

SE-344 Assignment 1 Report

META-INF

- 姓名：于喜千
- 学号：517030910168
- 任务：Assignment #1

具体工作

第一部分：搭建 OpenGL 编程环境

参考文档

- [OpenGL Official](#)
- [GLFW Official](#)
- [LearnOpenGL Manual](#)

环境设定

- Windows 10 x64 LTSC 1809 (17763.737)
- Visual Studio 2019 Community (16.2.4)
- GLFW (3.3)

操作步骤

1. 安装 Visual Studio 2019 及其附带的 C++ 工具链；
2. 安装 Windows 版 CMake (GUI) ；
3. 下载 glfw-3.3 Windows 版源代码，在 Configure 中配置所使用的 Visual Studio 版本（这里使用的版本号是 16，即 2019 版本），并点击 Generate 生成 Visual Studio 解决方案；

生成日志参见 `/ass1/logs/cmake.logs` 。

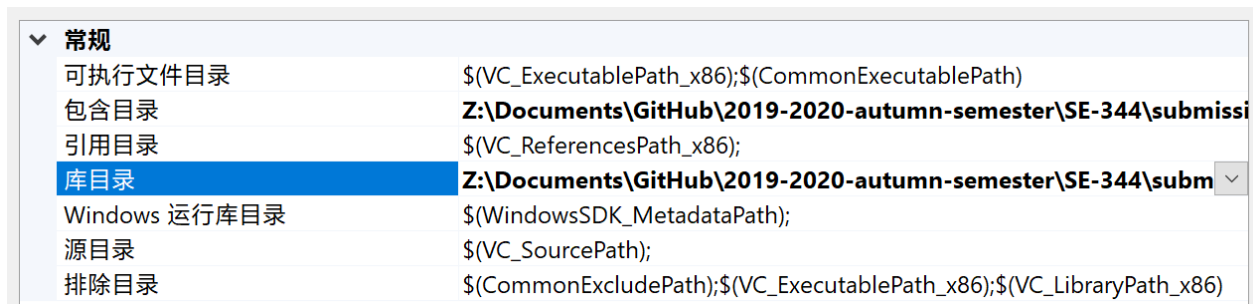
4. 使用 Visual Studio 开启 `/build/GLFW.sln` 解决方案文件，按下 `Ctrl + Shift + B` 组合键来生成解决方案。

编译日志参见 `/ass1/logs/vs.logs` 。

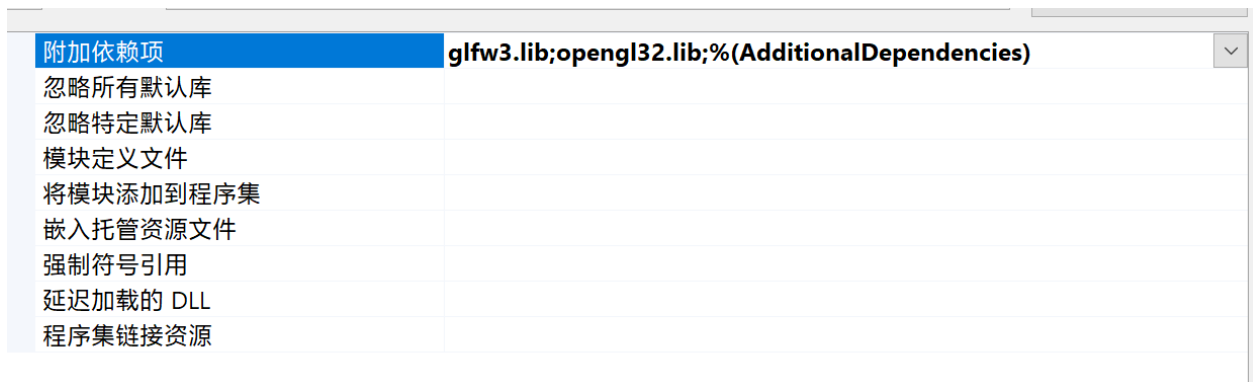
- 在 `/src/Debug` 目录中发现刚刚编译完成的 `glfw3.lib` 二进制文件，将其拷贝到方便操作的目录下待用。同时将解压出来的 `/glfw/include` 文件夹中的内容也拷贝到该目录待用。

此二进制文件参见 `/ass1/bin/glfw3.lib`。

- 再次打开 Visual Studio，新建一个 Visual C++ 空项目（Empty Project），将其存放在 `/ass1/src/` 目录下，命名为 `ImA_Teapot`。
- 在解决方案资源管理器中右击 `ImA_Teapot` 项目，在弹出的右键菜单中点按「属性」，在弹出窗口中依次寻找「配置属性」、「VC++ 目录」，在「包含目录」和「库目录」两个字段中加入 `glfw3.lib` 所在的绝对路径，如下图所示，并点按「应用」来保存更改。



- 同样在这个窗口，依次寻找并点按「配置属性」「链接器」「输入」项，在右侧字段列表中寻找「附加依赖」项字段，将我们刚刚编译的 `glfw3.lib` 和 OpenGL 库文件 `opengl32.lib` 加入该字段中来，如下图所示，并点按「应用」来保存更改。



- 开启 [GLAD 在线服务](#)，选择刚刚配置的 glfw 库版本（3.3），并将 Profile 项设定为 Core（仅核心），其余部分按默认配置，并点按 Generate 生成。随后下载生成的 `glad.zip`，将其解压后放置在项目目录附近。

GLAD 所做的工作是在众多 OpenGL 驱动版本的函数上层提供一个抽象，避免我们处理许多跨平台的问题（如，获取函数指针等）。

生成的 GLAD 库代码参见 `/ass1/lib/glad` 文件夹。

- 将 `/glad/include` 目录下的两个文件夹复制到第 7 步中制定的文件夹中，并将 `glad/src` 目录下的 `glad.c` 文件加入解决方案中。
- 新建一个 `main.cpp` 文件，尝试加入以下代码：

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
```

点按 Debug 运行发现不报错，则说明配置完成。

第二部分：测试 OpenGL 环境

操作步骤

12. 在 `main.cpp` 中填入以下代码，将配置管理器的下拉框设定为 `Debug`，`x64`，尝试生成并运行项目。

```

#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>

void framebuffer_size_callback( GLFWwindow* window, int width, int height );

int main() {
    glfwInit();
    // 初始化 glfw 库

    glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 3 );
    glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 3 );
    // 声明我们使用的 glfw 语义化版本: v3.3

    glfwWindowHint( GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE );
    // 设置 OpenGL 配置文件; 这里使用精简的 Core Profile。

    // glfwWindowHint( GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE );
    // macOS 所需要进行的 Workaround。
    // 由于本次 Assignment 使用 Windows 环境, 故无需这一行。

    // 创建 glfw 窗口
    GLFWwindow* window = glfwCreateWindow( 800, 600, "I'm A Teapot", NULL,
    NULL );
    if ( window == NULL ) {
        // 检查创建窗口是否成功
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }

    // 设定图形上下文为 window
    glfwMakeContextCurrent( window );

    if ( !gladLoadGLLoader( ( GLADloadproc )glfwGetProcAddress ) ) {
        // 利用 GLAD 来重建函数指针位置
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // 告诉 OpenGL 渲染窗口的大小
    glViewport( 0, 0, 800, 600 );

    glfwSetFramebufferSizeCallback( window, framebuffer_size_callback );

    while ( !glfwWindowShouldClose( window ) ) {
        glfwSwapBuffers( window );
        glfwPollEvents();
    }

