中山大學

SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程: 计算机图形学

专业名称: 计算机科学与技术

学生姓名:

学生学号:

教学班级: 计科 3 班

实验成绩:

报告时间: 2024 年 1 月 16 日

Edited by LaTeX

1 实验任务

1. 使用 GLSL 片元着色器实现 Phong 光照模型;
2. 使用 VBO 存储顶点与连接关系;
3. 对比 Phong shading 与 OpenGL 自带的 Smoothing shading 的区别;
4. 对比通过 VBO 进行渲染与通过 glVertex 进行渲染的区别;
5. 讨论 VBO 中是否使用 Index Array 的效率区别
6. 讨论 HW2 与 HW3 的渲染效果和效率方面的区别。

2 实验环境

本次作业完全基于 **HW2 的环境和框架**。

3 实验原理

3.1 GLSL

GLSL（OpenGL Shading Language）是一种类 C 语言的用于编写着色器程序的编程语言，编写顶点和片段着色器以控制图形渲染管线中的顶点和像素的处理过程。作为 OpenGL 的一部分，GLSL 在图形处理单元（GPU）上用于执行高性能图形渲染和计算任务。

顶点着色器是在图形渲染管线的顶点处理阶段执行的程序。它接收输入顶点的属性（如位置、颜色、法线等），并可以对它们进行变换和处理。顶点着色器通常用于执行模型变换、光照计算和其他与顶点相关的操作。

片段着色器是在图形渲染管线的片段处理阶段执行的程序。它接收由顶点着色器传递过来的顶点属性(如插值后的颜色、纹理坐标等),并用于计算最终的像素颜色。片段着色器通常用于执行光照模型、纹理采样和其他与像素相关的操作。

GLSL 提供了丰富的内置函数和数据类型，以及各种语法结构，使开发人员能够编写高效的图形处理代码。它支持向量和矩阵运算，提供了对纹理采样、光照计算和颜色处理等常见图形任务的支持。GLSL 代码通常嵌入在 OpenGL 应用程序中，通过编译和链接过程与图形渲染管线进行交互。开发人员可以使用 GLSL 来实现各种图形效果,包括阴影、反射、折射、模糊和其他高级渲染技术。

3.2 渲染管线

渲染管线（Rendering Pipeline）用于描述图形渲染的过程和流程，它定义了将输入的几何数据转化为最终图像的一系列阶段和操作。渲染管线通常包含以下几个主要阶段：

