**wxPython + pyOpenGL，打造三维数据分析的利器**

**文章目录**

* [1. 前言](https://xufive.blog.csdn.net/article/details/97020456#1__2)
* [2. 关于 wxPython](https://xufive.blog.csdn.net/article/details/97020456#2__wxPython_12)
* [3. 关于pyOpenGL](https://xufive.blog.csdn.net/article/details/97020456#3_pyOpenGL_20)
* [4. 架起沟通 wxPython 和 pyOpenGL 的桥梁](https://xufive.blog.csdn.net/article/details/97020456#4__wxPython__pyOpenGL__36)
* [5. 场景、视区和模型](https://xufive.blog.csdn.net/article/details/97020456#5__299)
* [6. 三维重建的实例](https://xufive.blog.csdn.net/article/details/97020456#6__379)
* [7. 后记](https://xufive.blog.csdn.net/article/details/97020456#7__460)

**1. 前言**

在三维显示领域，OpenGL 是神一样的存在，其地位就像编程语言里面的 C 一样。基于 OpenGL 衍生出来的分支、派系，林林总总，多如牛毛。Python 旗下，影响较大的三维库有 pyOpenGl / VTK / Mayavi / Vispy 等，它们各自拥有庞大的用户群体。VTK 在医学领域应用广泛，Vispy 在科研领域粉丝众多。VTK 和 Vispy 都是基于 OpenGL 的扩展，Mayavi 则是基于VTK 的，因此很多的医学影像应用都是采用 Python + VTK + ITK + Mayavi 的组合（ITK 是图像处理库，类似于 OpenCV 或 PIL）。

