Organ-Field GUI Help

0.1 概述

C++程序的交互界面为控制台(命令行),而对于初学者而言(或非计算机专业的同学),一个图形用户界面并不是必要的但的确能明显的提升程序的交互体验. 这一类型的群体自己编写图形界面或使用 C++的 GUI 库(如 QT)或多或少都觉得力不从心,从而难以兼顾程序核心功能与图形界面. 于是就诞生了基于很多初学者都在使用的 EasyX Graphics Library 的 GUI 库(Organ-Field GUI), 其最大的特点就是简单易用.初学者可以用其进行学习前期的过度,非计算机专业的同学可以把大部分精力投入于程序核心功能中的同时也能兼顾友好的交互.

Chapter 1 Organ-Field Button(OFB)

1.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFButton.cpp"与"OFButton.h"文件.

使用 Organ-Field Button 前需要引用头文件: `#include "OFButton.h"`

使用 Organ-Field Button 前需要引用类名: `OFButton OFB;`

1.1 宏

- **在创建按钮时,需要指定按钮的状态,按钮共有 5 种状态,以下是按钮状态宏.**

宏 Macro	值 Value	含义 Meaning
normal	0	正常状态: 表示按钮是可交互的，并且可用的
disable	1	禁用状态: 表示当前组件处于非交互状态，但是之后可以被启用
hover	2	悬停状态: 当用户使用光标或者其他的元素，置于其上方的时候，显示这样的状态(鼠标划过)
click	3	激活状态: 表示用户已经按下按钮,且还未结束按按钮的动作(鼠标点击)
loading	4	加载状态: 表示操作正在加载中，组件正在反映，但是操作还未完成

1.2 函数

1.2.1 CreateButton

- **这个函数用于创建一个按钮**

```
int CreateButton(
    string ButtonName,
    int x, int y,
    int width, int height,
    short SetState,
    int (*EventFunc)()
);
```

- **参数**
  - < ButtonName > 设置按钮名称
  - < x,y > 指定按钮坐标
  - < width > 设置按钮宽度
  - < height > 设置按钮高度
  - < SetSate > 设置按钮状态(此处填写按钮状态宏)
  - < *EventFunc > 指定要执行的事件函数指针(在此填写函数名)

1.2.2 ButtonDefaultStyle

- **按钮样式快速设置函数,此函数令所有按钮使用同一套已经预设好的样式,不再需要自己进行复杂的按钮样式设置.**

```
void ButtonDefaultStyle();
```

1.2.3 SetButtonStyle Series

- **这一系列的函数用于自定义设置各按钮状态的样式(不能与 1.2.2 ButtonDefaultStyle 同时使用)**

```
//设置正常状态按钮的样式
void SetNormalStyle(
    short buttonrim,
    COLORREF rimcolor,
    short fillstyle,
    short fillhatch,
    COLORREF fillcolor,
    COLORREF textcolor
);
```

```
//设置禁用状态按钮的样式
void SetDisableStyle (
    short buttonrim,
    COLORREF rimcolor,
    short fillstyle,
    short fillhatch,
    COLORREF fillcolor,
    COLORREF textcolor
);
```

```
//设置悬停状态按钮的样式
void SetHoverStyle (
    short buttonrim,
    COLORREF rimcolor,
    short fillstyle,
    short fillhatch,
    COLORREF fillcolor,
    COLORREF textcolor
);
```

```
//设置激活状态按钮的样式
void SetClickStyle (
    short buttonrim,
    COLORREF rimcolor,
    short fillstyle,
    short fillhatch,
    COLORREF fillcolor,
    COLORREF textcolor
);
```

```
//设置加载状态按钮的样式
void SetLoadingStyle (
    short buttonrim,
    COLORREF rimcolor,
    short fillstyle,
    short fillhatch,
    COLORREF fillcolor,
    COLORREF textcolor
);
```

- **参数**

< buttonrim >	设置按钮边框线粗细
< rimcolor >	设置按钮边框线颜色
< fillstyle >	设置按钮填充样式
< fillhatch >	指定填充图案
< fillcolor >	指定填充颜色
< textcolor >	指定按钮字体颜色

- **颜色**

```
COLORREF RGB(
    BYTE byRed,     //颜色的红色部分
    BYTE byGreen,   //颜色的绿色部分
    BYTE byBlue     //颜色的蓝色部分
);
//颜色值范围: 0~255
```

- **填充样式**

宏 Macro	含义 Meaning
BS_SOLID	固实填充
BS_NULL	不填充
BS_HATCHED	图案填充
BS_PATTERN	自定义图案填充
BS_DIBPATTERN	自定义图像填充

- **填充图案**

宏 Macro	含义 Meaning
HS_HORIZONTAL	填充横线
HS_VERTICAL	填充竖线
HS_FDIAGONAL	填充斜线
HS_BDIAGONAL	填充反斜线
HS_CROSS`	填充网格线
HS_DIAGCROSS	填充斜网格线

- **示例代码**

```
#include "Window.h"
#include "OFButton.h"
Window win;
OFButton OFB;

int main() {
    //初始化绘图窗口(窗口宽度,窗口高度,显示控制台);
    win.Initialize_Window(1200, 700, EW_SHOWCONSOLE);
    //设置各按钮状态的样式(按钮边框线粗细,边框线颜色,指定填充样式,指定填充图案,指定填充颜色,指定按钮字体颜色);
    OFB.SetNormalStyle(1, RGB(180, 180, 180), BS_SOLID, NULL, RGB(225, 225, 225), RGB(30, 30, 30));
    OFB.SetDisableStyle(1, RGB(191, 191, 191), BS_SOLID, NULL, RGB(204, 204, 204), RGB(130, 130, 130));
    OFB.SetHoverStyle(1, RGB(180, 180, 180), BS_SOLID, NULL, RGB(229, 243, 255), RGB(30, 30, 30));
    OFB.SetClickStyle(1, RGB(10, 89, 247), BS_SOLID, NULL, RGB(204, 232, 255), RGB(30, 30, 30));
    OFB.SetLoadingStyle(1, RGB(10, 89, 247), BS_SOLID, NULL, RGB(204, 204, 204), RGB(130, 130, 130));
    system("pause");
};
//注: 在不需要参数时必须填 NULL;
```

1.3 示例

- **以下代码展示如何创建两个按钮**

```
//编译环境: Visual Studio 2022  C++14  EasyX_20220116  使用多字节字符集;
#include "Window.h"
#include "OFButton.h"
Window win;
OFButton OFB;

void ClickEvent() { cout << "ButtonClick-1" << endl; } //第一个按钮点击事件;
void ClickEvent2() { cout << "ButtonClick-2" << endl; }//第二个按钮点击事件;

int main() {
    win.Initialize_Window(1200, 700, EW_SHOWCONSOLE); //初始化窗口(窗口宽度:1200px,窗口高度:700px,显示控制台);

    OFB.ButtonDefaultStyle();//快速设置按钮样式为预设默认值;
    while (1) {//这里一定要给循环;
        OFB.CreateButton("Test", 10, 10, 120, 60, normal, ClickEvent);   //创建第一个按钮;
        OFB.CreateButton("Test2", 10, 80, 120, 60, normal, ClickEvent2); //创建第二个按钮;
    }
};
```

Chapter 2 Organ-Field Window(OFW)

2.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"Window.cpp"与"Window.h"文件.