（1）几何阶段：

· 顶点输入：接收输入的几何数据，如顶点的位置、法线和纹理坐标等。

· 顶点处理：对输入的顶点数据进行变换、光照计算和其他操作，生成变换后的顶点数据。

（2）光栅化阶段：

· 图元装配：将顶点数据组装成基本图元，如点、线段和三角形。

· 光栅化：将基本图元转换为屏幕上的像素，并确定每个像素的位置和属性。

（3）片段阶段：

· 片段着色器：计算每个像素（片段）的最终颜色值，可以进行纹理采样、光照计算和其他像素级操作。

· 深度测试：根据片段的深度值，确定像素是否可见，以及是否更新深度缓冲区。

· Alpha 测试：根据片段的 Alpha 值（透明度），确定是否丢弃该片段。

· 模板测试：使用模板缓冲区来进行更复杂的像素丢弃和掩码操作。

（4）输出合成阶段：

· 像素操作：根据深度、颜色和模板测试的结果，决定最终像素的颜色值。

· 帧缓冲写入：将最终的像素颜色值写入帧缓冲区，生成最终的图像。

本次实验中，顶点着色器和片段着色器的渲染管线图如下：

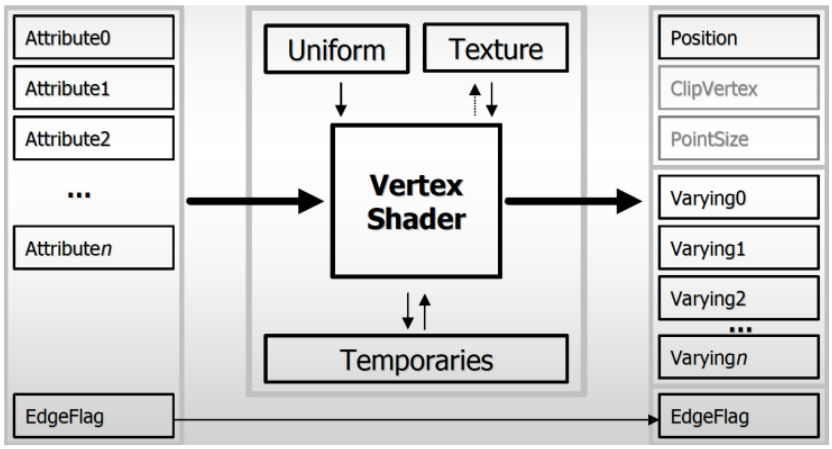


fig 1: 顶点着色器渲染管线

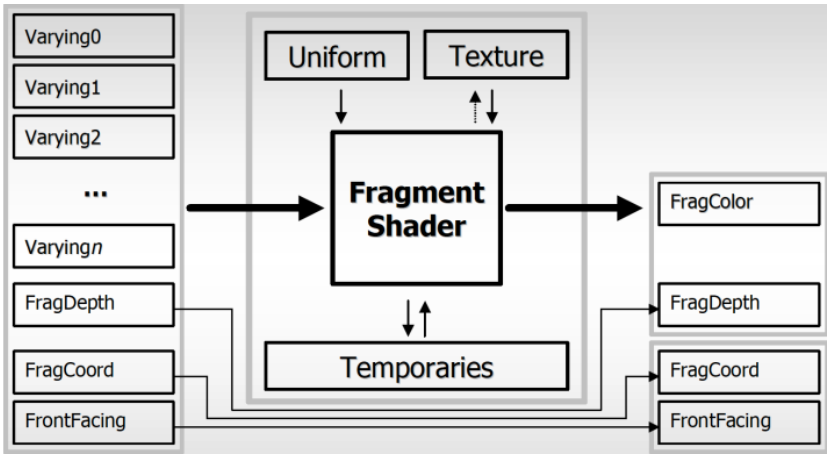


fig 2: 片段着色器渲染管线

3.3 VBO

VBO（Vertex Buffer Object）是在图形渲染中用于高效存储顶点数据的一种机制。VBO 将顶点数据直接存储在 GPU 的内存中，减少了数据传输的开销。通过使用 VBO，可以在应用程序初始化阶段将顶点数据加载到 GPU 内存中，然后在渲染过程中直接从 GPU 内存中读取数据，提高了渲染效率。

VBO 的基本用法如下：

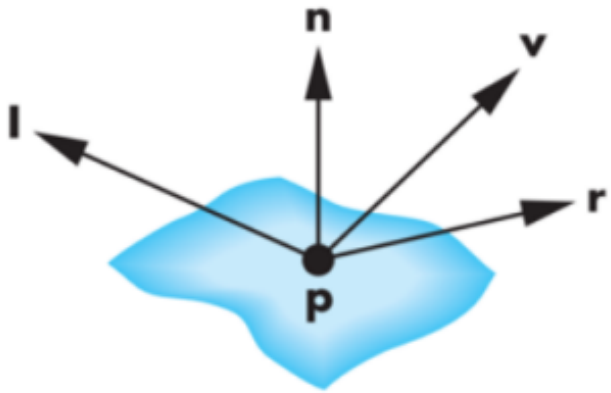
1. 生成 VBO：使用函数如 glGenBuffers() 生成一个唯一的 VBO 标识符；

2. 绑定 VBO：使用函数如 glBindBuffer() 将 VBO 绑定到 OpenGL 的上下文中，表示当前操作的 VBO 是该绑定的 VBO；
3. 分配内存：使用函数如 glBufferData() 为 VBO 分配一块内存空间，并指定数据的大小和使用方式（静态、动态、一次性等）；
4. 数据传输：使用函数如 glBufferSubData() 或 glMapBuffer() 将顶点数据传输到 VBO 中；
5. 使用 VBO：在渲染过程中，通过绑定 VBO 并配置顶点属性指针，将 VBO 中的顶点数据传递给顶点着色器；
6. 解绑 VBO：使用函数如 glBindBuffer() 将 VBO 解绑，表示当前操作的 VBO 为空；
7. 删除 VBO：使用函数如 glDeleteBuffers() 删除 VBO 并释放相关的 GPU 内存。

