Insight

由于才华有限，对C++的理解不够充分，历经尝试也未能将GUI、OpenGL、状态管理，三方面彻底剥离开，给大家提供一个简单的、明确的填空式开发体验。为了避免在抽象的途中半途而废，使用户在使用它的时候还不得不考虑背后的实现细节和抽象逻辑，不如就此将它以最单纯粗暴的形式交给大家。配以尽可能详细的文档，希望没有为你们添加额外的负担。

使用

这个项目依赖glew和wxWidget，安装它们在一个拥有健全包管理器的发行版上是颇为容易的：

```
sudo apt install make wx-common gcc g++ libwxbase3.0-dev
libwxgtk3.0-dev libglm-dev libglew-dev wx3.0-headers

make run
```

在成功运行之后，点击menu窗口中的Open键，选中一个自己喜欢的obj文件（比如demo/bunny.obj），它就会显示在menu窗口下方的列表里；同时Insight窗口中将会出现它的渲染效果。

可以在menu窗口中修改它的透明度、位置和缩放。在Insight窗口中使用WASDQE可以控制相机位置，使用鼠标左键和右键也可以对模型的位置和角度进行调整（如何使这种调整合理和顺手可能是有些复杂的）。

在把玩过这个项目之后，再来看看它的实现吧。

项目结构

该项目主要有两个子项目：一个是window，一个是obj_loader。其中window包括了创建窗口、绘图等部分，而obj_loader顾名思意就是读入并解析obj文件的部分。

window

window子项目应该是比较核心的一部分。在这个软件中使用了wxWidget来提供GUI组件，选择它的理由如下：

1. 相比GTK，它能更容易地跨平台、并且复杂度和侵入性远低于GTK。
2. wxWidget对c++支持友好。而GTK的native binding仅有C、js、vala，c++支持非常不友好。
3. 同样的c++跨平台图形库QT相比之下又太过复杂，会带来不小的学习负担。
4. 相比glfw这样专门为了opengl设计的GUI库，它提供了更多的功能：比如颜色选取框、文件选取框等

但同样地，它也有一些不足之处：

1. 缺乏文档和社区支持。一些地方可能需要通过查阅源码来猜测。
2. 排版方式复杂、丑陋、低级

不过幸运的是这些不足之处我都尽力隐藏了起来 :).

程序的入口在window.cpp中，它会负责创建GUI组件并把它们摆在合适的位置。在点击Open按钮之后`Insight::on_open_button_click(wxCommandEvent &evt)`被触发，它根据用户选择的文件构造了一个ObjLoader，并调用它的load函数，得到了一个`ObjLoader::Obj`的实例，最终将这个实例中的节点、法向量提取出来，存入了objs中并触发`GLComponent`的一次重绘。

在`GLComponent`被触发重绘时，它的render函数会被调用。这是渲染发生的核心部分。在讲解它（残存的部分）之前，我需要先提供一点关于OpenGL的知识：

OpenGL

不如考虑考虑在没有 OpenGL 存在的情况下，拿到了一堆三维三角形，我们会想要怎样渲染吧。

1. 依照相机的位置、物体的位置、物体的旋转，来对三角形的座标做一些调整
2. 将三角形依照三维前后关系排好顺序：把看不到的部分隐藏掉。
3. 给每个三角形适当的颜色。
4. 对它们作投影，投射到屏幕这个二维平面上并显示出来。

理想的情况下确实是这样的，非常简单。我们拆分一下每一步：

1. 根据相机的位置和物体的位置对物体的座标进行平移变换，根据物体的旋转对物体进行正交变换即可。
2. 如何确定哪些三角形看不见？部分看得见部分看不见的三角形怎么办？由于最终会投射到二维平面上，我们不妨在最后一步记录下某个像素点对应的深度是多少。此后再次想要在这个像素点上绘图的时候，如果太深（被挡住），就不画了。这种做法被称为Z-Buffer，即使用一个Buffer记录屏幕上每个像素点的深度（z值）。
3. 如何确定一个三角形的颜色？"确定合适的颜色"这句话囊括了渲染90%的问题。这不是这个项目关注的部分，于是我们用最简单的方案：将光线方向与法线方向点乘并限制在[0.5, 1.0]中，再与物体本身的颜色相乘。`clamp(dot(normalize(light_direction), normalize(normal)), 0.5, 1.0) * col`
4. 这一步相对来说也是容易的：无论你选择平行投影（Parallel projection）还是透射投影（Perspective projection），都只是计算矩阵乘法的事情。

