计算机图形学

Introduction to OpenGL

Assignment 0

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 作业任务与要求 **2**

2 Job1：解答 **2**

3 Job2：实验过程与结果 **2**

3.1 “你好，窗口” 的实现 2

3.2 “你好，三角形” 的实现 4

3.3 “着色器” 的实现 7

4 Job3：纹理实验过程与结果 **11**

4.1 纹理的实现 11

4.2 纹理单元 14

4.3 不同的纹理环绕方式 15

4.4 在箱子的角落放置 4 个笑脸 18

5 Job4：学习感想 **19**

1 作业任务与要求

(1) Job1：回答什么是 OpenGL，以及 OpenGL 与图形学之间的关系；

(2) Job2：实现教程中的 “你好，窗口”，“你好，三角形” 和 “着色器”；

(3) Job3：学习教程中的 “纹理”，尝试用不同的纹理环绕方式，设定一个从 0.0f 到 2.0f 范围内的（而不是原来的 0.0f 到 1.0f）纹理坐标。试试看能不能在箱子的角落放置 4 个笑脸。再试试其它的环绕方式。简述原因并贴上结果；

(4) Job4：谈谈学习这些教程章节的感想；

2 Job1：解答

OpenGL（Open Graphics Library，开放图形库）是一种用于渲染 2D 和 3D 矢量图形的跨语言、跨平台的 API（应用程序编程接口）。OpenGL 包含了一系列可以操作图形和图像的函数。然而，OpenGL 本身并不是一个 API，而是一个由 Khronos 组织指定并维护的规范。规范严格规定了每个函数应当如何执行、输出，但具体实现由开发者自行决定。实际的 OpenGL 库的开发者通常是显卡的生产商，购买者使用的显卡所支持的 OpenGL 库一般由他们专门开发。

OpenGL 和计算机图形学的关系是什么呢？在国内知识问答平台知乎上，用户 vczh 给出了这样的回答：“（计算机图形学和 OpenGL）分别是中国传媒大学和摄像机的关系。”[1] 计算机图形学作为一门学科，是一种是一种使用数学算法将二维或三维图形转化为计算机显示器的栅格形式的科学。简单地说，计算机图形学的主要研究内容就是研究如何在计算机中表示图形、以及利用计算机进行图形的计算、处理和显示的相关原理与算法。计算机图形学包含三部分内容：建模（modeling），渲染（rendering）和动画（animation），一个三维数字虚拟世界的构建和创造基本都由上述过程实现。而 OpenGL 是一种应用计算机图形学、实现了有关知识落地的软件，是个专业的图形程序接口。开发者可以通过调用 OpenGL 实现三维图像在计算机上的渲染与显示。总的来说，计算机图形学是一门包含各种知识的学科，而 OpenGL 是图形学有关内容的具体实现。

3 Job2：实验过程与结果

3.1 “你好，窗口” 的实现

窗口的显示包括以下步骤：

(1) 实例化 GLFW 窗口；

(2) 创建窗口对象；

(3) 初始化 GLAD；

(4) 设置窗口维度，注册回调函数；

[1] https://www.zhihu.com/question/66848391?sort=created

(5) 实现输入控制；

(5) 实现渲染循环；

(6) 退出时释放/删除分配的资源；

关键代码如下所示：

```
int main()
{
    // 实例化GLFW窗口
    glfwInit();// 初始化
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);// 配置GLFW：主版本号
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);// 配置GLFW：此版本号
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);// 配置GLFW：核心模式

    // 创建一个窗口对象
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "18308013", NULL, NULL);
    glfwMakeContextCurrent(window);
    // 注册回调函数
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
    // 初始化GLAD
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }
    // 渲染循环
    while (!glfwWindowShouldClose(window))
    {
        //输入
        processInput(window);
        glfwSwapBuffers(window);// 交换颜色缓冲
        glfwPollEvents();// 检查是否触发事件、更新窗口状态，调用对应的回调函数
    }
    // 释放/删除之前分配的所有资源
    glfwTerminate();
    return 0;
}
```