使用 Organ-Field Window 前需要引用头文件: `#include "Window.h"`

使用 Organ-Field Window 前需要引用类名: `Window win`

2.1 函数

2.1.1 Initialize_Window

- **这个函数用于创建窗口(内置默认窗口样式)**

```
void Initialize_Window(int Width, int Height, int Flag);
```

- **参数**

< Width > 指定窗口宽度

< Height > 指定窗口高度

< Flag > 绘图窗口的样式

- **绘图窗口的样式宏(Flag Macro)**

宏 Macro	含义 Meaning
EW_DBLCLKS	在绘图窗口中支持鼠标双击事件
EW_NOCLOSE	禁用绘图窗口的关闭按钮
EW_NOCLOSE	禁用绘图窗口的最小化按钮
EW_NOCLOSE	显示控制台窗口

2.1.2 SetWindowStyle

- **这个函数用于设置窗口样式,必须在 2.1.1 Initialize_Window 创建窗口后才可使用**

```
void SetWindowStyle(COLORREF backdrop_color, float xasp, float yasp);
```

- **参数**

< backdrop_color > 指定窗口背景颜色

< xasp > x 方向上的缩放因子(例如绘制宽度为 100 的矩形，实际的绘制宽度为 100 * xasp)

< yasp > y 方向上的缩放因子(例如绘制高度为 100 的矩形，实际的绘制高度为 100 * yasp)

- **备注**

① 如果缩放因子为负，可实现坐标轴翻转.

② 使用函数时为了更新背景颜色,此函数会清空一次窗口绘图画板.

2.1.3 SetWindowTilte

- **这个函数用于设置窗口标题**

```
void SetWindowTilte(const string &Title);
```

- **参数**

< &Title > 指定窗口标题(文本)

2.1.4 MoveWindow

- **这个函数用于设置窗口位置**

```
void MoveWindow(int X, int Y);
```

- **参数**

< X > 相对屏幕的横轴坐标

< Y > 相对屏幕的纵轴坐标

- **备注**

屏幕的最左上角为坐标原点,向下纵坐标增加,向右横坐标增加(此法则也适用于窗口).

2.1.5 Reset_Window

- **这个函数用于重新设置窗口大小并清空一次窗口绘图画板**

```
void Reset_Window(int Width, int Height);
```

- **参数**

< Width > 指定新的窗口宽度

< Height > 指定新的窗口高度

2.1.6 WindowOnTop

- **这个函数用于使窗口置顶**

```
void WindowOnTop();
```

2.1.7 GetWindowHWND

- **这个函数用于获取窗口句柄**

```
HWND GetWindowHWND();
```

- **返回值**

返回本窗口的窗口句柄(hWnd)

2.1.8 GetWindowRect

- **这个函数用于获取窗口边框矩形**

```
RECT GetWindowRect();
```

- **返回值**

返回本窗口的窗口边框矩形

```
typedef struct tagRECT{
    LONG left;    //窗口矩形左侧边框;
    LONG top;     //窗口矩形顶部边框;
    LONG right;   //窗口矩形右侧边框;
    LONG bottom;  //窗口矩形底部边框;
} RECT;
```

- **备注**

这里的矩形有两种表达方法: ①矩形的四边; ②矩形的左上角与右下角点的坐标;

2.1.9 GetWidth

- **这个函数用于获取窗口宽度**

```
static int GetWidth();
```

- **返回值**

返回本窗口的窗口宽度

2.1.10 GetHeight

- **这个函数用于获取窗口高度**

```
static int GetHeight();
```

- **返回值**

返回本窗口的窗口高度

Chapter 3 Organ-Field Message (OFM)

3.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFMessage.cpp"与"OFMessage.h"文件.

使用 Organ-Field Window 前需要引用头文件: `#include "OFMessage.h"`

使用 Organ-Field Window 前需要引用类名: `OFMessage OFM;`

3.1 函数

3.1.1 PeekOFMessageUP

- **这个函数用于启动一个不断扫描鼠标消息(包括位置,按键,滚轮)和键盘消息以及窗口消息的独立函数线程.**

```
void PeekOFMessageUP();
```

- **备注**

这个函数负责启动一个独立于主线程的线程,然后将扫描得到的消息储存于 ExMessage 结构体中,需要鼠标位置等消息时直接从结构体中调取,不会卡死主线程.

3.1.2 PeekOFMessage

- **这个函数用于获取鼠标消息(包括位置,按键,滚轮)和键盘消息以及窗口消息.**

```
ExMessage PeekOFMessage();
```

- **返回值**

立即返回保存有鼠标消息和键盘消息以及窗口消息的 ExMessage 结构体.

```
struct ExMessage{
     USHORT message; // The message identifier.
     union{// Data of the mouse message
          struct{
               bool ctrl   :1; // Indicates whether the CTRL key is pressed.
               bool shift  :1; // Indicates whether the SHIFT key is pressed.
               bool lbutton:1; // Indicates whether the left mouse button is pressed.
               bool mbutton:1; // Indicates whether the middle mouse button is pressed.
               bool rbutton:1; // Indicates whether the right mouse button is pressed.
               short x;    // The x-coordinate of the cursor.
               short y;    // The y-coordinate of the cursor.
               short wheel;// The distance the wheel is rotated, expressed in multiples or divisions of 120.
          };
          struct{// Data of the key message
               BYTE vkcode;// The virtual-key code of the key.
               BYTE scancode; // The scan code of the key. The value depends on the OEM.
               bool extended:1;
               // Indicates whether the key is an extended key, such as a function key or a key on the numeric key
               pad. The value is true if the key is an extended key; otherwise, it is false.
               bool prevdown:1; // Indicates whether the key is previously up or down.
          };
          TCHAR ch; // Data of the char message
```

```
        struct{// Data of the window message
            WPARAM wParam;
            LPARAM lParam;
        };
    };
};
```

- **备注**

此函数只会返回 PeekOFMessageUP 线程预先储存好的消息,对性能几乎没有影响.

3.3 示例

- **以下代码展示如何获取鼠标坐标**

```
//编译环境：Visual Studio 2022  C++14  EasyX_20220116  使用多字节字符集;
Window OFW;
OFMessage OFM;


ExMessage MouseMessage{ NULL };  //定义消息变量(它能够储存鼠标的坐标);
int main() {
    OFW.Initialize_Window(1200, 700, EW_SHOWCONSOLE);

    OFM.PeekOFMessageUP();//启动鼠标消息(坐标)扫描;


    while (true) {
        MouseMessage = OFM.PeekOFMessage()//获取鼠标信息,并将返回值传给 MouseMessage;

        //打印鼠标坐标值;
        cout << "鼠标当前坐标(X:" << MouseMessage.x << ",Y:" << MouseMessage.y << ")" << endl;
    }
};
```

Chapter 4 Organ-Field ToggleSwitch (OFTS)

4.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"ToggleSwitch.cpp"与"ToggleSwitch.h"文件.

