hiGui 设计思路

练洋

2020 年 6 月 28 日

目录

Chapter 1

整体框架和流程

hiGui 设计思路是在学习 BeGui 过程中的记录，主要将 BeGui 的实现方法进行解析，作为自己学习的记录，如果有幸也可以作为他人学习或开发 gui 引擎的参考。BeGui 是一款基于 win32 系统，用 opengl 进行渲染的简易 gui。因为简单，所以在此基础上进行解析，同时后期解析完成之后，在此基础上进行一些列的改进，作为自己一个可用的 gui 框架。

1.1 app 消息循环机制

win32 中，操作系统接收底层交互设备的输入，如鼠标、键盘等的输入。有消息输入时，windows 会将消息先存储到操作系统的消息队列中，同时有一个专门的操作系统线程将这些消息投递给对应的应用程序。转投过程中，会先判断消息来源于那个应用程序，确切来说是哪一个线程，windows 操作系统会给每一个 gui 线程维护一个消息队列，然后 gui 线程内部对这些消息进行内部处理，进行消息转投、处理等等，具体内容将会在下面展开。
Q: 操作系统如何识别消息来自于哪一个线程?
A:windows 在 PCB 进程控制块中会记录每一个线程打开的文件列表，新建的窗口等信息。首先操作系统接收消息之后就会通过窗口管理系统判断是哪一个窗口或组件获取信息，同时将该窗口和各个进程控制块进行比较，找到窗口所属的进程（gui 线程），同时 gui 线程拥有一个消息队列，操作系统就会把消息投递到该消息队列中。
Q: 什么会当成 gui 线程?

A: 当线程调用了 user 或 gdi 模块的 windows api 函数，那么该线程就被认为是 gui 线程，windows 会为该线程新建一个消息队列。
Q: 该过程的函数调用流程?
A:win32 有一个入口函数，该函数的定义如下:

```
int WINAPI WinMain(HINSTANCE hInstance, // Instance
                   HINSTANCE hPrevInstance, // Previous Instance
                   LPSTR lpCmdLine, // Command Line Parameters
                   int nCmdShow); // Window Show State
```