实现结果如图1所示。可见我们实现了一个无内容窗口的显示。

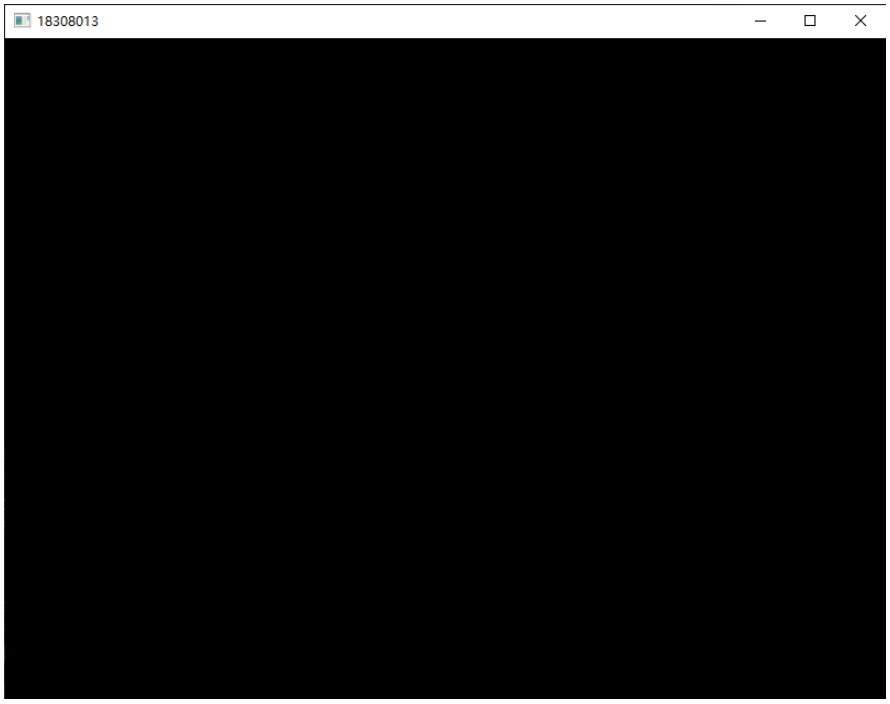

图 1: “你好，窗口” 实现结果 1

此外，在渲染循环中，我们可以使用 glClearColor() 函数设置清空屏幕的颜色，glClear() 清空颜色缓冲。实现结果如图2所示。

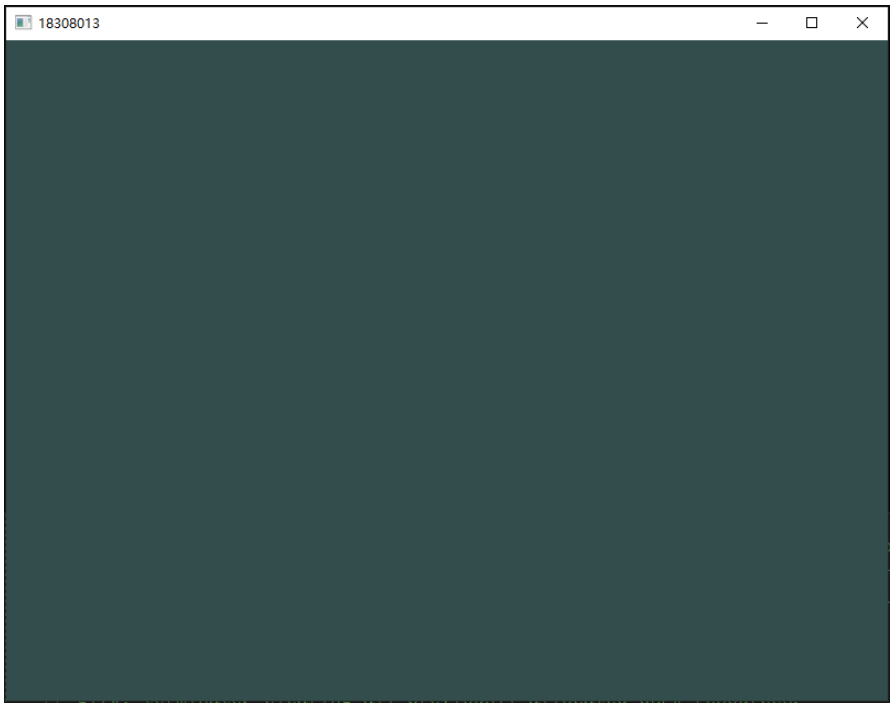

图 2: “你好，窗口” 实现结果 2

3.2 “你好，三角形” 的实现

接下来我们尝试在窗口中绘制一个三角形。在 OpenGL 中，任何事物都在 3D 空间中，而屏幕和窗口都是 2D 像素数组。3D 坐标转为 2D 坐标的处理过程是由图形渲染管线完成的。图形渲染管线接受一组 3D 坐标，然后把它们转变为屏幕上的有色 2D 像素输出。

绘制一个三角形的大致流程如下：

(1) 输入顶点数据；

(2) 生成 VAO（顶点数组对象）和 VBO（顶点缓冲对象），分别绑定，设置顶点数据指针；

(3) 创建和编译 Vertex Shader（顶点着色器）；

(4) 创建和编译 Fragment Shader（片段着色器）；

(5) 将顶点着色器和片段着色器链接，生成着色器程序；

(5) 在渲染循环中使用当前激活的着色器、顶点属性配置和顶点数据绘制图元；

关键代码如下所示：

```
{
    ......
    ......
    // 顶点着色器
    int vertexShader = glCreateShader(GL_VERTEX_SHADER);// 创建顶点着色器对象
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);// 将着色器源码附加到着色器
        对象
    glCompileShader(vertexShader);// 编译着色器源码
    // 片段着色器
    int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);// 创建片段着色器对象
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);// 将着色器源码附加到着
        色器对象
    glCompileShader(fragmentShader);// 编译着色器源码
    // 着色器程序
    int shaderProgram = glCreateProgram();// 创建程序对象，返回程序对象的ID引用
    glAttachShader(shaderProgram, vertexShader);// 附加顶点着色器
    glAttachShader(shaderProgram, fragmentShader);// 附加片段着色器
    glLinkProgram(shaderProgram);// 链接着色器
    glDeleteShader(vertexShader);// 删除顶点着色器对象
    glDeleteShader(fragmentShader);// 删除片段着色器对象
    // 三角形三个顶点的3D位置
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,
         0.5f, -0.5f, 0.0f,
         0.0f,  0.5f, 0.0f
    };
    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);// 创建VAO对象（顶点数组对象）
    glGenBuffers(1, &VBO);// 使用缓冲ID生成VBO对象（顶点缓冲对象）
    // #0: 绑定VAO
    glBindVertexArray(VAO);
    // #1: 复制顶点数组到缓冲中供OpenGL使用
    glBindBuffer(GL_ARRAY_BUFFER, VBO);// 把新创建的缓冲绑定到GL_ARRAY_BUFFER上
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);// 把之前定
        义的顶点数据复制到缓冲的内存中
```

```
    // #2: 设置顶点属性指针
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);// 解析顶点数据的方式
    glEnableVertexAttribArray(0);// 启用顶点属性
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

    while (!glfwWindowShouldClose(window)){
        // 输入
        processInput(window);
        // 渲染
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        // #3: 绘制物体
        glUseProgram(shaderProgram);// 激活程序对象
        glBindVertexArray(VAO);
        glDrawArrays(GL_TRIANGLES, 0, 3);// 使用当前激活的着色器、顶点属性配置和VBO的顶点数据绘制图元
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    ......
    ......
}
```

实现结果如图3所示：

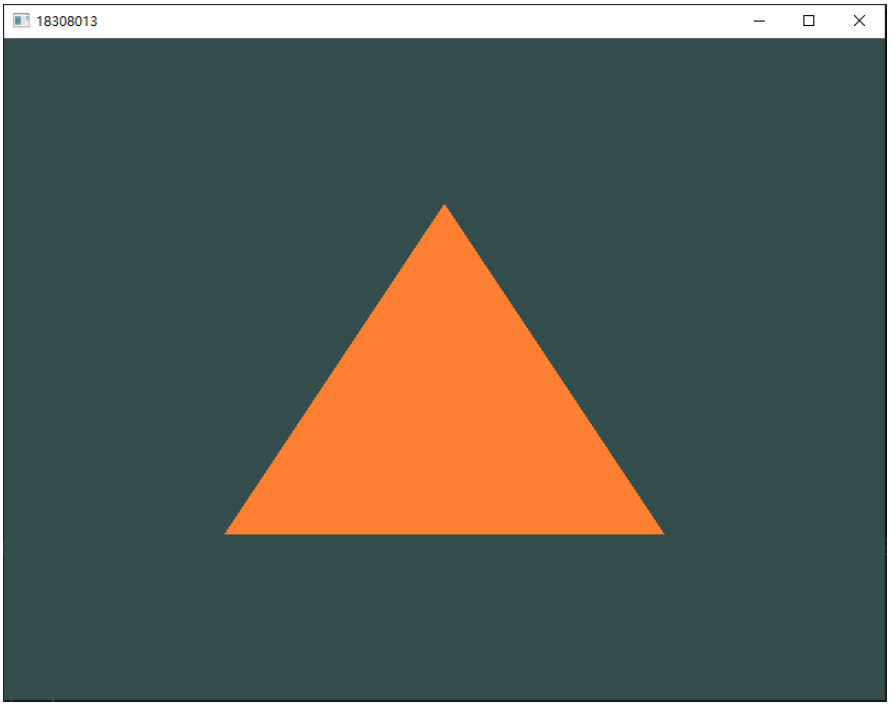

图 3: “你好，三角形” 实现结果 1

同样，我们可以创建 EBO（索引缓冲对象），将绘制顶点的索引复制到缓冲中，从而实现多个顶点的绘制顺序。我们用其通过绘制两个三角形构成一个矩形，实现结果如图4所示。启用线框模式，效果如