3.4 Phong 光照模型

在本次实验中，我们所用光照模型为局部光照模型，采用单点光源，整体光照包括环境光照、漫反射光照和镜面反射光照。

如图所示，曲面上 p 点的光照效果用以下几个向量进行描述：曲面法向量 **n**，视线方向 **v**，光线入射方向 **l**，光线反射方向 **r**。



环境光照公式：

$I_{ambient} = k_a L_a$

其中，k_a 表示环境光照系数，L_a 表示环境光照强度。

漫反射光照公式：

$I_{diffuse} = \frac{1}{a+bd+cd^2}(k_d L_d max(\mathbf{l} \cdot \mathbf{n}, 0))$

其中，k_d 表示漫反射光照系数，L_d 表示漫反射光照强度，a、b、c 表示距离衰减参数，$\mathbf{l}$ 表示光线入射方向，$\mathbf{n}$ 表示曲面法向量。

镜面反射光照公式：

$I_{specular} = \frac{1}{a+bd+cd^2}(k_s L_s max((\mathbf{l} \cdot \mathbf{n})^{\alpha}, 0))$

其中，k_s 表示镜面反射光照系数，L_s 表示镜面反射光照强度，a、b、c 表示距离衰减参数，$\mathbf{v}$ 表示视线方向，$\mathbf{r}$ 表示光线反射方向，α 表示高光系数。

整体光照公式：

$I = I_{ambient} + I_{diffuse} + I_{specular}$

4 代码设计

4.1 着色器代码实现

根据渲染管线以及 Phong 光照模型的原理，我们设计以下的顶点着色器和片段着色器：

顶点着色器（VertexShader.glsl）

```
#version 330 core

layout (location = 0) in vec3 Vertex;
layout (location = 1) in vec3 Color;
layout (location = 2) in vec3 Normal;

out vec3 ObjColor;
```

```
8  out vec3 FragNorm;
9  out vec3 FragPos;
10
11 uniform mat4 view;
12 uniform mat4 proj;
13
14 void main()
15 {
16     FragPos = Vertex;
17     ObjColor = Color;
18     FragNorm = Normal;
19     gl_Position = proj * view * vec4(Vertex, 1.0);
20 }
```

片段着色器（FragmentShader.glsl）

```
1  #version 330 core
2
3  out vec4 FragColor;
4
5  in vec3 ObjColor;
6  in vec3 FragNorm;
7  in vec3 FragPos;
8
9  uniform vec3 LightPos;
10 uniform vec3 LightColor;
11 uniform vec3 ViewPos;
12 uniform vec3 CamPos;
13
14 void main()
15 {
16     float Dist = distance(LightPos, FragPos);
17     float da = 1.0f, db = 0.001f, dc = 0.000001f;
18
```

```
19      // Ambient
20      float AmbientStrength = 0.2f;
21      vec3 Ambient = AmbientStrength * LightColor;
22
23      // Diffuse
24      float DiffuseStrength = 0.1f;
25      float DiffuseAlpha = 1.0f;
26      vec3 Vec_L = normalize(LightPos - FragPos);
27      vec3 Vec_N = normalize(FragNorm);
28      float Diff = pow(max(dot(Vec_N, Vec_L), 0.0f),
            DiffuseAlpha);
29      vec3 Diffuse = DiffuseStrength * Diff * LightColor
            ;
30
31      // Specular
32      float SpecularStrength = 0.8f;
33      float SpecularAlpha = 2.0f;
34      vec3 Vec_V = normalize(CamPos - ViewPos);
35      vec3 Vec_R = normalize(reflect(-Vec_L, Vec_N));
36      float Spec = pow(max(dot(Vec_V, Vec_R), 0.0f),
            SpecularAlpha);
37      vec3 Specular = SpecularStrength * Spec *
            LightColor;
38
39      vec3 Color = ObjColor * ((1 / (da + db * Dist + dc
             * Dist * Dist)) * (Diffuse + Specular) +
            Ambient);
40      FragColor = vec4(Color, 1.0);
41  }
```