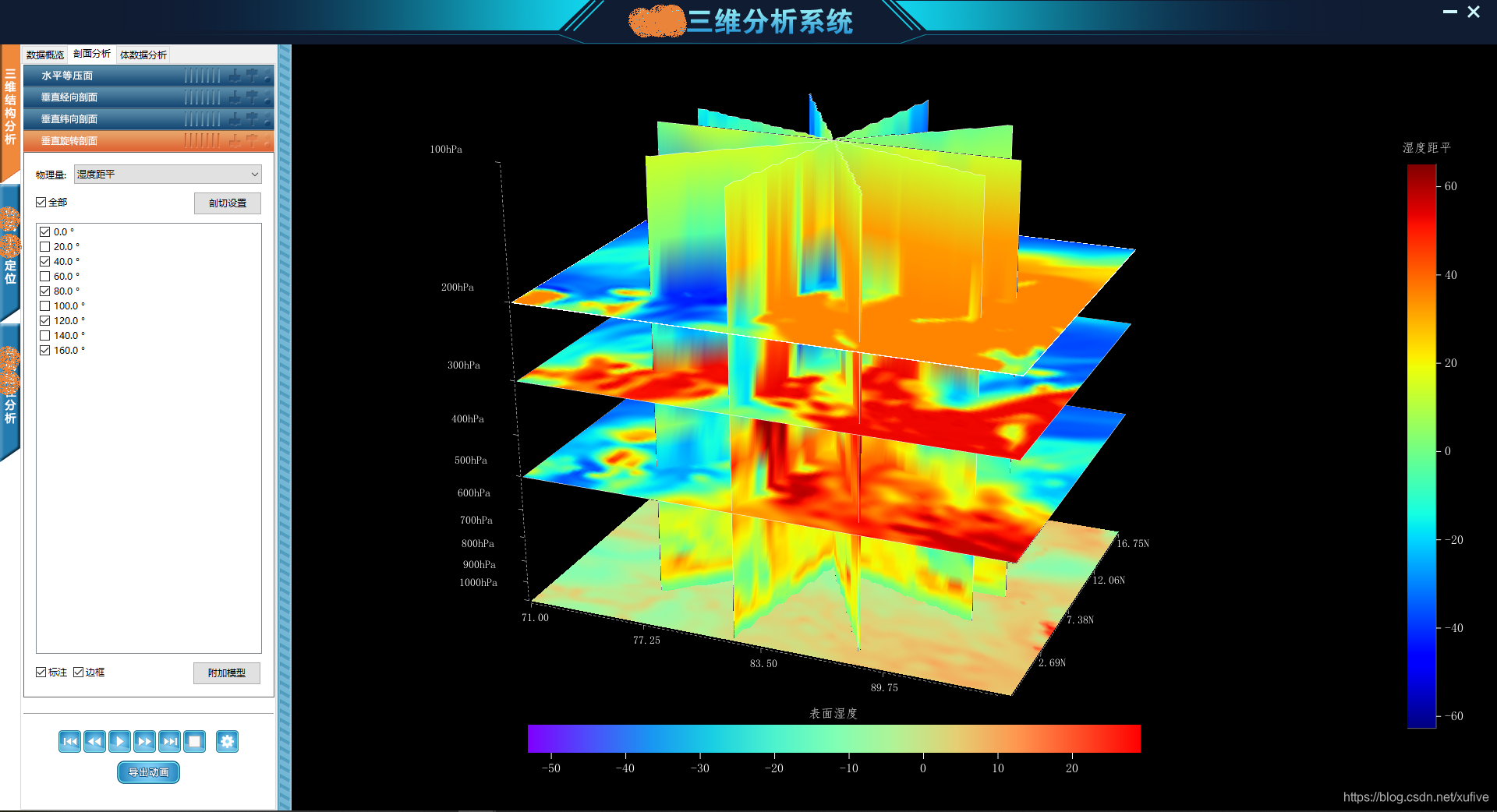
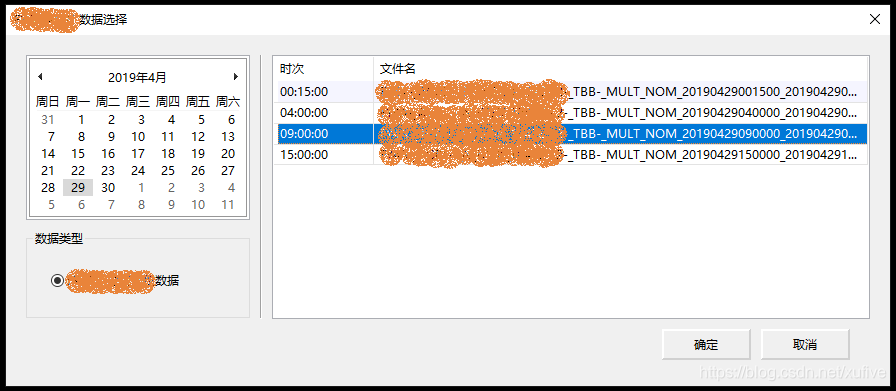
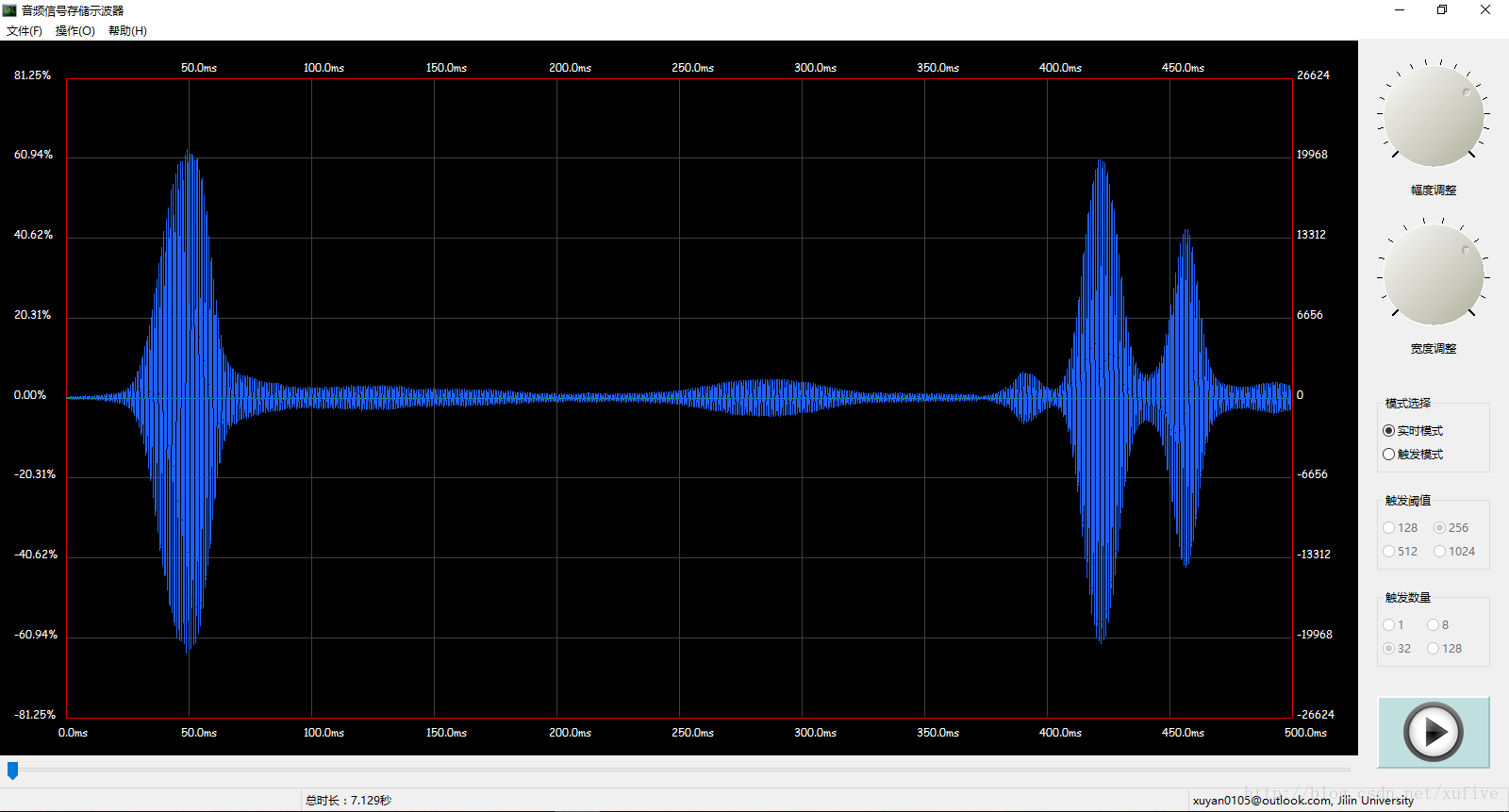
上述三维渲染库，包括 pyOpenGl，都有一个共同的特点，那就是只专注于三维功能的实现，而疏于对 UI 的支持。比 Vispy，虽然支持以 wx 或者 Qt 作为后端，但绑定后端以后，在窗口管理、交互操作等方面还是存在不少问题。pyOpenGl 做得更简单，提供一个 glut 库就算是对 UI 的支持了。

