计算机图形学与虚拟现实

实验报告

2025秋

课程类型	限选□ 选修□
实验题目	**圆的绘制**
姓　名	**邴海诺**
学　号	**2024112353**
学　院	**计算学部**

2025 年 12 月 21 日

一、实验目的

了解帧缓冲的概念

使用着色器绘制圆形

二、实验要求及实验环境

```
OpenGL version: 4.6 (Compatibility Profile) Mesa 25.2.7-cachyos1.2
GLEW version: 2.2.0
```

三、设计思想（本程序中的用到的主要算法及数据结构，没有填写无）

无

四、实验结果与分析（根据章节分节）

默认情况下，OpenGL 渲染到屏幕（默认帧缓冲）。OpenGL 允许我们定义我们自己的帧缓冲，也就是说我们能够定义我们自己的颜色缓冲，甚至是深度缓冲和模板缓冲。在能够自行实现帧缓冲后，就能够有更多方式来进行渲染。通过创建自定义帧缓冲对象 (FBO)，可以将场景渲染到纹理附件中。这个纹理随后可以像普通图片一样被采样，用于后处理效果或显示在屏幕的特定区域。

1. 创建铺屏四边形：

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec2 texCoords;
out vec2 TexCoords;

void main()
{
    gl_Position = vec4(position, 1.0f);
    TexCoords = texCoords;
}
```

顶点着色器输出 texcoords，而片元着色器接受 texcoords 并用 xy 坐标作为 RG 颜色。输出如下：

```
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;

void main()
{
    FragColor = texture(screenTexture, TexCoords);
}
```

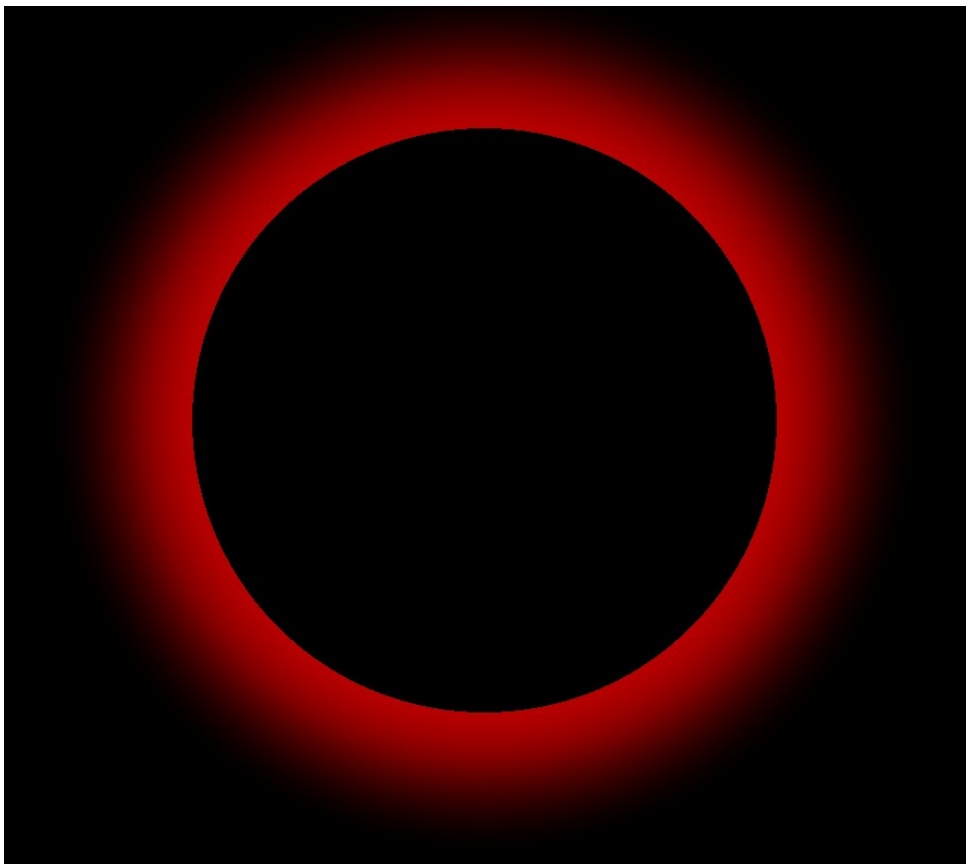
2. 画圆：

然后新建片元着色器用于画圆。这里稍微改了一下让它能返回渐变的颜色，但是总体思路还是不变：定义一个半径 radius 和半径宽度 edge，在半径$\pm edge$内调用 gradient 函数