图5所示，显然该矩形由两个三角形拼接而成。

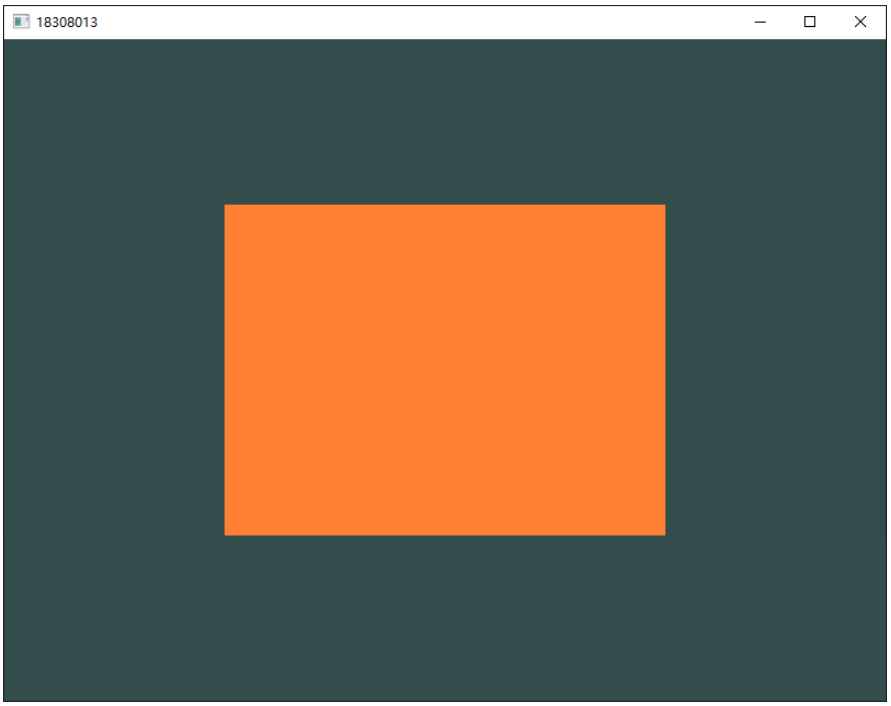

图 4: “你好，三角形” 实现结果 2

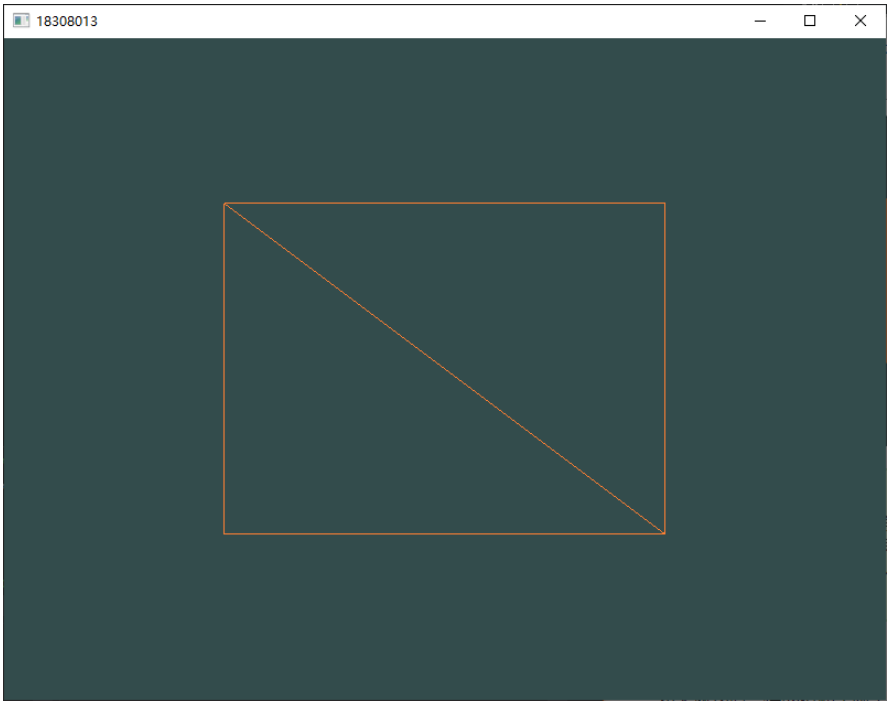

图 5: “你好，三角形” 实现结果 3

3.3 “着色器” 的实现

着色器是驻留在 GPU 上的小程序。这些程序针对图形管线的每个特定部分运行。其由 GLSL 语言编写。着色器虽然相互独立，但都是一个整体的一部分。每个着色器都有输入和输出，上一着色器阶段的输出应与下一阶段的着色器阶段的输入相匹配。顶点着色器从顶点数据中直接接收输入，而片段着色器需要生成一个最终输出的颜色，故需要一个 vec4 颜色输出变量。

我们可以通过修改着色器实现颜色的变化。例如，我们在顶点着色器定义一个颜色，将数据发送给片段着色器实现。顶点着色器代码如下：

```
#version 330 core
layout (location = 0) in vec3 aPos;
out vec4 vertexColor;
void main()
{
    gl_Position = vec4(aPos, 1.0);
    vertexColor = vec4(0.5,0.0,0.0,1.0);
}
```