事实上，在复杂的三维展示系统中，UI 的重要性并不亚于 OpenGL。如果能为 OpenGL 找到一位 UI 搭档，必将提高程序的可靠性和可操作性，增强用户感受。wxPython 和 pyOpenGL 就是这样的一对黄金搭档。有诗赞曰：

**面壁十年图破壁，宝剑霜刃未曾试。  
秋风策马出京师，开启三维新天地。**

**2. 关于 wxPython**

我一直认为，wxPython 是最适合 python 的GUI库，并为此专门写过一篇博文。详情见[《wxPython：python首选的GUI库》](https://blog.csdn.net/xufive/article/details/82665460)。这里不再讨论如何使用 wxPython，只贴出几张开发项目的截图，展示一下 wxPython 的风格。

下图为 wxPython + pyOpenGL 开发的项目截图（隐去敏感信息）：  
下图为界面细节展示（隐去敏感信息）：  
  
下图为 wxPython 的传统风格：  


**3. 关于pyOpenGL**

pyOpenGL 的入门教程有很多，我也有一篇博文滥竽充数，详见[《写给 python 程序员的 OpenGL 教程》](https://blog.csdn.net/xufive/article/details/86565130)。特别提醒一下，这篇博文最后提到顶点缓冲区对象 VBO，并有演示代码。VBO 的概念很重要很重要很重要，只有学会使用 VBO，才能真正进入 OpenGL 的精彩世界。

早期的 OpenGL 使用立即渲染模式（Immediate mode，也就是固定渲染管线），概念清晰易于理解，绘制图形也很方便，但效率太低。从 OpenGL3.2 开始，规范文档开始废弃立即渲染模式，并鼓励开发者在 OpenGL 的核心模式（Core-profile）下进行开发，这个分支的规范完全移除了旧的特性。

VBO 是 OpenGL 核心模式的基础。VBO 将顶点数据集存储在 GPU 中，这意味着渲染 VBO 数据会很快。不过，数据从 RAM 传送到 GPU 是有代价的。VBO 虽然在 GPU 上，但并没有使用 GPU 的运算功能。在 VBO 之上，还有 VAO 的概念，即Vertex Array Object，顶点数组对象。这个概念很复杂，我们可以简单把 VAO 理解为 VBO 管理者。由于 VAO 依赖于显卡，通用性较差，我选择绕过它。