```
in vec2 TexCoords;
out vec4 FragColor;

// 宽高比，保证在非正方形窗口下圆仍然是圆
uniform float w_div_h;
// 控制圆的半径（归一化坐标），边缘平滑宽度，以及颜色
uniform float radius; //  0.9
uniform float edge;   // 0.02
uniform vec3 innerColor; // 中心颜色
uniform vec3 outerColor; // 边缘颜色
// 计算给定距离 d 的径向渐变颜色（从 innerColor 到 outerColor）
vec3 radialGradient(float d) {
    float r = radius;
    if (r <= 0.0) r = 1.0; // 容错：若未设置 radius，则使用 1.0
    float t = clamp(d / r, 0.0, 1.0);

    vec3 inC = innerColor;
    vec3 outC = outerColor;

    if (length(inC) == 0.0 && length(outC) == 0.0) {
        inC = vec3(1.0, 1.0, 0.0);
        outC = vec3(1.0, 0.0, 0.0);
    }

    return mix(inC, outC, t);
}
// 原来的 step 函数现在返回 vec3（颜色），内部调用 radialGradient
vec3 step(float distance) {
    float e = edge;
    if (distance < radius - e/2) return vec3(0.0); // 圆内为黑色
    else if (distance > radius + e) return vec3(0.0); // 圆外为黑色
    else {
    float mask = 1.0 - smoothstep(radius - e, radius + e, distance);
    vec3 grad = radialGradient(distance);
    return grad * mask;
    }
    return vec3(0.0);
}
void main()
{
    vec2 uv = TexCoords * 2.0 - 1.0; // 把 [0,1] 映射到 [-1,1]
    uv.x *= w_div_h; // 宽高比调整
    float d = length(uv); // 当前片元到中心的距离
    vec3 color = step(d); // 调用 step 返回渐变颜色
    FragColor = vec4(color, 1.0f);
}
```

返回渐变颜色，其他地方设成黑色。

注意 TexCoords * 2.0 – 1.0 使其从[0,1]映射到[-1,1]，这样圆心才能在屏幕中间。uv.x *= w_div_h 根据屏幕的宽高比重新计算 x 坐标，不然画出来是椭圆形。绘制效果：



3. 自定义帧缓冲：

创建帧缓冲并绑定：

```
// 帧缓冲（FBO）设置
// 1) 生成并绑定帧缓冲对象：
//     生成一个 FBO，后续附件（颜色纹理 / 深度模板 RBO）
//     都会被附加到当前绑定的 FBO 上，之后渲染到 FBO 相当于渲染到这些附件。
unsigned int frameBuffer;
glGenFramebuffers(n: 1, framebuffers: &frameBuffer);
glBindFramebuffer(target: GL_FRAMEBUFFER, framebuffer: frameBuffer);
```

创建一个纹理图像，将它作为一个颜色附件附加到帧缓冲上：

```
// 2) 创建颜色附件：把渲染结果写入一个纹理（随后可以作为采样纹理使用）
//    这里创建了一个与窗口宽高相同的空纹理（NULL），作为 FBO 的颜色缓冲。
unsigned int texColorBuffer;
glGenTextures(n: 1, textures: &texColorBuffer);
glBindTexture(target: GL_TEXTURE_2D, texture: texColorBuffer);
// 分配纹理存储（不提供初始数据），格式为 RGB，尺寸为当前帧缓冲尺寸，其实就是一个空的2D纹理
glTexImage2D(target: GL_TEXTURE_2D, level: 0, internalformat: GL_RGB, width, height,
    border: 0, format: GL_RGB,type: GL_UNSIGNED_BYTE, pixels: NULL);

