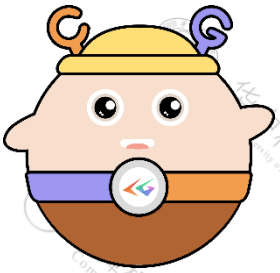
华中科技大学



计算机图形学课程

实验二：绘制四边形

目录

1 绘制四边形

在正式搭建环境之前，我们先来介绍一下读完下面的部分你会了解些什么。

- 基础绘制的过程，包括初始化，顶点输入，数据处理，着色器计算以及渲染
- 用 VAO/VBO 以及 EBO 两种方式绘制四边形

接下来，我们来介绍一下绘制四边形。绘制效果如下：

图 1 绘制四边形的效果

1.1 初始化

在代码的开始部分，我们依然对 OpenGL 进行一个初始化，其中就包括初始化 GLFW，创建窗口，初始化 GLAD，创建视口这四个部分。

```
// 初始化 GLFW
```

```
glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_RESIZABLE, FALSE);
// 创建窗口(宽、高、窗口名称)
auto window = glfwCreateWindow(screen_width, screen_height, "Quad",
nullptr, nullptr);
if (window == nullptr) {
std::cout << "Failed to Create OpenGL Context" << std::endl;
glfwTerminate();
return -1;
}
glfwMakeContextCurrent(window);
// 初始化 GLAD，加载 OpenGL 函数指针地址的函数
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}
// 指定当前视口尺寸(前两个参数为左下角位置，后两个参数是渲染窗口宽、高)
glViewport(0, 0, screen_width, screen_height);
```

1.2 顶点输入

初始化之后，我们需要给出我们四边形的顶点数据，并对数据做出一些处理，包括生成绑定 VAO、VBO 和属性设置，最后将其解绑。

```
// 四边形的顶点数据
float vertices[] = {
        // 第一个三角形
        0.5f, 0.5f, 0.0f,     // 右上


        0.5f, -0.5f, 0.0f,    // 右下
        -0.5f, -0.5f, 0.0f,   // 左下
```

```
        // 第二个三角形
        -0.5f, -0.5f, 0.0f,   // 左下
        0.5f, 0.5f, 0.0f,     // 右上
        -0.5f, 0.5f, 0.0f     // 左上
};
```

由于我们绘制的是一个四边形，因此我们的顶点数据由两组三角形组成。这里的顶点数据是标准化的设备坐标，也就是 x,y,z 轴坐标均映射到【-1,1】之间。

1.3 数据处理

1.3.1 VAO、VBO

我们有了顶点数据，接下来就是要将这些顶点数据发送到 GPU 中去处理，这里我们生成了一个顶点缓冲对象 VBO，并且将其绑定到顶点缓冲对象上，使用这个顶点缓冲对象的好处是我们不用将顶点数据一个一个的发送到显卡，而是可以借助 VBO 一次性的发送一大批数据过去，然后使用 glBufferData 将顶点数据绑定到当前默认的缓冲上，这里的 GL_STATIC_DRAW 表示我们的四边形位置数据不会被改变。

```
// 生成并绑定 VBO
GLuint vertex_buffer_object;
glGenBuffers(1, &vertex_buffer_object);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object);
// 将顶点数据绑定至当前默认的缓冲中
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
```

这里我们还生成了一个顶点数组对象 VAO，使用 VAO 的原因是首先我们使用的核心模式要求我们需要使用 VAO，其次使用 VAO 的好处在于我们在渲染的时候只需要调用一次 VAO 就可以了，之前的数据都对应存储在了 VAO 中，

不用再调用 VBO。那么 VAO 的生成过程也跟 VBO 一样，需要先生成再绑定，等到这些操作都进行完，我们可以解绑我们的 VAO，VBO。

```
// 生成并绑定 VAO
GLuint vertex_array_object;
glGenVertexArrays(1, &vertex_array_object);
glBindVertexArray(vertex_array_object);
```

发送到 GPU 之后我们还需要告诉 OpenGL 我们如何解释这些顶点数据。因此我们用 glVertexAttribPointer 这个函数告诉 OpenGL 我们如何解释这些顶点数据。

1.3.2 顶点属性

```
// 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
glEnableVertexAttribArray(0);
```