    // 循环终结时, 同时终结 glfwTerminate。

```

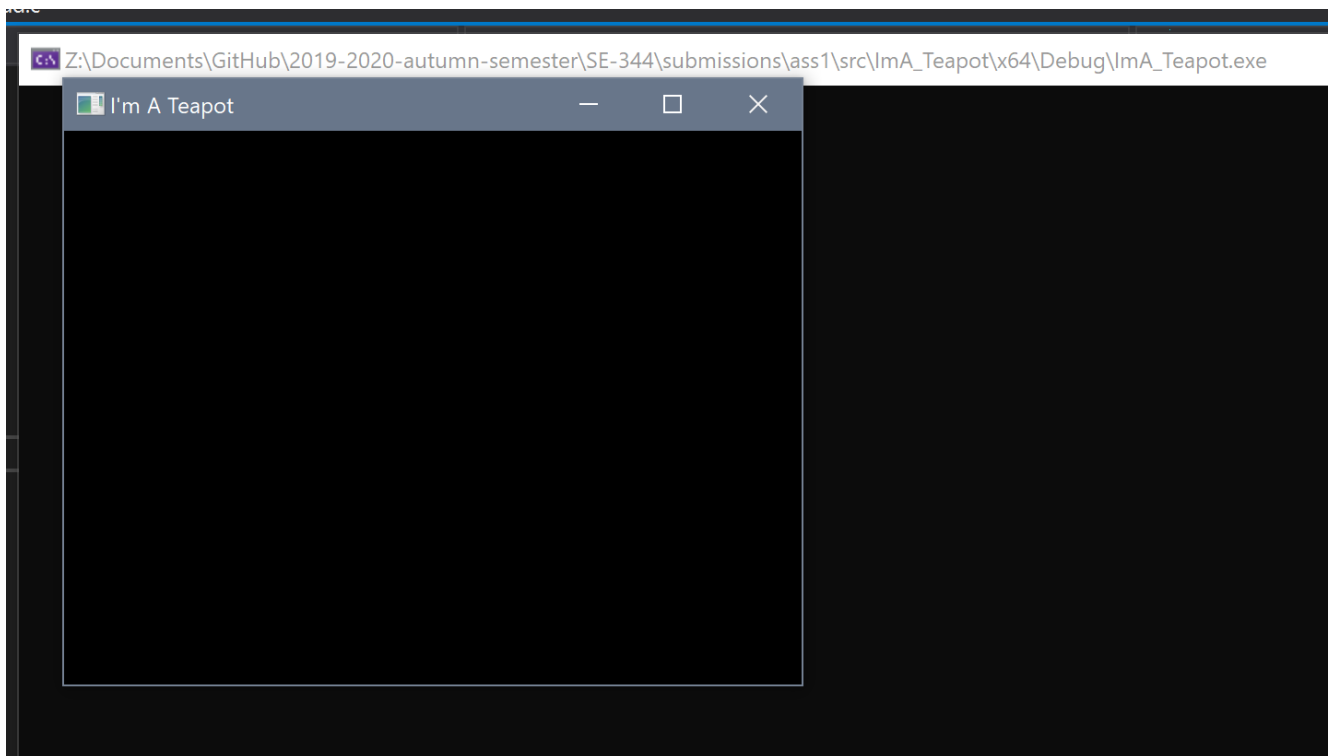
```
    glfwTerminate();
    return 0;
}

// 回调函数，在窗口大小改变时通知 OpenGL
void framebuffer_size_callback( GLFWwindow* window, int width, int height )
{
    glViewport( 0, 0, width, height );
}
```

⚠ 此段代码系 `glfw` 官方文档提供的环境测试代码，以 `CC BY-SA 4.0` 授权。注释为自行补充。

13. 预期结果：出现了一个纯黑的、`800×600` 大小的窗口。

屏幕截图



第三部分：设置背景色

考虑到我们之后会需要动态改变窗口背景色，并且调节窗口大小需要背景被重新渲染，我们不应该将窗口背景颜色硬编码在 `OpenGL` 启动之前，而应该将其放在渲染循环中（就是上面那个 `while(!glfwShouldClose(window))` 循环里）。因此我们进行下列操作：

操作步骤

14. 在渲染循环代码中加入下列代码：

```
// 渲染循环
while(!glfwWindowShouldClose(window))
{
    glClearColor(0.7f, 0.7f, 1.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // 检查并调用事件，交换缓冲
    glfwPollEvents();
    glfwSwapBuffers(window);
}
```

其中 `glClearColor` 函数用来设置我们需要使用的笔刷（Brush）颜色，其参数列表为

```
void glClearColor(GLclampf red,
                  GLclampf green,
                  GLclampf blue,
                  GLclampf alpha);
```

即我们将笔刷颜色设定为了 `alpha = 0.0f`（完全透明）的淡紫色。但此时尚未将其刷上背景，我们需要调用 `glClear(GL_COLOR_BUFFER_BIT)` 函数将其上色。

Q：为什么这里设置 `alpha` 的值无论是 `0.0f` 还是 `1.0f`，对最终上色的结果没有影响？

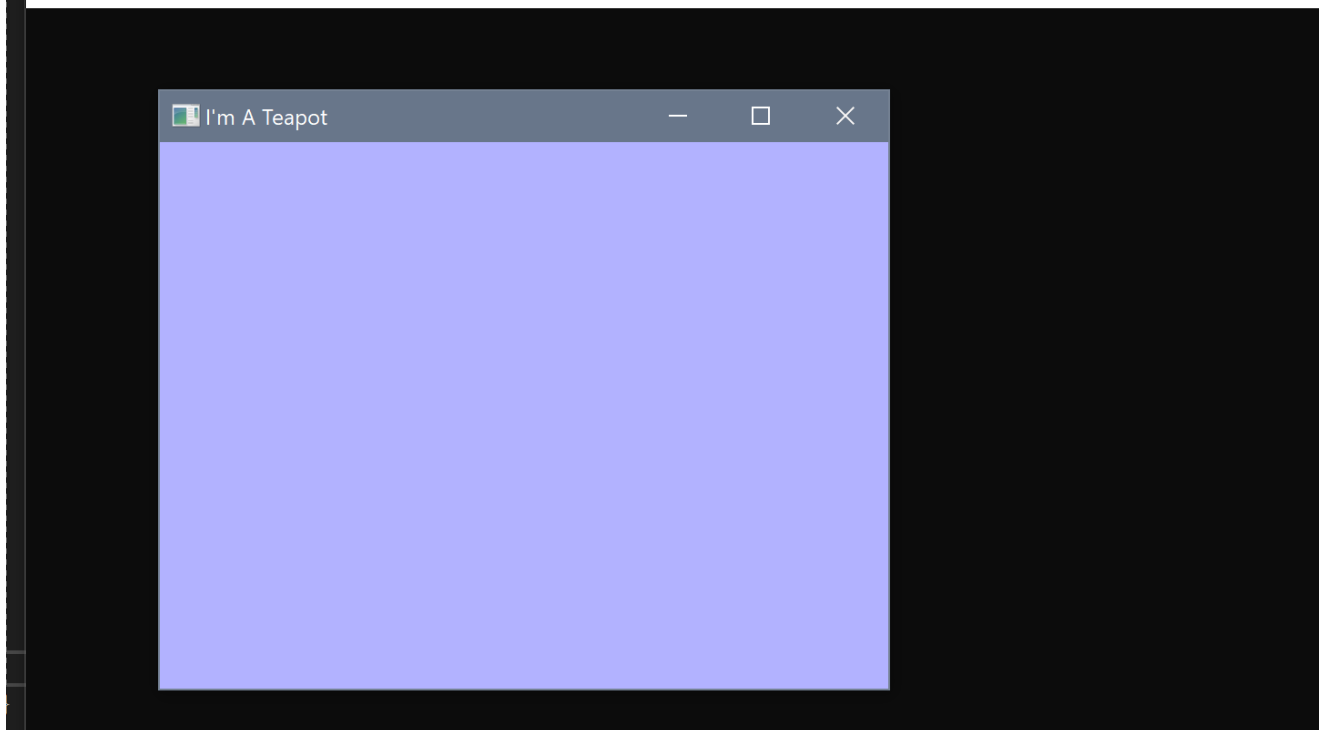
A：见第十一部分的讨论。

事实上，这里 `glClear` 所做的事情是「清除颜色缓冲区」，即将缓冲区中原有的颜色值清空，并用 `glClearColor` 所设置的颜色来填充整个屏幕。

通过设置 `glClear` 的参数，我们还可以选择清空深度信息（将参数设定为 `GL_DEPTH_BUFFER_BIT`）或清除贴图信息（将参数设定为 `GL_STENCIL_BUFFER_BIT`）。这是后话。

现在我们可以看到背景色已经被成功设置，而且在调节窗口大小时，颜色会被重新渲染以充满整个窗口。

屏幕截图



第四部分：设定双缓冲

概念：什么是双缓冲？

最直观朴素的渲染机制可能是：每过一个 `Tick` 就计算一次屏幕上所有像素点的信息，并将其填充到显存中以供显卡显示。

但这样就存在一个问题：显示驱动取走数据是一次拿走对应一整个屏幕的像素点信息，但是我们的程序渲染像素点却是一点一点渲染的。这就导致如果我们仅仅按照上面的思路制作渲染器的话，显卡驱动可能会在取走一帧画面时发现这帧画面由一半填充好的新像素和遗留的一半旧像素组合在一起，这就导致了画面锯齿，严重的话可能导致画面撕裂。

因此我们在将像素点放入显存中之前，需要先进行一个缓存，待一整幅画面被渲染完毕之后，再将其一并送入缓存才可以。

那么我们就有了另一个思路：在渲染过程中不要将其送入显存，而是将其放在另一块缓冲区中，等所有像素点都渲染完成之后，再将其一并送入显存。

听起来挺不错的，但事实上这就导致每一个像素点都要被移动两次：第一次是渲染完成后立即将其放入的那个缓冲区，第二次是我们将其挪到显卡实际读取的缓冲区。这就带来了严重的时间延迟。

因此我们引出了另一样技术：渲染器和上面一样维护两块缓冲区，但却不做任何缓冲区的拷贝操作。反过来，让显卡驱动来决定从哪一块缓冲区来刷新屏幕；这样在完成一块缓冲区的渲染操作之后，渲染器则无需拷贝缓冲区，只需要通知显卡驱动「切换舞台」，并转向另一块缓冲区进行下一次渲染即可。

这就是双缓冲！

实践

那么我们来给我们的程序引入双缓冲吧！

GLFW 默认使用双缓冲区。

事实上我们上面的代码已经使用了双缓冲技术了。观察 `glfwSwapBuffers` 函数的名字就大概能猜出来，它的作用就是交换前景（正在被显卡读取的）和背景（正在往里塞数据的）两个缓冲区。

第五部分：画茶壶

Huh I'm a Teapot

第三方开源库使用说明

由于肖老师在布置作业时临时改动了 Assignment 内容，要求画一个茶壶来代替原题目中要求的球。作为 GLUT 的替代品，GLFW 本身并没有预设「画茶壶」这项功能。

又因为 GLUT 已经死透了，因此将选择使用 `FreeGLUT + glew` 的组合来代替上面的 `glfw + glew` 组合。

操作步骤

15. 按照第一部分中编译 `glfw` 同样的办法编译 `FreeGLUT` 和 `glew`，并放置在上述位置。

动态链接库文件参见 `ass1/dll/` 目录。

⚠ 还需要将编译得到的 `freeglutd.dll` 和 `glew32.dll` 放置在 `%SystemRoot%/Windows/System32` 和 `%SystemRoot%/Windows/SysWOW64` 目录下。

16. 现在我们需要用 `FreeGLUT + glew` 重写上面的代码。首先，所有的 `glut` 程序都需要首先初始化这个库。第一行会把 `main` 函数的两个参数传入，是标准的初始化方法。