使用 Organ-Field Window 前需要引用头文件: `#include "ToggleSwitch.h"`

使用 Organ-Field Window 前需要引用类名: `ToggleSwitch OFTS;`

4.1 函数

4.1.1 CreateToggle

- **这个函数用于创建一个拨动开关**

```
int CreateToggle(int ToggleName, int x, int y, short SetInitialState, int (*EventFunc)());
```

- **参数**

< ToggleName > 设置拨动开关名称(此名称只能是非负整数数值,并且应遵从由小到大取名;如:0,1,2,3,4,……)
< x,y > 指定拨动开关坐标
< width > 设置拨动开关宽度
< height > 设置拨动开关高度
< SetSate > 设置拨动开关状态(此处填写拨动开关状态宏)
< *EventFunc > 指定要执行的事件函数指针(在此填写函数名)

- **拨动开关状态宏**

宏 Macro	含义 Meaning
open	拨动开关打开
close	拨动开关关闭
disable	拨动开关禁用
loading	正在加载: 拨动开关被拨动后所执行的事件函数工作未完成

- **返回值**

返回事件函数的返回值.

4.1.2 SetToggleState

- **这个函数用于设置拨动开关的状态.**

```
void SetToggleState(int ToggleName, short SetState);
```

- **参数**

< ToggleName > 指定拨动开关名称(此名称只能是非负整数数值)
< SetState > 指定拨动开关的状态(此处填写拨动开关状态宏)

4.1.3 GetToggleState

- **这个函数用于获取拨动开关的状态.**

```
short GetToggleState(int ToggleName);
```

- **参数**

< ToggleName > 指定拨动开关名称(此名称只能是非负整数数值)

- **返回值**

返回目标拨动开关的状态宏

4.1.4 ToggleDefaultStyle

- **拨动开关样式快速设置函数,此函数令所有拨动开关使用同一套已经预设好的样式,不再需要自己进行复杂的样式设置.**

```
void ToggleDefaultStyle();
```

4.1.5 SetToggleStyle Series

- **这一系列的函数用于自定义设置各拨动开关状态的样式(不能与 4.1.4 ToggleDefaultStyle 同时使用)**

4.1.5.1 SetOpenStyle

- **设置拨动开关打开状态的样式**

```
void SetOpenStyle(COLORREF fillcolor, COLORREF slidercolor);
```

- **参数**

< fillcolor > 指定拨动开关填充底色
< slidercolor > 指定拨动开关滑块颜色

4.1.5.2 SetCloseStyle

- **设置拨动开关关闭状态的样式**

```
void SetCloseStyle(short Togglerim, COLORREF rimcolor, COLORREF slidercolor);
```

- **参数**

< Togglerim > 指定拨动开关边框线粗细(px)
< rimcolor > 指定拨动边框线颜色
< slidercolor > 指定拨动开关滑块颜色

4.1.5.3 SetDisableStyle

- **设置拨动开关禁用状态的样式**

```
void SetDisableStyle(short Togglerim, COLORREF rimcolor, COLORREF slidercolor);
```

- **参数**

< Togglerim > 指定拨动开关边框线粗细(px)
< rimcolor > 指定拨动边框线颜色
< slidercolor > 指定拨动开关滑块颜色

4.1.5.4 SetLoadingStyle

- **设置拨动开关加载状态的样式**

```
void SetLoadingStyle(short Togglerim, COLORREF rimcolor, COLORREF slidercolor);
```

- **参数**

< Togglerim > 指定拨动开关边框线粗细(px)
< rimcolor > 指定拨动边框线颜色
< slidercolor > 指定拨动开关滑块颜色

4.1.5.5 SetHoverStyle

- **设置当鼠标悬停在拨动开关上方时的状态样式**

```
void SetHoverStyle(short Togglerim, COLORREF MainColor);
```

- **参数**

< Togglerim > 指定拨动开关边框线粗细(px)
< MainColor > 指定鼠标划过时的主题色

4.2 示例

- **以下代码展示如何创建拨动开关**

```
//编译环境：Visual Studio 2022  C++14  EasyX_20220116  使用多字节字符集;
#include "Universal.h"
#include "Window.h"
#include "ToggleSwitch.h"
Window OFW;
ToggleSwitch TS;
OFMessage OFM;

int ToggleEvent1() { cout << "ToggleOpen-1" << endl; return 0; }
int ToggleEvent2() { cout << "ToggleOpen-2" << endl; return 0; }

int main() {
    OFW.Initialize_Window(400, 400, EW_SHOWCONSOLE);
    TS.ToggleDefaultStyle();//快速设置拨动开关样式;

    OFM.PeekOFMessageUP();//启动鼠标消息(坐标)扫描;

    while (1) {
        TS.CreateToggle(0, 10, 40, close, ToggleEvent1);   //创建第一个拨动开关;
        TS.CreateToggle(1, 10, 70, disable, ToggleEvent2); //创建第二个拨动开关;
    }
};
```

Chapter 5 Organ-Field Slider (OFS)

5.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFSlider.cpp"与"OFSlider.h"文件.

使用 Organ-Field Window 前需要引用头文件: `#include "OFSlider.h"`

使用 Organ-Field Window 前需要引用类名: `OFSlider OFS;`

5.1 函数

5.1.1 CreateSlider

- **这个函数用于创建一个滑动条或进度条**

```
float CreateSlider(int SliderName, int x, int y, int width, int height, short SetState);
```

- **参数**

< SliderName > 设置滑动条或进度条名称(此名称只能是非负整数数值,并且应遵从由小到大取名;如:0,1,2,3,4,……)
< x,y > 指定滑动条或进度条坐标
< width > 设置滑动条或进度条宽度
< height > 设置滑动条或进度条高度
< SetSate > 设置滑动条或进度条状态(此处填写滑动条或进度条状态宏)

- **滑动条或进度条状态宏**

宏 Macro	含义 Meaning
normal	滑动条正常状态(可拖动)
disable	滑动条禁用拖动(仅作为进度条)

- **返回值**

返回滑动条或进度条当前进度位置(0~100)

5.1.2 SetSliderLocation

- **这个函数用于设置滑动条或进度条进度**

```
void SetSliderLocation(int SliderName, float LocationValue);
```

- **参数**

< SliderName > 指定滑动条或进度条名称(此名称只能是非负整数数值)
< LocationValue > 设置滑动条或进度条进度位置(0~100)

5.1.3 PeekSliderLocation

- **这个函数用于获取当前滑动条或进度条进度**

```
float PeekSliderLocation(int SliderName);
```

- **参数**

< SliderName > 指定滑动条或进度条名称(此名称只能是非负整数数值)

- **返回值**

返回目标滑动条或进度条进度位置(0~100)

5.1.4 SliderDefaultStyle

- **滑动条或进度条样式快速设置函数,使用同一套已经预设好的样式,不再需要自己进行复杂的样式设置.**

```
void SliderDefaultStyle ();
```

5.1.5 SetSliderStyle Series

- **这一系列的函数用于自定义设置各滑动条状态的样式(不能与 5.1.4 SliderDefaultStyle 同时使用)**

5.1.5.1 SetNormalStyle

- **设置滑动条或进度条正常状态状态的样式**

```
void SetNormalStyle(COLORREF rimcolor, COLORREF fillcolor, COLORREF bkcolor, COLORREF TextColor);
```

- **参数**

< rimcolor > 边框线粗细(px)
< fillcolor > 边框线颜色
< bkcolor > 滑动条或进度条填充底色
< TextColor > 进度字体颜色

5.1.5.2 SetDisableStyle

- **设置滑动条或进度条禁用状态状态的样式**

```
void SetDisableStyle(COLORREF rimcolor, COLORREF fillcolor, COLORREF bkcolor, COLORREF TextColor);
```