说实话，我对 OpenGL 的核心模式了解不多，对于着色器语言 GLSL 更是畏之如虎，对 VBO 的理解也不见得正确。虽然在模型拾取、体数据绘制、三维重建等方面，我的代码跑出来的效果还算差强人意，我仍然觉得我的方法与主流思路不同。很多时候，我喜欢说我的方法是“独辟蹊径”。

下面是我在工作中绘制的一些三维效果图：

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**4. 架起沟通 wxPython 和 pyOpenGL 的桥梁**

wx.glcanvas.GLCanvas 是 wxPython 为显示 OpenGL 提供的类，顾名思义，我们可以将其理解为 OpenGL 的画板。有了这个画板，我们就可以使用 OpenGL 提供的各种工具在上面绘制各种三维模型了。

下面这段代码，从 wx.glcanvas.GLCanvas 派生了新类 WxGLScene，绑定了鼠标滚轮事件，并以立即渲染模式（Immediate mode）画了两个三角形。受限于篇幅，删去了鼠标拖拽操作，仅保留了滚轮缩放功能。

# -\*- coding: utf-8 -\*-

import wx

from wx import glcanvas

from OpenGL.GL import \*

from OpenGL.GLU import \*

class WxGLScene(glcanvas.GLCanvas):

"""GL场景类"""

def \_\_init\_\_(self, parent, eye=[0,0,5], aim=[0,0,0], up=[0,1,0], view=[-1,1,-1,1,3.5,10]):

"""构造函数

parent - 父级窗口对象

eye - 观察者的位置（默认z轴的正方向）

up - 对观察者而言的上方（默认y轴的正方向）

view - 视景体

"""

glcanvas.GLCanvas.\_\_init\_\_(self, parent, -1, style=glcanvas.WX\_GL\_RGBA|glcanvas.WX\_GL\_DOUBLEBUFFER|glcanvas.WX\_GL\_DEPTH\_SIZE)

self.parent = parent # 父级窗口对象

self.eye = eye # 观察者的位置

self.aim = aim # 观察目标（默认在坐标原点）

self.up = up # 对观察者而言的上方

self.view = view # 视景体

self.size = self.GetClientSize() # OpenGL窗口的大小

self.context = glcanvas.GLContext(self) # OpenGL上下文

self.zoom = 1.0 # 视口缩放因子

self.mpos = None # 鼠标位置

self.initGL() # 画布初始化

self.Bind(wx.EVT\_SIZE, self.onResize) # 绑定窗口尺寸改变事件

self.Bind(wx.EVT\_ERASE\_BACKGROUND, self.onErase) # 绑定背景擦除事件

self.Bind(wx.EVT\_PAINT, self.onPaint) # 绑定重绘事件

self.Bind(wx.EVT\_LEFT\_DOWN, self.onLeftDown) # 绑定鼠标左键按下事件

self.Bind(wx.EVT\_LEFT\_UP, self.onLeftUp) # 绑定鼠标左键弹起事件

self.Bind(wx.EVT\_RIGHT\_UP, self.onRightUp) # 绑定鼠标右键弹起事件

self.Bind(wx.EVT\_MOTION, self.onMouseMotion) # 绑定鼠标移动事件

self.Bind(wx.EVT\_MOUSEWHEEL, self.onMouseWheel) # 绑定鼠标滚轮事件

def onResize(self, evt):

"""响应窗口尺寸改变事件"""

if self.context:

self.SetCurrent(self.context)

self.size = self.GetClientSize()

self.Refresh(False)

evt.Skip()

def onErase(self, evt):

"""响应背景擦除事件"""

pass

def onPaint(self, evt):

"""响应重绘事件"""

self.SetCurrent(self.context)

glClear(GL\_COLOR\_BUFFER\_BIT|GL\_DEPTH\_BUFFER\_BIT) # 清除屏幕及深度缓存

self.drawGL() # 绘图

self.SwapBuffers() # 切换缓冲区，以显示绘制内容

evt.Skip()

def onLeftDown(self, evt):

"""响应鼠标左键按下事件"""

self.CaptureMouse()

self.mpos = evt.GetPosition()

def onLeftUp(self, evt):

"""响应鼠标左键弹起事件"""

try:

self.ReleaseMouse()

except:

pass

def onRightUp(self, evt):

