实验：
延迟渲染

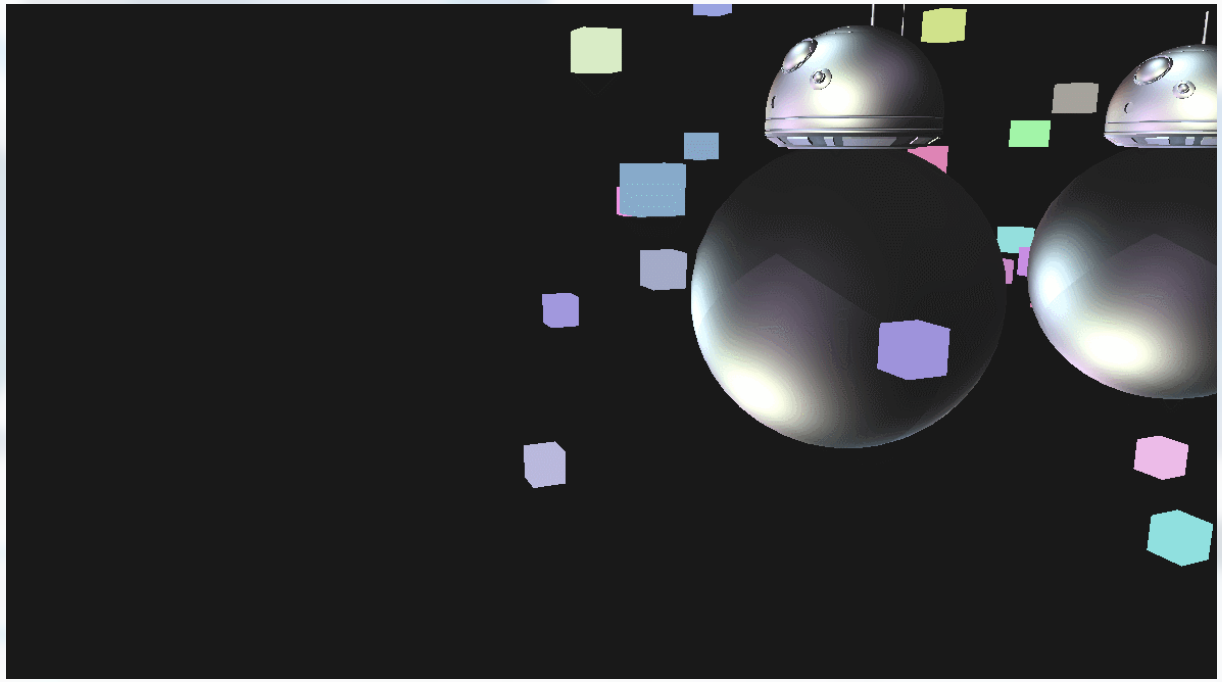
华中科技大学软件学院　万琳

提纲

1. 实验要求
2. 程序流程
3. 要点解析
4. 程序演示

1 实验要求

1.理解延迟渲染原理；
2.实现延迟渲染；



2 程序流程

准备阶段　　几何阶段　　光照处理阶段

3 要点解析

Step1 准备阶段

G-buffer创建

着色器、模型
和光源设置

3 要点解析

G-buffer创建

创建了G-buffer，因为接下来的操作要在该帧缓冲上进行，第三行我们在这里绑定到G-buffer帧缓冲上

Gbuffer创建

```
// G-buffer的创建
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
```

绑定该帧缓冲

3 要点解析

G-buffer创建

为G-BUFFER创建三个纹理附件（或叫颜色附件）：

gPosition顶点位置信息；

gNormal法线信息；

gAlbedoSpec颜色信息和镜面强度值；

纹理创建及绑定

```
//position-位置信息
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, gPosition, 0);
```

将相应纹绑定到帧缓冲的对应缓冲区

3 要点解析

G-buffer创建

用多渲染目标（MRT）方法将对应数据分配到各个颜色附件

将每像素的数据保存到帧缓冲不同的缓冲区中，使得这些缓冲区的数据由此可用于后续的计算。

```
//MRT-Multiple Render Targets 多渲染目标技术
unsigned int attachments[3] =
{ GL_COLOR_ATTACHMENT0,GL_COLOR_ATTACHMENT1,GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, attachments);
```

