实验1：Spinning Pinwheel(绘制流水线实验)

AI

任俊峰

201700301123

实验介绍

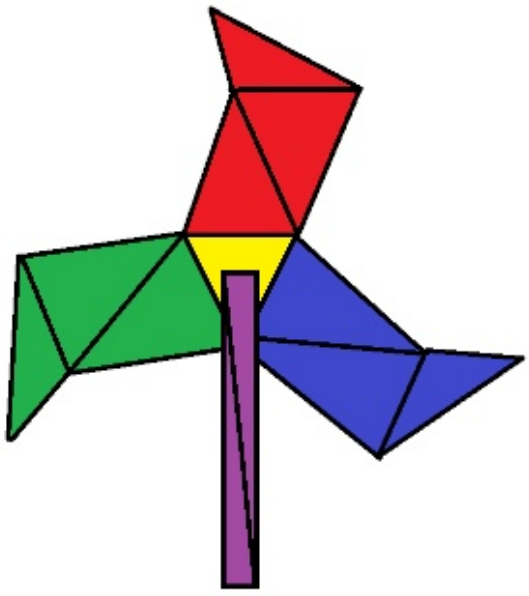
实验目的

1. 学会简单的openGL 3D图形渲染编程
2. 学会编程实现简单的动画功能
3. 学会编程实现简单的交互功能

实验内容

利用openGL编写一个程序，实现以下功能：

1. 1.构建且渲染一个3D风车模型（如下图所例示）。其中，三个叶片（包括中间的黄色三角形）在一个2D平面上，手柄在另外一个离相机更近的2D平面上。三个叶片、中心的三角形、手柄分别用不同的颜色显示。注：图中画的黑色线条是为了方便演示，不要求绘制。



2. 实现风车的旋转动画。要求风车的三个叶片以及中间的黄色三角形（在其所在平面上）一起绕着中心一点不停旋转，且将此动画渲染出来。

3. 实现通过键盘对动画的交互控制，包括切换旋转方向、增大旋转速度以及减小旋转速度。
4. 设计按钮和菜单两个控件，用于动画的交互控制：点击按钮可以切换旋转方向；选择三个菜单项分别可以切换旋转方向、增大旋转速度以及减小旋转速度。

实验环境

1. **System :** macOS Catalina 10.15
2. **Frameworks** : glfw + glad + dearimgui

Notes:使用glad 有一点是它在函数定义里把gl库很多函数名字重定义为glad前缀

实验环境搭建

学习资料主要来源：
https://learnopengl.com (采取glad+glfw)

1. 框架
    - glfw 是轻量级框架，相对与glut与freeglut更新，在OpenGL的核心模式（**(Core-profile)**）下进行开发，但在GUI方面无法创建菜单按钮等。
    - glad 相较于 glew更新，用于管理Opengl函数的指针
    - dearimgui 对于glfw无GUI支持的补充，同样也是跨平台工具，将github上相关头文件拖入项目即可
2. 其他方面支持
    - 自己跟着教程将着色器封装至shader.h文件中
    - stb_image.h 用于读取文件（用于纹理）
    - glm数学库

实现功能

1. 使用glfw创建窗口

初始化glfw

```
// glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    #ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // uncomment this statement to fix compilation on OS X
    #endif
```