这个函数第一个参数是我们后面会用到的顶点着色器的位置值，3 表示的是顶点属性是一个三分量的向量，第三个参数表示的是我们顶点的类型，第四个是我们是否希望数据被标准化，就是映射到 0-1 之间，第五个参数叫做步长，它表示连续顶点属性之间的间隔，因为我们这里只有顶点的位置，所以我们将步长设置为这个，表示下组数据在 3 个 float 之后。最后一个是数据的偏移量，这里我们的位置属性是在数组的开头，因此这里是 0，并且由于参数类型的限制，我们需要将其进行强制类型转换。

而下面 Enable 的函数则是表明我们开启了 0 的这个通道，默认状态下是关闭的，因此我们在这里要开启一下。

等到设置属性指针完成之后，我们这里需要解绑 VAO 和 VBO。那么，我们可以思考一下，为什么我们在这里要解绑 VAO 和 VBO 呢？

一个原因是因为在防止之后再继续绑定 VAO 的时候会影响当前的 VAO，另一个原因是为了使代码更加灵活规范，在渲染需要的时候我们会再绑定 VAO。

我们已经通过 VAO、VBO 将顶点数据储存在显卡的 GPU 上了，接下来我们会创建顶点和片段着色器真正处理这些数据。这里我们会给出着色器的源码，然后生成并编译着色器，最后将顶点和片段着色器链接到一个着色器程序，在之后的渲染流程中我们会使用这个着色器程序，最后将之前的着色器删除。

1.4 顶点着色器和片段着色器

这里我们给出的这两段分别是顶点着色器的源码和片段着色器的源码，这个是用 GLSL 语言来编写的，而且这个 GLSL 语言看起来与 C 语言的风格类似，很容易懂。我们先看顶点着色器，第一行表示我们使用的是 OpenGL3.3 的核心模式，第二行就是我们之前说到的位置值。Main 函数中的部分就是将我们之前的顶点数据直接输出到 GLSL 已经定义好的一个内建变量 gl_Position 中，这个就是我们顶点着色器的输出，也就是说我们在顶点着色器这里什么都没做，就只是将顶点位置作为顶点着色器的输出。

```
// 顶点着色器源码
const char *vertex_shader_source =
    "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos, 1.0);\n"
    "}\n\0";
```

接下来是片段着色器，前面两行类似，这里的 out 表示输出变量，就像之前的 in 表示输入变量。然后我们这里的四分量向量就是我们之前看到的四边形是红色的来源，是一个四分量的 RGBA，那么我们也可以将其更改一下，我们输出的四边形颜色就会发生变化。

```
// 片段着色器源码
const char *fragment_shader_source =
        "#version 330 core\n"
        "out vec4 FragColor;\n"
        "void main()\n"
        "{\n"
```

```
"    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n\0";
```

有了顶点和片段着色器的源码，我们还需要生成和编译着色器，那么这里我们先生成了顶点着色器，并且附加上了之前的源码，将其进行编译，然后我们这里对其进行检测，是否成功编译，如果编译不成功就打印错误信息。

同样的片段着色器也是一样的。

最后我们将顶点和片段着色器链接到一个着色器程序中，这样我们在渲染时只需要调用一个着色器程序就可以了，同样这里我们检测了一下链接是否成功。

最后删除掉顶点和片段着色器。因为在后面渲染的时候我们只需要用那个我们之前链接好的着色器程序就可以了，不需要再使用顶点和片段着色器了。

1.5 渲染

接下来我们进入我们的渲染阶段，当窗口没有关闭的时候我们就一直进行渲染。首先我们先清空颜色缓冲，我们这里用的是黑色的背景色来清空屏幕颜色缓冲，当然这里我们可以更换颜色。接下来我们使用我们之前已经链接好的着色器程序，和 VAO，来绘制四边形，绘制四边形其实只要一句话，就是这个 glDrawArrays，当然还有其他的绘制方式，我们这里用的是这种方式。这里的第一个参数表示我们是要绘制四边形，第二个参数表示我们顶点数组的起始索引值，第三个参数表示我们要绘制的顶点数量，这里绘制四边形我们要绘制三个顶点。绘制结束后解除绑定。最后我们会交换一下缓冲，这里我们使用的是一个双缓冲的做法，前缓冲保存着输出的图像，而渲染指令都在后缓冲中进行，当指令执行完毕后我们交换前后缓冲，最后我们还会检测是否有触发一些回调函数。

```
// 使用着色器程序
glUseProgram(shader_program);
// 绘制四边形
glBindVertexArray(vertex_array_object);
```

```
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindVertexArray(0);
```