经过这些步骤，我们终于生成了一个二维的颜色阵列。每一个x和y座标，我们都能给出相应格点上的颜色。接下来只要把这些颜色输出给屏幕就好了。

不过稍加考虑（你也可以写一个程序验证一下试试），对每个三角形进行这样的计算是非常耗时的。而实时渲染却对延迟、计算性能有非常高的要求：希望你能在30 ms内计算完毕，这简直是痴人说梦。很显然，再给五年CPU也发展不出这样的计算速度。那么为什么人们在九十年代就能玩上 3D 游戏了呢？

这是因为有专门的硬件在做这些事情：为了专门的算法设计专门的硬件已经是计算机界的传统艺能了。考虑到以上算法的特性：可以对每个三角形分别计算、充斥着大量的线性计算（矩阵乘、向量加）... 能够高度并行计算的显卡被制造出来。但显卡作为一张硬件设备通过 PCIe 连接到你的电脑之后，你的软件要如何与它交互呢？OpenGL 作为一个通用的、开放的接口，起到了这样的作用，你通过 OpenGL 告诉你的显卡你想要算什么、怎么算，它帮你计算完成并直接输出在屏幕上。

我们现在看看余下的代码中与 gl 相关的部分：从init开始

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

这两句告诉 OpenGL 打开深度检测。它便会自己使用Z Buffer的算法来进行消隐。

```
vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, &vertex_shader_text, nullptr);
glCompileShader(vertex_shader);

fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1, &fragment_shader_text, nullptr);
glCompileShader(fragment_shader);

program = glCreateProgram();
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);
glLinkProgram(program);
```

这一段分别创建了Vertex Shader和Fragment Shader，并将它们编译、传给显卡。糟糕，突然冒出了两个从未出现过的词语：Vertex Shader和Fragment Shader。简单来说，它们就是你用来定义“怎么算”的部分。在Vertex Shader中我们需要给出一个点的位置，在Fragment Shader中我们需要给出一个点的颜色。查看`vertex_shader_text`和`fragment_shader_text`这两个变量，我们依次讲解其中的内容：

```
#version 110
uniform mat4 mvp;
uniform vec4 col;
uniform vec3 cameraPosition;

attribute vec3 pos;
attribute vec3 normal;

varying vec4 color;

void main(){
    vec3 light_direction = vec3(3.0, 3.0, 3.0);

    gl_Position = mvp * vec4(pos, 1.0);
```

```
    color =
vec4(clamp(dot(normalize(light_direction),normalize(normal)), 0.5,
1.0) * col.xyz, col.w);
}
```

这是我们正使用的Vertex Shader，与C非常相似。上方定义了一些矩阵、向量作为变量，它们有许多不同的修饰符：

- uniform，这是可以从外界设置的一个一个的变量
- attribute，这是从外界以列表的形式传入的变量
- varying，这是在渲染的过程中设置的，在不同阶段的Shader之间共享的变量

varying和其它两者的区别很大（可以看到在main中我们设置了color，马上在Fragment Shader中我们将用到它）。但许多人无法理解uniform变量和attribute变量的差别，举一个可能形像的例子：

考虑一个流水线，从一端不断涌来各式各样零件，需要工人加工。有的零件需要用两千目砂纸加工，有的零件需要用五百目砂纸加工。低效的做法是：为每个零件购买配套的砂纸，在把零件传给工人的时候也把相应的砂纸传递给它们。高效的做法是：工头把一张两千目砂纸给工人，然后传输数百个需要用两千目砂纸的零件给他们。工头再把一张五百目砂纸给工人，再把需要用五百目砂纸加工的零件传给他们。uniform变量就是砂纸的目数、attribute变量就是零件。在我们的情景中，由于不同物体的位置、朝向、颜色可能不同，但同一个物体中的三角形们位置、朝向、颜色是相同的。所以那些三角形使用attribute变量传入，而位置、朝向、颜色使用uniform变量传入。