片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;
in vec4 vertexColor;
void main()
{
    FragColor = vertexColor;
}
```

实现结果如图6所示。

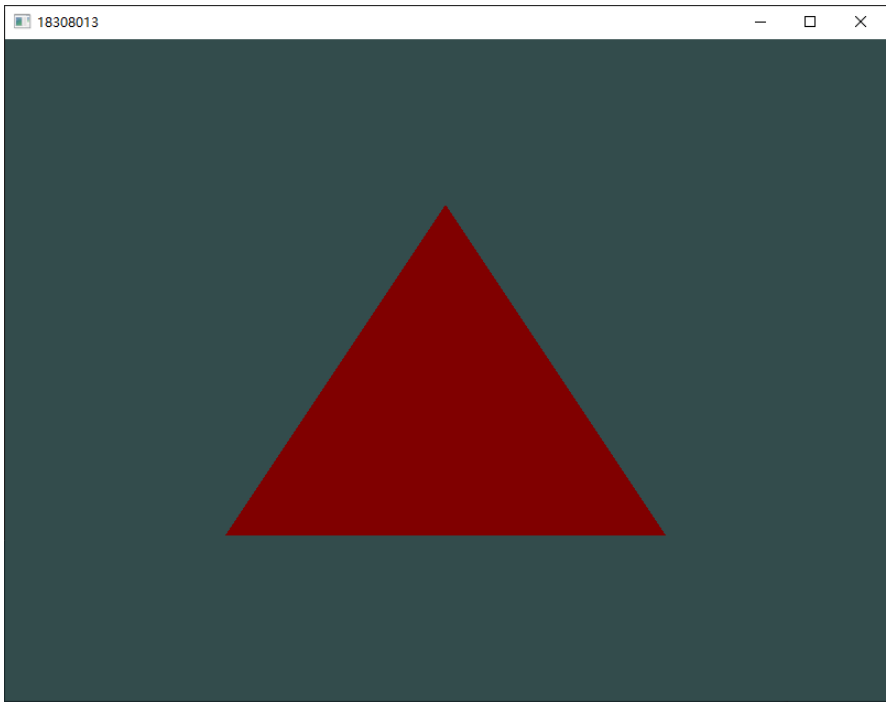

图 6: “着色器” 实现结果 1

我们也可以使用 uniform，实现从 CPU 中的应用向 GPU 中的着色器发送数据。这样的话，我们就可以实现在程序中实时改变显示的颜色。顶点着色器代码如下：

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
```

```
{
    gl_Position = vec4(aPos, 1.0);
}
```

片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;
uniform vec4 ourColor;
void main()
{
   FragColor = ourColor;
}
```

在渲染循环中，每次循环对 uniform 值进行更新：

```
while (!glfwWindowShouldClose(window))
{
    // 输入
    processInput(window);
    // 渲染
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    // 激活着色器
    glUseProgram(shaderProgram);
    // 更新uniform颜色
    float timeValue = glfwGetTime();// 获取运行的秒数
    float greenValue = sin(timeValue) / 2.0f + 0.5f;// 让颜色在0.0到1.0之间改变
    int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");// 查询
        uniform ourColor的位置
    glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);// 设置uniform值
    // 绘制三角形
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

最终实现结果如图7所示。三角形随着时间变化由绿变黑再变绿。

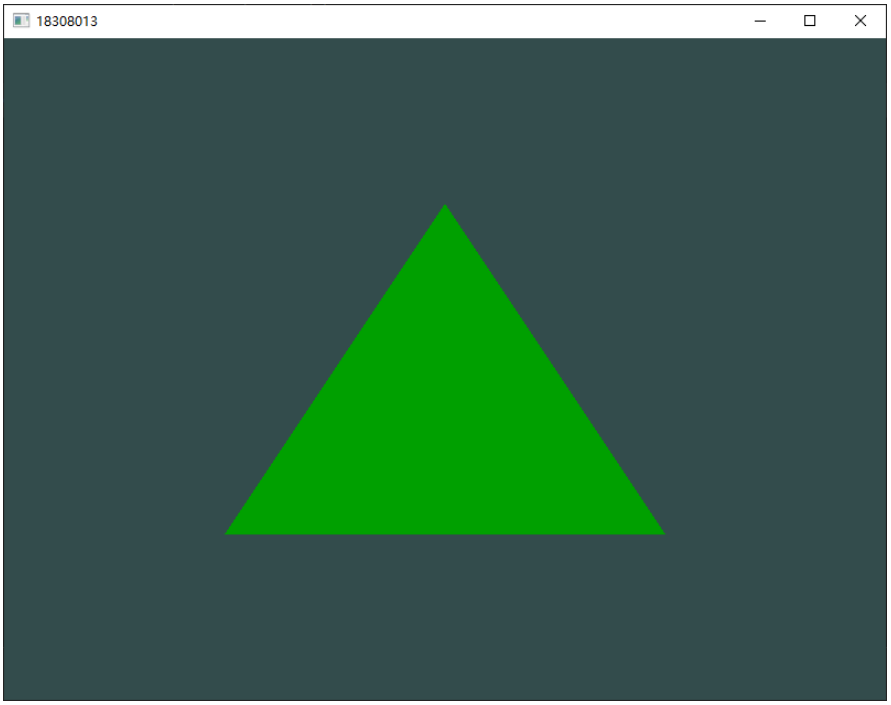

图 7: “着色器” 实现结果 2

除了在顶点着色器/片段着色器定义颜色、通过 uniform 变量从当前运行程序向 GPU 着色器传递数据，我们还有第三种方式实现颜色的定义，就是在顶点数据中定义颜色。对此，在顶点数组大小增加的同时，我们还要修改顶点着色器、片段着色器和顶点属性指针，以适配新的顶点数据格式。最终实现结果如图8所示。