glTexParameteri(target: GL_TEXTURE_2D, pname: GL_TEXTURE_MIN_FILTER, param: GL_LINEAR);
glTexParameteri(target: GL_TEXTURE_2D, pname: GL_TEXTURE_MAG_FILTER, param: GL_LINEAR);
// 解绑纹理
glBindTexture(target: GL_TEXTURE_2D, texture: 0);
// 将该纹理附加到当前绑定的 FBO 的颜色附件 0 上（GL_COLOR_ATTACHMENT0）
glFramebufferTexture2D(target: GL_FRAMEBUFFER, attachment: GL_COLOR_ATTACHMENT0, textarget: GL_TEXTURE_2D,
                       texture: texColorBuffer, level: 0);
```

中间还有添加深度附件到帧缓冲中。由于暂时不需要对深度缓冲进行采样，可以使用 opengl 的渲染缓冲对象，将它创建为一个深度和模板附件渲染缓冲对象。并将内部格式设置为 GL_DEPTH24_STENCIL8

当所有缓冲都绑定完成后，检查帧缓冲的完整性，顺利执行后，就可以取消帧缓冲的绑定，帧缓冲的创建过程就完成了：

```
// 4) 检查 FBO 是否完整（所有必须的附件是否正确附加）
if (glCheckFramebufferStatus(target: GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;

// 完成设置后解绑 FBO（以后渲染需要写入该 FBO 时再绑定）
glBindFramebuffer(target: GL_FRAMEBUFFER, framebuffer: 0);
```

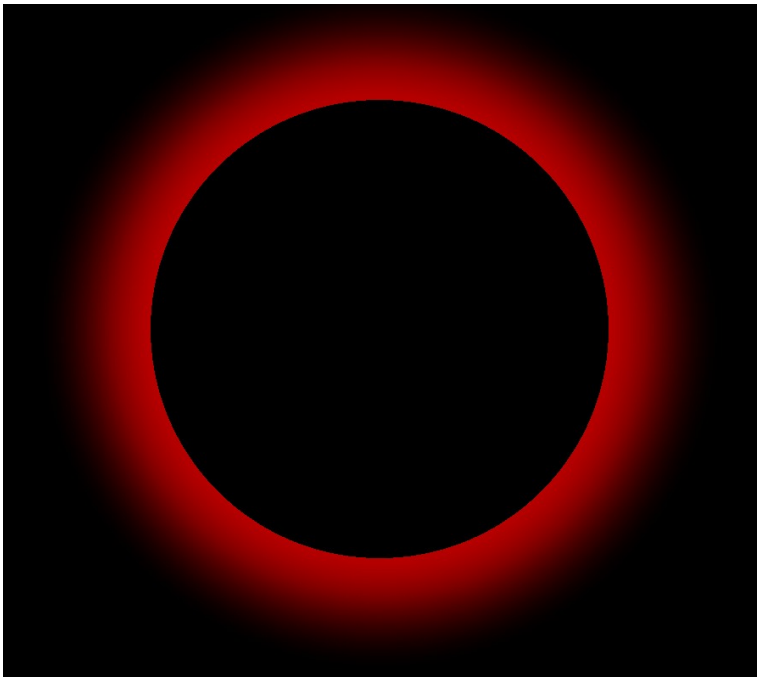
4. 使用自定义的帧缓冲绘制：

在完成帧缓冲的创建后，只需要绑定这个帧缓冲对象，让渲染到帧缓冲的缓冲中而不是默认的帧缓冲中。之后的渲染指令将会影响当前绑定的帧缓冲。

```
    processInput(window);
    glBindFramebuffer(target: GL_FRAMEBUFFER, framebuffer: frameBuffer);
    glClearColor(red: 0.1f, green: 0.1f, blue: 0.1f, alpha: 0.1f);
    glClear(mask: GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(cap: GL_DEPTH_TEST);

    circle_shader.use();
    circle_shader.setFloat(name: "radius", value: 0.7f);
    circle_shader.setFloat(name: "edge", value: 0.2f);
    circle_shader.setVec3(name: "innerColor", value: glm::vec3(x: 0.0f, y: 0.0f, z: 0.0f));
    circle_shader.setVec3(name: "outerColor", value: glm::vec3(x: 1.0f, y: 0.0f, z: 0.0f));
    circle_shader.setFloat(name: "w_div_h", value: (float)width / (float)height);

    glBindVertexArray(array: VAO);
    glDrawArrays(mode: GL_TRIANGLE_STRIP, first: 0, count: 4);
    glBindVertexArray(array: 0);

    glBindFramebuffer(target: GL_FRAMEBUFFER, framebuffer: 0);
    glClearColor(red: 0.2f, green: 0.3f, blue: 0.3f, alpha: 1.0f);
    glClear(mask: GL_COLOR_BUFFER_BIT);

    quad_shader.use();
    glBindTexture(target: GL_TEXTURE_2D, texture: texColorBuffer);
    glDisable(cap: GL_DEPTH_TEST);
    glBindVertexArray(array: VAO);
    glDrawArrays(mode: GL_TRIANGLE_STRIP, first: 0, count: 4);
    glBindVertexArray(array: 0);