创建glfw窗口

```
// glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
```

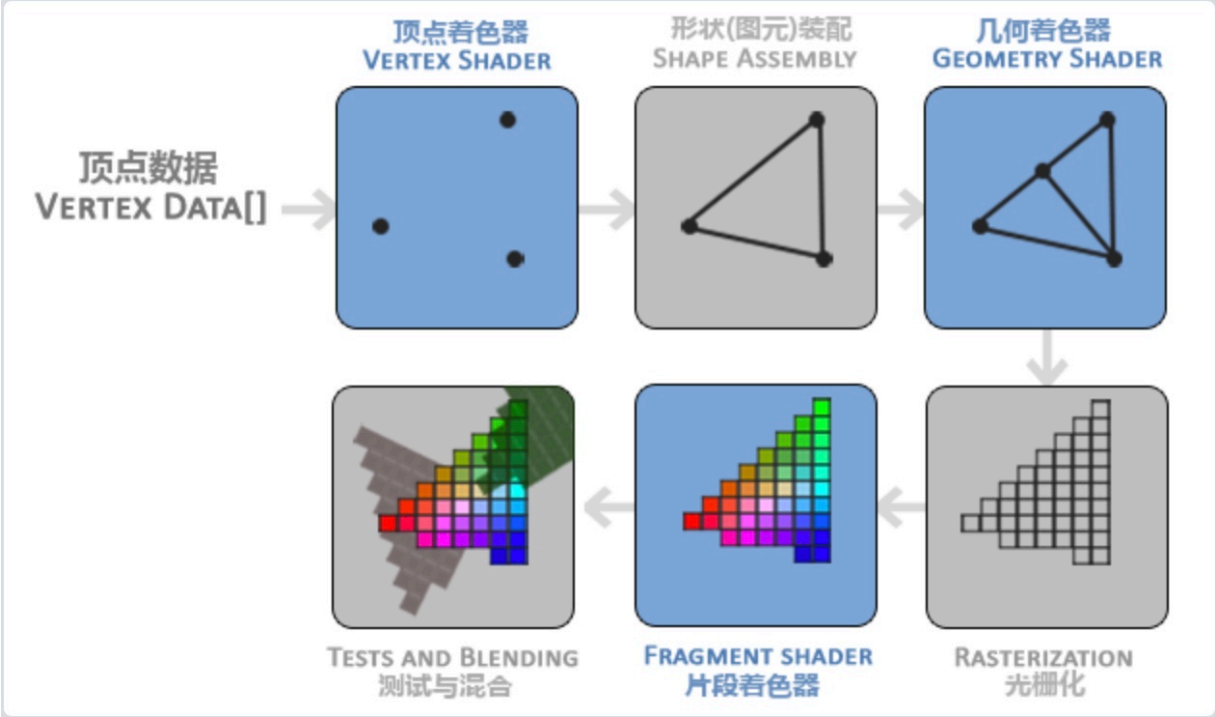
```
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
```

初始化glad管理函数指针

```
// glad: load all OpenGL function pointers
    // ---------------------------------------
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }
```

2. 着色器

首先我们要了解图形的渲染管线的抽象展示



我们可编辑的有

顶点着色器　几何着色器　片段着色器

这里我们编辑顶点着色器和片段着色器，使用**GLSL（GL Shader Language)**

顶点着色器

```
#version 410 core                    //使用opengL core 4.1
layout (location = 0) in vec3 aPos;   // 位置变量的属性位置值为 0
```

```

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{   //对输入值进行简单处理，不能直接把传入顶点着色器的值传给片段着色器，要简单处理
    //注意矩阵左乘。乘法y从右向左读
    gl_Position = projection * view * model * vec4(aPos,1.0);

}
```

片段着色器

```
#version 410 core
out vec4 FragColor;

in vec3 ourColor;

void main() {
    FragColor = vec4(ourColor, 1.0f);
}
```

着色器程序对象

- 着色器程序对象(Shader Program Object)是多个着色器合并之后并最终链接完成的版本。如果要使用刚才编译的着色器我们必须把它们链接(Link)为一个着色器程序对象，然后在渲染对象的时候激活这个着色器程序。
- 渲染时要激活程序对象

着色器的编译与连接

```
const char *vertexShaderSource = "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "out vec4 vertexColor;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
    "   vertexColor = vec4(0.5,0.0,0.0,1.0);\n"
    "}\0";
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "in vec4 vertexColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vertexColor;\n"
    "}\n\0";
```

```
//vertex shader
    unsigned int vertexShader;
    vertexShader = glad_glCreateShader(GL_VERTEX_SHADER);
    glad_glShaderSource(vertexShader,1,&vertexShaderSource,NULL);
```

```
    glad_glCompileShader(vertexShader);
    //检测编译是否成功
    int success;
    char infoLog[521];
    glad_glGetShaderiv(vertexShader,GL_COMPILE_STATUS,&success);
    if(!success){
        glad_glGetShaderInfoLog(vertexShader,521,NULL,infoLog);
    std::cout<<"ERROR::SHADER::VERTEX::COMPLIATION_FAILED\n" << infoLog << std::endl;
    }
    //fragment shader
    unsigned int fragmentShader;
    fragmentShader = glad_glCreateShader(GL_FRAGMENT_SHADER);
    glad_glShaderSource(fragmentShader,1,&fragmentShaderSource,NULL);
    glad_glCompileShader(fragmentShader);
    // check for shader compile errors
    glad_glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glad_glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog <<
std::endl;
    }
    //link shader
    unsigned int shaderProgram;
    shaderProgram = glad_glCreateProgram();
    glad_glAttachShader(shaderProgram,vertexShader);
    glad_glAttachShader(shaderProgram,fragmentShader);
    glad_glLinkProgram(shaderProgram);
    // check for linking errors
    glad_glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
    if (!success) {
        glad_glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
    }
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
```