在 gui 框架中，一般包含 app 类、frame 窗口类，其中 app 是单例。在程序启动后就会进入上述的 WinMain 函数中，app 就会初始化 frame 窗口（下文将此称做 framewindow），framewindow 在初始化中就会在创建一个窗口，在创建窗口的过程中会给窗口指定一个回调函数，回调函数的原型如下:

```
LRESULT     CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

该回调函数就是当该窗口接收到相关信息时，如果当前线程去 peekmessage，就会调用此回调函数，然后此回调函数就会根据消息的类型进行判断和分类，比如是鼠标消息还是键盘消息，内部 gui 经过分类后，就需要对消息是在哪个组件上响应的做出判断，以响应对应的事件。在没有消息时，程序会处理一些 idle 时相关的事项，同时会对需要同步的内容进行渲染。在 app 的 run 循环中，如果有消息为 WM_QUIT 则退出程序。
app 消息的整体框图如下所示:

1.2 窗口管理与交互

窗口管理与交互主要包括这几个方面: 窗口消息传递；激活窗口处理；窗口更新及同步问题；domodal 模态窗口的实现。窗口消息传递是指 gui 线程获得操作系统消息之后，怎么将消息进行投递，将消息转化为 OnMouseMove(int x, int y, ...)，OnKeyUp(...)，OnKeyDown(...) 等等消息，让对应的组件响应这些事件，同时这些事件发生后，如果绑定不同事件产生时控件需要执行的任务。激活窗口处理主要包括在如何响应操作系统给出的激活相关消息，但鼠标点击或键盘 tab 键时，焦点或激活应该落在哪个窗口或组件上。窗口更新及同步问题：响应时间需要组件重绘，整个 hiGui 是如何进

表 1.1: 消息类型

类型	Character Block Name
WM_ACTIVE	窗口激活或未被激活时操作系统会产生这样的消息
WM_SYSCOMMAND	系统命令
WM_CLOSE	窗口接收到关闭的命令
WM_PAINT	窗口接收到绘制消息，有时候该消息是程序自动产生的
WM_KEYDOWN	键盘按下事件
WM_KEYUP	键盘释放事件
WM_SIZE	窗口大小发生变化
WM_MOUSEDOWN	鼠标按下
WM_MOUSEMOVE	鼠标移动事件
WM_LBUTTONUP	鼠标左键释放

行的；窗口有时候大部分不需要重绘，只需要在发生改变的地方重绘即可，这样可以极大提高 gui 线程的交互效率，我们该如何处理；同时有的窗口它不接受消息，但是它的显示需要和其他组件进行同步。模态窗口是 gui 使用过程中常见的一种，主要表现为当前窗口阻塞其他窗口的消息接收，同时在渲染时，应该对 modal 窗口进行特殊处理，让用户更加明确。

1.2.1 关于各个控件的 UI 交互信息的传递

当有消息并利用 user32.dll 调用之前的回调函数进行, 原型如下:

$$WndProc(HWNDhWnd, UINTuMsg, WPARAMwParam, LPARAMlParam);$$

该回调函数会根据调用 app 中的 framewindow 中的与之原型相同的函数进行处理，并进行消息分类。先是对 uMsg 进行判断，uMsg 包括的消息类型在 win32 中如下表所示。

该表列举了一部分内容，具体的内容可以查阅 MSDN 及 WinUser.h 文档。在 hiGui 处理这些消息时，会将消息进行简单处理再转投到内部。对于非鼠标和键盘事项，hiGui 会通过 input 命名空间中的函数，对当前点击的位置、时间和按下的鼠标进行记录，然后在 FrameWindow 中对应处理。FrameWindow 处理完成之后会转到 Window 再处理。Window 中定义有 Container 类的 m_contents，它将使用 m_contents 进行处理，Container 是组件容器类，Container 会对里面的组件的可见进行遍历，找出鼠标位置

是位于哪一个组件中，其中组件又包括有容器或组件，这时候会一直往下遍历寻找，如果找到了最终的组件（不可再细分），那么最终的组件将会接收到响应时间并作出动作。 组件会对特定的事项进行响应，以执行相关的任务，在 hiGui 中就是在对应的 OnMouseDown(...)，OnButtonDown(...) 等函数中执行特定的回调函数，如果回调函数为 null，那么不执行。

1.2.2 关于激活窗口

激活窗口

1.2.3 窗口更新和同步

1.2.4 关于 domodal 的实现

在执行类似 dlg.showModal() 这样的命令时，hiGui 首先做的就是找到整个应用程序的主窗口，主窗口中有 container 组件，该组件中包括了两个指针，一个是 domodal 组件一个是激活组件，设置当前 dlg 为当前 domodal 组件，在消息循环处理过程中会首先判定是有有 domodal 组件，如果没有则消息传递到子控件中，如果有，则所有消息由 domodal 接收处理并直接返回。

1.3 opengl 渲染过程

hiGui 采用 opengl 渲染，在创建窗口的过程中同时将 opengl 上下文创建好，并存放在主窗口中。在应用程序退出时，就会执行相关的释放动作，也包括将 opengl 相关资源释放。opengl 渲染内容包括窗口边框、文字、以及各个部件。开始由 app 调用 framewindow 的 frameRender，frameRender 的渲染作为所有渲染的开始，在渲染之初就会清理所有的缓存（颜色缓存和深度缓存），用背景颜色进行填充。之后，frameRender 先渲染主窗口，主窗口如果有标题，那么渲染标题，然后渲染主要区域；主窗口本体部分渲染完成之后，会渲染其 container 相关内容，container 会先遍历渲染其子控件，最后如果有 modal 控件，那么这时候采用的方式混合，及将之前渲染的内容和即将要渲染的 modal 控件进行混合，以达到 modal 控件的效果。每个组件的渲染方式主要是设置 opengl 的相关状态，渲染边框，渲染图标

等，如果组件在不同的状态下（获取焦点、失效等）还会根据这些状态选择渲染的颜色及对应的图标，表示不同的状态。

1.4 动画系统

在 BeGui 中没有动画系统，我们在实现 hiGui 一开始就考虑将动画系统加上，动画系统的整体思路就是，给每一个需要渲染过程进行判定，查看是否需要实现动画，采用定时器或者插帧方式实现。

1.5 各个组件的实现介绍

hiGUi 中实现的组件有：button、checkbox、combobox、container、dialog、window、group、imageBox、label、listbox、menu、radiobutton、propertySheet、slider、scrollbar、tab。同时计划后期增加:treeCtrl、progressbar；同时增加对应的 3D 状态的组件。

1.6 存在的问题和改进

因为刚开始是基于 BeGui，这是一个很简单的 gui 框架，但同时也存在很多问题，我们需要达到一个基本好用的一个水平，需要对其进行改进，同时参考其他比较好的 gui 框架，构建我们的 gui 框架平台。
接下来，每章节就以流程结构和关键代码讲解方式逐一说明。

Chapter 2

app 消息循环实现与解读

2.1 windows 下的消息循环机制

windows 操作系统的消息机制如图所示，图中略去了操作系统接收事件，然后维护应用程序消息队列这一块。应用程序启动后，不断地从消息队列里面去取消息，然后处理消息。但是从图中我们发现有一个 WndProc 回调函数在 Run 和 OnXyz 之间，是因为 run 函数调用 peekmessge 去获取消息队列中的消息，而消息队列是操作系统维护的，必须由操作系统进行处理，但调用 peekMessage 时，就变成系统调用 user32 中的相关函数了，而由 WndProc 转为普通函数调用，继续处理。

2.1.1 windows 消息

window 中定义的消息类型为 MSG, 它的定义如下：