```
glutInit(&argc, argv);
```

17. 随后我们初始化 `glut` 的显示设置：

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

这几个掩码的意思分别是：

宏名	意义
GLUT_RGBA	将窗口设定为 RGBA 彩色模式。这是默认情况下的采用值。
GLUT_DOUBLE	使用双缓冲机制（见上）。
GLUT_DEPTH	使用深度缓冲机制。

Q：什么叫做深度缓冲机制呢？

A：深度缓冲指的是在生成每一个像素的时候，都把它深度信息存储在一个缓冲区之中；如果另一个物体也被渲染到同一个像素处，则通过深度缓冲区对二者的深度进行比较，并消隐那个被遮挡的物体。

如果这个 OpenGL 程序需要处理多个物体间的遮挡关系，那么大概率会需要将 `GLUT_DEPTH` 掩码设定上。

18. 接下来我们开启一个新窗口：通过下列两个函数调用决定窗口的尺寸以及标题文字。

```
glutInitWindowSize(windowWidth, windowHeight);  
// windowHeight 和 windowHeight 是两个静态变量，用于维护窗口当前大小  
  
glutCreateWindow("I'm a Teapot");  
// 执行此步骤之后，窗口被正式创建
```

19. 接下来我们设定背景颜色。此处和第 14 步中的操作完全一样。

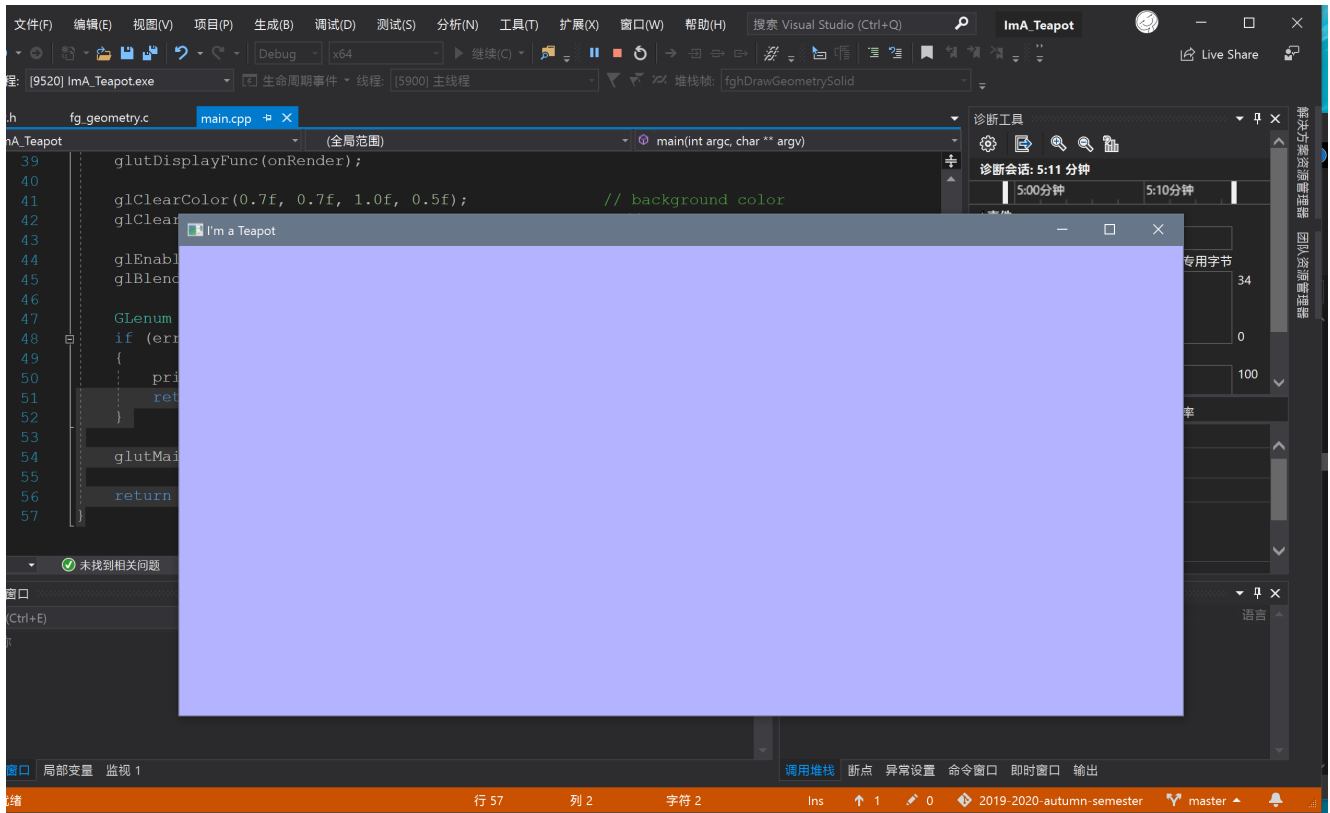
```
glClearColor(0.7f, 0.7f, 1.0f, 1.0f);
```

20. 最后我们需要调用函数

```
glutMainLoop();
```

来开始 GLUT 的循环。

21. 现在我们回到 Windows。由于 Windows 平台下背景透明功能不起作用，我们现在能看到的窗口如图所示。



21. 由于题目要求我们需要动态进行渲染生成，因此我们注册了两个回调函数：`onWindowResized` 和 `onRender`。其中窗口被重新拉伸之后将调用函数 `onWindowResized`，当需要重新渲染下一帧画面时调用 `onRender`。

```
static int windowHeight = 1280, windowHeight = 720;

void onWindowResized(int w, int h)
{
    // 更新窗口尺寸变量
    windowHeight = w, windowHeight = h;
}

void onRender()
{
    // 清除上一帧的缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