为保证简单，我们将着色器封装至**Shader.h**头文件中，使用时创建着色器对象即可，将声明顶点着色器和片段着色器的部分写至**shader.vs**和**shader.fs**文件中，使用时读取即可。

```
Shader ourShader("homework1/shader.vs", "homework1/shader.fs");
```

---

3. 顶点数据的输入、绑定和解析

定义一个数组来存储三角形的三个点的坐标数据

画中间的黄色三角形

```
float vertices2[]{
        -0.5f,  0.289f, 0.0f,
         0.5f,  0.289f, 0.0f,
```

```
    0.0f,  -0.578f, 0.0f
};
```

1. 通过顶点缓冲对象VBO管理内存

```
//利用glGenBuffers函数和一个缓冲ID生成一个VBO对象:
unsigned int VBO;
glGenBuffers(1, &VBO);
//使用glBindBuffer函数把新创建的缓冲绑定到GL_ARRAY_BUFFER目标上
glBindBuffer(GL_ARRAY_BUFFER, VBO);
//调用glBufferData函数，它会把之前定义的顶点数据复制到缓冲的内存中
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

还需要调用glVertexAttribPointer，告诉OpenGL怎么解析数组中的数据。（应用到逐个顶点属性上）
glEnableVertexAttribArray以顶点属性位置值作为参数，启动顶点属性，顶点属性默认是禁用的。

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

```
//参数一：指定我们要配置的顶点属性，前面GLSL声明location=0 的那个值
//参数二：指定顶点属性大小,这里是vec3所以选3
//参数五：步长，顶点属性组之间的距离，这里的下一组位置在3个float之后，所以是3 * sizeof(float) ,
//如果我们要在传入的数组中包括vec3 颜色变量 和 vec2 纹理 就是 8 * sizeof(float)了
//参数六：offset，如果传入颜色和纹理，则要分别传入和解析数据，这里需要用到偏移量

//第二个函数以顶点属性位置值作为参数，启动顶点属性，顶点属性默认是禁用的。
```

这里我们看一下加入之后如何调用

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 *
sizeof(float)));
glEnableVertexAttribArray(1);
// texture coord attribute
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 *
sizeof(float)));
glEnableVertexAttribArray(2);
```

2. 通过绑定顶点数组对象VAO，可以将多个VBO集成到顶点数组中，方便多次绘制重复的图形。

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
```

使用While循环进行多帧渲染。先激活程序、绑定VAO，然后输入三角形图元，用glDrawArrays生成图形。

```
shader.use();                           //封装后
glBindVertexArray(VAO);                 //绑定VAO
glDrawArrays(GL_TRIANGLES, 0, 3);       //画图
```

3. 通过索引缓冲对象(Element Buffer Object，EBO，也叫Index Buffer Object，IBO)节省开销

实验要求我们绘制风车，显然绘制风车的叶子部分需要三个三角形，9个点，但是有4个点重合，这是一种额外的开销。

我们可以通过利用EBO存储索引，使OpenGl根据顶点的索引来决定绘制哪个顶点。那么我们需要定义不重复的顶点对象和对应的索引两个数组。

```
float vertices1[] = {
        // positions
        -0.5f,  0.289f, 0.0f,
        0.5f,   0.289f, 0.0f,
        0.0f,   1.789f, 0.0f,
        1.0f,   1.789f, 0.0f,
        -0.20,   2.539f, 0.0f
};

    unsigned int indices[] = {
        0, 1, 2,  // first    triangle
        1, 2, 3,  // second   triangle
        2, 3, 4   // third    triangle
};
```

创建和绑定EBO（绑定到对应VAO方便操作）

```
unsigned int EBO;
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

渲染

```
shader.use();
glBindVertexArray(VAO);
glad_glDrawElements(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0);   //glDrawArrays(GL_TRIANGLES, 0, 3);
```