- **参数**

< rimcolor > 边框线粗细(px)
< fillcolor > 边框线颜色
< bkcolor > 滑动条或进度条填充底色
< TextColor > 进度字体颜色

5.1.5.3 SetHoverStyle

- **设置当鼠标悬停在滑动条或进度条上方时的状态样式**

```
void SetHoverStyle(COLORREF rimcolor, COLORREF fillcolor, COLORREF bkcolor, COLORREF TextColor);
```

- **参数**

< rimcolor > 边框线粗细(px)
< fillcolor > 边框线颜色
< bkcolor > 滑动条或进度条填充底色
< TextColor > 进度字体颜色

5.1.5.4 SetDragStyle

- **设置滑动条被拖动状态的样式**

```
void SetDragStyle (COLORREF rimcolor, COLORREF fillcolor, COLORREF bkcolor, COLORREF TextColor);
```

- **参数**

< rimcolor > 边框线粗细(px)
< fillcolor > 边框线颜色
< bkcolor > 滑动条或进度条填充底色
< TextColor > 进度字体颜色

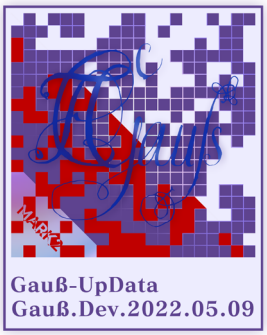
5.2 示例

- **以下代码展示如何创建进滑动条和度条**

```
//编译环境：Visual Studio 2022  C++14  EasyX_20220116  使用多字节字符集;
#include "Universal.h"
#include "Window.h"
#include "OFMessage.h"
#include "OFSlider.h"
Window OFW;
OFMessage OFM;
OFSlider OFS;

int main() {
    OFW.Initialize_Window(400, 400, EW_SHOWCONSOLE);
    OFS.SliderDefaultStyle();//快速设置拨动开关样式;
    OFS.SetSliderLocation(0, 0); //设置滑动条初始进度为 0;
    OFS.SetSliderLocation(1, 0); //设置进度条初始进度为 0;
    OFM.PeekOFMessageUP();//启动鼠标消息(坐标)扫描;
    while (1) {
        OFS.CreateSlider(0, 10, 360, 380, 30, normal);  //创建一个滑动条;
        OFS.CreateSlider(1, 10, 360, 380, 30, disable); //创建一个进度条;
    }
};
```

Chapter 6 Organ-Field Blur (OFBlur)



6.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFBlur.cpp"与"OFBlur.h"文件.

使用 OFBlur 前需要引用头文件: `#include "OFBlur.h "`

使用 OFBlur 前需要引用类名: `OFBlur OFBr;`

6.1 public 函数

6.1.1 WriteBlurCache

- **这个函数用于将原始图像写入模糊缓存(相当于截图)**

```
void WriteBlurCache(int BlurName, int x, int y, int width, int height);
```

- **参数**

< BlurName > 指定模糊名称(只能设置非负整形值)
< x,y > 指定模糊坐标
< width > 设置模糊宽度
< height > 设置模糊高度

6.1.2 ColorFilter

- **线性颜色滤镜使原始图像的颜色往设定值颜色偏移(该函数用于操作模糊缓存中的原始图像)**

```
void ColorFilter(int BlurName, int R, int G, int B);
```

- **参数**

< BlurName > 指定模糊名称(只能指定非负整形值)
< R,G,B > 指定 RGB 颜色偏移系数(可为任何整形值,具体效果请自行尝试)

- **附加 CPP 内部色彩偏移算法(方便理解)**

```
DWORD *pMem = GetImageBuffer(&IMG[BlurName].ColorFilterIMG);
float WeightR = NULL, WeightG = NULL, WeightB = NULL;
for (int i = 0; i < IMG[BlurName].OrigIMG.getwidth() * IMG[BlurName].OrigIMG.getheight(); ++i) {
float r = (GetRValue(pMem[i]) + static_cast<float>(R) * 0.5); if (r > 255) { r = 255; } else if (r < 0) { r = 0; }
float g = (GetGValue(pMem[i]) + static_cast<float>(G) * 0.5); if (g > 255) { g = 255; } else if (g < 0) { g = 0; }
float b = (GetBValue(pMem[i]) + static_cast<float>(B) * 0.5); if (b > 255) { b = 255; } else if (b < 0) { b = 0; }
pMem[i] = RGB((BYTE)r, (BYTE)g, (BYTE)b);
}
```

6.1.3 CreateGaussBlur

- **创建高斯模糊效果(该函数会创建一个独立于程序主线程之外的线程用于高斯模糊计算)**

```
void CreateGaussBlur(int BlurName, const int radius, const double SigmaDis);
```

- **参数**

< BlurName > 指定模糊名称(只能指定非负整形值)
< radius > 模糊半径(模糊半径越大模糊效果越明显,计算速度越慢)
< SigmaDis > Sigma 值越小模糊质量越好,计算速度越慢

- **性能说明**

由于本人 C++编程能力实在太渣,所以写不出高性能的高斯模糊算法,因此 OFBlur 的模糊功能存在较为严重的性能问题,计算速度比较慢无法进行实时模糊处理,小面积,小半径的高斯模糊计算速度其实可以接受.不过我也做了多线程优化,并且所有的模糊计算都是独立于程序主线程之外的,经过我的暴力测试最多可以使用 400(左右)个线程来进行高斯模糊计算,可以完全榨干 6 核 12 线程的 R5-4600H CPU 的性能.总之不建议过多的使用模糊效果.

6.1.4 ShowBlur

- **显示模糊(将已经计算完成的模糊图像显示出来)**

```
void ShowBlur(int BlurName, float ALPHA);
```

- **参数**

< BlurName > 指定模糊名称(只能指定非负整形值)
< ALPHA > 不透明度(0:完全透明~1:完全不透明)

- **说明**

此函数只会显示已经全部计算完成的高斯模糊效果,若模糊计算未完成,则该函数会等待计算完成后再进行显示;该函数的线程同样也是独立于程序主线程之外的,该函数的等待不会影响主线程.

6.1.5 ShowOrig

- **显示模糊前原始图像**

```
void ShowOrig (int BlurName, float ALPHA);
```

- **参数**

< BlurName > 指定模糊名称(只能指定非负整形值)
< ALPHA > 不透明度(0:完全不透明~1:完全透明 000)

6.2 “模糊缓存(BlurCache)”的结构

```
typedef struct BlurCache {
      IMAGE OrigIMG; unsigned int x; unsigned int y;                  //原始图片缓存;
      IMAGE ColorFilterIMG;                                           //原始图片滤色后的缓存;
      IMAGE BlurIMG; int radius; double SigmaDis; double SigmaCol;    //模糊后的图片缓存;
      IMAGE ShowIMG;                                                    //最终显示模糊的图片缓存;
      bool BlurDone;                                                    //模糊运算是否完成;
}BlurCache;
```

6.3 示例

- **以下代码展示如何创建模糊.** //编译环境：Visual Studio 2022 C++14 EasyX_20220116 使用多字节字符集;

```
#include "Universal.h"
#include "Window.h"
#include "OFMessage.h"
#include "OFBlur.h"
Window win;
OFMessage OFM;
OFBlur OFBr;

int main() {
      win.Initialize_Window(800, 800, EW_SHOWCONSOLE);
      win.SetWindowTilte("Organ-Field GUI");

      loadimage(NULL, _T("C:\\Test.png"), 800, 800);        //从硬盘读取并显示原始图片;

      OFBr.WriteBlurCache(0, 0, 0, 800, 800);               //将原始图片缓存到 BlurCache 中;
      OFBr.ColorFilter(0, -130, -130, -130);                //对原始图片进行色彩偏移处理;
      OFBr.CreateGaussBlur(0, 40, 20);                      //对色彩偏移后的图像计算高斯模糊;
      OFBr.ShowBlur(0, 1);                                  //模糊计算完成后立即显示模糊效果;

      system("pause");
};
```

Chapter 7 Organ-Field TaskManager (OFTM)

7.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFTools.cpp"与"OFTools.h"文件.