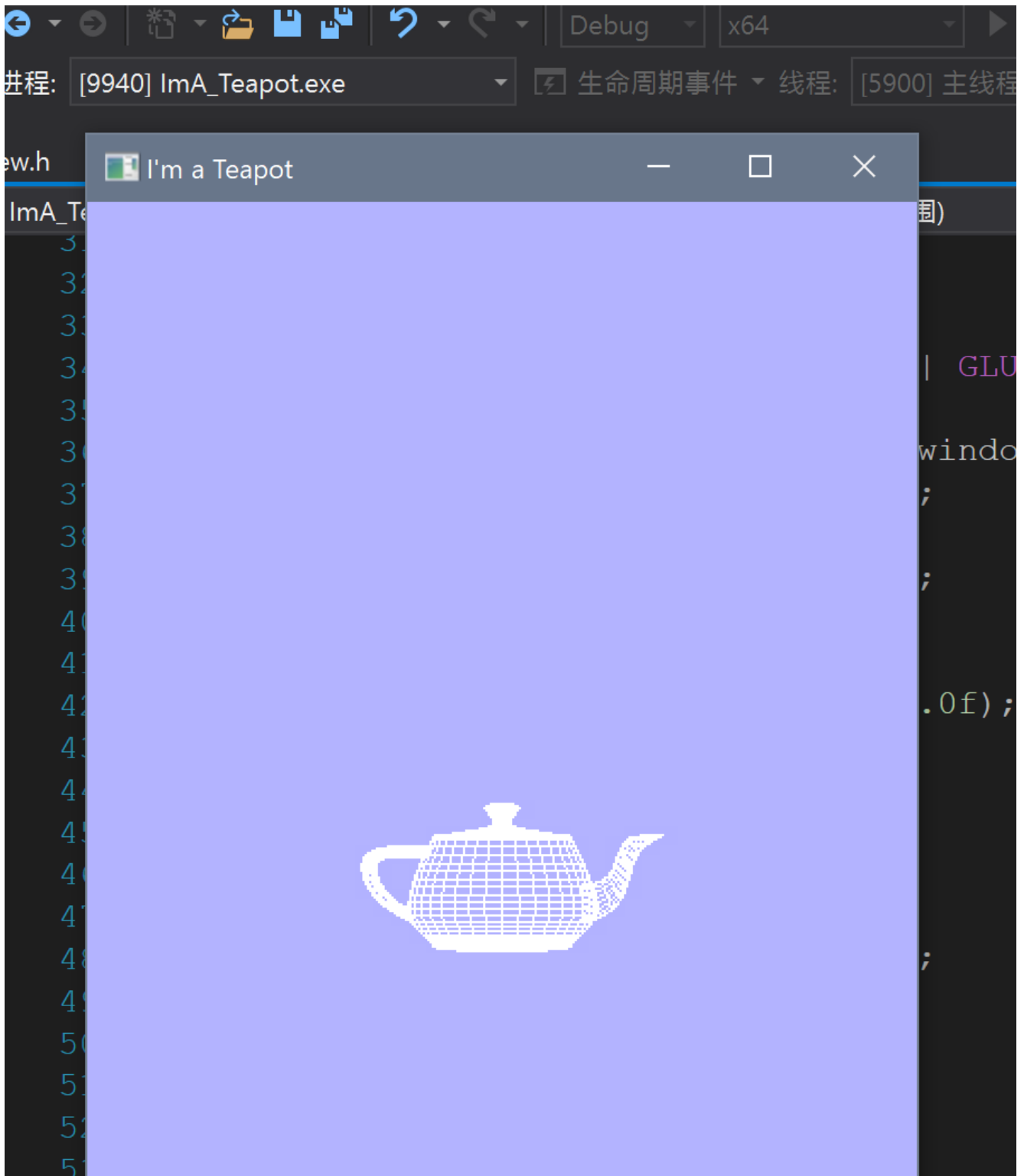
    // 启用双帧缓存
    glutSwapBuffers();
}

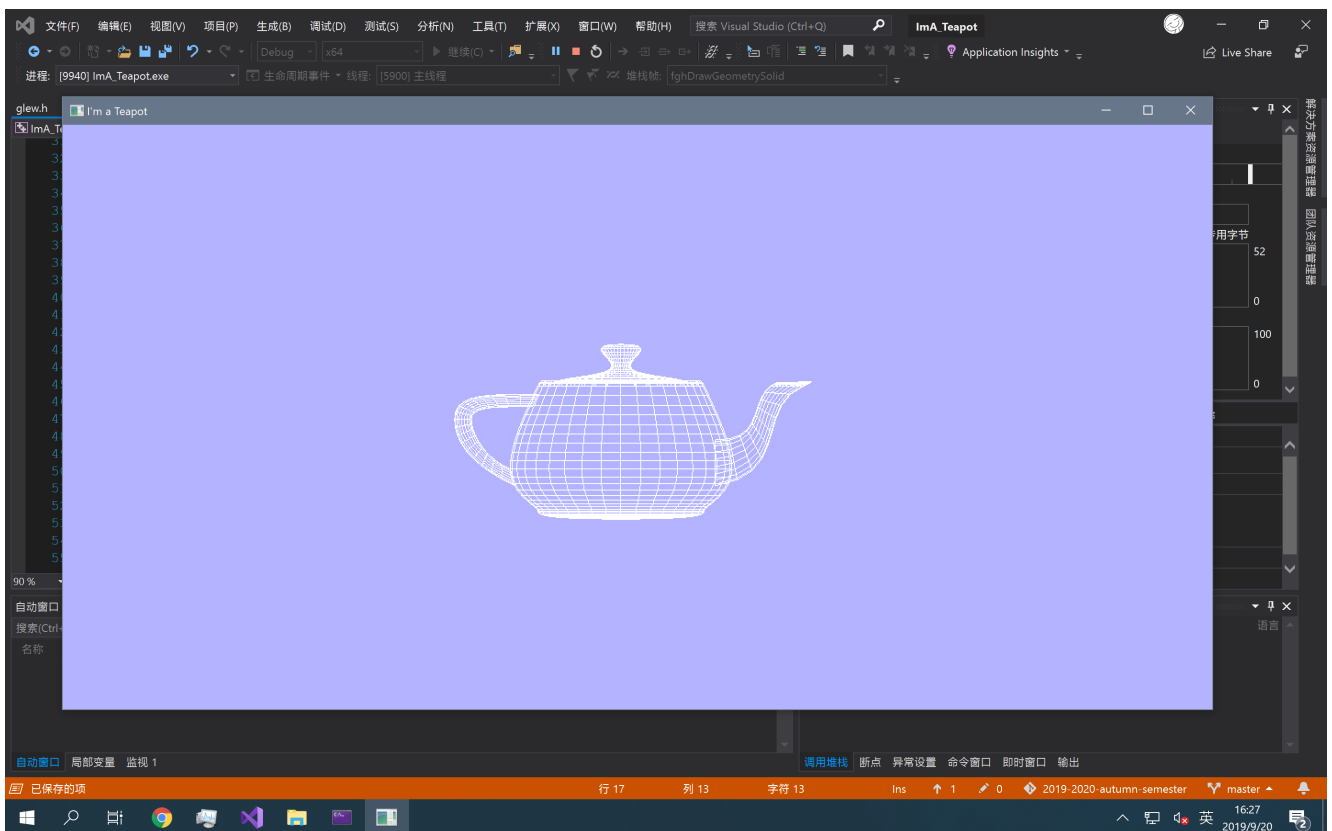
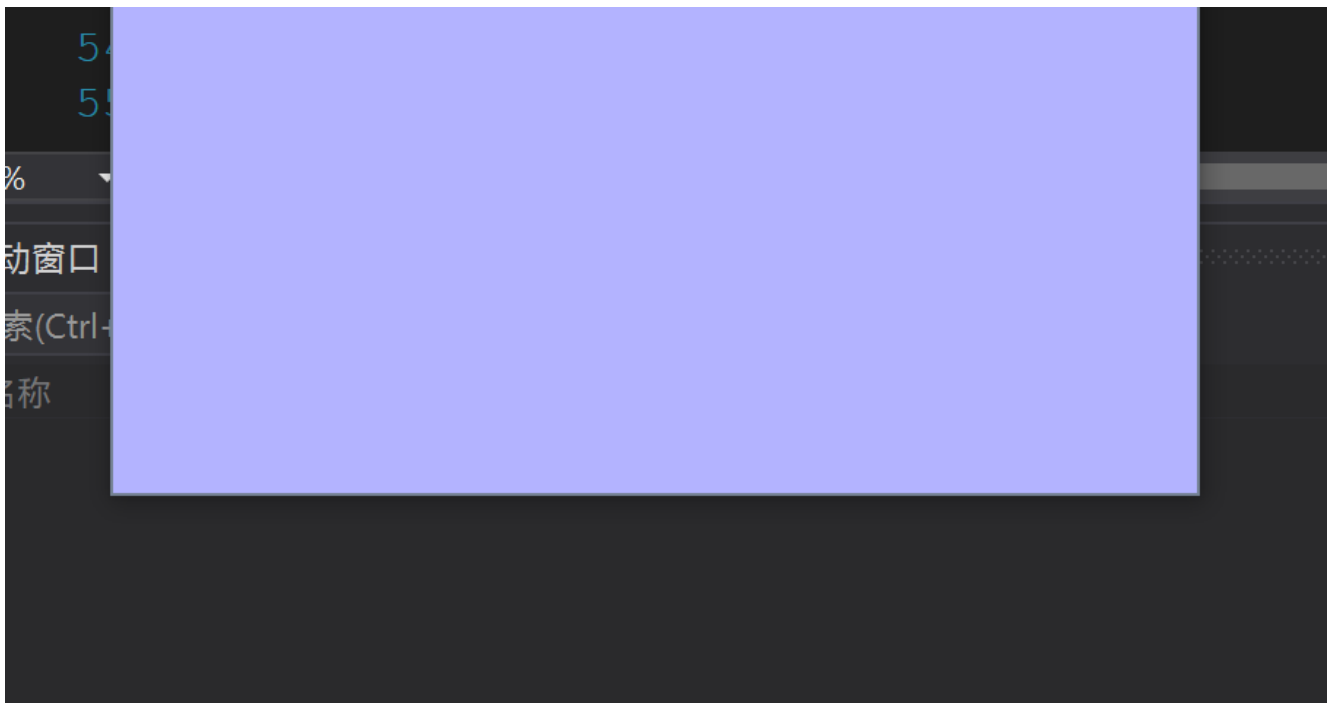
int main() {
    ...
    glutReshapeFunc(onWindowResized);
    glutDisplayFunc(onRender);
    ...
}
```

22. 完成之后，我们可以开始画 Teapot 了。在 onRender 函数中我们绘制一帧茶壶，我们需要动态计算出合适的茶壶大小，并使用 `glutSolidTeapot` 函数将其画到画面中。

