接收 x 按键并调用回调函数

```
int MyClass::handle(int event)
```

```
{
    switch(event)
    {
        case FL_PUSH:
            highlight = 1;
            redraw();
            return 1;
        case FL_DRAG:
            {
                int t = Fl::event_inside(this);
                if (t != highlight)
                {
                    highlight = t;
                    redraw();
                }
            }
        return 1;
        case FL_RELEASE:
            if(highlight)
            {
                highlight = 0;
                redraw();
                do_callback();
                // never do anything after a callback, as the callback
```

```

        // may delete the widget!
    }
    return 1;
    case FL_SHORTCUT:
    if(Fl::event_key() == 'x')
    {
        do_callback();
        return 1;
    }
    return 0;
    default: return Fl_Widget::handle(event);
}
}

```

当你的 `handle()` 函数处理某事件后不能返回 0，若是返回 0，父控件将会把该事件发送给其他控件。

4.4 画控件

当 FLTK 需要重画控件时将调用虚拟函数 `draw()`。只有在 `damage()` 返回非 0 值时调用该函数，`draw()` 返回后，`damage()` 被清 0。`Draw()` 应该被声明为保护成员函数，避免在不需要写画图代码时用到。

`Damage()` 将包含从最后一次调用 `draw()` 后 `damage(n)` 调用产生的所有与或位信息，根据该信息只重画需要重画的位置，只有 FLTK 认为需要全部重画时才打开 `FL_DAMAGE_ALL` 位，比如收到 `expose` 事件。

修改控件的尺寸

`resize(int x,int y,int w,int h)` 在控件被移动和改变大小时被调用，这些参数分别是新位置，宽度和高度。但是 `x()`,`y()`,`w()`,`h()`，还是以前的值，若要改变这些值，必须在基类中也调用 `resize()` 函数

不需要调用 `redraw()` 函数，至少只改变 `x()`,`y()` 时不需要，因为一个组合控件有一套更有效的方法来画新的位置

如何制作一个组合控件

一个组合控件包括一个或多个子控件。制作组合控件必须继承 `Fl_Group` 类。不继承 `Fl_Group` 类当然也可能可以制作一个组合控件，但是你还是需要重新写 `Fl_Group` 类里面的工作

子控件可能在类里面声明

```

class MyClass : public Fl_Group
{
    Fl_Button the_button;
    Fl_Slider the_slider;
    ...
};

```

构造函数要初始化这些子控件。他们将被自动的 `add()` 到 `group` 中。因为 `Fl_Group` 构造函数调用了 `begin()`。在构造函数中不要忘记调用 `end()` 函数

`MyClass::MyClass(int x, int y, int w, int h) :`

```
Fl_Group(x, y, w, h),
the_button(x + 5, y + 5, 100, 20),
the_slider(x, y + 50, w, 20)
{
    ...(you could add dynamically created child widgets here)...
    end(); // don't forget to do this!
}
```