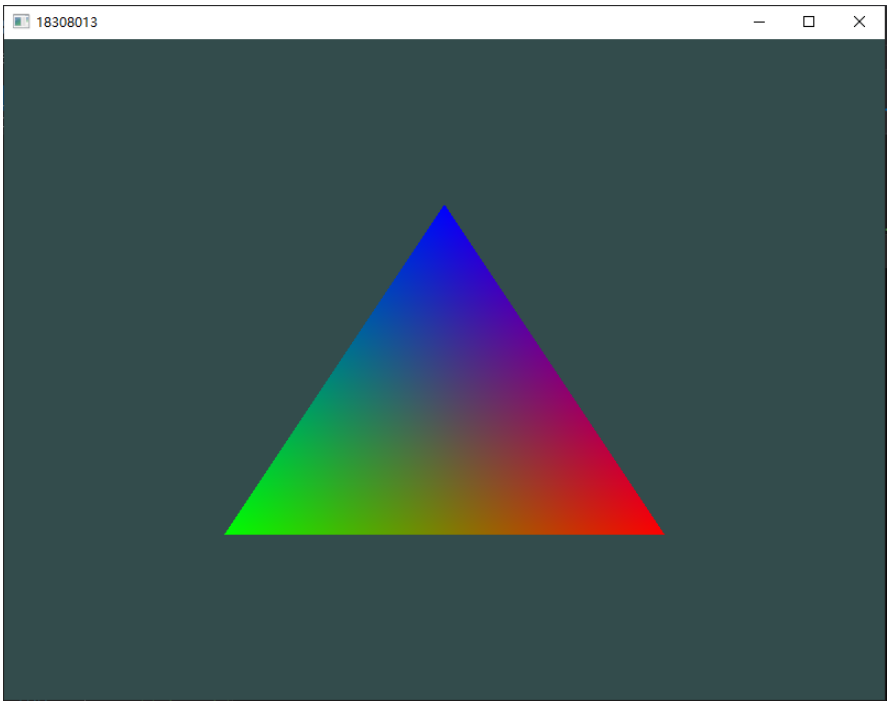

图 8: “着色器” 实现结果 3

4 Job3：纹理实验过程与结果

4.1 纹理的实现

纹理是一个 2D 图片（也有 1D 和 3D 的纹理）。我们可以将其附加在 OpenGL 中创建的图形上，从而增加物体细节。为了能够把纹理映射到图形上，我们需要为每个顶点配置一个纹理坐标，用来指明从纹理图像的哪个部分进行采用，然后在图形其他片段上进行片段插值。

纹理坐标的范围通常是 (0,0) 到 (1,1)。当纹理坐标超过默认范围时，OpenGL 提供了不同的环绕方式任君选择。

为物体图形使用纹理的过程大致如下:

(1) 使用 stb_image.h 库的函数加载纹理图片；

(2) 创建、绑定纹理；

(3) 为当前绑定的纹理对象设置环绕、过滤方式；

(4) 使用纹理图片数据生成纹理；

(5) 如需要，生成多级渐远纹理；

(5) 在顶点数据中设置纹理坐标；

(6) 修改顶点着色器和片段着色器，使得着色器能够接收纹理坐标的同时，可以通过纹理采样器访问纹理，对纹理颜色采样、输出；

(7) 渲染循环中，在 glDrawElements 前绑定纹理，纹理会被自动赋值给采样器；

实现代码如下：

```
int main(){
    ......
    // 顶点数组
    float vertices[] = {
        // 位置                 // 颜色                 // 纹理坐标
         0.5f,  0.5f, 0.0f,    1.0f, 0.0f, 0.0f,    1.0f, 1.0f, // 右上角
         0.5f, -0.5f, 0.0f,    0.0f, 1.0f, 0.0f,    1.0f, 0.0f, // 右下角
        -0.5f, -0.5f, 0.0f,    0.0f, 0.0f, 1.0f,    0.0f, 0.0f, // 左下角
        -0.5f,  0.5f, 0.0f,    1.0f, 1.0f, 0.0f,    0.0f, 1.0f  // 左上角
    };
    ......
    // 设置顶点格式
    // 顶点坐标位置
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // 颜色设置位置
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(
        float)));
    glEnableVertexAttribArray(1);
```

```
    // 纹理坐标位置
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(
        float)));
    glEnableVertexAttribArray(2);
    // 生成纹理
    unsigned int texture;
    glGenTextures(1, &texture);// 创建纹理
    glBindTexture(GL_TEXTURE_2D, texture); // 绑定纹理
    // 为当前绑定的纹理对象设置环绕方式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // 为当前绑定的纹理对象设置过滤方式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    int width, height, nrChannels;
    // 加载图片
    unsigned char* data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
    // 生成纹理
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
        data);
    // 自动生成多级渐远纹理
    glGenerateMipmap(GL_TEXTURE_2D);
    stbi_image_free(data);
    // 渲染循环
    while (!glfwWindowShouldClose(window)){
        ......
        glBindTexture(GL_TEXTURE_2D, texture);// 绑定纹理
        ourShader.use();// 激活着色器程序
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
        ......
    }
    ......
}
```

顶点着色器代码如下：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;
out vec3 ourColor;
out vec2 TexCoord;
void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
void main()
{
    FragColor = texture(texture1, TexCoord);
}
```

最终实现效果如图9所示。

图 9: “纹理” 实现结果 1

我们也可以在片段着色器中实现顶点颜色和纹理颜色的混合，产生出奇异的效果，实现效果如图10所示。

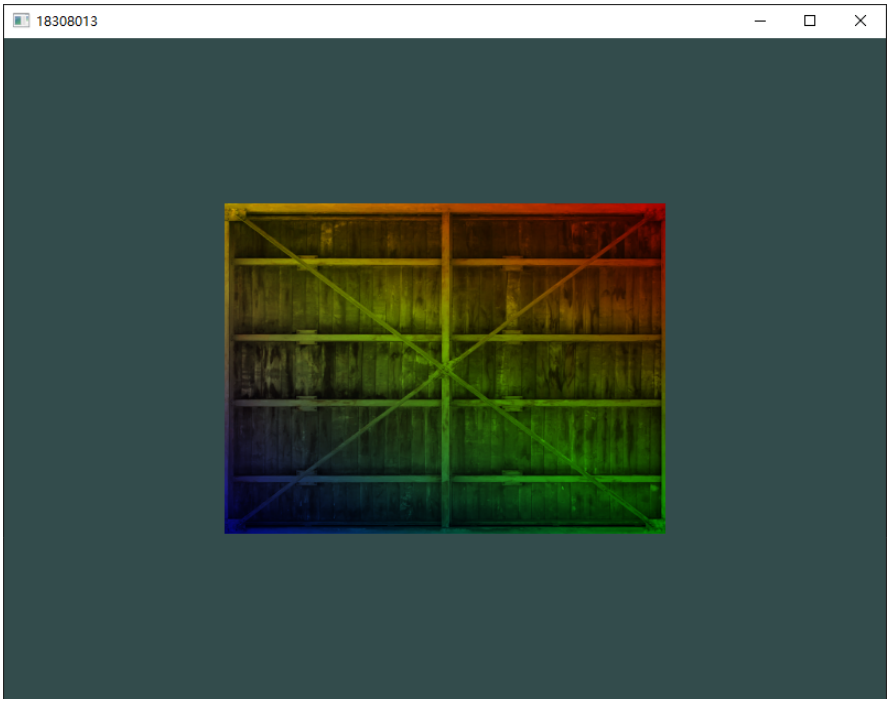

图 10: “纹理” 实现结果 2

4.2 纹理单元

如果要实现多个纹理在着色器中使用，我们就需要纹理单元。纹理单元的主要目的是让我们在着色器中可以使用多于一个的纹理。使用纹理单元的过程如下：

(1) 创建、绑定、生成多个纹理；

(2) 设置采样器，使得片段着色器的每个采样器对应一个纹理单元；

(3) 在渲染循环中激活纹理单元，分别绑定不同的纹理到不同的纹理单元；

关键代码如下所示：