```
typedef struct tagMsg
{
    HWND hwnd; //接收该消息的句柄
    UINT message;//消息常量标识
    WPARAM wParam; //消息附加信息
    LPARAM lParam; //消息附加信息
    DWORD time; //消息创建的时间
    POINT pt; //消息创建是鼠标在屏幕系统中的位置
```

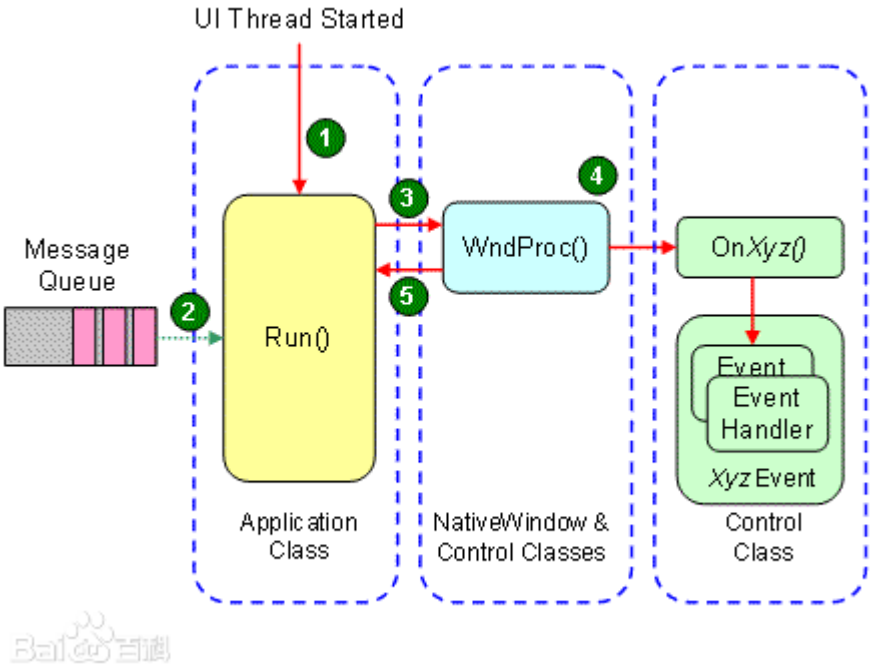

图 2.1: windows 消息机制 [1]