"""响应鼠标右键弹起事件"""

pass

def onMouseMotion(self, evt):

"""响应鼠标移动事件"""

if evt.Dragging() and evt.LeftIsDown():

pos = evt.GetPosition()

try:

dx, dy = pos - self.mpos

except:

return

self.mpos = pos

# 限于篇幅省略改变观察者位置的代码

self.Refresh(False)

def onMouseWheel(self, evt):

"""响应鼠标滚轮事件"""

if evt.WheelRotation < 0:

self.zoom \*= 1.1

if self.zoom > 100:

self.zoom = 100

elif evt.WheelRotation > 0:

self.zoom \*= 0.9

if self.zoom < 0.01:

self.zoom = 0.01

self.Refresh(False)

def initGL(self):

"""初始化GL"""

self.SetCurrent(self.context)

glClearColor(0,0,0,0) # 设置画布背景色

glEnable(GL\_DEPTH\_TEST) # 开启深度测试，实现遮挡关系

glDepthFunc(GL\_LEQUAL) # 设置深度测试函数

glShadeModel(GL\_SMOOTH) # GL\_SMOOTH(光滑着色)/GL\_FLAT(恒定着色)

glEnable(GL\_BLEND) # 开启混合

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA) # 设置混合函数

glEnable(GL\_ALPHA\_TEST) # 启用Alpha测试

glAlphaFunc(GL\_GREATER, 0.05) # 设置Alpha测试条件为大于0.05则通过

glFrontFace(GL\_CW) # 设置逆时针索引为正面（GL\_CCW/GL\_CW）

glEnable(GL\_LINE\_SMOOTH) # 开启线段反走样

glHint(GL\_LINE\_SMOOTH\_HINT, GL\_NICEST)

def drawGL(self):

"""绘制"""

# 清除屏幕及深度缓存

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT)

# 设置视口

glViewport(0, 0, self.size[0], self.size[1])

# 设置投影（透视投影）

glMatrixMode(GL\_PROJECTION)

glLoadIdentity()

k = self.size[0]/self.size[1]

if k > 1:

glFrustum(

self.zoom\*self.view[0]\*k,

self.zoom\*self.view[1]\*k,

self.zoom\*self.view[2],

self.zoom\*self.view[3],

self.view[4], self.view[5]

)

else:

glFrustum(

self.zoom\*self.view[0],

self.zoom\*self.view[1],

self.zoom\*self.view[2]/k,

self.zoom\*self.view[3]/k,

self.view[4], self.view[5]

)

# 设置视点

gluLookAt(

self.eye[0], self.eye[1], self.eye[2],

self.aim[0], self.aim[1], self.aim[2],

self.up[0], self.up[1], self.up[2]

)

# 设置模型视图

glMatrixMode(GL\_MODELVIEW)

glLoadIdentity()

# ---------------------------------------------------------------

glBegin(GL\_LINES) # 开始绘制线段（世界坐标系）

# 以红色绘制x轴

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.8, 0.0, 0.0) # 设置x轴顶点（x轴负方向）

glVertex3f(0.8, 0.0, 0.0) # 设置x轴顶点（x轴正方向）

# 以绿色绘制y轴

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.0, -0.8, 0.0) # 设置y轴顶点（y轴负方向）

glVertex3f(0.0, 0.8, 0.0) # 设置y轴顶点（y轴正方向）

# 以蓝色绘制z轴

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.0, -0.8) # 设置z轴顶点（z轴负方向）

glVertex3f(0.0, 0.0, 0.8) # 设置z轴顶点（z轴正方向）

glEnd() # 结束绘制线段

# ---------------------------------------------------------------

glBegin(GL\_TRIANGLES) # 开始绘制三角形（z轴负半区）

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.5, -0.366, -0.5) # 设置三角形顶点

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, 0.5, -0.5) # 设置三角形顶点

glEnd() # 结束绘制三角形

# ---------------------------------------------------------------

glBegin(GL\_TRIANGLES) # 开始绘制三角形（z轴正半区）

glColor4f(1.0, 0.0, 0.0, 1.0) # 设置当前颜色为红色不透明

glVertex3f(-0.5, 0.5, 0.5) # 设置三角形顶点

glColor4f(0.0, 1.0, 0.0, 1.0) # 设置当前颜色为绿色不透明

glVertex3f(0.5, 0.5, 0.5) # 设置三角形顶点

glColor4f(0.0, 0.0, 1.0, 1.0) # 设置当前颜色为蓝色不透明

glVertex3f(0.0, -0.366, 0.5) # 设置三角形顶点

glEnd() # 结束绘制三角形

WxGLScene 类的使用示例：

#-\*- coding: utf-8 -\*-

import wx

from scene import \*

APP\_TITLE = u'架起沟通 wxPython 和 pyOpenGL 的桥梁'

class mainFrame(wx.Frame):