Notes:render渲染循环后，记得释放空间

```
// optional: de-allocate all resources once they've outlived their purpose:
// ------------------------------------------------------------------------
glDeleteVertexArrays(3, VAOs);
glDeleteBuffers(3, VBOs);
glDeleteBuffers(2, EBOs);
```

4.使用Uniform从CPU向GPU发送数据

1. 与顶点属性的不同：

1. uniform是全局的
   即uniform在每个着色器程序对象中都独一无二，且可以被着色器程序的任意着色器在任意阶段访问
2. 无论你把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。可以在任何着色器中定义uniform，无需通过顶点着色器作为中介。

Notes:如果你声明了一个uniform却在GLSL代码中没用过，编译器会静默移除这个变量，导致最后编译出的版本中并不会包含它，这可能导致几个非常麻烦的错误，记住这点！

2.如何给uniform添加数据

- 我们知道，前面的提到的着色器部分，是将数组传入VBO，由VBO这种缓冲对象（EBO也是缓冲对象，类型不同）将数据传输给顶点着色器，通过`layout (location = i) in`关键词接收数据
- 而uniform则是通过
  a. 找到着色器中uniform属性的索引/位置值。
  b. 根据索引/位置值更新他的值。

Notes: uniform 的设置是在当前激活的着色器中设置，所以设置前需要激活再设置

```
int vertexColorLocation = glGetUniformLocation(shaderProgram, "chooseColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation,  glm::vec3(0.8f,0.8f,0.0f));
```

这里我们都封装到Shader.h文件中，代码可以简化为以下形式，

```
ourShader.use();
ourShader.setVec3("chooseColor", glm::vec3(0.8f,0.8f,0.0f));
```

综上，着色器部分实现，如下所示，略有不同是因为举例时使用的是单一的VAO、VBO、VCO,实际实现时使用多个，采用数组对象，数组对象名当地址使用，不需要取地址符&.（渲染后面再说）

```
//风车叶子部分
    float vertices1[] = {
        // positions
        -0.5f,  0.289f, 0.0f,
        0.5f,   0.289f, 0.0f,
        0.0f,   1.789f, 0.0f,
        1.0f,   1.789f, 0.0f,
        -0.20,   2.539f, 0.0f
    };

    unsigned int indices[] = {
        0, 1, 2, // first triangle
        1, 2, 3,  // second triangle
        2, 3, 4  // third triangle
    };
    //中间三角形
    float vertices2[]{
        -0.5f,  0.289f, 0.0f,
        0.5f,  0.289f, 0.0f,
        0.0f,  -0.578f, 0.0f
    };

    //风车杆子
    float vertices3[]{
        -0.1f,0.0f,0.0f,
        0.1f, 0.0f,0.0f,
        -0.1f,-3.0f,0.0f,
        0.1f,-3.0f,0.0f
    };
    unsigned int  indice3[]{
        0,1,2,  //first triangle
```

```
    1,2,3   //second triangle
};


unsigned int VBOs[3], VAOs[3], EBOs[2];
glGenVertexArrays(3, VAOs);
glGenBuffers(3, VBOs);
glGenBuffers(2, EBOs);//第一个参数，根据输入数组长度,第二个参数取地址

glBindVertexArray(VAOs[0]);

glBindBuffer(GL_ARRAY_BUFFER, VBOs[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices1), vertices1, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBOs[0]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);



//绑定第二个VAO
glBindVertexArray(VAOs[1]);
glBindBuffer(GL_ARRAY_BUFFER,VBOs[1]);
glBufferData(GL_ARRAY_BUFFER,sizeof(vertices2),vertices2,GL_STATIC_DRAW);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,3 * sizeof(float),(void*)0);
glEnableVertexAttribArray(0);//着色器只用了location=0，都是0

//绑定第三个VAO
glBindVertexArray(VAOs[2]);
glBindBuffer(GL_ARRAY_BUFFER,VBOs[2]);
glBufferData(GL_ARRAY_BUFFER,sizeof(vertices3),vertices3,GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,EBOs[1]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(indice3),indice3,GL_STATIC_DRAW);
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

```

4. 利用模型矩阵、观察矩阵、投影矩阵进行3D化，实现旋转、不同平面

使用GLM库实现数学细节

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

```
glm::mat4 model = glm::mat4(1.0f);//初始化为单位矩阵
model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));//将角度值转化为弧度制传给rotate
```