```
int teaPotSize = int(std::min(int(windowWidth * 9.0 / 16.0), windowHeight));
glViewport((windowWidth - teaPotSize) / 2, (windowHeight - teaPotSize) / 2,
teaPotSize, teaPotSize);
glutWireTeapot(0.4f);
```

我们来逐行分析代码：其中第一行计算出按照 16: 9 重分配比例之后窗口的较短边作为 Restriction；然后将 ViewPort 设定在这个画幅的正中央处；最后按照 0.4 倍的比例画出线框茶壶。这个算法经过测试可以保证在任何窗口比例下显示合理的画面。





23. 最后，为了以后实现变色功能，我们将硬编码的颜色值抽离出来成为变量，并在绘制茶壶之前先设置笔刷颜色，使用 `glColor3f` 函数：

```

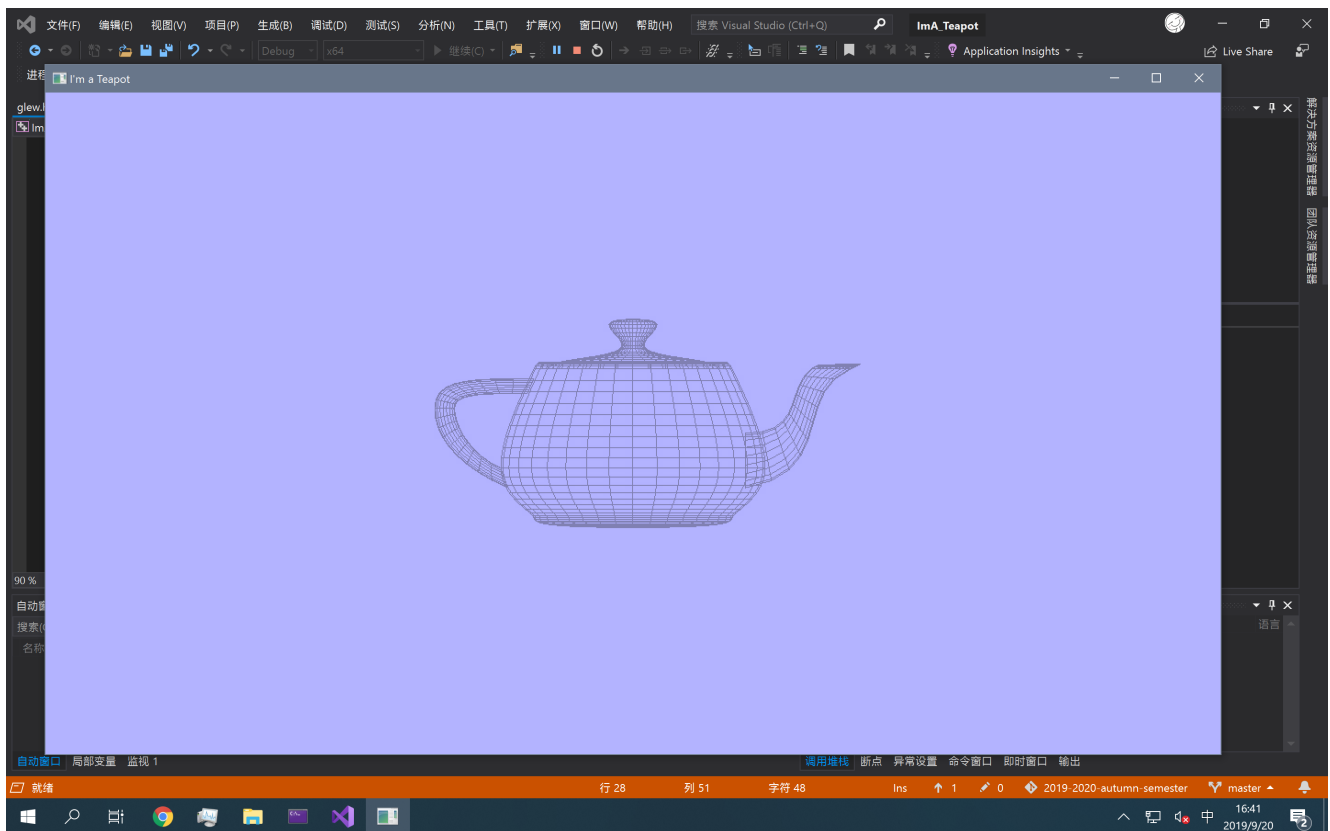
static GLfloat teapotRed = 0.5f;
static GLfloat teapotGreen = 0.5f;
static GLfloat teapotBlue = 0.7f;

static GLfloat bgRed = 0.7f;
static GLfloat bgGreen = 0.7f;
static GLfloat bgBlue = 1.0f;

static void onRender() {
    ...
    glColor3f(teapotRed, teapotGreen, teapotBlue);
    ...
}

```

此部分完成后的效果如下图所示。



第六部分：转茶壶

代码更改

首先，OpenGL 中的旋转跟一般的 3D 渲染器类似，都使用三个坐标来表示 x、y、z 三轴方向的旋转分量。因此我们首先定义三个静态变量用于存储旋转值：

```

static GLfloat rotationX = 0.0f;
static GLfloat rotationY = 0.0f;
static GLfloat rotationZ = 0.0f;

```

之后在 `onRender` 函数中调用 `glRotatef` 即可。注意后面的三个浮点参数即为坐标轴 x、y、z 方向上的分量。因此更新角度部分的代码写作：

```
glRotatef(rotationX, 1.0, 0.0, 0.0);
glRotatef(rotationY, 0.0, 1.0, 0.0);
glRotatef(rotationZ, 0.0, 0.0, 1.0);
```

我们要做的就只是更新三个角度值就能实现旋转了。

问题发现

然而在实现代码之后发现，如果直接让角度进行随机数增减，会造成运动的不平滑，即速度会发生突变，看起来很生硬。

因此我们考虑引入「角速度」的概念，让茶壶旋转的角速度随机变化，而旋转的角度值一阶平滑渐变。

代码解读

```
static GLfloat speedX = 0.0f;
static GLfloat speedY = 0.0f;
static GLfloat speedZ = 0.0f;
// 初始状态下将速度设置为 0
```

计算速度随机变化值调整为：

```
srand(time(NULL));
/* 使用 time(NULL) 作为随机数种子，以免伪随机数出现循环 */

double random_value = double(rand()) / RAND_MAX;
speedX += random_value - 0.5;
/* 保证速度变化值的数学期望为 0，防止出现「一边倒」 */

// speedY, speedZ 类比
```

同时为了防止速度过快导致的难看，设置了一个正反方向的速度上下界。

```
// 设定的速度最大值
static GLfloat const maxSpeedLimit = 3.5f;

// 手动计算溢出
if (speedX > maxSpeedLimit) {
    speedX = maxSpeedLimit;
}
else if (speedX < -maxSpeedLimit) {
    speedX = -maxSpeedLimit;
}
```

最后按照角速度更新每一时刻的角度。这里需要处理 0 和 360 的上下溢出情况。

```
rotationX += speedX * 0.5;
if (rotationX >= 360.0) {
    rotationX -= 360.0;
} else if (rotationX < 0.0) {
    // 注意 rotation 的合法范围为 0 到 360 的前闭后开区间,
    // 因此这里的小于号不取等号
    rotationX += 360.0;
}
```

另一个问题

在进行这一步操作之后，我们留意到只有在我们点击屏幕或是拖动窗口的时候，图形才会旋转，而在保持不动时不会旋转。为什么？

因为我们之前注册的 `onRender` 回调函数是以 `glutDisplayFunc` 注册的。这个函数只会在「他认为」需要重新绘制画面的时候才会去调用回调函数。在鼠标不点击、画面不缩放的情况下，他认为不需要重新绘制画面，就没有去调用我们的更新函数，自然茶壶也就不转了。

解决方法很简单：用 `glutIdleFunc` 来注册 `onRender` 回调函数。在渲染器闲置的时候，就会去调用我们的更新旋转函数，这个问题也就解决了。

第七部分：颜色渐变

代码解读

跟上面的旋转类似，也是一个三维向量，只是需要小心选择好参数防止出现不连续的变动即可。

```
// 每次只增加或减少 0.01 范围以内的值
teapotRed += (random_value - 0.5) / 50.0;