当我们的窗口关闭之后，我们还会进行一些善后工作，这里包括删除我们之前所创建的 VAO、VBO，以及调用 GLFW 的函数来清理所有的资源并退出程序。

整个绘制四边形的程序就到此为止。

我们回头再来梳理一遍绘制四边形流程，刚开始我们先初始化 OpenGL，接下来对我们的数据进行处理，通过 VAO、VBO 将其发送至 GPU 中，并设置属性指针告诉 GPU 我们会如何解释这些数据，然后在着色器中通过顶点和片段着色器对数据进行处理，最后进行渲染，渲染之后做好我们的善后工作，一个四边形就绘制成功了。效果如下：

图 5 VAO/VBO 绘制四边形

1.6 EBO

细心的人可能已经看出了些问题，左下和右上角的顶点我们使用了两次，一个矩形只有四个顶点，如果是大量的复杂模型计算就会产生很多浪费。如何

解决这个问题呢？其实我们只要存储矩形的四个顶点，然后指定绘制顺序就好了，EBO 帮助我们实现了这个功能。

我们采用了一种索引绘制的方式，首先要定义不重复的顶点

```
// 四边形的顶点数据
float vertices[] = {
        // 第一个三角形
        0.5f, 0.5f, 0.0f,     // 右上
        0.5f, -0.5f, 0.0f,    // 右下
        -0.5f, -0.5f, 0.0f,   // 左下
        -0.5f, 0.5f, 0.0f     // 左上
};
// 索引数据(注意这里是从 0 开始的)
unsigned int indices[] = {
        0, 1, 3,              // 第一个三角形
        1, 2, 3               // 第二个三角形
};
```

接下来我们要创建索引缓冲对象，这里同 VBO 类似。

```
GLuint element_buffer_object; // == EBO
glGenBuffers(1, &element_buffer_object);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
```

我们先绑定 EBO 然后用 glBufferData 把索引复制到缓冲里。这次我们把缓冲的类型定义为 GL_ELEMENT_ARRAY_BUFFER。

运行程序则会显示如下的结果。

图 6-1 EBO 绘制四边形(填充模式)

开启线框模式如下：

```
// 线框模式
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

其中第一个参数表示我们应用到所有三角形的正面和背面，第二个参数则是我们选择用线来绘制还是使用填充模式（GL_FILL）来设置回默认模式。

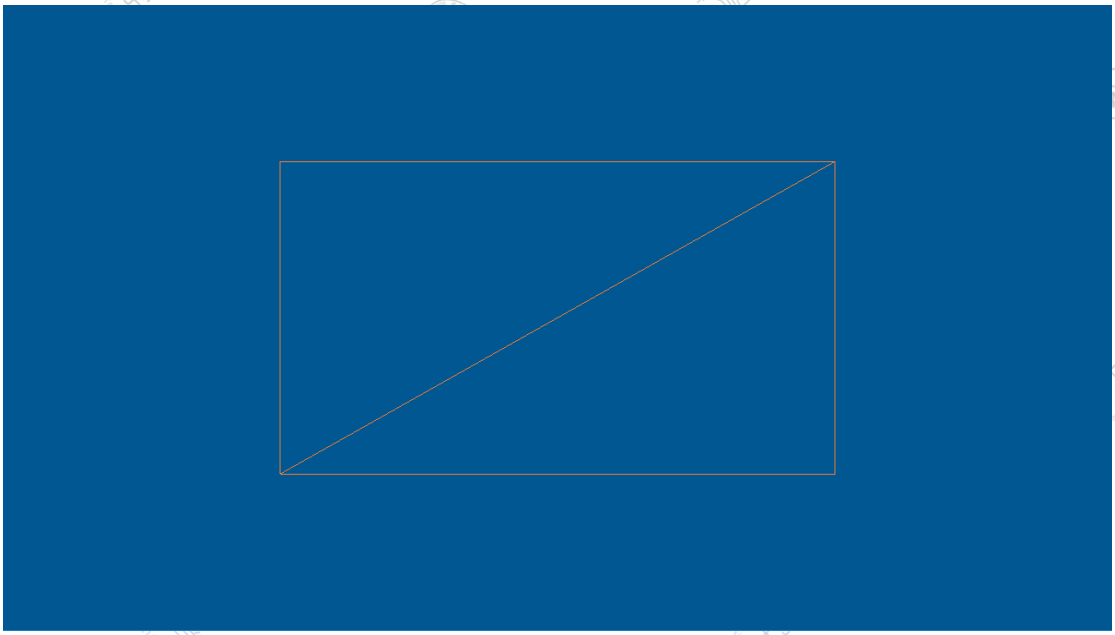


图 6-2 绘制四边形(线框模式)