MRT代码实现，分别表示缓冲区的个数和缓冲区名称

3 要点解析

着色器、模型和光源设置

A **着色器设置**

B **模型设置**

C **光源设置**

3 要点解析

A

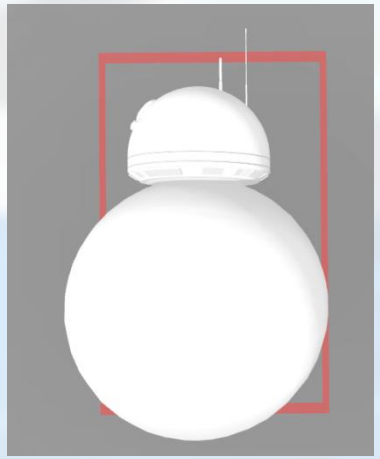
着色器设置


```
//载入着色器和模型
Shader shaderGeometryPass("res/shader/g_buffer.vs",
 "res/shader/g_buffer.fs");
Shader shaderLightingPass("res/shader/deferred_shading.vs",
"res/shader/deferred_shading.fs");
```

3 要点解析

B

模型设置



```
Model BB8("res/models/BB8 New/bb8.obj");
std::vector<glm::vec3> objectPositions;
objectPositions.push_back(glm::vec3(-1.0, -3.0, 0.0));
objectPositions.push_back(glm::vec3(2.0, -3.0, 0.0));
```

3 要点解析

C

光源设置

```
//多光源
std::vector<glm::vec3> lightPositions;
std::vector<glm::vec3> lightColors;
srand(13);
```

3 要点解析

C

光源设置

```
for (unsigned int i = 0; i < NR_LIGHTS; i++)
{
        float xPos = ((rand() % 100) / 100.0) * 6.0 - 3.0;
        float yPos = ((rand() % 100) / 100.0) * 6.0 - 4.0;
        float zPos = ((rand() % 100) / 100.0) * 6.0 - 3.0;
        lightPositions.push_back(glm::vec3(xPos, yPos, zPos));
        float rColor = ((rand() % 100) / 200.0f) + 0.5;
        float gColor = ((rand() % 100) / 200.0f) + 0.5;
        float bColor = ((rand() % 100) / 200.0f) + 0.5;
        lightColors.push_back(glm::vec3(rColor, gColor, bColor));
}
```

3 要点解析

Step2

几何阶段

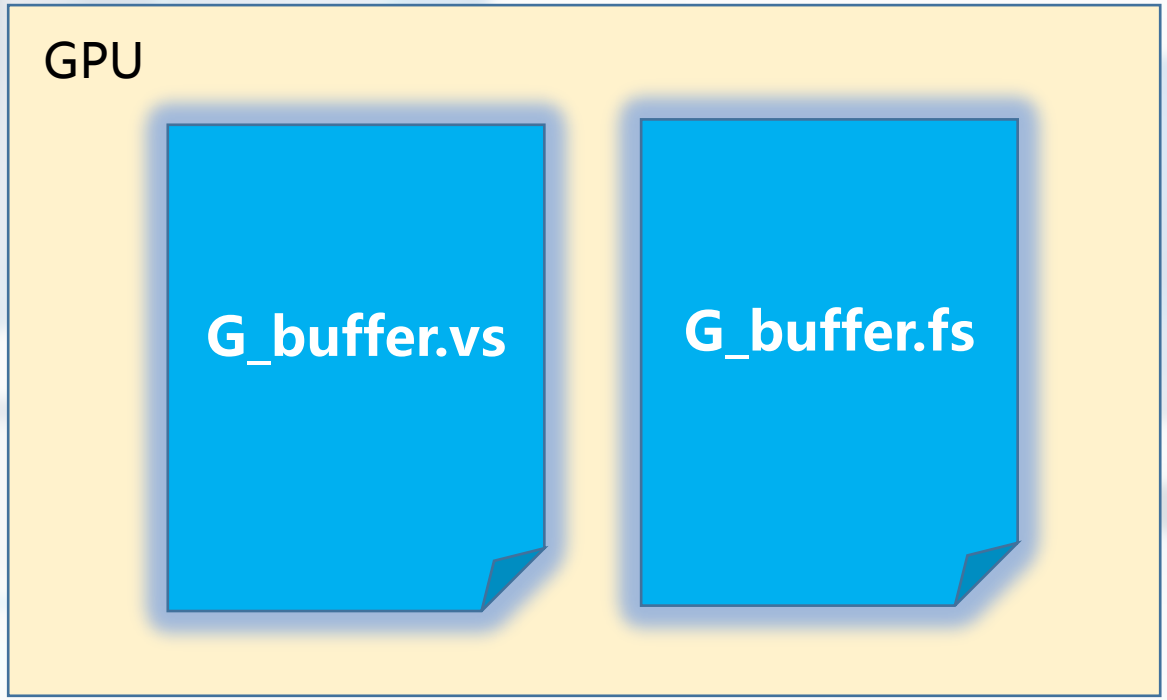
激活着色器

```
shaderGeometryPass.use();
shaderGeometryPass.setMat4("projection", projection);
shaderGeometryPass.setMat4("view", view);
for (unsigned int i = 0; i < objectPositions.size(); i++)
{
        model = glm::mat4(1);
        model = glm::translate(model, objectPositions[i]);
        model = glm::scale(model, glm::vec3(0.02f));
        shaderGeometryPass.setMat4("model", model);
        BB8.Draw(shaderGeometryPass);
}
```