使用 OFTM 前需要引用头文件: `#include "OFTools.h "`

使用 OFTM 前需要引用类名: `OFTaskManager OFTM;`

7.1 public 函数

7.1.1 ProcessKill

- **这个函数用于结束指定进程**

```
void ProcessKill(LPCTSTR strProcessName);
```

- **参数**

< strProcessName > 目标进程名称

- **备注**

此函数为非主线程堵塞函数;

7.1.2 ProcessTraversal

- **这个函数用于获取全部进程的列表**

```
void ProcessTraversal();
```

- **备注**

此函数为非主线程堵塞函数;

7.1.3 GetTaskManagerResult

- **这个函数用于调用(获取)在 TaskManagerType 中保存的相关信息**

```
TaskManagerType GetTaskManagerResult();
```

- **返回值**

返回 TaskManagerType TMT;

- **说明**

ProcessKill 函数的返回值[1.要结束的目标进程名称],[2.结束进程的结果(进程是否被结束)]; 以及通过 ProcessTraversal 获取到的进程列表均保存在 TaskManagerType 类型中,此函数提供了一个外部接口来调用这些信息.

7.2 “TaskManagerType”的结构

```
typedef struct TaskManagerType {
      string Processlist;              //系统进程列表;
      LPCTSTR Target_strProcessName;   //要结束的目标进程名称;
      BOOL ProcessKill_Result;         //结束进程的结果(进程是否被结束);
}TaskManagerType;
```

TaskManagerType (TaskManager 类型)是保存 TaskManager 类中所有成员信息的数据类型,其中外部有两种调用这些信息的方式: 一是通过 GetTaskManagerResult 函数; 二是因为 TaskManagerType TMT 是 public 中的成员变量,故可以通过类似 OFTM.TMT.Processlist 的方式调用.

7.3 示例

- **以下代码展示获取进程列表并且结束一个进程.** //编译环境: Visual Studio 2022 C++14 EasyX_20220116 使用多字节字符集;

```
#include "Universal.h"
#include "OFTools.h"
OFTaskManager OFTM;

int main() {
      OFTM.ProcessTraversal();                  //获取系统进程列表;
      OFTM.ProcessKill("Taskmgr.exe");          //结束"Taskmgr.exe(任务管理器)"的进程;

      //这里是以两种不同的方式获取 TaskManagerType 中 ProcessKill_Result 与 Processlist(进程列表)的信息;
      cout << OFTM.TMT.ProcessKill_Result << "\n" << OFTM.TMT.Processlist << endl;
      cout << OFTM.GetTaskManagerResult().ProcessKill_Result << "\n" << OFTM.GetTaskManagerResult().Processlist
      << endl;

      system("pause");
};
```

Chapter 8 Organ-Field EXPGraph (OFEG)

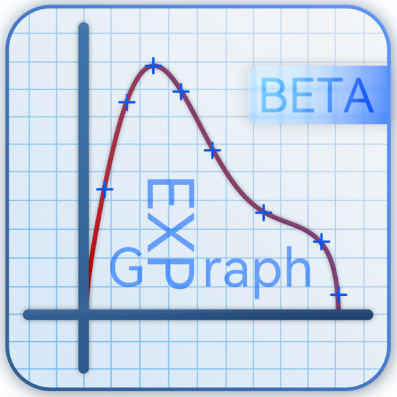
8.0 引用与类

目前 Organ-Field GUI 没有开发完成,因此在使用前需手动在你的项目中添加"OFEXPGraph.cpp"与"OFEXPGraph.h"文件.

使用 OFEG 前需要添加头文件: `#include "OFEXPGraph.h "`

使用 OFEG 前需要添加类名: `OFEXPGraph OFEG;`

8.1 EXPGraph 简要说明



EXPGraph 模块是 Organ-Field GUI 中的一个数据可视化工具,EXPGraph 并不是静态的,它是一个动态工具,能够可视化数据的动态变换过程,并渲染图片或视频. 可以使用 EXPGraph 来绘制各种函数的静态和动态图像,也可以从外部文件导入数据进行图像绘制. Organ-Field GUI 使用 C++开发,所有函数都尽量以易用为目的封装,渲染一个复杂函数图像的视频仅需要简单的几行代码(前提是使用者要具有一定 C++基础), EXPGraph 拥有较高的自动化能力,使用者无需关心内部的实现过程,因此本文档主要描述的是如何使用 EXPGraph,对其底层原理不做深入解释.

由于本 GUI 的开发者 C++水平较渣,自然 EXPGraph 的性能不会好到那里去.值得警告的是当前整个项目仍处于开发阶段(BETA),所以存在很多的 Bug 以及性能问题,要是使用时发现 CPU 占用 100% ,或者爆内存均属正常现象,若很不幸造成了您的经济或精神损失请还您自行消化.当然这属于极低概率事件.另外本作者有严重的鸽子倾向(B 站视频都是年更的那种),所以几个月不更新属于正常现象.

8.2 public 函数

8.2.1 CreateFunctionGraphBOX

- **创建一个用于绘制函数的图形盒**

```
void CreateFunctionGraphBOX(int BoxName, int x, int y, int width, int height);
```

- **参数**

< BoxName > 指定图形盒名称(名称只能为整形值)
< x > 设置位置,指定图形盒的 x 坐标值
< y > 设置位置,指定图形盒的 y 坐标值
< width > 设置图形盒宽度
< height > 设置图形盒高度

- **备注**

图形盒为矩形,图形盒的坐标指的的图形盒矩形的左上角点的坐标.这里科普一下,在计算机窗口图形中一般以屏幕左上角的点为坐标原点,原点向下为+y 方向,原点向右为+x 方向.

8.2.2 ImportValue

- **导入离散点坐标数值到图形盒,该函数用来确定离散块位置**

```
void ImportValue(int BoxName, double xValue, double yValue);
```

- **参数**

< BoxName > 指定图形盒名称(函数会将坐标值导入到指定的图形盒中)
< xValue > 给定一个离散点的 x 坐标值
< yValue > 给定一个离散点的 y 坐标值

- **备注**

本函数调用一次只会导入一个点的坐标,当导入多点坐标时一般应使用 for 循环. 如果想在一个图形盒中绘制多个图形函数,那么只需要在一个 for 循环体中调用多个 ImportValue 函数,且每个函数都指向相同的图形盒(BoxName 参数相同).