// 防止颜色分量溢出
teapotRed = std::max(std::min(teapotRed, 1.0f), 0.3f);
```

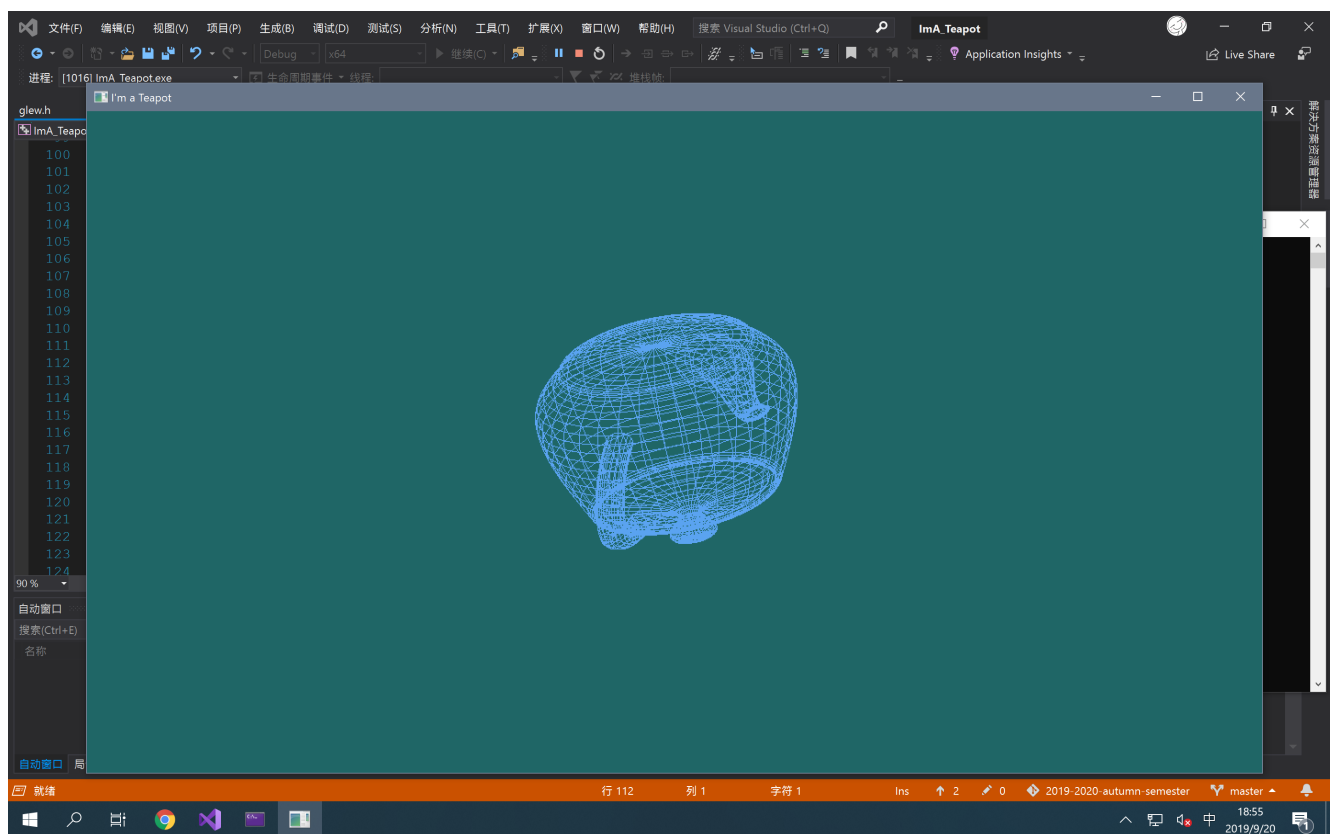
这里将每种颜色的分量都设定为不小于 0.3f，是因为为保证背景颜色协调一致，采取了这样的背景颜色算法：

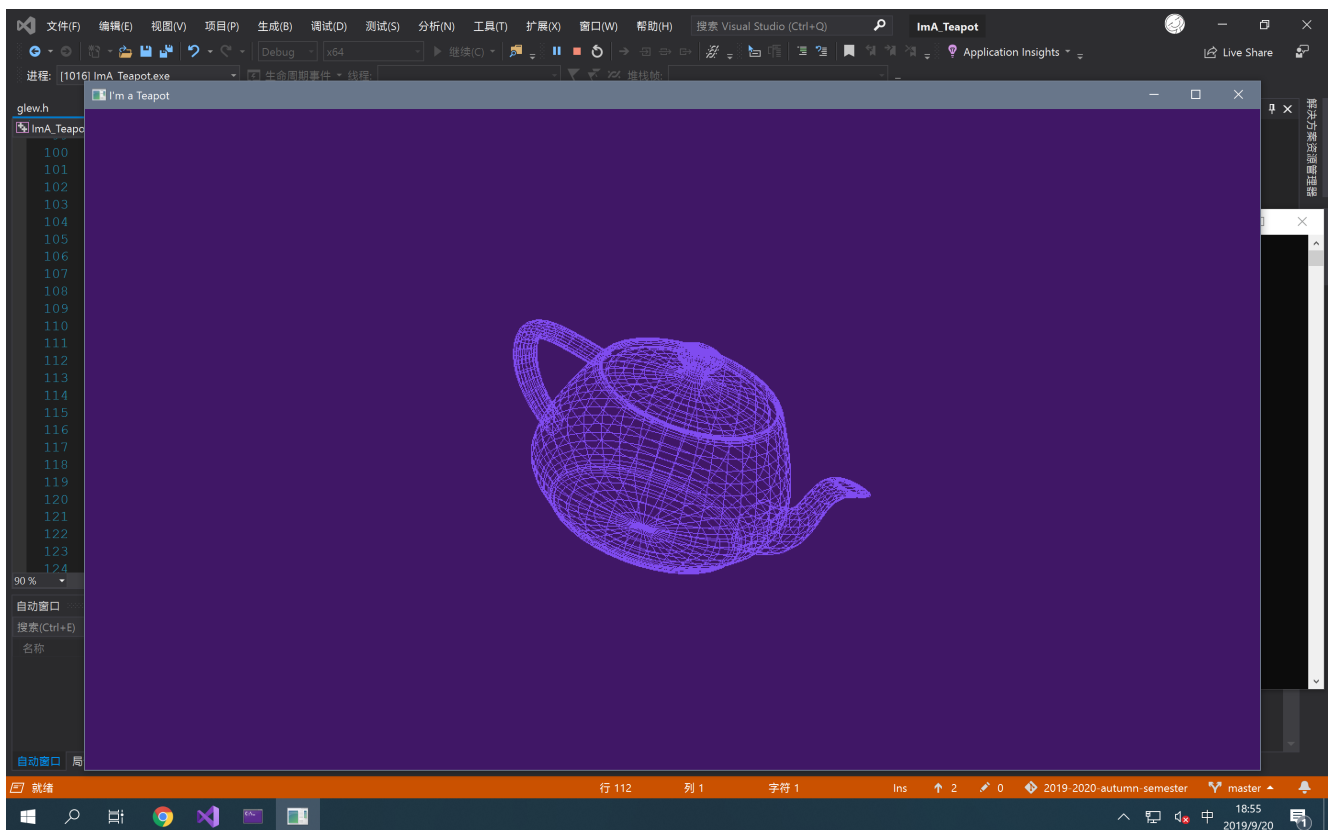
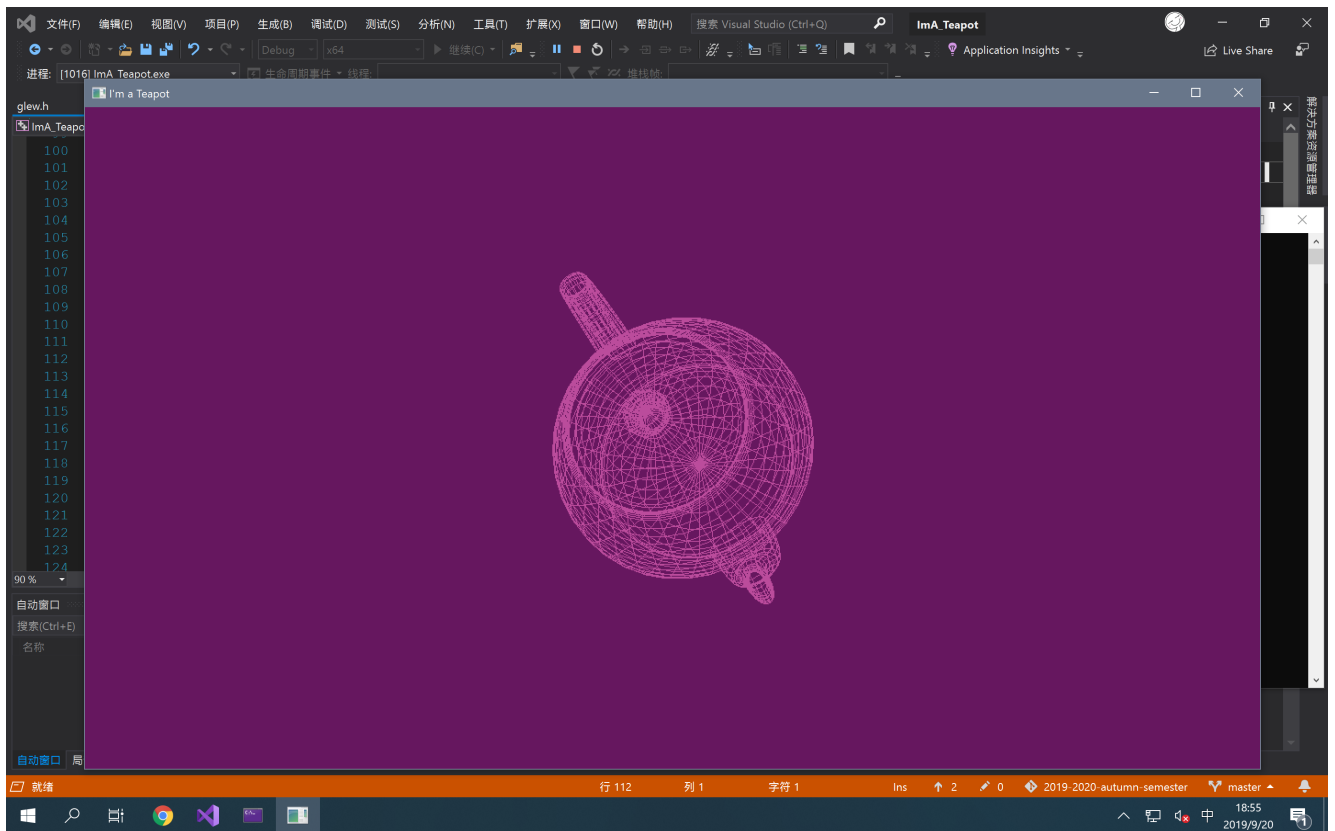
```
bgRed = std::min(pow(teapotRed, 2), 0.4f);  
bgGreen = std::min(pow(teapotGreen, 2), 0.4f);  
bgBlue = std::min(pow(teapotBlue, 2), 0.4f);
```

即前景颜色平方值，更靠近 0（暗）端之后作为背景颜色，且最大值不超过 40%。

由于之前限制了前景颜色的各分量不小于 30%，因此不会产生前景背景都很暗淡的难以辨认情况。

效果图





第八部分：发布注记

程序发布于 `/ass1/release/ImA_Teapot.exe`。

依赖的动态链接库：

- `/ass1/release/glew32.dll`

- `/ass1/release/freeglutd.dll`

可能需要将它们放置在 `%SystemRoot%/Windows/System32` 和 `%SystemRoot%/Windows/SysWOW64` 目录下。

录制的效果视频参见 `/ass1/movie/rolling.teapots.mov` 。

第九部分：笔记

Sep 12

Teacher

计算机图形学

肖双九 @`xsjiu99@cs.sjtu.edu.cn`

TA

葛续荣、许泽资

Homework

按具体要求发邮件上交给 `cg_dalab@163.com`，或是上传到交大云盘里。

命名规范：学号+姓名+作业编号，打包成一个 `.zip` / `.tar.gz`。

Text Book

《计算机图形学》(Computer Graphics)

written by *Donald Hearn*, Translated by *Cai Shijie*, 电子工业出版社

> 平时作业占比 `50%`！请务必认真进行。

> 请不要抄袭互联网上或是同学的代码。

> **Warning：**课堂笔记都是每次平时作业报告的一部分！且占比很高。

>

> 要求每次笔记都需要提出一个有价值的问题。而且最终需要回答他。

课件

逐部分放置在交大云盘上。

Today: Chapter 0

计算机图形学入门·计算机绘图基础

Contents

- 计算机图形学入门
 - > 今天的这节课
- 光栅化图形学
 - > 我们的显示器只是一堆光栅...怎么表现东西？
- 三维建模及其变换
 - > 在一个三维坐标系中，如何进行坐标变换，如何进行具体的计算？
- 光照计算

> 要有光，看起来能更真实些

- 图形的细节渲染

> 有了这些，看起来会更真实

- 光影处理

- 纹理

- 图形特效

- 阴影

-

进入第一章：计算机图形学入门

计算机绘图基础

Definition：啥是计算机绘图？

为啥要绘图？对真实世界的映射。相比于语言文字，人类总能更轻易地理解图形。
另外，绘图能帮助人类完成许多了不起的工程，传达珍贵的工程经验。

为啥要用计算机绘图？

算力强，发展快。

自 1940s 开始，计算机辅助绘图已经非常常见且普遍了。

反过来，计算机图形界面也反过来使得计算机更加普及、易用了。

计算机图形硬件：支撑一切的基础

没有纸，没有笔，咱们怎么画东西？

- 图形输出设备 (Image Output Devices)

> 没有输出设备您画了半天画了个啥？

最简单最常见的：显示器、绘图仪、打印机，等等。都可以把计算机绘图的结果展示出来。

论及显示器，到现在还是没有本质变化...还是由大量发光像素的组合来显示图形的。

无论是 CRT（阴极射线显示器）、LCD（液晶显示器），.....皆如此。

目前基本都是给予 2D 显示。即，如果用显示器展示 3D 模型的话，等于是损失了一个维度。

- 图形输入设备 (Image Input Devices)

计算机输入设备基本都包括在内，如鼠标、手柄、触摸屏、触摸板、绘图板，等等.....

以及一些光学采集设备，如：摄像头、扫描仪、相机，等等.....

还有一些比较偏门的：数据手套、坐标数字化仪（没听说过）.....

- 图形处理设备 (Image Processing Devices)

好了，有了图形数据的输入输出，就该由计算机来进行处理，实现所需的制图效果了。

摩尔定律、并行计算的发展使得图形处理的能力爆炸式发展；

而人类对真实感和娱乐的强烈需求也刺激了图形处理的发展。

这部分工作该交给谁呢？

显卡！显卡！Graphics Card！专业点讲，Graphics Processing Unit。

> 插入：显卡的历史：

> 最早的显卡真的就只是用来显示（还是单色的.....），由 IBM 推出于 1981 年，至于显示什么还是交给 CPU 的。

> 但是 CPU 全才，并不是单单用来做显示计算的，所以其效率较低。

> 那么，我们逐渐把那些跟图形有关的内容单独拉出来放在另一张芯片里，美其名曰「图形加速卡」，

> 可以分担一部分 CPU 的功能...最后它的功能越来越多，反走样，3D 计算，光照计算，特效.....

> 这些全部都写在硬件里，效率很高很高。（虽然很死板，但效率高啊）

> 最后，这些东西都交给了 GPU。

> 但是如果所有的这些东西全部写死了，那就失去了灵活性。我们希望他可以兼具灵活性和效率，因此就出现了

> 可编程 GPU！（Yeah！）直接对硬件编程；加上 GPU 的晶体管已经大大超过了 CPU，Intel 感到极大压力（雾

这就是一点简单的计算机图形历史了。

计算机图形软件

谈完硬件谈软件（用户和机器之间的交互层

- 计算机图形 API

首先，咱们来介绍几个著名 API：

- 跨平台、通用：OpenGL

- Windows 专用、邪恶、微软家的：DirectX

现在所有的硬件产品、图形引擎基本都接受两种标准。

其他一些（没什么人用的）API：Java 3D、Java Mobile 3D、Windows Mobile（已死）APIs