设置PVM矩阵

3 要点解析

Step2 几何阶段



3 要点解析

Step2 几何阶段

G_buffer.vs

世界坐标系下的坐标

```
void main()
{
    vec4 worldPos = model * vec4(aPos, 1.0);
    FragPos = worldPos.xyz;
    TexCoords = aTexCoords;

    mat3 normalMatrix = transpose(inverse(mat3(model)));
    Normal = normalMatrix * aNormal;

    gl_Position = projection * view * worldPos;
}
```

3 要点解析

Step2 几何阶段

G_buffer.fs

输出到3个颜色缓冲区

```
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

void main()
{
    gPosition = FragPos;

    gNormal = normalize(Normal);

    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;

    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

从模型的贴图中提取相应数据

3 要点解析

Step3

光照处理阶段

将帧缓冲绑定到默认帧缓冲即屏幕显示的帧缓冲上

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderLightingPass.use() );//激活shaderLightingPass
glActiveTexture(GL_TEXTURE0);//激活纹理通道
glBindTexture(GL_TEXTURE0, gPosition);//绑定纹理附件
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
```

3 要点解析

Step3 光照处理阶段

num参数用于削弱最后光照结果，因为光源数量较多时叠加后亮度过高

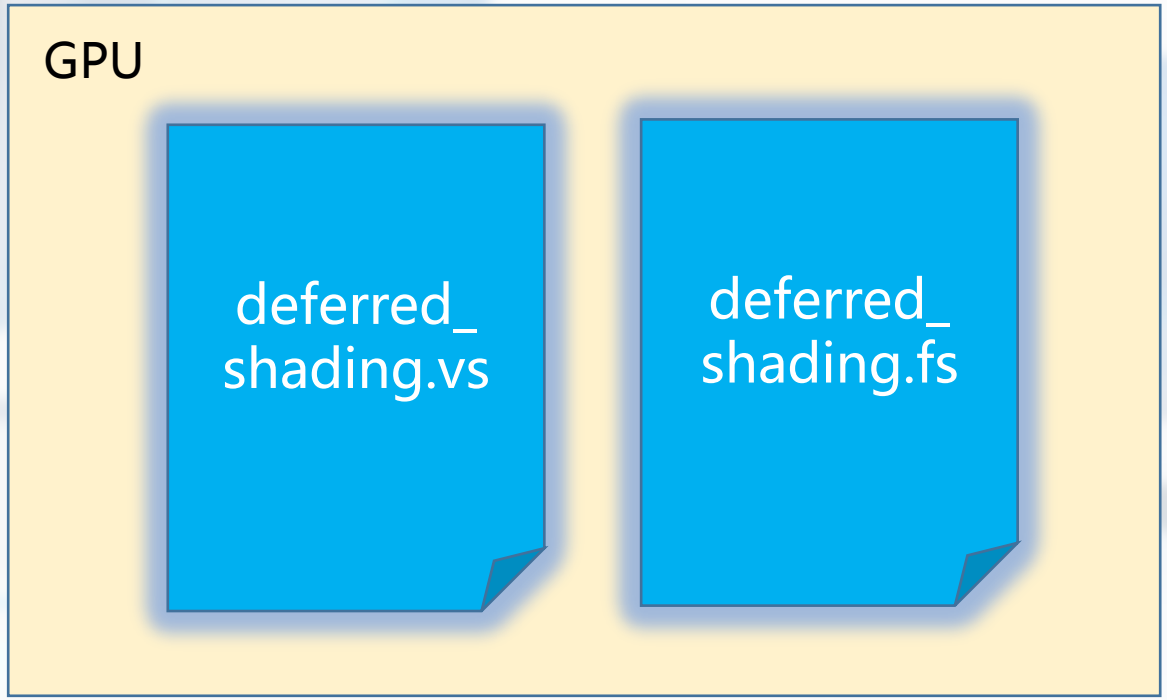
```
shaderLightingPass.setFloat("num", NR_LIGHTS / (NR_LIGHTS / 3));

for (unsigned int i = 0; i < lightPositions.size(); i++)
{
        shaderLightingPass.setVec3("lights[" + std::to_string(i) + "].Position", lightPositions[i]);
        shaderLightingPass.setVec3("lights[" + std::to_string(i) + "].Color", lightColors[i]);;
}
shaderLightingPass.setVec3("viewPos", camera.Position);
renderQuad();
```

直接从3个纹理附件提取每个像素的数据进行绘制

3 要点解析

Step3 光照处理阶段



3 要点解析

Step3 光照处理阶段

deferred_
shading.vs

```
void main()
{
    TexCoords = aTexCoords;
    gl_Position = vec4(aPos, 1.0);
}
```