8.2.3 SSAARender

- **将使用超级采样抗锯齿进行渲染**

```
void SSAARender(int BoxName, const float Override, const short FPS);
```

- **参数**

< BoxName > 指定图形盒名称(告诉函数需要进行渲染的对象)
< Override > SSAA 渲染放大倍率(可以输入(0, 18)之间的任意数值. 数值<1 意味着缩小分辨率渲染;数值 >1 意味着放大分辨率渲染)
< FPS > 渲染视频帧率(当需要渲染视频时该参数应大于 0; 当只渲染图片时该参数应设为 0)

- **备注**

[原理] 超级采样抗锯齿就是把当前分辨率成倍提高,再把画缩放到当前的显示器上. 这样的做法实际上就是在显示尺寸不变的情况提高分辨率,让单个像素变得极小,这样就能够大幅减轻画面的锯齿感了. 该函数同时也用于渲染高分辨率图像.

[1] 当调用该函数表示将会在接下来的图形绘制(AutoVisualDraw)中导出渲染图到程序自身路径下.

[2] 当调用该函数表示将会在接下来的图形绘制(AutoVisualDraw)中启用 SSAA 渲染,目前的渲染仅为单线程,因此速度可能会较慢(特别是在渲染视频时).

[3] 当一个程序窗口中存在两个或两个以上图形盒时,不能使用 SSAARender.否则会闪退.(这是目前已发现的 Bug 之一).

[4] SSAA 视频渲染的原理是逐帧的渲染图片,当全部渲染完成后再将所有已渲染的图片按一定的顺序合并为视频.

8.2.4 AutoVisualDraw

- **当一切准备完毕后,调用此函数图形绘制的所有计算将开始进行**

```
void AutoVisualDraw(int BoxName, bool TransformMode);
```

- **参数**

< BoxName > 指定图形盒名称

< TransformMode > 图形盒变换模式(自适应缩放模式)

- **宏定义**

TransformMode	值 Value	含义 Meaning
RadiusTransformMode	false	自适应 x 轴, y 轴等比例缩放模式,即绘制图形时 X 方向与 Y 方向的拉伸比例相同.
AxisTransformMode	true	自适应 x 轴, y 轴非等比例缩放模式.图像可能会被拉伸以充盈整个图形盒空间.

- **备注**

CreateFunctionGraphBOX, ImportValue, SSAARender 只是在设置和准备数据并未进行任何的计算. 当执行 AutoVisualDraw 时,所有的计算才会开始,相当于火箭的点火按钮.

8.2.5 IMGtoVIDEO

- **将渲染图合并为视频**

```
void IMGtoVIDEO(int BoxName);
```

- **参数**

< BoxName > 指定图形盒名称(这里用于给导出的视频文件命名防止文件名冲突)

- **备注**

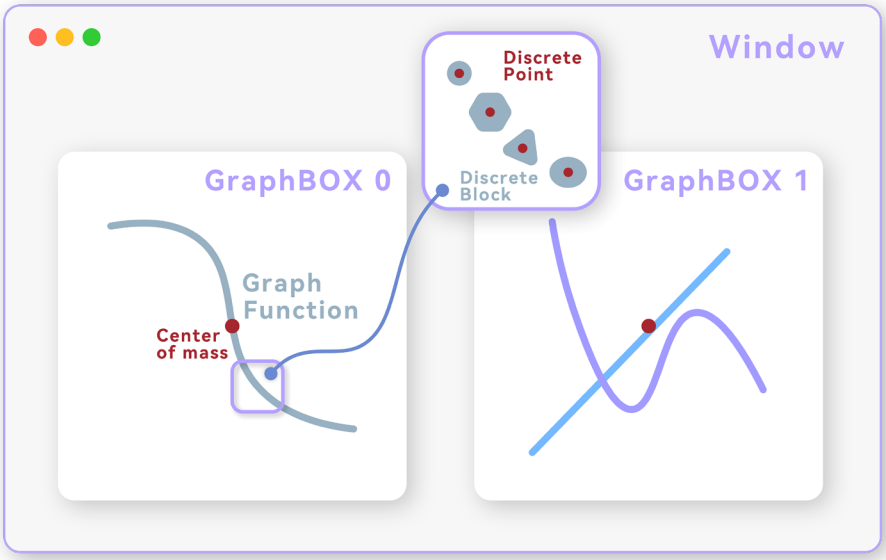
使用前提:

[1] 这是用于将 SSAA 渲染时所导出的众多图片文件合并为视频的函数,因此使用本函数的前提是必须启用 SSAARender.

[2] 将图片合并为视频的操作依赖于 FFmpeg,因此要确保你的计算机中存在正确安装的 FFmpeg.

[3] FFmpeg 官网链接: https://ffmpeg.org/

8.3 EXPGraph 名词解释



EXPGraph 是基于离散点的数据可视化工具,它的绘图层级结构如右图.

窗口(Window): 它在程序图形界面中是绝对的,为内部的一切提供了一个绝对参考系.(坐标变换这里只会简单带过一下,不会深入说明).

图形盒(GraphBOX): 在使用 EXPGraph 编写数据可视化程序时最先被创建,它能够被创建多个是容纳图形的独立容器,内部只能容纳一个图形函数(GraphFunction).图形盒左上角点在窗口中的坐标表示了它自己的位置.

离散点(DiscretePoint): 在创建完图形盒后就可以向其导入离散点坐标.离散点在物理意义上具有质点的属性.它表示离散块(DiscreteBlock)的位置(拥有在图形盒中唯一确定的坐标),并且代表了离散块的质量.

离散块(DiscreteBlock): 它理论上拥有任意形状(在 EXPGraph 中就是椭圆),是具有任意给定位置、固定密度、任意给定面积、和确定质量的理想化均质平面. 其中它质量和位置的表示移交给了离散点(离散点在离散块的几何中心处). 离散块的意义在于其质量与面积是正相关的,将面积与质量在某种意义上等同了起来,能够用于质量的直观可视化(当然我也提供了使用颜色进行质量可视化的方式).离散块的形状其实并不重要.(每个离散块的位置、面积、颜色都是可控可操纵的).

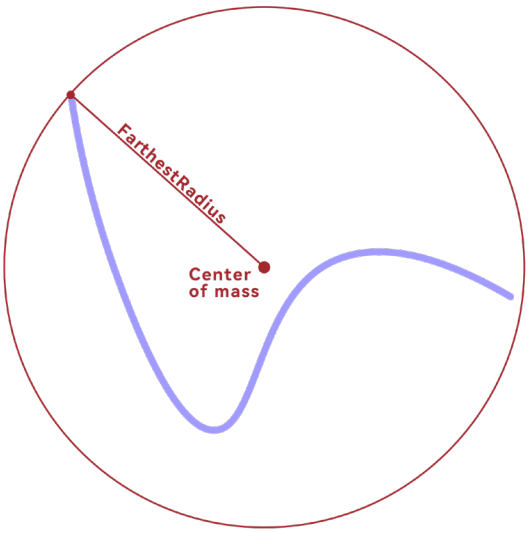
图形函数(GraphFunction): 一个图形盒中所有(有序)离散块的集合. 在计算机眼中,它就是一个储存了离散块位置、面积(质量)、颜色的数组,因此即使你向一个图形盒中导入了多个函数(例如: 上图中的 GraphBOX 1),并且在程序绘制完成后你看到它绘制出了多条函数曲线,不要被表象迷惑了,你看到的多条函数曲线只不过是你已有认知对其做的划分,对于图形盒来说这些函数曲线是一体的,它们仅是成千上万离散块组成的一个有序数组.(为什么要这样做?因为开发者懒,这样计算质心方便…)

质心(Center of mass): 准确来说是图形函数的质量(重力作用)中心(Center of mass of graph function).质心主要用于图形函数的自适应显示,无论图形函数如何变化,它的质心永远会被固定在图形盒的中央(形心),这意味着图形函数如果在不断变化,那么它的原点坐标在图形盒中的位置也会不断变化.