可以通过model矩阵调整物体出现在世界坐标系的位置

```
glm::mat4 view= glm::mat4(1.0f);//初始化为单位矩阵
// 注意，我们将矩阵向我们要进行移动场景的反方向移动。
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

调整view矩阵以变相调整摄像机位置，利用这个实现风车与杆子出现在不同平面或者model也可

```
glm::mat4 projection= glm::mat4(1.0f);//初始化为单位矩阵
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f,
100.0f);
```

1. *Fov参数比较容易看出，我的理解是,眼睛睁开的角度,即,视角的大小,如果设置为0,相当你闭上眼睛了,所以什么也看不到,如果为180,那么可以认为你的视界很广阔,你观察的物体就显得很小，而为负数会发现物体颠倒*
2. *aspect为实际窗口大小，最好与显示窗口长宽比相等，否则观察到的物体会被压缩或拉伸*
3. *近平面过近会穿模*

运用公式$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$左乘运算，从右向左读。
得出的结果被赋给顶点着色器中的gl_Position,OpenGL会自动进行透视除法和裁剪

- 那么我们利用模型矩阵设置rotate，则可以将前面设置的风车叶子部分按旋转120度作循环渲染出三个不同颜色的叶子。颜色和三个矩阵都是uniform变量，在渲染循环中设置即可。
- 且可以设置全局变量speed和revolve调整旋转速度以及旋转方向，再加一个colorAlpha更改颜色。

同样的，对于这几个uniform变量，我们依旧需要找到对应的地址进行设置，这里我们也可以封装起来进行set使用。

```
ourShader.setMat4("model", model);
ourShader.setVec3("chooseColor", myColor[i]*colorAlpha);

OR

int viewLoc = glGetUniformLocation(ourShader.ID,"view");
glUniformMatrix4fv(viewLoc,1,GL_FALSE,glm::value_ptr(view));
int projectionLoc = glGetUniformLocation(ourShader.ID,"projection");
glUniformMatrix4fv(projectionLoc,1,GL_FALSE,glm::value_ptr(projection));
```

画出来的效果很丑，感觉可能是平面的原因利用z缓冲没什么效果，

```
/**
Z缓冲
    存储OpenGL深度信息，允许OpenGL决定何时覆盖一个像素而何时不覆盖
    • OpenGL存储它的所有深度信息于一个Z缓冲(Z-buffer)中，也被称为深度缓冲(Depth Buffer).
    • GLFW会自动为你生成这样一个缓冲（就像它也有一个颜色缓冲来存储输出图像的颜色）
    • 深度测试：深度值存储在每个片段里面（作为片段的z值），当片段想输出它的颜色时，
    将深度值与z缓冲进行比较，如果当前的片段在其他片段之后，它将会被丢弃，否则将会覆盖。

glEnable & glDisable
    • glEnable和glDisable函数允许我们启用或禁用某个OpenGL功能。这个功能会一直保持启用/禁用状态，直到另一
```

```
个调用来禁用/启用它。
    • 我们现在用glEnable来开启深度测试：
    glEnable(GL_DEPTH_TEST);
    • 因为我们使用了深度测试，我们也想要在每次渲染迭代之前清除深度缓冲（否则前一帧的深度信息仍然保存在缓冲中）。就像清除颜色缓冲一样，我们可以通过在glClear函数中指定DEPTH_BUFFER_BIT位来清除深度缓冲：
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    就是这样！一个开启了深度测试，各个面都是纹理，并且还在旋转的立方体！
*/
```

5. 键盘交互

设定`processInput`函数，在渲染循环中调用它实现键盘交互,用`glfwGetKey`检测

```
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
    if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
    {
        speed += 0.11f;
        if(speed >= 100.0f){
            speed = 100.0f;
        }
    }
    if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
    {
        speed -= 0.11f;
        if(speed <= 0.0f){
            speed = 0.0f;
        }
    }
    if(glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        colorAlpha += 0.01f;
        if(colorAlpha >= 1.0f)
            colorAlpha = 1.0f;
    }
    if(glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        colorAlpha -= 0.01f;
        if(colorAlpha <= 0.0f)
            colorAlpha = 0.0f;
    }