如果还不理解的话，就继续看下去，我们会讲到不同变量的传入方式，那时就应该明白了。

main中还出现了不熟悉的东西：gl_Position，我没有定义这个变量，它自然就有。设置gl_Position表示设置当前这个点的位置，正如此前所说，Vertex Shader的目标便是计算出它来。

再来看Fragment Shader：

```
#version 110

varying vec4 color;

void main() {
    gl_FragColor = color;
}
```

它非常的简单。由于我们在Vertex Shader中就已经计算好了当前点的颜色，并赋予了color这个varying变量，这里只需要把varying变量color传递给gl_FragColor就好了。与gl_Position类似，gl_FragColor表示这个点的颜色。

在看完Shader之后我们接着向下看

```
mvp_location = glGetUniformLocation(program, "mvp");
col_location = glGetUniformLocation(program, "col");
camera_position_location =
glGetUniformLocation(program,"cameraPosition");
```

在这里我们使用`glGetUniformLocation`函数获取了那些uniform变量对应的地址，此后我们可以使用`glUniformMatrix4fv`、`glUniform4fv`、`glUniform3fv`等函数来为这些uniform变量赋值。

接下来是为attribute变量赋值的部分：

我们先如同为Uniform变量赋值一样，先获取它们的地址

```
pos_location = glGetAttribLocation(program, "pos");
normal_location = glGetAttribLocation(program, "normal");
```

由于attribute变量是极大批量地传入的，我们需要一个数组/缓冲区来储存它们

```
glGenBuffers(1, &vertex_buffer);
glGenBuffers(1, &normal_buffer);
```

然后再将缓冲区与attribute变量绑定起来，告诉OpenGL当你想要得到新的attribute变量的时候，就从这个缓冲区中读吧！以attribute变量pos为例：

```
glEnableVertexAttribArray(pos_location);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glVertexAttribPointer(pos_location, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(GLfloat), nullptr);
```

注：其中真正起到绑定作用的，不是名字有Bind的那个函数，而是它下面的glVertexAttribPointer。glBindBuffer起到的作用只是告诉OpenGL，之后的操作都是对vertex_buffer这个buffer的操作。

这下当OpenGL感到空虚，需要为attribute变量pos赋新值的时候，就会从vertex_buffer中读。读多少呢？读三个Float。

准备工作已经完毕，接下来我们看render函数中剩余的部分：

```
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) *
obj.vertices.size(), &obj.vertices[0], GL_STATIC_DRAW);
```

注：如果在上一段中没有理解glBindBuffer的作用，这里它又出现了，该明白了吧。

这两句话将obj.vertices这个数组载入到vertex_buffer中。同样的

```
glBindBuffer(GL_ARRAY_BUFFER, normal_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) *
obj.normals.size(), &obj.normals[0], GL_STATIC_DRAW);
```

这两句话将obj.normals这个数组载入到normal_buffer中。

接下来是一些计算的代码：计算出对三角形进行仿射变换的矩阵（命名为mvp），这一部分需要你们自行完成。

然后把这些东西作为uniform变量传入OpenGL程序中：

```
glUniformMatrix4fv(mvp_location, 1, GL_FALSE, glm::value_ptr(mvp));
glUniform4fv(col_location, 1, glm::value_ptr(obj.color));
glUniform3fv(camera_position_location, 1,
glm::value_ptr(camera_position));
```

这里使用了glm，当然你也可以不用——使用单纯的float[]是更简单的选择。

最后便是绘图了！

```
glDrawArrays(GL_TRIANGLES, 0, obj.vertices.size());
```

这个函数告诉OpenGL咱们要一个一个三角形地画，画多少下。

于是计算开始了——使用已经传入的uniform变量，从buffer中读取attribute变量，进行计算得出`gl_Position`和`gl_FragColor`，最终绘制在屏幕上，就是这么简单。在专门为这种计算设计过的硬件上，这些计算运行地飞快。

透明

透明看上去是个小问题：我们的gl_FragColor本身就是四维向量——这意味着它自带alpha通道，能够表示透明度。可惜的是，它是个远远比这复杂的问题：这也是游戏中透明物件不常见的原因。