```
int main(){
    ......
    ourShader.use(); // 激活着色器程序
    glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0);// 手动设置采样器
        texture1对应纹理单元0
    ourShader.setInt("texture2", 1);// 使用着色器类设置采样器2对应纹理单元1
    while (!glfwWindowShouldClose(window))
    {
        ......
        glActiveTexture(GL_TEXTURE0);// 激活纹理单元0（默认已激活）
        glBindTexture(GL_TEXTURE_2D, texture1);// 绑定纹理到纹理单元0
        glActiveTexture(GL_TEXTURE1);// 激活纹理单元1
        glBindTexture(GL_TEXTURE_2D, texture2);// 绑定纹理到纹理单元1
        ourShader.use();// 激活着色器程序
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

```
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    ......
}
```

片段着色器代码如下所示：

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
uniform sampler2D texture2;
void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

实现结果如图11所示。

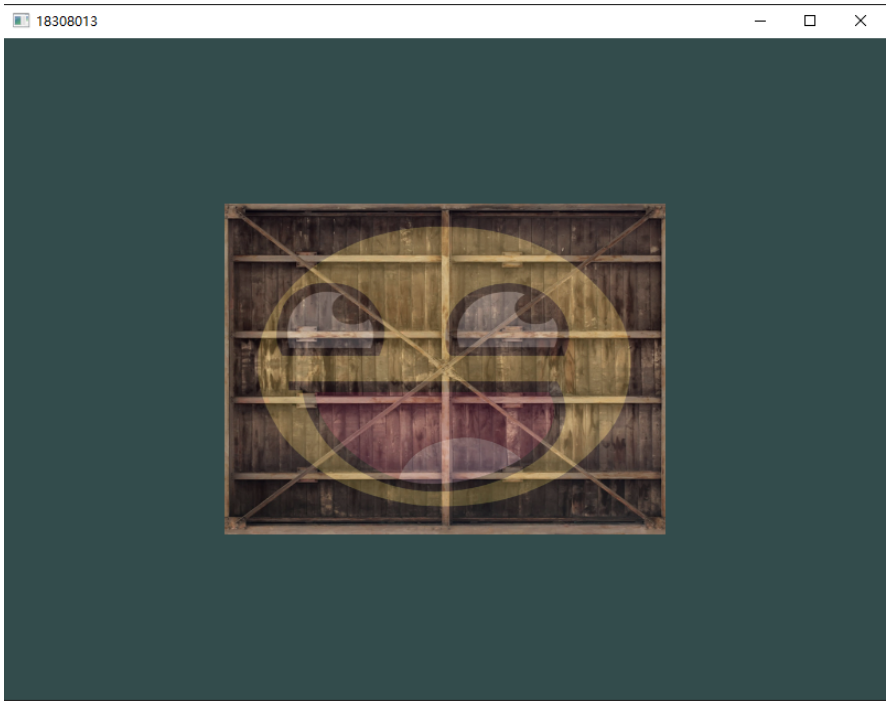

图 11: “纹理” 实现结果 3

4.3 不同的纹理环绕方式

纹理的环绕方式有四种，如表1所示。

环绕方式	描述
GL_REPEAT	对纹理的默认行为。重复纹理图像。
GL_MIRRORED_REPEAT	同 GL_REPEAT 一致，但镜像重复。
GL_CLAMP_TO_EDGE	超出坐标重复纹理坐标的边缘，产生拉伸效果。
GL_CLAMP_TO_BORDER	超出坐标为用户指定的边缘颜色。