```
9     }MSG;
```

消息由操作系统或应用程序产生，但系统中发生输入事件；应用程序可以产生消息，用于通信或是窗使窗体执行任务。
windows 消息类型包括以下两种:

- windows 系统定义消息：包括窗口消息、命令消息和控件消息
- 程序定义消息：用户自定义消息，但数值范围有规定

2.1.2 消息的相关处理

每个 ui 线程维护一个消息队列, 并且消息队列并非是抢占式的, 而是按照先进先出原则进行处理。但需要注意的是:线程消息队列中的 WM_PAINT, WM_TIMER 只有在没有其他消息时才会被处理, 并且 WM_PAINT 还会被合并用以提高效率。对于大部分消息都会进入操作系统消息队列，然后转发给线程消息队列。但是还有一部分消息不会入队，就是产生该消息之后，绕过消息队列，直接就会调用 ui 线程的窗口回调函数 WndProc 执行，这一类消息包括:WM_ACTIVE, WM_SETFOCUS, WM_SETCURSOR, WM_WINDOWPOSCHANGED。并且 postMessage 函数发送的是队列消息，发送完之后立即返回；而 sendMessage 发送的是非队列消息，只有等待接收线程将消息处理完成之后才返回，它是阻塞型。
Q: 线程如何去取消息呢?
A: 可以使用 windows 这两个 API，一个是 PeekMessage，另外一个是 GetMessage。其中 PeekMessage 就是从消息队列里面查看是否有消息，如果有进行处理，如果没有那么就行另外的 idle 处理。而 GetMessage 就是会等待是否有消息，等待有消息就才返回。
tips: 非常重要的一点,就是不能用 PeekMessage 从消息队列中删除 WM_PAINT 消息，如果队列中包含 WM_PAINT 消息，PeekMessage 就会进入死循环，这时候需要借助 DefaultWndProc 进行处理，移除掉该消息。
tips: 注意消息死锁！如果线程 A SendMessage 给线程 B, 此时线程 A 在等待线程 B 的返回；线程 B 收到线程 A 的消息之后进行处理，在处理的过程中 sendMessage 给线程 A，这时候因为线程 A 是此时挂起，无法响应线程 B 的请求，导致了 AB 相互等待，造成死锁。如果有多个线程一次发送消息，则可能造成环形死锁。
Q: 取得消息之后如何处理呢?

A:windows 系统对所有键盘编码都采用虚拟键的定义，这样当按键按下时，得到的并不是字符消息，此时需要映射消息，TranslateMessage 用于将虚拟键消息转换为字符消息，并将转换后的新消息投递到调用线程的消息队列中。也就是说如果 Translate 看是否有字符键消息，如果有则产生 $\mathrm{WM}_C HAR$ $DispatchMessage\ TranslateMessage$

2.2 linux 下的消息循环机制

linux 本身其实并不包括消息机制，最主要的是。

2.3 mac 系统洗的消息循环机制

2.4 windows 下 app 框架结构及实践

hiGui 主要讲解的是 windows 下的 gui 实现方式，重点讲解 windows 相关 app 框架结构和实践。该部分内容主要从代码编写入手讲解每一个过程，从新建 App 应用程序、frame 窗口建立以及消息循环建立三个方面讲述 app 框架，为了便于观看，同时每一步中我会将不必要出现的代码删除。

2.4.1 BaseAppWin 类的建立

BaseAppWin 类是应用程序框架类，该类是一个单例，并且在程序刚开始启动的时候就在 winmain 函数中实例化。该类的主要功能包括：初始化主窗口和维持整个消息循环。

```
1   class BaseAppWin{
2    private:
3        static BaseAppWin* m_pInstance;
4        BaseAppWin(const BaseAppWin&);
5    public:
6        virtual bool initialize(const std::string& title, size_t width,
7                size_t height, Window::Style frame_style = Window::MULTI
8                FrameWindow::Options* opt = 0);
9        virtual int run();
10       static BaseAppWin* instance(){
```