第一个问题：颜色

但如果考虑考虑它的实现方式，又会觉得它没那么简单——在绘制一个三角形的时候如何知道它的背后有哪些三角形、它们是什么颜色呢？事实上，是不知道的。由于点一个一个的被传入Shader，在计算一个点、一个面片、一个像素的颜色的时候，我无从得知这个像素与多少个三角形有关，无从得知那些三角形是什么颜色、透明度多少。知道的有什么呢？只有当前点的颜色（包括透明度）和已经绘制的像素的颜色。

如何通过这些来组成新的颜色呢？最想当然的做法是这样：

$$A_s \times RGB_s + (1 - A_s) \times RGB_d$$

其中RGB_d为当前像素点已经绘制的颜色，RGB_s为当前像素点想要绘制的颜色。A_s为想要绘制的颜色的透明度。这样看似效果很好，但会带来严重的问题：考虑一个黑色的背景上，先绘制上0.5透明度的绿色，再绘制上0.5透明度的红色；或是先绘制上0.5透明度的红色，再绘制上0.5透明度的绿色，它们的颜色是不同的。我无法断言哪种是正确、哪种是错误，但为了保证结果的一致，将三角形由远及近排好顺序，按照这种固定的顺序渲染即可。

当然，你也可以选用其他的和顺序无关的混合方式：

$$RGB_d = RGB_s \cdot RGB_d$$

虽然渲染的结果有些奇怪，但它至少与顺序是无关的。

如何在OpenGL中定制自己的混合方式呢？使用以下几行

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

第一行启用GL_BLEND（即混合），第二行指定混合函数为“加性”的（即形如$\alpha_1 \times RGB_s + \alpha_2 \times RGB_d$）。第三行指定$\alpha_1, \alpha_2$这两个参数，看名字就知道这时选用的混合函数是

$$A_s \times RGB_s + (1 - A_s) \times RGB_d$$

如果你想用

$$RGB_d = RGB_s \cdot RGB_d$$

那么只需要使用`glBlendFunc(GL_ZERO, GL_SRC_COLOR);`就好了，即指定第一个参数为0，第二个参数为RGB_s.

第二个问题：消隐

由于Z Buffer算法的存在，更深的部分可能会被挡住，从而被剪裁掉——哪怕挡住它的是个透明物件。幸运的是OpenGL提供了`glDepthMask(GL_FALSE)`函数，它能够暂时地允许/禁止写入Z Buffer。显然，如果一个三角形被绘制时禁止写入Z Buffer了，那么它将无法挡住它身后的三角形。

obj_loader

这个项目用于读取、解析obj文件。obj文件的格式在维基百科上有完整的解释，我们只需要支持一个obj文件的子集：不需要考虑vt、vp、纹理和没有提供法向量的face。

现在`ObjLoader`的load函数与`Obj`的get_normals函数和get_vertices函数需要自行完成。

我需要做什么

上面的文档中已经零散提及了你们需要做什么，下面列出一个整理后的清单：

1. 完成obj_loader的load函数、get_normals函数和get_vertices函数。（这只是简单的字符串处理）
2. 完成`gl_component.hpp`中的mouse_moved、mouse_wheel、left_down、left_released、right_down、right_released、key_pressed、key_released函数。你希望在用户触发这些事件的时候，程序做什么事情呢？
3. 完成`GLComponent::render`函数。它几乎只剩下了计算仿射变换矩阵的这一部分。

任何可能需要你完成的部分都使用/**/进行了注释。这是一个颇为轻松的作业，不是吗？

其他操作系统

由于完全的使用了跨平台的组件，有心人可以不用修改代码就将这个项目编译至各个不同的平台上，包括Windows、Mac OS、Linux的各种发行版。当然，项目的编译选项可能需要做一些修改。

多余的问题

main函数在哪里

在`window.cpp`文件中有一行`IMPLEMENT_APP(Insight)`，其中`IMPLEMENT_APP`是一个宏，这个宏的展开中蕴含了一个main函数。听上去确实很黑魔法，但这好像是C++ GUI框架的常规操作。

OpenGL真有趣，我还想学

很高兴在结束一个OpenGL的项目之后你还会这样想。市面上OpenGL的教程/教材非常多，其中有些简单的：OpenGL Tutorial 能让你对OpenGL有个全面的大概的认识，不再是这样的填空式开发。