- 高级图形渲染器

基于以上的那些 APIs，有人构建了一些更高层的引擎：

- Pixar 家的 RenderMan
 - > 只给 Disney/Pixar 自家用
- NVidia 家的 Gelato
 - > 与 RenderMan 兼容的，（某些方面）更高效的渲染器（法务部警告
- German mental images 公司推出的 MentalRay 渲染器
 - > 已经被 NVidia 买下来了（

> Question: API 跟 图形渲染器的关系？

> A: 图形渲染器基于具体的 API。

- 游戏引擎（另一码事儿）

也在 API 上层，而且除去图形相关的东西，还包含了很多其他的跟游戏内容有关的东西，包括交互、资源管理、逻辑等等等等，（像是 Unity, Unreal 之类的东西）有些杂。

因此这门课里不用游戏引擎来讲。

OpenGL

- 开放的图形 API，工业化的标准
- 完全独立于窗口系统，OS，等等一切东西。
- 从点线面开始，构建一切东西
- 能力：创建二维、三维物体，布置（layout）并观察场景

OpenGL 的库函数

核心库：名为 gL，所有的函数名开头都是 GL。

实用库：GLU，所有函数名开头均为 GLU。

辅助库：GLaux，一些窗口相关的东西

工具库：GLUT，创建窗口和其他 GUI 的库函数

另有基于 GLUT 的 GUI 库名为 GLUI。

面向 Windows 的扩展：WGL，仅仅包括很旧版本的 OpenGL。

OpenGL 扩展库并没有它本身那么标准...基本也是群魔乱舞...

Warning: 提交作业时附上你使用的库。（否则 TA 跑不起来...

Sep 19

2019/09/19

SE-344

> 计算机图形学

理论性基础

主要讲一些计算机图形学是啥？研究些什么？跟几何学的渊源？发展和应用？

Definition

`CG` -> `Computer Graphics`

用计算机来生成、处理、显示「图形」。

Problems

主要问题：模型、动画、渲染。

- * Modeling

- * Animating

- * Rendering

Solutions

CG 的研究范畴

- * Physical Simulation (物理仿真)

- * Physical Hardware & Software (图形软硬件)

- * 硬件部分

- * 数据采集和测量

- > 从真实世界中通过图形的方式来获取数据

- * 传感控制

- > 通过上面获取的数据来干预真实世界

- * 软件部分

- * 矢量绘图软件

- > 绘图软件不用多说，实在太多了。

- > 矢量图指的是用顶点和曲线等几何信息来描述的图形，放大不走形

- * 3D 建模

- > 3ds max, Maya, etc.

- * Graphics Algorithms (图形算法)

- 算法就是上面各种应用程序的底层支撑了。

- 包括矢量图形处理、渲染基础、动画制作云云。

- * Graphics Standard (图形标准)

标准化是非常要紧的一件事情。

生成

处理

显示

第十部分：笔记问题和解答

Q：OpenGL、GLUT、FreeGLUT、glfw、glew、mesa、glad 分别是什么？应该如何选择？分别处于图形渲染的哪一层？

A：逐个来看。

- OpenGL

OpenGL，译名为「开放式图形库」，它本身只是定义了一套简单而强大的接口的 API，可以用来渲染各种图形。它本身是跨语言、跨平台的。OpenGL 标准没有规定任何平台、任何语言的具体实现，所有实现都是由各显卡驱动厂商提供的。

- GLUT

GLUT 是基于 OpenGL 和各平台 OS 的更上一层实现。它的全称是 OpenGL Utility Toolkit，目的在于隐藏各操作系统之间的区别性，并且简化部分 OpenGL 调用。然而这个软件已经于数十年前停止更新了。

- FreeGLUT

由于 GLUT 是闭源的，因此作者停止更新之后一直没有被维护。之后出现的 OpenGLUT、FreeGLUT 基本上是 GLUT 的开源版本，具有非常类似的函数调用，且仍在持续更新。

- glfw

glfw 也是在 OpenGL 之上提供的一层封装，和 GLUT 功能类似，意在简化窗口创建等平台相关的操作。

- glew

glew 全称是 The OpenGL Extension Wrangler Library，大意就是加载一些 OpenGL 的扩展功能，方便调用一些更新的 OpenGL 函数接口。

- glad

由于 Windows 提供的 OpenGL 头文件非常旧，因此需要 glad 来解析更新版本的 OpenGL 函数指针地址。

- mesa

以上这些都是 OpenGL 及其衍生软件。而 mesa 则是一个纯通过软件开源地实现了 OpenGL API 的图形库。

总结：

- OpenGL 是接口，各驱动厂商提供了高性能的硬件实现；
- mesa 纯软件地实现了类似于 OpenGL 的接口；
- 在 OpenGL 接口上为了简化部分函数调用及跨平台问题，出现了 glfw、GLUT 和 FreeGLUT；
- 随着 OpenGL 演化发展，出现了一些新的扩展接口，需要 glew 来调用这些时髦接口；
- 最后，因为 Windows 头文件太老，glad 提供了使用新版 OpenGL 的方法。

本次采用的库组合：

首先，由于使用 Windows，一定会使用 glad；

因此一开始使用了 glfw + glew + glad 的库组合；

后来因为需要一个茶壶模型，而 glfw 并未提供这项没用的功能，

因此改用 FreeGLUT + glew + glad 的组合来实现。

在选择这些库的时候，采用这个原则：

GLUT、FreeGLUT、glfw 选一个；

要调用新 API，加上一个 glew；

要在 Windows 下面跑，加一个 glad。

第十一部分：报告问题记录

Q1：为什么背景色的 alpha 值设定没有生效？

A1：为什么 glClearColor 设定透明背景的代码不起作用？

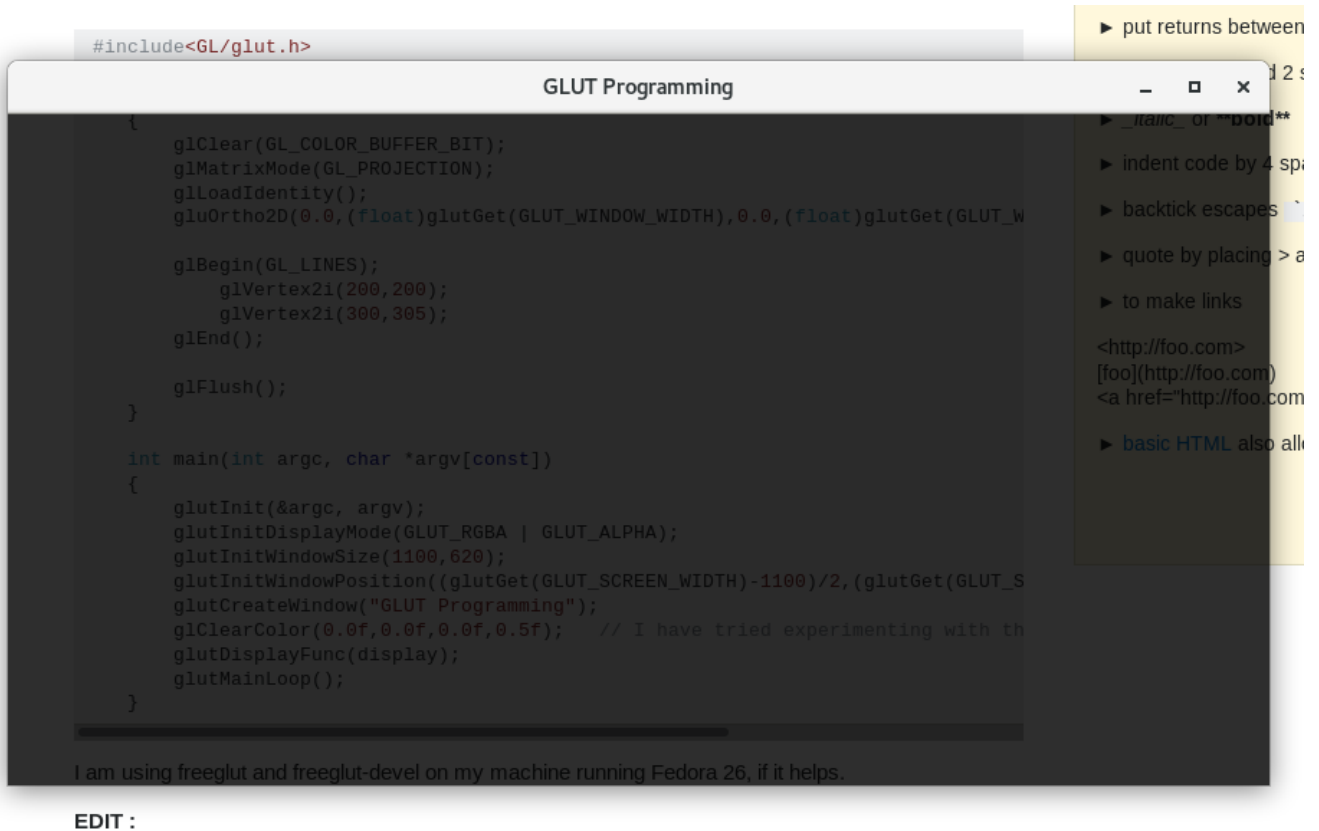
事实上这个函数的实现中，调节 alpha 值并非用于生成另一种纯色，而是为了实现「半透明效果」，即可以透过背景看到半透明的其他窗口内容，类似于 Windows 8 之前版本中的 Aero 效果。为了使这项功能有效，我们需要使用