表 1: 环绕方式

对箱子纹理和笑脸纹理的 S、T 方向都使用 GL_REPEAT，设定从 0.0f 到 2.0f 范围内的纹理坐标。效果如图12所示。

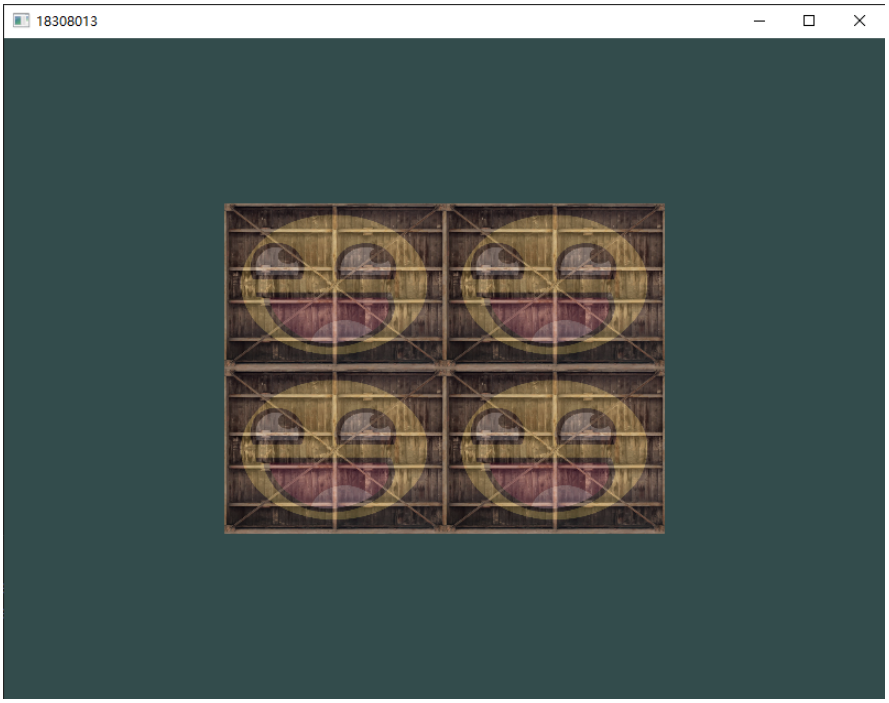

图 12: 不同的环绕方式 1

可见，因为使用了 GL_REPEAT，箱子纹理和笑脸纹理在 X、Y 轴坐标大于 1.0f 时均出现直接重复。

对笑脸纹理的 S、T 方向使用 GL_MIRRORED_REPEAT。效果如图13所示。

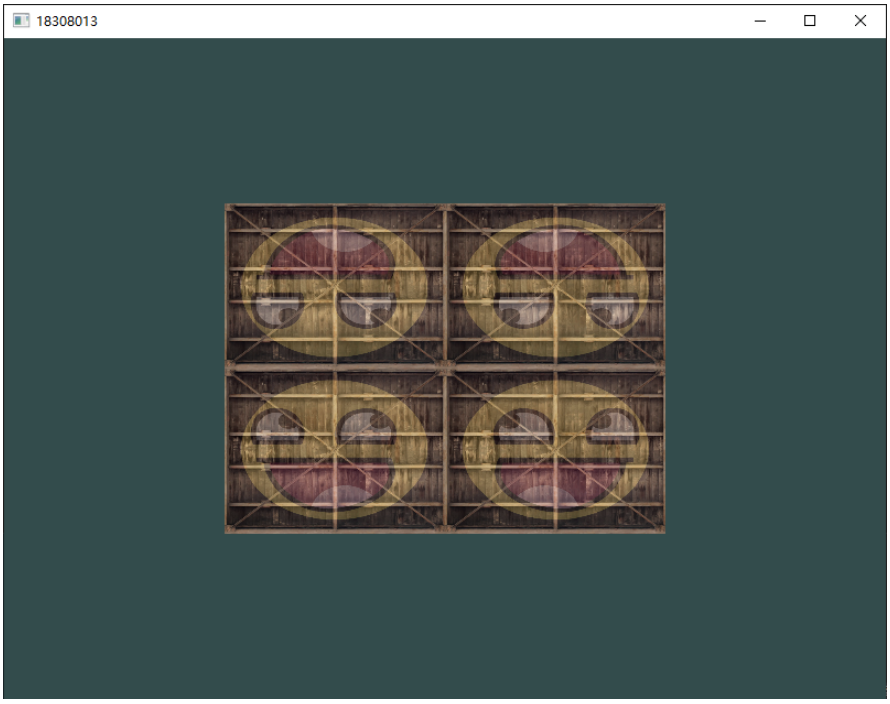

图 13: 不同的环绕方式 2

可见，因为使用了 GL_MIRRORED_REPEAT，上方的笑脸纹理因为 Y 轴方向超过标定值，故在 Y 轴方向镜像反转。右侧的笑脸纹理则在 X 轴镜像反转。右上方的笑脸纹理在 X、Y 轴均超过标定值，故在 X、Y 轴均镜像反转。

对箱子纹理的 S、T 方向使用 GL_CLAMP_TO_EDGE，效果如图14所示。

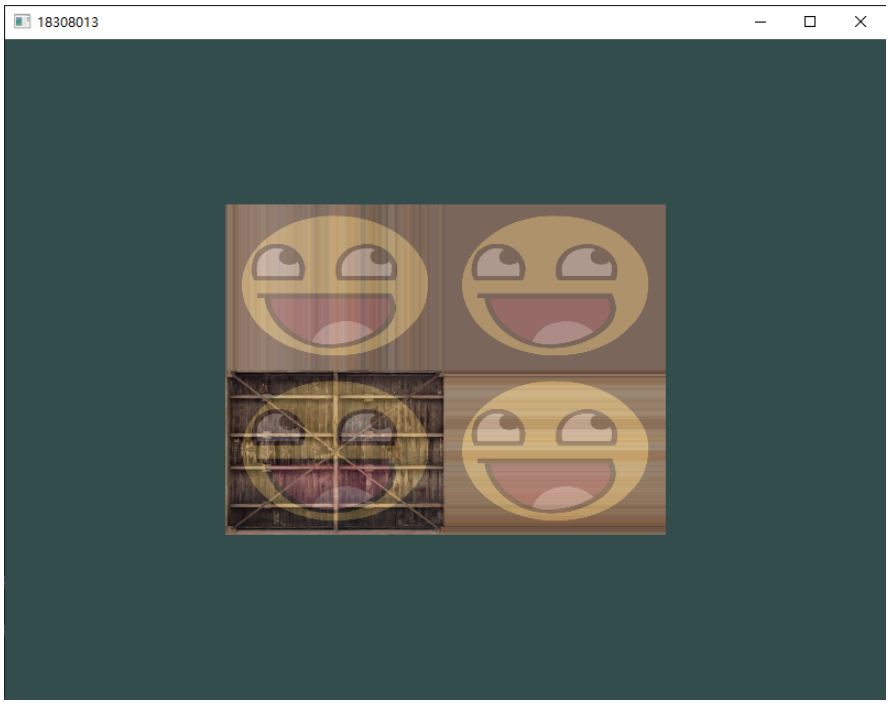

图 14: 不同的环绕方式 3

可见上方图像和右方图像均是对箱子纹理的上方边缘纹理和右方边缘纹理进行拉伸。而右上方图像是对右上角箱子纹理进行拉伸。

对箱子纹理的 S、T 方向使用 GL_CLAMP_TO_BORDER，效果如图15所示。

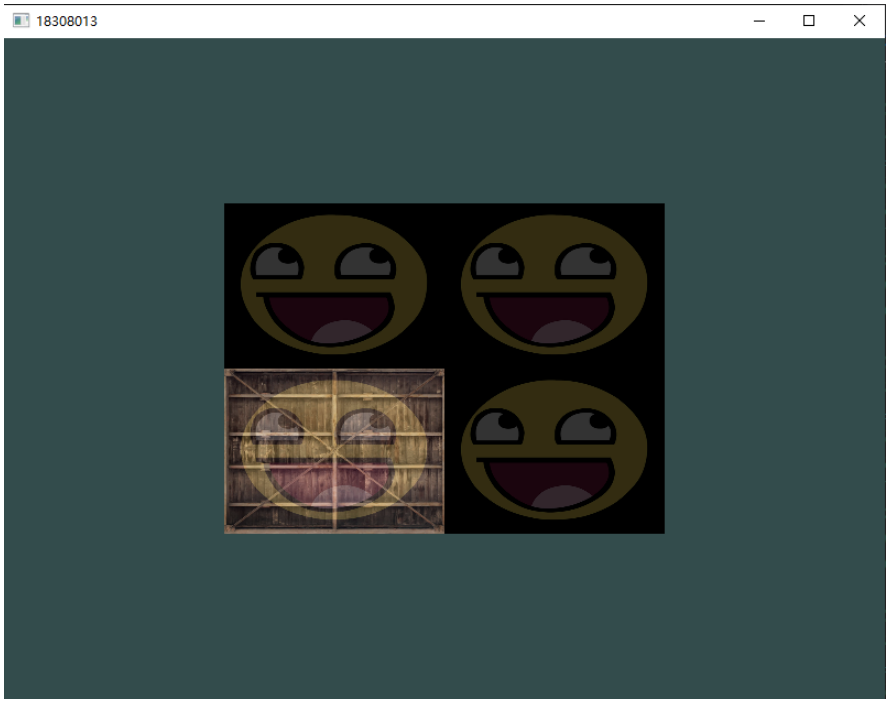

图 15: 不同的环绕方式 4

可见，上方、右方和右上方的箱子纹理均变成黑色，这是因为超出标定值的范围自动填充指定默认颜色。

当然，我们也可以在 S、T 方向上使用不同的环绕方式，例如在箱子的 S 方向上使用 GL_REPEAT，在 T 方向上使用 GL_CLAMP_TO_EDGE，效果如图16所示。