}
```

6.设计按钮和菜单控件，用于动画交互控制

这一块是觉得最困难的部分，前面的部分按步就班，理清概念很容易，如果打算采用对鼠标位置进行检测，实现对固定位置的一个渲染出的正方形进行点按操作，感觉按钮和菜单没什么区别，实现效果会很迷，最好还得检测在该物体位置时，正方形颜色出现变化，然后检测点击，还有一个问题是拉伸窗口大小后，也会有不利的影响。

所以本能地想利用GUI 的 API

因为需要添加菜单和按钮，而glfw属于轻量级框架，不提供GUI方面的函数，

在百度、stackOverflow、Google、github多方浏览后，大致得出以下结论：

1. windows端可以通过修改glfw源文件通过win32API添加按钮和菜单
2. mac似乎更多的是建议利用苹果自带的Cocoa框架进行绘制。有一点问题是openGL更多的是一种接口规范，利用Cocoa框架就得使用苹果的objective-c和swift进行代码重构，而且没有很多实例可供参考，有点盲人摸🐃的感觉。其次，苹果在18年10.14系统中正式禁用opengl，而是推荐开发者使用苹果自带框架Metal，所以两个框架在苹果系统上的文档都显得很少，整个的进行去写opengl的各个操作对初学者很不友好
3. 为什么不使用GLUT进行开发操作呢？GLUT版本相对较老，而想使用opengl的core渲染模式，显然glfw更好一些，拥有对于管线更多的理解和自由度
4. 网上针对glfw添加UI主要有两点建议
5. QT
6. dear imgui
7. 如果实在弄不明白，打算自己画正方形和鼠标输入检测当作按钮、菜单操作

于是开始检索dear imgui与glfw一起合用的可能性，看他的官方代码也很不清楚，因为他主题部分还是选择用dear imgui处理，这可能意味着得对之前的代码进行大范围重构，思路也不是很清晰。慢慢尝试之后，得到了成效，大概理清了自己实现菜单和按钮的大致操作。

1. dear imgui 使用的是弹出小窗口形式的交互界面，在网上没有很多的教程
2. 但在跨平台方面上不需要使用到mac或者windows特有的API，相对跨平台优势更大
3. 大致操作如下：
    1. 首先将他的各种文件移动到工作目录中，（它们内部互相引用）
    2. 引用需要使用的相关头文件

```
//imgui include
#include "imgui.h"
#include "imgui_impl_opengl3.h"
#include "imgui_impl_glfw.h"
```

    3. 在渲染循环外 初始化dear imgui

```
//初始化ImGui
    ImGui::CreateContext();
    ImGuiIO &io = ImGui::GetIO(); (void)io;
    ImGui::StyleColorsDark();
    ImGui_ImplGlfw_InitForOpenGL(window, true);
    ImGui_ImplOpenGL3_Init("#version 410");
```

    4. 在循环内设置dear ImGui样式

```
//设置ImGui样式
    ImGui_ImplOpenGL3_NewFrame();
    ImGui_ImplGlfw_NewFrame();
    ImGui::NewFrame();
```

5. 利用相关begin end 创建不同子窗口， 在子窗口内选择不同子控件

```
ImGui::Begin("button Setting");
    ImGui::SliderFloat("speed", &speed, 1.0f, 10000.0f);
    ImGui::Selectable("clockWide", &revovle);
    ImGui::End();
```

6. 最后调用渲染gui即可