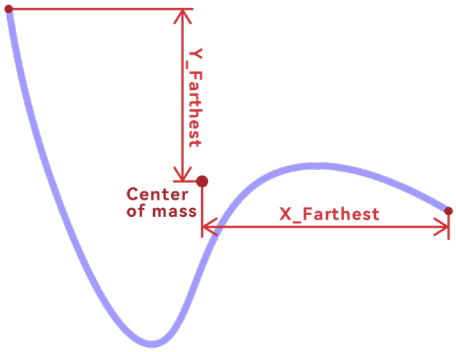
8.4 EXPGraph 的大致计算流程

[1] GravityCenterSolver: 进行图形函数在图形盒平面上唯一质心点位置的求解;

[2] FarthestSolver: ①若 TransformMode 设置为 RadiusTransformMode 将会执行图形函数距质心点最远距离的求解(也就是求距离质心最远的离散块到质心的直线距离),我把这段距离称为最远半径(FarthestRadius); ②若 TransformMode 设置为 AxisTransformMode,那么将会求解图形函数分别向“以质心为原点的坐标系”的 X 轴和 Y 轴投影后到质心距离的最大值,它们分别称为 X 方向最远值(X_Farthest)和 Y 方向最远值(Y_Farthest).

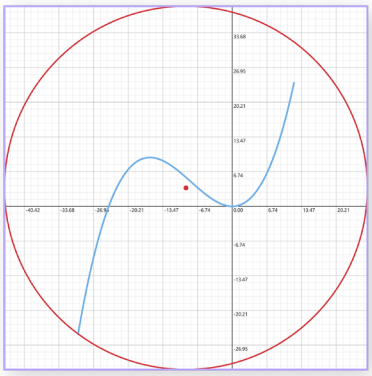


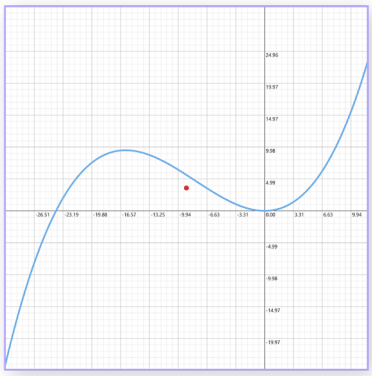
RadiusTransformMode



AxisTransformMode

[3] BoxSpaceTransformation: 进行图形盒空间变换,用于图形函数的自适应框显. 若在 FarthestSolver 中如果使用了 RadiusTransformMode 模式,程序则会自适应框显半径为 FarthestRadius 的圆,以质心为基点等比缩放图形函数,缩放的结果是半径为 FarthestRadius 的圆与图形盒边界相切. 若在 FarthestSolver 中如果使用了 AxisTransformMode 模式,程序则会将图形函数双向拉伸以匹配图形盒的长度和宽度.





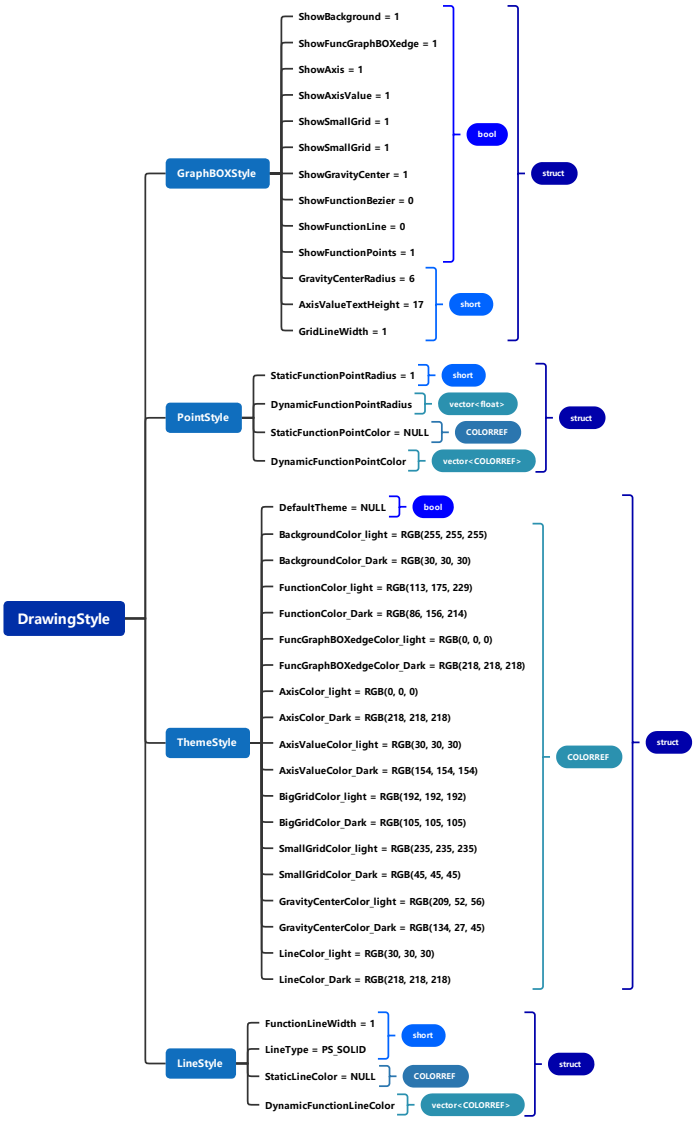
[4] AutoGrid: 自适应栅格.该程序会自动确定图形盒栅格的等比缩放或拉伸,自动确定栅格的大小和数目以及坐标轴数值.

[5] DrawFunction: 开始进行图形绘制.这里要说明的是前四步进行的是矩阵计算(也可以说是矢量计算),并将计算结果缓存,到了这里程序则开始利用之前缓存的数据进行图像绘制. 程序按照如下顺序进行图像绘制: ①填充图形盒背景颜色; ②绘制小栅格网; ③绘制大栅格网& XY 坐标轴数值文本; ④绘制 XY 坐标轴; ⑤绘制图形盒边框; ⑥绘制图形函数质量中心; ⑦绘制离散点的两两连线; ⑧绘制离散点的贝塞尔曲线; ⑨绘制离散块;

[6] FreeCache: 释放内存.当渲染与绘制任务全部完成后,程序最后的工作就是从内存中释放之前矩阵计算产生的缓存数据.