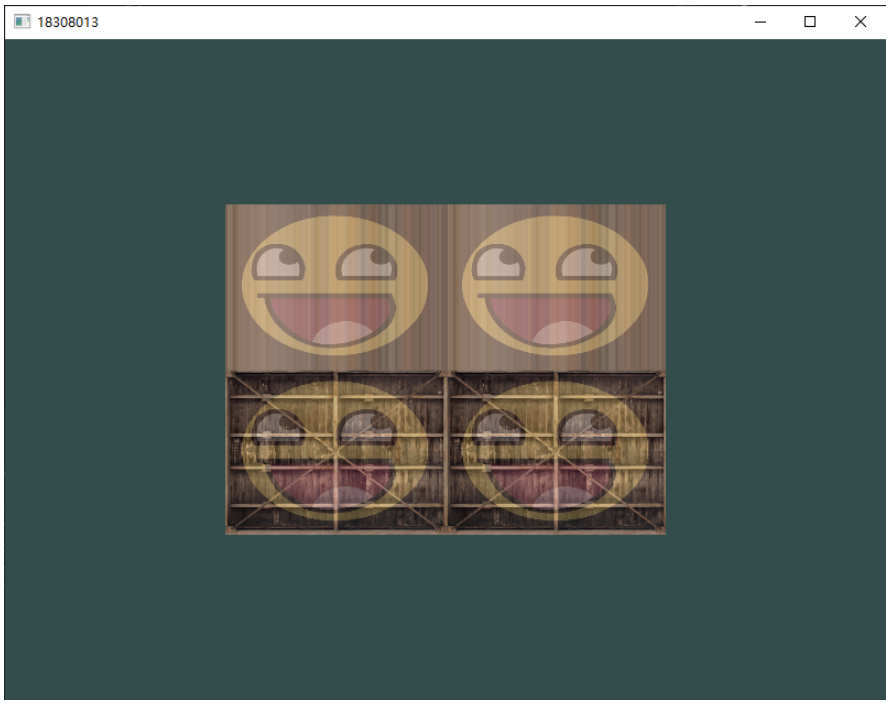

图 16: 不同的环绕方式 5

4.4 在箱子的角落放置 4 个笑脸

对于这个问题，有两种理解方式：

(1) 通过 GL_REPEAT 方式在超出标准范围的箱子角落放置笑脸；

(2) 在 0.0f 到 1.0f 标准范围内的箱子角落放置笑脸；

对于理解方式 1，我们已在4.3实现，见图12。故我们尝试实现方式 2。一开始我的思路是使用 9 个顶点构成 8 个三角形来形成 4 个小矩形，然后四个笑脸纹理分别贴到四个小矩形；最后再将箱子纹理贴到四个小矩形拼接成的大矩形上。后来我发现我太菜了做不到……于是尝试从片段着色器入手。首先将矩形的四个顶点的纹理坐标设置回标准范围，然后设置笑脸纹理为 GL_REPEAT 环绕模式。在片段着色器中，对箱子纹理我们还是使用原始纹理坐标，而对笑脸纹理我们乘 2，这样就可以使得笑脸纹理重复图像。片段着色器代码如下：

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
uniform sampler2D texture2;
void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord*2), 0.2);
}
```

最终实现效果如图17所示。

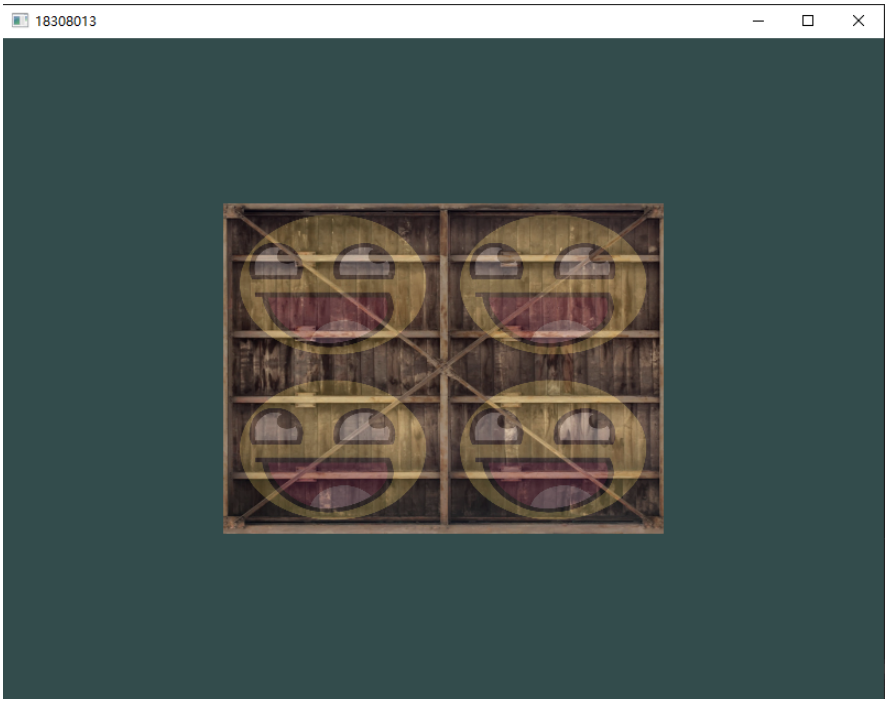

图 17: 箱子的 4 个角落放置笑脸

5 Job4：学习感想

本次作业任务量较大，要求我们阅读 OpenGL 教程，并实现前面的几个工作。由于零基础，我光是配置环境就花了很长时间，然后又花了将近以一星期阅读教程里的代码，尝试弄懂 OpenGL。幸好，在

电影《信条》的台词 “不要试着理解，而是去感受” 的精神指引下，我大致理解了 OpenGL 里的每一步工作的目的和完成方式，例如什么是顶点缓冲对象、顶点着色器和片段着色器的作用、如何渲染显示图像等。我还明白了如何创建一个窗口，如何从顶点开始构建一个图形、并使用着色器进行上色、贴图，用 uniform 实现颜色的变换。当实验结果是一幅幅漂亮的图像而不是黑底白字的数据文本时，内心还是有一些成就感的。

本次作业可以说是图形学的一次入门，并没有涉及到太多理论知识，但我们也可以从中看到图形学这门课程的复杂与精妙。在此感谢老师和助教在作业过程中提供的帮助，希望我能顺利完成下一次作业。