3 要点解析

Step3 光照处理阶段

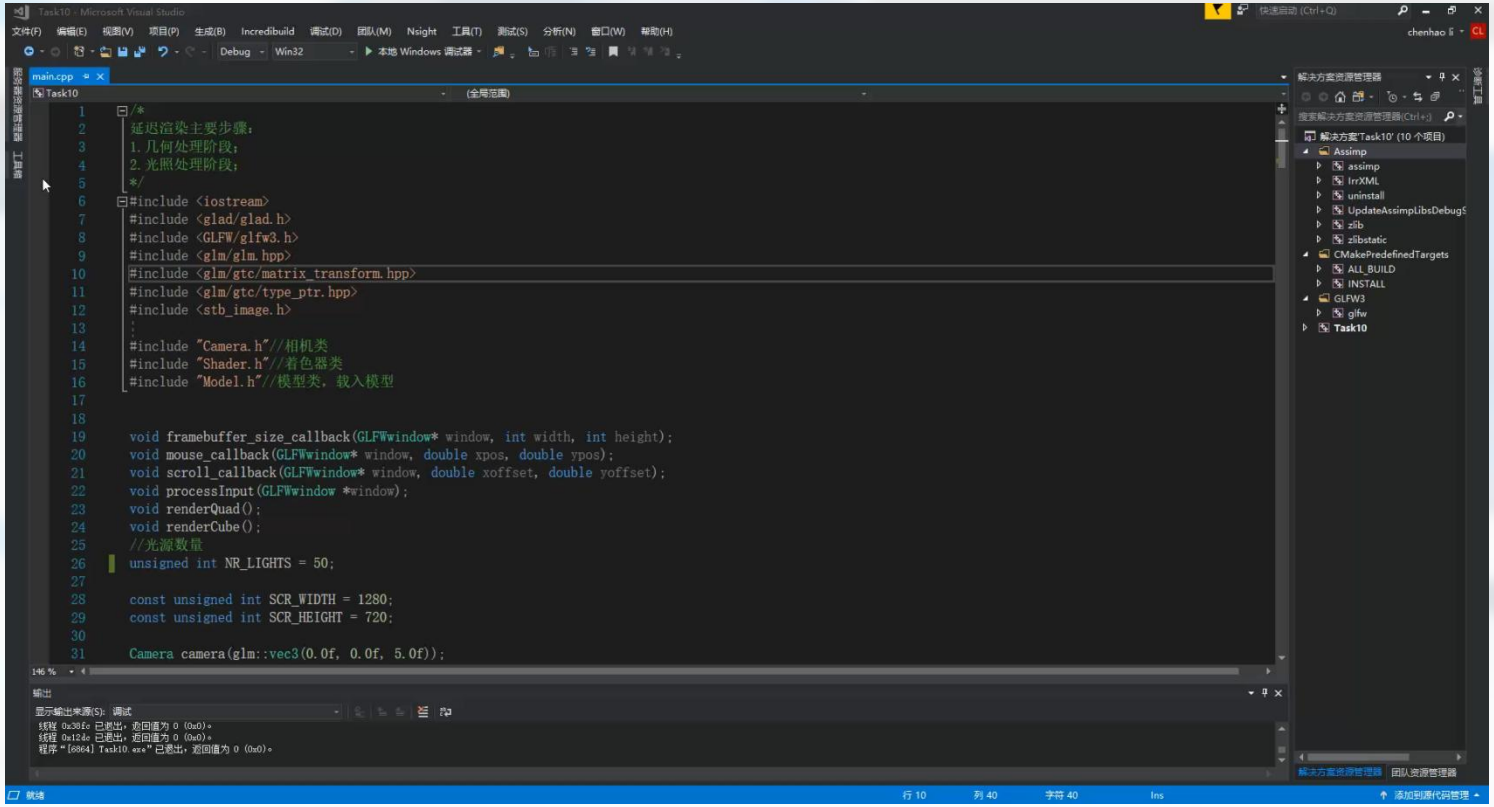
deferred_
shading.fs

从纹理附件提取数据 → 光照计算

```
vec3 FragPos = texture(gPosition, TexCoords).rgb;
vec3 Normal = texture(gNormal, TexCoords).rgb;
vec3 Diffuse = texture(gAlbedoSpec, TexCoords).rgb;
float Specular = texture(gAlbedoSpec, TexCoords).a
```

4 程序演示

最终得到的结果可见如下视频：



说明：

1.Z 开始光源显示；
  X 关闭光源显示；

2. 修改NR_LIGHTS改变光源数量，查看更多光源情况下延迟渲染帧数。


谢谢
软件学院　万琳