```
11                  if(!m_pInstance)
12                  {
13                          m_pInstance = new BaseAppWin();
14                  }
15                  return m_pInstance;
16          }
17      protected:
18          BaseAppWin();
19          virtual ~BaseAppWin();
20  };
```

该类为单例，同时作为其他 app 类的基类，因此构造函数和复制构造函数在子类中就变成了 private，同时复制构造函数为 private，不实现；这样保证了单例只能通过 instance() 函数获得。initialize() 是初始化窗口函数，run() 是维持消息循环。接下来我们看着两部分的实现过程。

```
1     bool BaseAppWin::initialize(const std::string& title, size_t width, size_t he
2     Window::Style frame_style /* = Window::MULTIPLE */, FrameWindow::Options* pop
3   {
4           FrameWindow::Options opt;
5           if(popt)
6           {
7                   opt = * popt;
8           }
9           else
10          {
11                  opt.bOwnDraw = true;
12                  opt.bFullScreen = true;
13                  opt.nColorBits = 16;
14                  opt.nDepthBits = 16;
15                  opt.nStencilBits = 0;
16          }
17          FrameWindow::createInstance();
18          sd::string wnd_style = "std_framewnd";
```

```
19          if(opt.windowStyleName.length() > 0)
20                  wnd_style = opt.windowStyleName;
21          try{
22                  FrameWindow::instance()->create(0, 0, (int)width, (int)h
23          }
24          catch(std::exception& e)
25          {
26                  //to do error process!
27                  return false;
28          }
29          //do extra initialization
30          //做渲染
31          return true;
32  }
```

在 app 创建窗口的过程中，指定了窗口的大小和风格，窗口如果没有创建成果，则进行错误处理，返回 false；如果创建成功则需要做渲染相关的工作。

```
1   int BaseAppWin::run()
2   {
3           MSG msg;        //windows消息结构体
4           BOOL done = FALSE; //用于指示是否完成，需要退出消息循环
5           while(!done)
6           {
7                   WaitMessage();//等待消息到来，如果没有消息，则线程挂起
8                   if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
9                   {
10                          /*message是消息的标识号，为了方便记忆，msg.messa
11                          它是一个空消息
12                          */
13                          if(msg.message == 0)
14                          {
15                                  if(FrameWindow::instance()->isActive())
16                                  {
```

```
17                                          //做渲染,todo
18                                  }
19                          }
20                          if(msg.message == WM_QUIT)
21                          {
22                                  done = TRUE;
23                          }
24                          else
25                          {
26                                  TranslateMessage(&msg);
27                                  DispatchMessage(&msg);
28                          }
29                          //做渲染,todo
30                  }
31                  else
32                  {
33                          //如果没有消息,做其他相关处理
34                  }
35          }
36          FrameWindow::instance()->free();
37          return msg.wParam;
38  }
```

整个过程中，如果接收到了 WM_QUIT 消息，done 就变成 TRUE, 同时结束 while 消息循环，并释放窗口退出主程序。当 while 运行中，如果存在消息, 它会对消息进行处理和转发, 一直会将消息转发到窗口回调函数中。

2.4.2 FrameWindow 类的建立

frameWindow 类主要是创建 opengl 窗口、被回调函数调用处理消息以及管理各级组件，这部分主要是查看该类与 app、回调函数以及各级组件之间的消息关系，清楚整个消息流动方向。

```
class FrameWindow: public Window{
public:
        struct Options{
                bool bOwnDraw;
                bool bFullScreen;
                int nColorBits;
                int nDepthBits;
                int nStencilBits;
                std::string windowStyleName;
        };
        virtual void create(int left, int top, int width, int height, co
                const std::string &style_name = "std_framewnd");
        static FrameWindow* FrameWindow::createInstance();
        static FrameWindow* instance(){ return m_pInst; }

        virtual void free();
        Options getOptions()const{return m_options;}
        virtual bool onMouseDown(int x, int y, int button);
        virtual bool onMouseMove(int x, int y, int prevx, int prevy);
        virtual bool onMouseUp(int x, int y, int button);
        virtual void onKeyDown(int key);
        virtual void onKeyUp(int key);
protected:
        FrameWindow();
        virtual void createGLWindow(int left, int top, int width, int he
        virtual void freeGLWindow() = 0;
        Options m_options;
        static FrameWindow* m_pInst;
};
```

create 函数调用 CreateGLWindow 进行窗口初始化，free 调用 freeGLWindow 销毁 opengl 和窗口相关资源；在回调函数过程中，会根据消息标识号判定哪个是鼠标消息，哪个是键盘消息，以及将相关的参数传递过来，这时候 onMousexxx 和 onKeyxxx 就是用来处理对应的事件。