以上代码中，顶点着色器的输入为 VBO 中存储的各个顶点的空间坐标、颜色和法向量，输出为输入的信息以及计算得到的内置参量 gl_Position；片段着色器的输入为顶点着色器输出的空间坐标、颜色和法向量信息，输出为片段颜色 FragColor。

此外，光源位置、光源颜色、视角方向及摄像机位置等信息通过 uniform 类型数据进行传递。

4.2 着色器编译及使用

创建顶点着色器和片段着色器：

```
1 GLuint VertexShader = glCreateShader(GL_VERTEX_SHADER)
    ;
2 GLuint FragmentShader = glCreateShader(
    GL_FRAGMENT_SHADER);
```

.glsl 文件读取：

```
1 std::string getShader(const char* file) {
2   std::ifstream fileStream(file, std::ios::in);
3   std::stringstream buffer;
4   buffer << fileStream.rdbuf();
5   fileStream.close();
6   return buffer.str();
7 }
```

获取.glsl 文件中的着色器代码：

```
1 std::string VertexShaderCode = getShader("VertexShader
    .glsl");
2 std::string FragmentShaderCode = getShader("
    FragmentShader.glsl");
3 const GLchar* VertexShaderCodeArrary =
    VertexShaderCode.c_str();
4 const GLchar* FragmentShaderCodeArrary =
    FragmentShaderCode.c_str() ;
```

```
5 glShaderSource(VertexShader, 1, &
      VertexShaderCodeArrary, NULL);
6 glShaderSource(FragmentShader, 1, &
      FragmentShaderCodeArrary, NULL);
```

编译着色器:

```
1 glCompileShader(VertexShader);
2 glCompileShader(FragmentShader);
```

创建着色器程序:

```
1 GLuint ShaderProgram = glCreateProgram();
2 glAttachShader(ShaderProgram, VertexShader);
3 glAttachShader(ShaderProgram, FragmentShader);
4 glLinkProgram(ShaderProgram);
5 glDeleteShader(VertexShader);
6 glDeleteShader(FragmentShader);
```

指定着色器程序:

```
1 glUseProgram(ShaderProgram);
```

渲染:

```
1 glDrawArrays(GL_TRIANGLES, 0, 3 * num);
```

4.3 VBO 创建及使用

创建 VBO（和 VAO）:

```
1 GLuint VBO[3], VAO;
2 glGenVertexArrays(1, &VAO);
```

```
glGenBuffers(3, VBO);
```