8.5 DrawingStyle 结构体

随着 EXPGraph 开发与功能的逐渐增多,我们就需要这样一个工具:“DrawingStyle 结构体”. 也就是说 DrawingStyle 结构体是 EXPGraph 的“控制面板”.并且结构体内部保存了 EXPGraph 的默认参数,通过该结构体我们可以控制图形盒中一些元素的显示,控制绘图样式(比如修改背景颜色,栅格颜色,图形函数的颜色).能够单独控制到每一个离散块的面积与颜色.




```
typedef struct DrawingStyle {
    struct GraphBOXStyle {
        //显示 GraphBOX 背景(值为 0 时背景透明);
        bool ShowBackground = 1;
        //显示 GraphBOX 边框;
        bool ShowFuncGraphBOXedge = 1;
        bool ShowAxis = 1;//显示坐标轴;
        bool ShowAxisValue = 1;//显示坐标轴数值;
        bool ShowBigGrid = 1;//显示大栅格;
        bool ShowSmallGrid = 1;//显示小栅格;
        bool ShowGravityCenter = 1;//显示质心(重心)点;
        //显示函数离散点的贝塞尔曲线
        bool ShowFunctionBezier = 0;
        //显示函数离散点的两两连线;
        bool ShowFunctionLine = 0;
        bool ShowFunctionPoints = 1;//显示函数离散点;
        short GravityCenterRadius = 6;//重心显示半径;
        //坐标轴数值字高;
        short AxisValueTextHeight = 17;
        short GridLineWidth = 1;//栅格线宽;
    } GraphBOXStyle;

struct PointStyle {
        //'点'的静态半径;
        short StaticFunctionPointRadius = 1;
        //'点'的动态半径;
        vector<float> DynamicFunctionPointRadius;
        //'点'的静态颜色;
        COLORREF StaticFunctionPointColor = NULL;
        //'点'的动态颜色;
        vector<COLORREF> DynamicFunctionPointColor;
    } PointStyle;

    struct ThemeStyle {
        bool DefaultTheme = NULL;
        COLORREF BackgroundColor_light = RGB(255, 255, 255);
        COLORREF BackgroundColor_Dark = RGB(30, 30, 30);
        COLORREF FunctionColor_light = RGB(113, 175, 229);
        COLORREF FunctionColor_Dark = RGB(86, 156, 214);
        COLORREF FuncGraphBOXedgeColor_light = RGB(0, 0, 0);
        COLORREF FuncGraphBOXedgeColor_Dark = RGB(218, 218, 218);
        COLORREF AxisColor_light = RGB(0, 0, 0);
        COLORREF AxisColor_Dark = RGB(218, 218, 218);
        COLORREF AxisValueColor_light = RGB(30, 30, 30);
        COLORREF AxisValueColor_Dark = RGB(154, 154, 154);
        COLORREF BigGridColor_light = RGB(192, 192, 192);
        COLORREF BigGridColor_Dark = RGB(105, 105, 105);
        COLORREF SmallGridColor_light = RGB(235, 235, 235);
        COLORREF SmallGridColor_Dark = RGB(45, 45, 45);
        COLORREF GravityCenterColor_light = RGB(209, 52, 56);
        COLORREF GravityCenterColor_Dark = RGB(134, 27, 45);
        COLORREF LineColor_light = RGB(30, 30, 30);
        COLORREF LineColor_Dark = RGB(218, 218, 218);
    } ThemeStyle;

    struct LineStyle {
        short FunctionLineWidth = 1;//函数线线宽;
        short LineType = PS_SOLID;//函数线线型;
        COLORREF StaticLineColor = NULL;//函数线静态颜色(静态颜色较优先,当静态颜色不存在时默认使用主题颜色);
        vector<COLORREF> DynamicFunctionLineColor;//函数线动态颜色(动态颜色最优先,当动态颜色不存在时默认使用静态颜色);
    } LineStyle;
} DrawingStyle;
```

RMSHE
2022.09.07
22:55:35
+08'00'
2021.001.2013
8

Struct	Member	Type	Defaults	Notes
GraphBOXStyle	ShowBackground	bool	true	显示图形盒背景颜色,当值为false时图形盒背景透明
	ShowFuncGraphBOXedge	bool	true	显示图形盒边框线
	ShowAxis	bool	true	显示笛卡尔坐标系的X和Y坐标轴
	ShowAxisValue	bool	true	显示X和Y坐标轴的数值
	ShowBigGrid	bool	true	显示图形盒大栅格网
	ShowSmallGrid	bool	true	显示图形盒小栅格网
	ShowGravityCenter	bool	true	显示图形函数的质心
	ShowFunctionBezier	bool	false	显示离散点的贝塞尔曲线
	ShowFunctionLine	bool	false	显示离散点之间的两两连线
	ShowFunctionPoints	bool	true	显示离散块
	GravityCenterRadius	short	6	设置图形函数质心点的显示半径
	AxisValueTextHeight	short	17	设置X和Y坐标轴数值字体的字高
	GridLineWidth	short	1	设置图形盒栅格线的线宽
PointStyle	StaticFunctionPointRadius	short	1	设置离散块的静态半径(静态半径指的是图形函数中任意离散块的面积不随时间变化)
	DynamicFunctionPointRadius	vector<float>		设置离散块的动态半径(动态半径指的是图形函数中任意离散块的面积随时间变化或不变化)
	StaticFunctionPointColor	COLORREF	NULL	设置离散块的静态颜色
	DynamicFunctionPointColor	vector<COLORREF>		设置离散块的动态颜色
ThemeStyle	DefaultTheme	bool	NULL	当值为DarkTheme(false)时,使用深色默认主题; 当值为lightTheme(true)时,使用浅色主题; 当值为NULL时为程序自动,会根据窗口背景色在上述两种主题之中自动选择合适的主题
	BackgroundColor_light	COLORREF	#ffffff	浅色主题 图形盒背景填充颜色
	BackgroundColor_Dark	COLORREF	#1e1e1e	深色主题 图形盒背景填充颜色
	FunctionColor_light	COLORREF	#71afe5	浅色主题 图形函数的颜色
	FunctionColor_Dark	COLORREF	#569ce5	深色主题 图形函数的颜色
	FuncGraphBOXedgeColor_light	COLORREF	#000000	浅色主题 图形盒边框线的颜色
	FuncGraphBOXedgeColor_Dark	COLORREF	#dadada	深色主题 图形盒边框线的颜色
	AxisColor_light	COLORREF	#000000	浅色主题 X和Y坐标轴的轴线颜色
	AxisColor_Dark	COLORREF	#dadada	深色主题 X和Y坐标轴的轴线颜色
	AxisValueColor_light	COLORREF	#1e1e1e	浅色主题 X和Y坐标轴数值字体的颜色
	AxisValueColor_Dark	COLORREF	#9a9a9a	深色主题 X和Y坐标轴数值字体的颜色
	BigGridColor_light	COLORREF	#c0c0c0	浅色主题 大栅格线的颜色
	BigGridColor_Dark	COLORREF	#696969	深色主题 大栅格线的颜色
	SmallGridColor_light	COLORREF	#ebebeb	浅色主题 小栅格线的颜色
	SmallGridColor_Dark	COLORREF	#2d2d2d	深色主题 小栅格线的颜色
	GravityCenterColor_light	COLORREF	#d13438	浅色主题 离散块质心点颜色
	GravityCenterColor_Dark	COLORREF	#861b2d	深色主题 离散块质心点颜色
	LineColor_light	COLORREF	#1e1e1e	浅色主题 静态画线颜色
	LineColor_Dark	COLORREF	#dadada	深色主题 静态画线颜色
LineStyle	FunctionLineWidth	short	1	设置画线宽度(线宽)
	LineType	short	PS_SOLID	设置画线样式
	StaticLineColor	COLORREF	NULL	设置静态画线颜色
	DynamicFunctionLineColor	vector<COLORREF>		设置动态画线颜色(这里的画线指的是离散点之间的两两连线,动态颜色指连接第一个和第二个离散点使用一种颜色,当连接到第二个与第三个离散点时使用另一种颜色,以此类推……)