"""程序主窗口类，继承自wx.Frame"""

def \_\_init\_\_(self):

"""构造函数"""

wx.Frame.\_\_init\_\_(self, None, -1, APP\_TITLE, style=wx.DEFAULT\_FRAME\_STYLE)

self.SetBackgroundColour(wx.Colour(224, 224, 224))

self.SetSize((800, 600))

self.Center()

self.scene = WxGLScene(self)

class mainApp(wx.App):

def OnInit(self):

self.SetAppName(APP\_TITLE)

self.Frame = mainFrame()

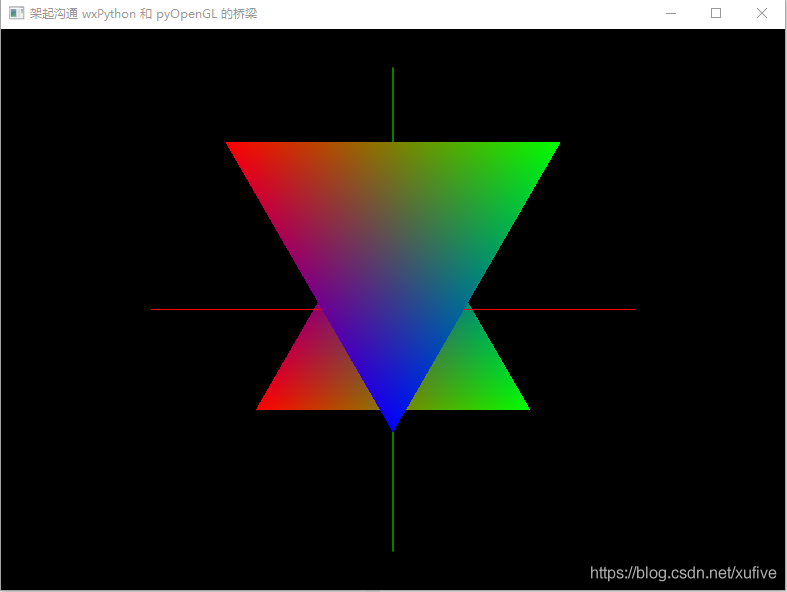
self.Frame.Show()

return True

if \_\_name\_\_ == "\_\_main\_\_":

app = mainApp()

app.MainLoop()

界面效果如下：  


**5. 场景、视区和模型**

OpenGL 允许用户使用 glViewport() 命令设置多个视口，这意味着我们可以在显示屏幕上分割出多个显示区域，这些区域可以相互重叠，在逻辑上是完全独立的。我们可以将 WxGLScene 称作场景（scene），由 glViewport() 命令创建的视口称为视区（region），拥有相同名字的三维部件定义为模型（model）。一个场景可以添加多个视区，一个视区可以创建多个模型。

以曲面模型为例，函数原型如下：

def drawSurface(self, name, v, c=None, t=None, texture=None, method='Q', mode=None, display=True, pick=False):

"""绘制曲面

name - 模型名

v - 顶点坐标集，numpy.ndarray类型，shape=(cols,3)

c - 顶点的颜色集，numpy.ndarray类型，shape=(3|4,)|(cols,3|4)

t - 顶点的纹理坐标集，numpy.ndarray类型，shape=(cols,2)

texture - 2D纹理对象

method - 绘制方法

'Q' - 四边形

0--3 4--7

| | | |

1--2 5--6

'T' - 三角形

0--2 3--5

\/ \/

1 4

'Q+' - 边靠边的连续四边形

0--2--4

| | |

1--3--5

'T+' - 边靠边的连续三角形

0--2--4

\/\_\/\_\

1 3 5

'F' - 扇形

'P' - 多边形

mode - 显示模式

None - 使用当前设置

'FCBC' - 前后面填充颜色FCBC

'FLBL' - 前后面显示线条FLBL

'FCBL' - 前面填充颜色，后面显示线条FCBL

'FLBC' - 前面显示线条，后面填充颜色FLBC

display - 是否显示

pick - 是否可以被拾取

"""

