调试 shadow map

一、调试核心思路

阴影贴图本质上是一张存储深度信息的纹理，其可视化原理与普通纹理（如牛模型的漫反射贴图、平板的棋盘格贴图）完全一致。既然普通纹理能正常贴在模型上显示，我们也可以将阴影贴图“贴”在一个屏幕空间的矩形上，通过观察灰度分布来验证阴影贴图是否正确生成——这是定位阴影渲染问题的高效手段。

核心实现逻辑：在屏幕空间绘制一个全屏四边形，通过片段着色器对阴影贴图进行采样，将深度值转换为灰度颜色输出，直观判断阴影贴图的生成效果。

> 感兴趣的同学可以尝试自行实现，当然也可以直接使用以下代码

二、着色器代码配置

1. 顶点着色器

负责传递顶点位置和纹理坐标，直接使用标准化设备坐标（NDC）确保四边形覆盖整个屏幕。

在 `src/shaders/` 目录下新建 `debugger.vert` 文件：

```
#version 330 core

// 输入顶点属性：位置（3分量）和纹理坐标（2分量）
layout (location = 0) in vec3 aPos;       // 顶点位置（与VAO属性索引0绑定）
layout (location = 1) in vec2 aTexCoord;  // 纹理坐标（与VAO属性索引1绑定）

// 输出到片段着色器的纹理坐标（需与片段着色器输入变量完全匹配）
out vec2 TexCoord;

void main()
{
    // 直接使用输入位置作为NDC坐标，确保四边形全屏显示
    gl_Position = vec4(aPos, 1.0);
    // 将纹理坐标传递给片段着色器，用于后续纹理采样
    TexCoord = aTexCoord;
}
```

2. 片段着色器

对阴影贴图（深度纹理）进行采样，将深度值映射为灰度色输出，便于观察深度分布。

在 `src/shaders/` 目录下新建 `debugger.frag` 文件：

```
#version 330 core
// 从顶点着色器接收的纹理坐标（名称、类型需与顶点着色器输出一致）
in vec2 TexCoord;
// 输出最终像素颜色
out vec4 FragColor;

// 2D纹理采样器（绑定到指定纹理单元，用于采样阴影贴图）
uniform sampler2D ourTexture;
```

```
void main() {
    // 采样阴影贴图的深度值（深度信息存储在r通道，范围[0,1]）
    float depth = texture(ourTexture, TexCoord).r;

    // 将深度重新映射为线性，否则贴图中几乎所有区域都接近1.0
    // 0.1/100.0是近/远平面的值，需要和main.cpp中阴影计算时的数值对应
    depth = depth * 2.0 - 1.0;
    depth = (2.0 * 0.1 * 100.0) / (100.0 + 0.1 - depth * (100.0 - 0.1));
    depth = (depth - 0.1) / (100.0 - 0.1);

    // 将深度值转换为灰度颜色输出（默认近物暗、远物亮）
    FragColor = vec4(vec3(depth), 1.0f);

    // 调试备用：输出纯色（用于验证着色器是否正常执行）
    // FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);
}
```

三、调试数据准备

在 `utils/` 下新建 `texture_debugger.h` 和 `texture_debugger.cpp` 文件。

`texture_debugger.h` 可以如下完成：

```
#ifndef DEBUG_TEXTURE_DEBUGGER_H
#define DEBUG_TEXTURE_DEBUGGER_H

#include <memory>
#include <map>
#include <string>
#include <vector>

#include "utils/shader.h"
#include "utils/gl/texture.h"
#include "utils/gl/vertex_array.h"
#include "utils/tools.h"

class TextureDebugger {
public:
    TextureDebugger();
    ~TextureDebugger() = default;

    void render(const Utils::GL::Texture2D& texture_to_debug);

private:
    std::unique_ptr<Utils::Shader> debug_shader;
    std::unique_ptr<Utils::GL::VertexArray> quad_va;
    std::unique_ptr<Utils::GL::ElementBuffer> quad_eb;
    std::map<std::string, std::unique_ptr<Utils::GL::VertexBuffer>> quad_vbos;
};

#endif // DEBUG_TEXTURE_DEBUGGER_H
```

接下来在 `texture_debugger.cpp` 中实现着色器的初始化和数据定义（`TextureDebugger()` 构造函数）和渲染配置（void render() 函数）