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

在 main 函数中绑定创建好的帧缓冲，绘制圆形，再绑定默认的帧缓冲，将创建的帧缓冲的颜色纹理作为着色器输入，绘制铺屏四边形，采样颜色纹理并输出。

五、结论



六、参考文献

七、附录：源代码

```
int main() {

// 初始化 GLFW

if (!glfwInit()) {

std::cerr << "Failed to initialize GLFW\n";

return -1;

}


// 创建窗口

GLFWwindow*  window  =  glfwCreateWindow(SCR_WIDTH,  SCR_HEIGHT,
```

```
"Learn OpenGL",
nullptr, nullptr);
if (!window) {
std::cerr << "Failed to create window\n";
glfwTerminate();
return -1;
}

// 设置当前上下文
glfwMakeContextCurrent(window);

// 初始化 GLEW
GLenum err = glewInit();
if (err != GLEW_OK) {
std::cerr << "Failed to initialize GLEW: " <<
glewGetErrorString(err)
<< "\n";
return -1;
}
int width, height;
glfwGetFramebufferSize(window, &width, &height);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

```
glViewport(0, 0, width, height);

std::cout << "OpenGL version: " << glGetString(GL_VERSION) <<
"\n";
std::cout << "GLEW version: " << glewGetString(GLEW_VERSION) <<
"\n";

float quadVertices[] = {
// positions // texture Coords
-1.0f, 1.0f, 0.0f, 0.0f, 1.0f,
-1.0f, -1.0f, 0.0f, 0.0f, 0.0f,
1.0f, 1.0f, 0.0f, 1.0f, 1.0f,
1.0f, -1.0f, 0.0f, 1.0f, 0.0f,
};

glEnable(GL_DEPTH_TEST);

unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

```
glBufferData(GL_ARRAY_BUFFER,                sizeof(quadVertices),
&quadVertices,
GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0,    3,    GL_FLOAT,    GL_FALSE,    5    *
sizeof(float),
(void*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1,    2,    GL_FLOAT,    GL_FALSE,    5    *
sizeof(float),
(void*)(3 * sizeof(float)));
glBindVertexArray(0);
Shader circle_shader("../res/exp4_circle.vert",
"../res/exp4_circle.frag");
// unsigned int texture = loadTexture("../res/yellow.png");
// unsigned int texture = loadTexture("../res/diamond.png");
// 帧缓冲（FBO）设置
// 1) 生成并绑定帧缓冲对象：
// 生成一个 FBO，后续附件（颜色纹理 / 深度模板 RBO）
// 都会被附加到当前绑定的 FBO 上，之后渲染到 FBO 相当于渲染到这些附件。
unsigned int frameBuffer;
glGenFramebuffers(1, &frameBuffer);
```

```
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);

// 2) 创建颜色附件：把渲染结果写入一个纹理（随后可以作为采样纹理使用）
// 这里创建了一个与窗口宽高相同的空纹理（NULL），作为 FBO 的颜色缓冲。
unsigned int texColorBuffer;
glGenTextures(1, &texColorBuffer);
glBindTexture(GL_TEXTURE_2D, texColorBuffer);
// 分配纹理存储（不提供初始数据），格式为 RGB，尺寸为当前帧缓冲尺寸，其实就是一个
空的 2D 纹理
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
0, GL_RGB,GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// 解绑纹理
glBindTexture(GL_TEXTURE_2D, 0);
// 将该纹理附加到当前绑定的 FBO 的颜色附件 0 上（GL_COLOR_ATTACHMENT0）
glFramebufferTexture2D(GL_FRAMEBUFFER,      GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D,
texColorBuffer, 0);
// 3) 创建并附加深度和模板渲染缓冲（RBO）
// RBO 用于保存深度/模板信息，通常比纹理更高效（但不能被采样）。
unsigned int rbo;
```

```
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
// 为 RBO 分配存储（深度 24 + 模板 8）并设置与窗口一致的大小
glRenderbufferStorage(GL_RENDERBUFFER,       GL_DEPTH24_STENCIL8,
width, height);
glBindRenderbuffer(GL_RENDERBUFFER, 0);
// 将 RBO 附加为 FBO 的深度模板附件
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);

// 4) 检查 FBO 是否完整（所有必须的附件是否正确附加）
if          (glCheckFramebufferStatus(GL_FRAMEBUFFER)          !=
GL_FRAMEBUFFER_COMPLETE)
std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!"
<< std::endl;

// 完成设置后解绑 FBO（以后渲染需要写入该 FBO 时再绑定）
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 注意:
// - 如果窗口大小发生变化，需要重新调整或重新创建颜色纹理和 RBO 的尺寸（与
width/height 保持一致），否则渲染会拉伸或不完整。
// - 渲染到 FBO 时，视口（glViewport）应与 FBO 大小一致（如果它们不同，需要调
```

```
用 glViewport 进行调整）。
// - 若需要从生成的颜色纹理中采样（在屏幕四边形上显示或后处理），记得在绘制时将
纹理绑定到合适的纹理单元并设置 shader 的 sampler。

Shader quad_shader("../res/expr4_quad.vert",
"../res/expr4_quad.frag");

while (!glfwWindowShouldClose(window)) {

processInput(window);
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
glClearColor(0.1f, 0.1f, 0.1f, 0.1f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
circle_shader.use();
circle_shader.setFloat("radius", 0.7f);
circle_shader.setFloat("edge", 0.2f);
circle_shader.setVec3("innerColor", glm::vec3(0.0f, 0.0f, 0.0f));
circle_shader.setVec3("outerColor", glm::vec3(1.0f, 0.0f, 0.0f));
circle_shader.setFloat("w_div_h", (float)width / (float)height);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glBindVertexArray(0);
```