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

生成曲面模型顶点集、索引集的函数原型如下：

def \_createSurface(self, v, c, t):

"""生成曲面的顶点集、索引集、顶点数组类型

v - 顶点坐标集，numpy.ndarray类型，shape=(clos,3)

c - 顶点的颜色集，None或numpy.ndarray类型，shape=(3|4,)|(cols,3|4)

t - 顶点的纹理坐标集，None或numpy.ndarray类型，shape=(cols,2)

"""

创建 VBO 和 EBO 的方法如下：

def \_createVBO(self, vertices):

"""创建顶点缓冲区对象"""

id = uuid.uuid1().hex

buff = vbo.VBO(vertices)

self.buffers.update({id: buff})

return id

def \_createEBO(self, indices):

"""创建索引缓冲区对象"""

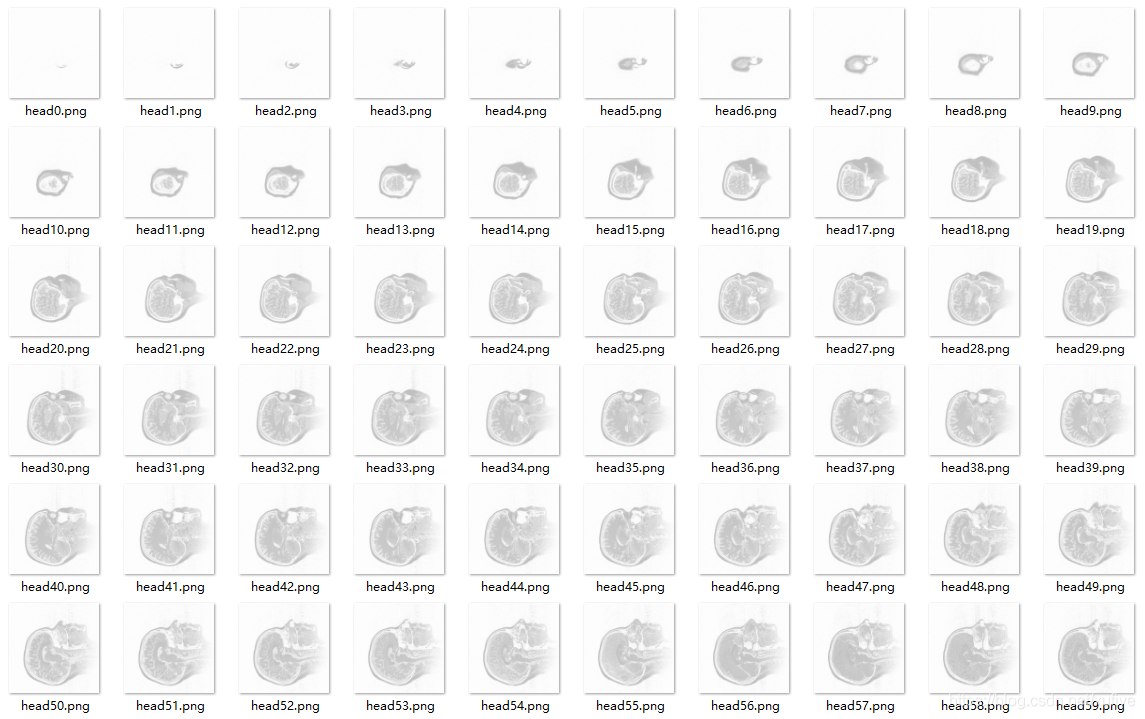
id = uuid.uuid1().hex

buff = vbo.VBO(indices, target=GL\_ELEMENT\_ARRAY\_BUFFER)

self.buffers.update({id: buff})

return id

**6. 三维重建的实例**

手头有 109 张头部 CT 的断层扫描图片，我打算用这些图片尝试头部的三维重建。基础工作之一，就是要把这些图片数据读出来，组织成一个三维的数据结构（实际上是四维的，因为每个像素有 RGBA 四个通道）。  
  
这个数据结构，自然是 numpy 的 ndarray 对象，读取图像文件我习惯使用 PIL。因此，需要导入两个模块：

import numpy as np

from PIL import Image

接下来，我用一行代码就把 109 张图片读到了一个 109x256x256x4 的 numpy 数组中，耗时 172 毫秒：

data = np.stack([np.array(Image.open('head%d.png'%i)) for i in range(109)], axis=0)