绑定 VBO：

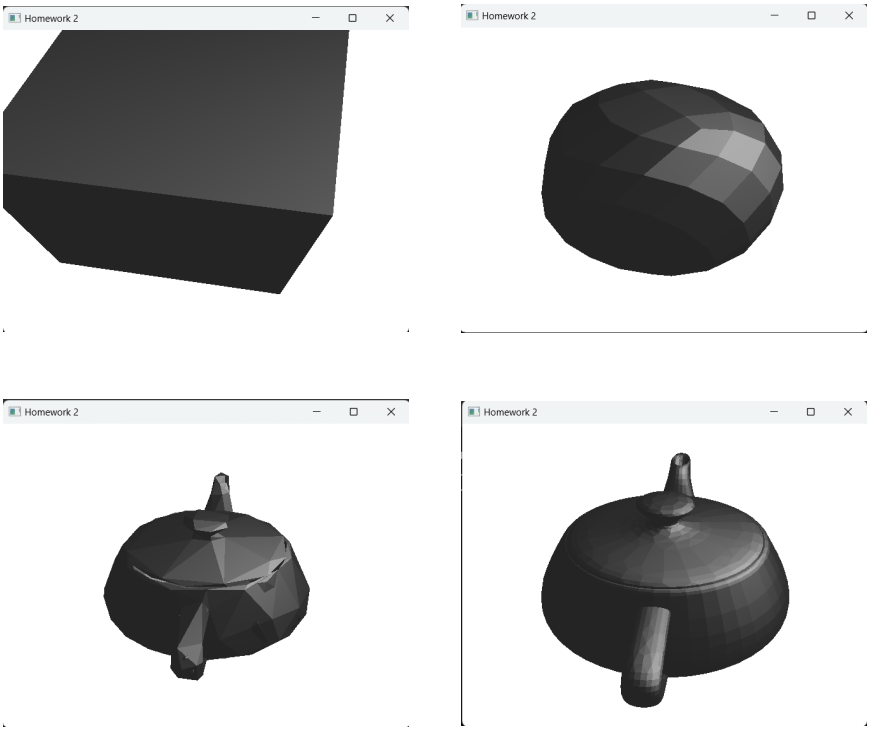
```
// 将顶点位置与VBO[0]绑定
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vec3) * num * 3,
    vertex, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

// 将顶点颜色与VBO[1]绑定
glBindBuffer(GL_ARRAY_BUFFER, VBO[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vec3) * num * 3,
    color, GL_STATIC_DRAW);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

// 将顶点法向量与VBO[2]绑定
glBindBuffer(GL_ARRAY_BUFFER, VBO[2]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vec3) * num * 3,
    normal, GL_STATIC_DRAW);
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(2);
```

清除 VBO（和 VAO）：

```
glDeleteBuffers(3, VBO);
glDeleteVertexArrays(1, &VAO);
```

5 实现效果

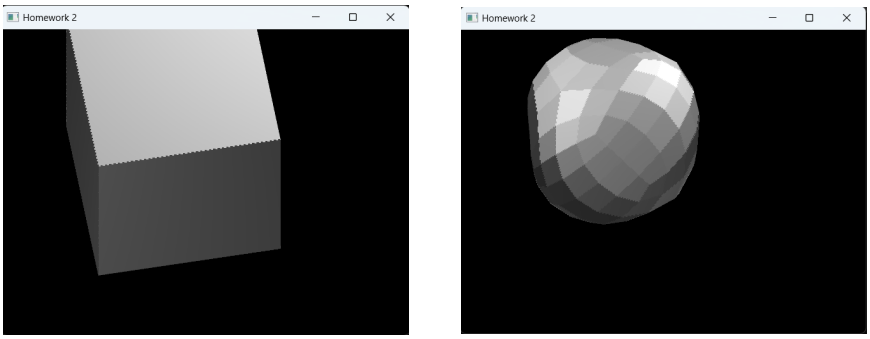
GLSL 渲染实现效果：

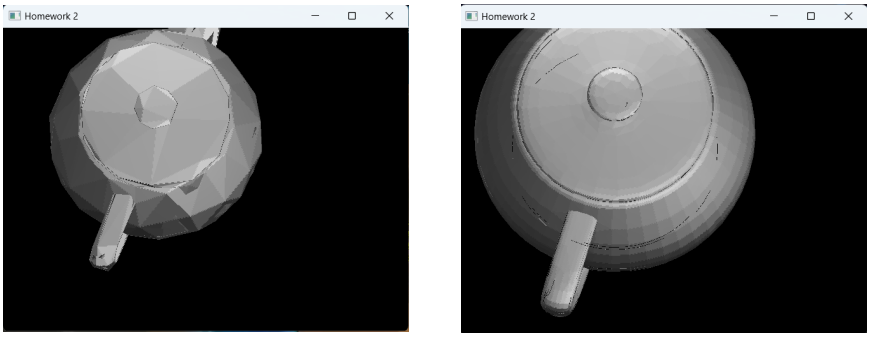


6 性能分析

6.1 渲染效果比较

*HW2 实现效果：





将 HW3（GLSL）中的效果与之前在 HW2 中实现的效果进行比较，我们发现一般的基于 glVertex 的渲染效果中片元之间的颜色变化较明显，而 GLSL 和 VBO 实现的渲染效果中，邻接片元的颜色变化更加缓和，因此整体上显得更为圆润。