```
//render gui
    ImGui::Render();
    ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());
```

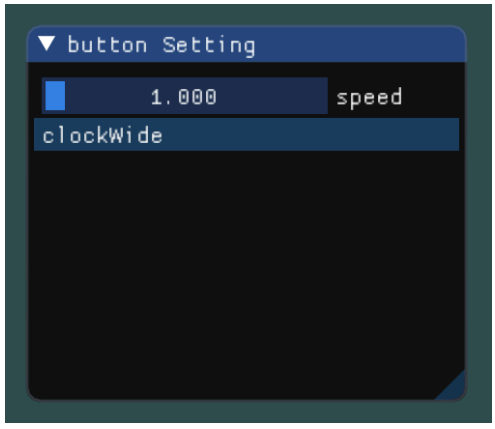
7. 具体交互方法

1. 键盘

1. W 和 S 可更改三个扇子的颜色，明亮OR暗淡
2. 上方向键 和 下方向键可实现调整风车转速
3. 空格调整风车转动方向

2. 按钮和菜单

1. 子窗口有滑动条slider调整转速 1~1000
2. clockWide按钮调整转动方向

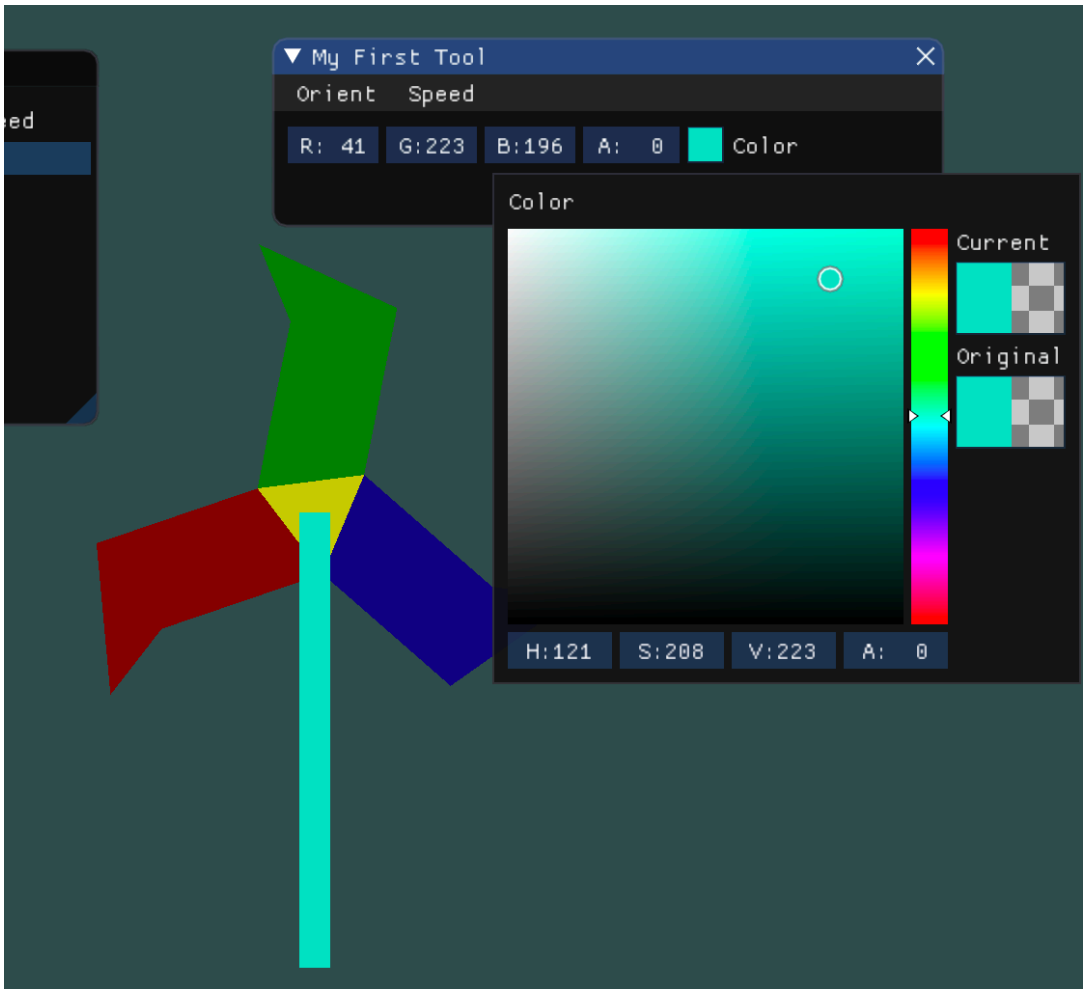


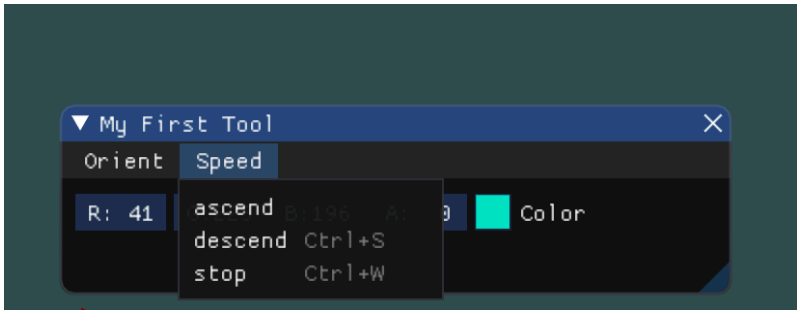
3. 子窗口2有一个彩色块调整杆子颜色，可以输入RGB值，或者调整色块

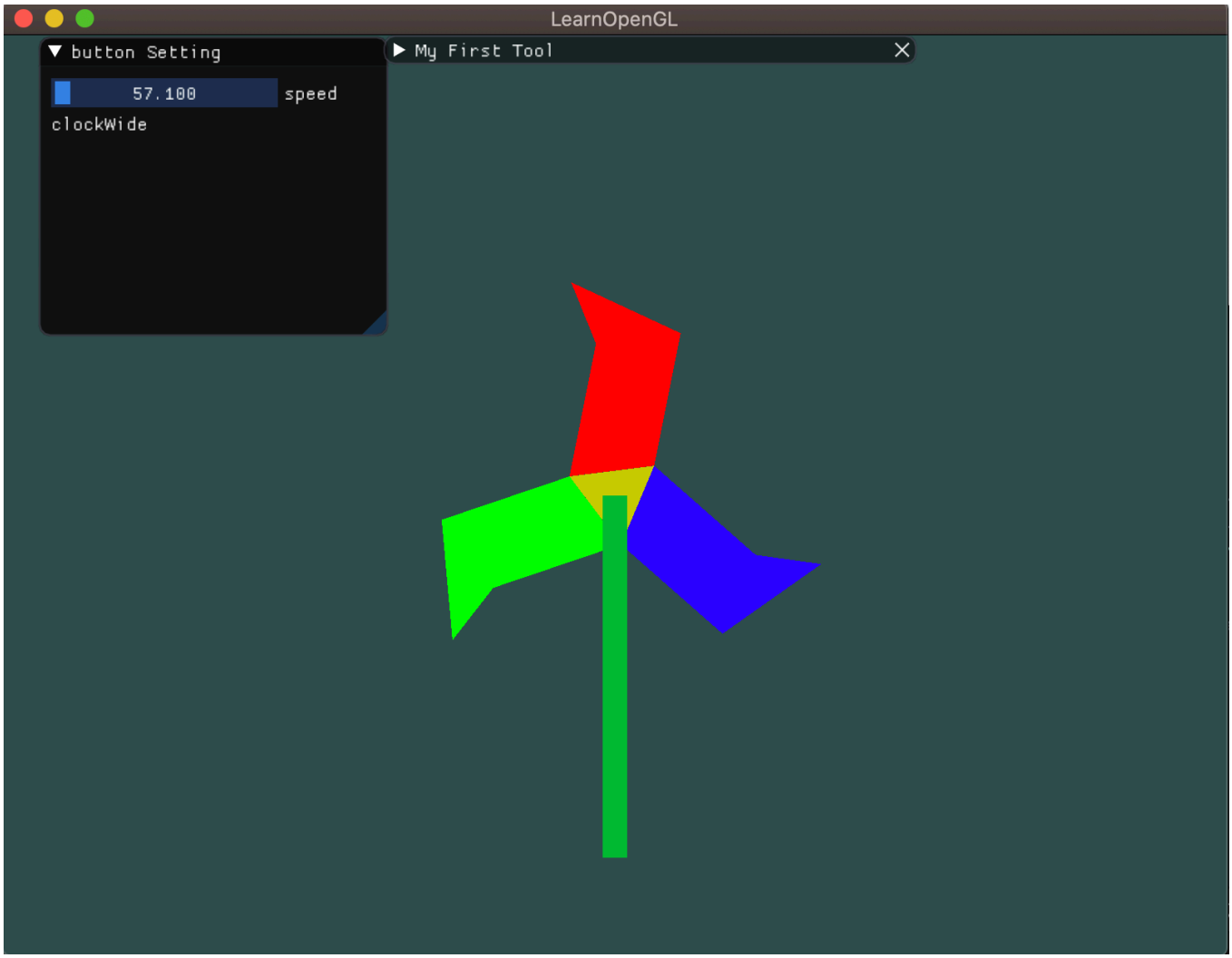
4. 有菜单项

orient	*speed*
change	*ascend*
forward	*descend*
diverse	*stop*

控制方向和速度还有停止，效果如下：






LearnOpenGL
button Setting
57.100
speed
clockWide
My First Tool