三维重建代码如下：

# -\*- coding: utf-8 -\*-

import numpy as np

from PIL import Image

import wx

import win32api

import sys, os

from wxgl.scene import \*

from wxgl.colormap import \*

FONT\_FILE = r"C:\Windows\Fonts\simfang.ttf"

APP\_TITLE = u'CT断层扫描三维重建工具'

APP\_ICON = 'res/head.ico'

class mainFrame(wx.Frame):

'''程序主窗口类，继承自wx.Frame'''

def \_\_init\_\_(self):

'''构造函数'''

wx.Frame.\_\_init\_\_(self, None, -1, APP\_TITLE, style=wx.DEFAULT\_FRAME\_STYLE)

self.SetBackgroundColour(wx.Colour(224, 224, 224))

self.SetSize((800, 600))

self.Center()

# 以下代码处理图标

if hasattr(sys, "frozen") and getattr(sys, "frozen") == "windows\_exe":

exeName = win32api.GetModuleFileName(win32api.GetModuleHandle(None))

icon = wx.Icon(exeName, wx.BITMAP\_TYPE\_ICO)

else :

icon = wx.Icon(APP\_ICON, wx.BITMAP\_TYPE\_ICO)

self.SetIcon(icon)

self.scene = WxGLScene(self, FONT\_FILE, bg=[1,1,1,1])

#self.scene.setView([-1,1,-1,1,2,500])

#self.scene.setPosture(elevation=30, azimuth=-45, save=True)

self.master = self.scene.addRegion((0,0,1,1))

# 读取109张头部CT的断层扫描图片

data = np.stack([np.array(Image.open('res/head%d.png'%i)) for i in range(109)], axis=0)

# 三维重建（本质上是体数据绘制）

self.master.drawVolume('volume', data/255.0, method='Q', smooth=False)

self.master.update()

class mainApp(wx.App):

def OnInit(self):

self.SetAppName(APP\_TITLE)

self.Frame = mainFrame()

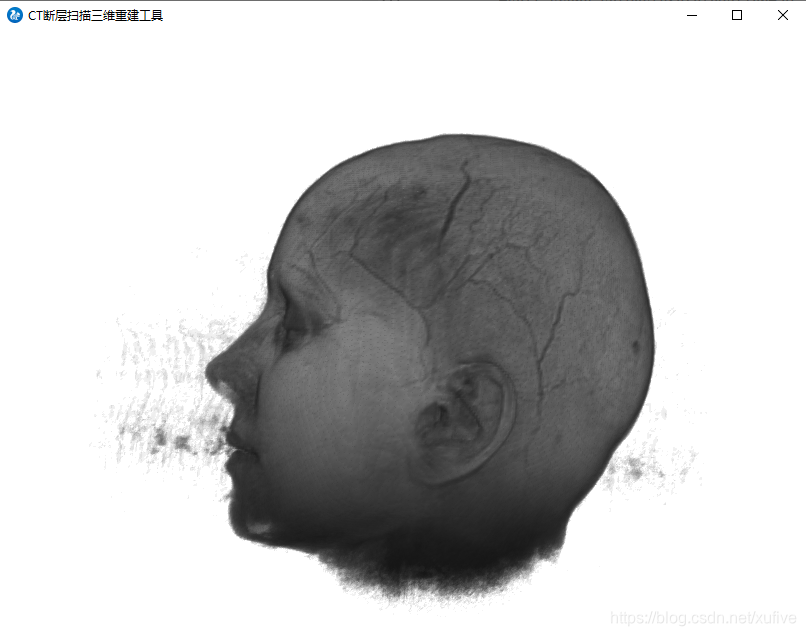
self.Frame.Show()

return True

if \_\_name\_\_ == "\_\_main\_\_":

app = mainApp()

app.MainLoop()

三维重建后的效果如下图：  


**7. 后记**

原本打算好好写一写纹理、拾取、体数据绘制的，结果写完第4章的时候，就已经感觉写不动了。这个题目实在太大，不是一篇一两万字的博文就可以说清楚的。如果不是因为有朋友在我的博客上留言说，想了解三维重构，本文也许会止于第4章。第5章简单展示了我的创作思路，只是为了讲解第6章的三维重建——事实上，也没有说明白。

行文至此，深为没有掰扯清楚关键问题而满怀愧疚。如果对这个话题感兴趣，请直接联系我吧：xufive@sdysit.com