如果你对更高级的实时渲染技术感兴趣，GPU Gems是很推荐的一本读物，里面的小topic可以分开阅读，每个小部分都很有趣。

如果你觉得实时渲染的效果满足不了你，想要探索一下电影级的渲染技术。那么我推荐的一本离线渲染的入门书籍是：Physically Based Rendering，离线渲染能够达到的效果是实时渲染远不能比的！

找到一个全局的统一的渲染模型实在是太困难了；但局部的单一材质的渲染模型又没有什么意思（渲染沾水的动物皮毛和汽车烤漆裂纹）。其中的数学知识也不怎么新鲜、深刻，应该是满足不了各位数学大仙的。

显卡能被用于通用的计算吗？

你应该已经注意到我们向显卡中传入了类C编写的程序（那门语言称作glsl），它还能承载更加复杂的逻辑吗？当然可以！现代显卡的功能过于强大，仅被用于显示计算有些浪费。当你的算法、程序具有与显示计算类似的特征的时候（可以充分并行、不用共享全局状态等），可以编写你的程序让它运行在显卡上。如果

你想要这样做，那么有很多工具可以使用：Nvidia提供了CUDA，LLVM有NVPTX后端，有适配多种硬件的opencl，专注于图像信息处理的halide...

shared_ptr是什么

我猜会有人问到这个问题，你可以将shared_ptr理解为一种更高级的指针。它高级在哪里呢？它维护了一个计数器，在调用它的一些构造器的时候（比如拷贝构造），会将这个计数器加一，在析构的时候会将这个计数器减一。在数量归零的时候意味着再也没有任何指针指向这个对象，那么它内部的对象将会被析构。

这是一种更加安全地使用堆内存的方式：可以有效的减少内存泄漏。但可惜的是它不是万能的，比如两个shared_ptr互相指的时候（比如双向链表中的节点），它的计数器至少为1，再也不会被析构了。

如何管理动态分配的内存是一个非常有趣的话题。内存泄漏（分配而不释放）是常见问题之一：最轻松的答案是引入一个垃圾回收器（许多编程语言自己携带一个，比如Java、Go和大部分脚本语言），它会在恰当的时机开始检测你再也无法访问到的内存（所有的变量间引用形成一个有向图，从栈上的变量出发对这个图作搜索，找不到的变量就该被回收了）。更危险的问题是悬垂指针、空指针、不安全的多线程访问，Rust编程语言给了一个很好的答案：使用Substructural type system和所有权机制，避免了这两个问题。

当然，解决问题的终极方案是一个足够可靠、稳定的大脑（很遗憾，我没有）。

怎样更好/熟练地编程

这是个过于generic的问题。编程是一项实践技术，多读多写就完事了。但日常生活中好像没有什么好写的东西？其实不是这样的：觉得浏览器太慢了吗？掌握C++的你能够查看/改写Firefox的代码，看看能不能让它跑得更快。觉得操作系统太慢了吗？打开kernel.org查看/修改Linux内核代码，让它更符合你的需求。其中会不断地涉及到各个领域的知识，用作消遣还是挺有趣味的。

代码风格问题

为什么没有注释

大部分逻辑代码都应当是简洁清晰的：哪怕没有注释也应该能让人看懂。我一向这样要求自己，如果我的能力不足没有达到这样的要求，希望你们能够指出，一定改正！

.hpp是什么，为什么不是.h和.cpp

.h和.cpp是C时代的遗留物，对于一门现代的编程语言来说是历史包袱。基于链接与字符串拼接技术的模块化是完全不够用的。但更加现代的模块化方式（modules）在C++20中才引入。使用hpp是一种向过去告别但新时代还未来的镇痛吧。

为何混用驼峰和下划线

因为C++没有官方的统一的风格。所以我可以按着自己的性子来：类名用首字母大写的驼峰、变量名和函数名用下划线。

你写的一点也不C++

接受这项指责，C++并不是我的常用语言，我对面向对象编程的理解也并不深刻。我的常用编程语言是JavaScript、Rust和Go，比C++简单太多了（但各有各的好处）。