1. 着色器程序初始化和数据定义

创建调试专用着色器程序，加载上述顶点/片段着色器文件。

```
TextureDebugger::TextureDebugger() {
    // 初始化调试着色器（传入顶点/片段着色器文件路径）
    debug_shader = std::make_unique<Utils::Shader>(SHADER_DIR"/debugger.vert",
SHADER_DIR"/debugger.frag");

    // 顶点数据：位置（x,y,z）+ 纹理坐标（u,v）
    std::vector<vecf3> positions = {
        { 1.0f,  1.0f, 0.0f},  // 右上角
        { 1.0f, -1.0f, 0.0f},  // 右下角
        {-1.0f, -1.0f, 0.0f},  // 左下角
        {-1.0f,  1.0f, 0.0f}   // 左上角
    };
    std::vector<vecf2> texcoords = {
        {1.0f, 1.0f},
        {1.0f, 0.0f},
        {0.0f, 0.0f},
        {0.0f, 1.0f}
    };

    // 索引数据：通过索引复用顶点，组成两个三角形
    std::vector<veci3> indices = {
        {0, 1, 3},  // 第一个三角形（右上角、右下角、左上角）
        {1, 2, 3}   // 第二个三角形（右下角、左下角、左上角）
    };

    // 创建并配置顶点数组对象（VAO）、顶点缓冲对象（VBO）和索引缓冲对象（EBO），确保顶点数据能正确传递给着
色器。
    auto vb_pos = new Utils::GL::VertexBuffer(positions.size() * sizeof(vecf3),
positions.data());
    quad_vbos["position"] = std::unique_ptr<Utils::GL::VertexBuffer>(vb_pos);

    auto vb_uv = new Utils::GL::VertexBuffer(texcoords.size() * sizeof(vecf2),
texcoords.data());
    quad_vbos["texcoord"] = std::unique_ptr<Utils::GL::VertexBuffer>(vb_uv);

    quad_eb = std::make_unique<Utils::GL::ElementBuffer>(GL_TRIANGLES, indices.size(),
(GLuint*)indices.data());

    Utils::GL::VertexArray::Format format;
    format.attr_ptrs.emplace_back(quad_vbos["position"]->attr_ptr(3, GL_FLOAT, GL_FALSE,
sizeof(vecf3)));
    format.attr_ptrs.emplace_back(quad_vbos["texcoord"]->attr_ptr(2, GL_FLOAT, GL_FALSE,
sizeof(vecf2)));
    format.eb = quad_eb.get();

    quad_va = std::make_unique<Utils::GL::VertexArray>(std::vector<GLuint>{0, 1}, format);
```

```
}
```

2. 调试渲染循环配置

核心步骤包括帧缓冲切换、状态配置、纹理绑定和绘制调用。

```
void TextureDebugger::render(const Utils::GL::Texture2D& texture_to_debug) {
    // 1. 临时禁用干扰性渲染状态（避免深度测试、面剔除等导致绘制失败）
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_CULL_FACE);

    // 2. 清除颜色缓冲，设置蓝色背景（与灰度阴影贴图形成明显对比，便于观察）
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 3. 激活调试着色器程序（关键步骤，确保使用目标着色器进行渲染）
    debug_shader->active_texture(0, const_cast<Utils::GL::Texture2D*>(&texture_to_debug));
    debug_shader->set_tex("ourTexture", 0);

    // 4. 绘制调用
    quad_va->draw(*debug_shader);

    // 5. 恢复原有渲染状态（避免影响后续其他渲染逻辑）
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}
```

在主渲染循环中进行修改：

```
    // init shadow map
    auto shadow_map = load_shadow_map(SHADOW_TEXTURE_SIZE);
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, vecf4(1.0f, 1.0f, 1.0f,
1.0f).data());
    FrameBuffer shadow_fbo;
    shadow_fbo.attach(GL_DEPTH_ATTACHMENT, &shadow_map);

    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // debugger 初始化
    auto texture_debugger = std::make_unique<TextureDebugger>();

    while (!glfwWindowShouldClose(window)) {
        // record time
        auto current_frame = static_cast<float>(glfwGetTime());
        delta_time = current_frame - last_frame;
        last_frame = current_frame;

        process_input(window);
        process_release(window);

        ////////////////////////////////////////////////
        // render shadow map
        // 渲染 shadow map 这部分逻辑需要保留
```

```
        // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        // debugger 调用
        glViewport(0, 0, SCR_HEIGHT, SCR_HEIGHT);
        texture_debugger->render(shadow_map);

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