此外，在 HW2 中我们并未实现抗锯齿处理，因此在部分片元边缘出现锯齿，但在 HW3（GLSL）中，并没有呈现出明显的锯齿，因此表明 GLSL 和 VBO 方法进行渲染时会实现一定的抗锯齿效果。

6.2 渲染效率比较

通过以下加在渲染代码前后的计时器记录运行时间：

```
1 DWORD t1, t2;
2 t1 = GetTickCount();
3
4 // 渲染
5
6 t2 = GetTickCount();
7 printf("Runtime = %lf s.\n", ((t2 - t1) * 1.0 / 1000))
    ;
```

基于 600 个片元的茶壶模型，针对运行时间比较 HW2 渲染算法与 HW3（GLSL）渲染算法的渲染效率，统计结果如下：

Table 1: HW2 渲染效率

运行次数	1	2	3	4	5
运行用时 (s)	4.453	4.140	4.063	4.093	4.140
平均用时 (s)	**4.178**				

Table 2: HW3(GLSL) 渲染效率

运行次数	1	2	3	4	5
运行用时 (s)	0.016	0.016	0.015	0.016	0.016
平均用时 (s)	**0.016**				

可见，在较多的细分迭代次数下，GLSL 的渲染效率要远远高于 HW2 中的渲染效率。

这是因为使用 GLSL 渲染时，顶点数据通常存储在 GPU 的缓冲区（VBO）中，并通过着色器程序进行处理。这种方式避免了将数据从 CPU 传输到 GPU 的开销，进而提高性能。而基于 glVertex 的渲染需要将顶点数据通过函数调用传输到 GPU，每次绘制都会产生额外的开销。

6.3 VBO 与 glVertex 的区别

（1）数据传输

· VBO：使用 VBO 时，顶点数据存储在 GPU 的缓冲区中。数据只需传输一次到 GPU 内存，然后可以在需要时重复使用。这减少了 CPU 到 GPU 之间的数据传输次数，提高了性能。

· glVertex：使用 glVertex 进行绘制时，每个顶点的数据都需要通过函数调用从 CPU 传输到 GPU。这可能导致频繁的数据传输，增加了 CPU 与 GPU 之间的开销。

（2）内存占用和效率

· VBO：使用 VBO 可以有效地管理和优化内存占用。顶点数据存储在 GPU 内存中，可以在需要时直接使用，无需重复传输。这对于大规模几何数据和复杂场景来说尤其有益，可以减少内存占用并提高渲染效率。

· glVertex：使用 glVertex 进行绘制时，顶点数据通常存储在 CPU 内存中。每次绘制时，数据都需要从 CPU 传输到 GPU，可能导致内存占用更高，并且在每次绘制时都要进行数据传输，可能降低渲染效率。

（3）灵活性

- VBO：使用 VBO 时，可以通过修改顶点缓冲区中的数据来实现动态的顶点变化，从而实现动画效果或实时交互。此外，VBO 可以与着色器程序（如 GLSL）结合使用，提供更大的灵活性和自定义渲染效果的能力。
- glVertex：使用 glVertex 进行绘制时，顶点数据的处理相对受限。它主要适用于简单的静态几何体，功能相对较弱，无法实现复杂的渲染效果。

6.4 Index Array 对效率的影响

- 内存占用：使用索引数组可以减少内存占用，因为同一个顶点可以通过索引在多个三角形之间共享。相比之下，不使用索引数组需要存储每个顶点的完整数据，可能导致内存占用增加。
- 数据传输：使用索引数组可以减少数据传输量。通过发送顶点数据和索引数据到 GPU 并进行一次绑定，可以在渲染时使用索引来引用顶点。这减少了传输到 GPU 的数据量，并减少了 CPU 与 GPU 之间的通信开销。
- 缓存命中率：使用索引数组可以提高缓存命中率。当索引数组与顶点数据一起存储在 VBO 中时，连续的顶点数据可以更好地利用缓存，因为它们在内存中彼此接近。这有助于提高内存访问效率，减少数据读取的延迟。