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

quad_shader.use();
glBindTexture(GL_TEXTURE_2D, texColorBuffer);
glDisable(GL_DEPTH_TEST);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glBindVertexArray(0);

glfwSwapBuffers(window);
glfwPollEvents();
}

glfwDestroyWindow(window);
glfwTerminate();
return 0;
}
```

顶点着色器:

```
#version 330 core
layout(location = 0) in vec3 position;
```

```
layout(location = 1) in vec2 texCoords;

out vec2 TexCoords;


void main()

{

gl_Position = vec4(position, 1.0f);

TexCoords = texCoords;

}
```

片元着色器

```
#version 330 core


in vec2 TexCoords;

out vec4 FragColor;


// 宽高比，保证在非正方形窗口下圆仍然是圆

uniform float w_div_h;

// 控制圆的半径（归一化坐标），边缘平滑宽度，以及颜色

uniform float radius; // 0.9

uniform float edge; // 0.02

uniform vec3 innerColor; // 中心颜色

uniform vec3 outerColor; // 边缘颜色

// 计算给定距离 d 的径向渐变颜色（从 innerColor 到 outerColor）

vec3 radialGradient(float d) {
```

```
float r = radius;
if (r <= 0.0) r = 1.0; // 容错：若未设置 radius，则使用 1.0
float t = clamp(d / r, 0.0, 1.0);

vec3 inC = innerColor;
vec3 outC = outerColor;

if (length(inC) == 0.0 && length(outC) == 0.0) {
inC = vec3(1.0, 1.0, 0.0);
outC = vec3(1.0, 0.0, 0.0);
}

return mix(inC, outC, t);
}
// 原来的 step 函数现在返回 vec3（颜色），内部调用 radialGradient
vec3 step(float distance) {
float e = edge;
if (distance < radius - e/2) return vec3(0.0); // 圆内为黑色
else if (distance > radius + e) return vec3(0.0); // 圆外为黑色
else {
float mask = 1.0 - smoothstep(radius - e, radius + e, distance);
vec3 grad = radialGradient(distance);
return grad * mask;
```

```
}

return vec3(0.0);

}

void main()

{

vec2 uv = TexCoords * 2.0 - 1.0; // 把 [0,1] 映射到 [-1,1]

uv.x *= w_div_h; // 宽高比调整

float d = length(uv); // 当前片元到中心的距离

vec3 color = step(d); // 调用 step 返回渐变颜色

FragColor = vec4(color, 1.0f);

}
```