```
glEnable(GL_BLEND)
```

```
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA ); // = color * alpha +
background * (1-alpha)
```

来表明如何计算被遮挡窗口的颜色。

⚠ Windows 平台不支持此项功能，需要在 Linux 下执行测试。



```
#include<GL/glut.h>

{
    glClearColor(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (float)glutGet(GLUT_WINDOW_WIDTH), 0.0, (float)glutGet(GLUT_W

    glBegin(GL_LINES);
        glVertex2i(200,200);
        glVertex2i(300,305);
    glEnd();

    glFlush();
}

int main(int argc, char *argv[const])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_ALPHA);
    glutInitWindowSize(1100,620);
    glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-1100)/2, (glutGet(GLUT_S
    glutCreateWindow("GLUT Programming");
    glClearColor(0.0f,0.0f,0.0f,0.5f); // I have tried experimenting with th
    glutDisplayFunc(display);
    glutMainLoop();
}

I am using freeglut and freeglut-devel on my machine running Fedora 26, if it helps.
```

EDIT :

最终效果如上图所示。

Q2：什么叫做深度缓冲机制？

A：深度缓冲指的是在生成每一个像素的时候，都把它的深度信息存储在一个缓冲区之中；如果另一个物体也被渲染到同一个像素处，则通过深度缓冲区对二者的深度进行比较，并消隐那个被遮挡的物体。

如果这个 OpenGL 程序需要处理多个物体间的遮挡关系，那么大概率会需要将 `GLUT_DEPTH` 掩码设定上。