```
1 //todo
```

Chapter 3

窗口管理与交互

3.1 opengl 窗口的创建和资源释放

FrameWindow 是所有主窗体的基类，opengl 渲染的实现，需要在该基类的子类中实现对应的 opengl 创建和销毁代码。因此创建了 FrameWindow_Win32 类，该类的定义如下：

```
1    class FrameWindow_Win32: public FrameWindow{
2  public:
3          LRESULT wndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
4  private:
5          HINSTANCE m_hInstance;
6          HWND m_hWnd;
7          HDC m_hDC;
8          HDC m_hMemDC;
9          HGLRC m_hRC;
10 protected:
11         FrameWindow_Win32();
12         virtual void createGLWindow(int left, int top, int width, int height, co
13         virtual void freeGLWindow();
14 };
```

该类中定义有窗口回调函数，实现 createGLWindow 和 freeGLWindow 两个 gl 函数；在该类中定义的变量中，m_hRC 就是 opengl 对于的 context，

m_hDC 指的是当前窗口具有的上下文，m_hWnd 是当前窗口具有的句柄，m_hInstance 是模块基地址，m_hMemDC 是内存绘图上下文。
代码段,todo. 从 createGLWindow 中可以看出，创建过程包括这几个部分，第一部分是创建窗口，获取当前模块基地址 m_hInstance，然后根据该基地址及定义的 WNDCLASS 结构体，调用 windows api createWindowE 去创建窗体，获得 m_hWnd; 第二部分就是创建 opengl 的上下文，根据创建窗口的一些参数初始化 PIXELFORMATDESCRIPTOR 结构体，找出并设置该结构体，之后调用 win GDI 中的 api wglCreateContext 创建 opengl 上下文；第三部分就是设置上下文，进行校对。同时 createGLWindow 该函数还做了窗体是否重叠显示的部分，该部分是用混合实现的。最后面给 opengl 创建了一个离线渲染 surface。

3.2 窗口组件层级管理

3.3 事件的流转及处理

Chapter 4

opengl 渲染实现及解读

4.1 opengl 渲染流程

4.2 图片的加载

4.3 字体加载及渲染

Chapter 5

动画系统的实现

Chapter 6

组件的实现介绍

Chapter 7

存在的问题与改进

7.1 一个好的 gui 引擎标准

一个好的 gui 引擎，应该是简单易用，开发过程容易上手；同时功能强大，效率高，稳定性好。所以，笔者认为好的 gui 引擎应该具备的基本条件是：

- 类接口简单
- 方便的自动化布局模块
- 支持工作线程和 gui 线程同步机制
- 较好的事件与组件的绑定机制
- 很高的渲染效率
- 跨平台（方便一次学习，多地使用）
- gui 编辑工具和相关环境
- 稳定的性能
- 支持动画
- 组件风格可以自定义
- 如果采用 3D 图形 api 渲染，那么 3D 相关内容应该强化
- 3D api 可以进行替换

7.2 效率方面的提高

效率方面的提高包括两个部分，一个是 gui 线程本身在做渲染时不能占用太多的资源，需要对齐进行优化；另外就是应用程序包括 gui 部分，同时也要处理其他任务，gui 整体框架能否将两者较好地分离开，能对效率有很大的提升。

7.3 3D 渲染内容的提高

7.4 跨平台修改

7.5 自动化布局

7.6 具有相关工具

参考文献

[1] Timothy I Murphy. Line spacing in latex documents. [EB/OL]. https://baike.baidu.com/item/windows%E6%B6%88%E6%81%AF%E5%A4%84%E7%90%86%E6%9C%BA%E5%88%B6/9420571?fr=aladdin Accessed April 4, 2010.