



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

计算机图形学

第三次作业

陶钧

taoj23@mail.sysu.edu.cn

中山大学 数据科学与计算机学院
国家超级计算广州中心

◉ 作业要求

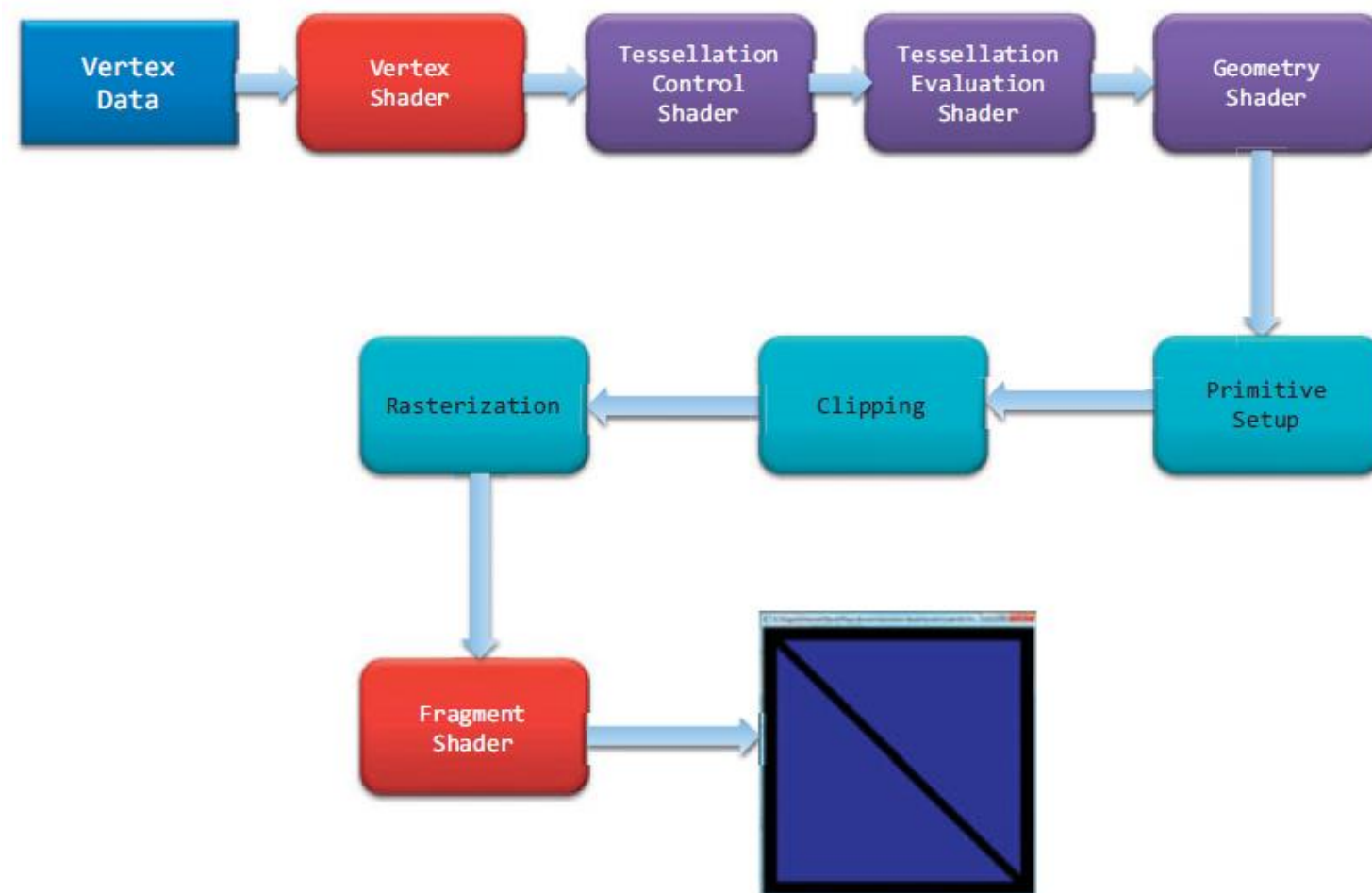
- 绘制一系列沿固定轨迹运动的小球（如太阳系）
- 通过GLSL实现Phong-shading
 - 比较Phong-shading与OpenGL自带的flat与smooth shading的区别
 - 可选做：比较Blinn-Phong shading与Phong shading的区别
- 使用VBO对小球进行绘制
 - 使用足够的细分产生充足的顶点和三角面片，便于计算绘制时间
 - 比较不同细分层级下shading的区别
 - 讨论是否使用VBO及index array的效率区别

◉ 交作业时间：1月13日晚23时59分

◉ 评分标准及slip days的使用与此前作业相同

回顾此前介绍的渲染管线（讲义5）

- 显卡是高度并行化的硬件，对所有数据采用同一工作流程进行处理
- GLSL (GLslang)
 - OpenGL Shading Language
 - 对着色器（shader）进行编程
- Vertex shader
 - 顶点变换，法向量变换
 - 光照
 - 纹理坐标的产生与变换
- Fragment shader
 - 纹理访问及颜色计算，等



- Vertex shader: 齐次坐标顶点→屏幕坐标顶点
 - 同时产生顶点相关的属性（经过插值后将输入至fragment shader）

```
out vec3 normal;  
out vec3 vertex_to_light_vector;  
  
void main()  
{  
    // Transforming The Vertex  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
  
    // Transforming The Normal To ModelView-Space  
    normal = gl_NormalMatrix * gl_Normal;  
  
    // Transforming The Vertex Position To ModelView-Space  
    vec4 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;  
  
    // Calculating The Vector From The Vertex Position To The Light Position  
    vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_modelview_space);  
}
```

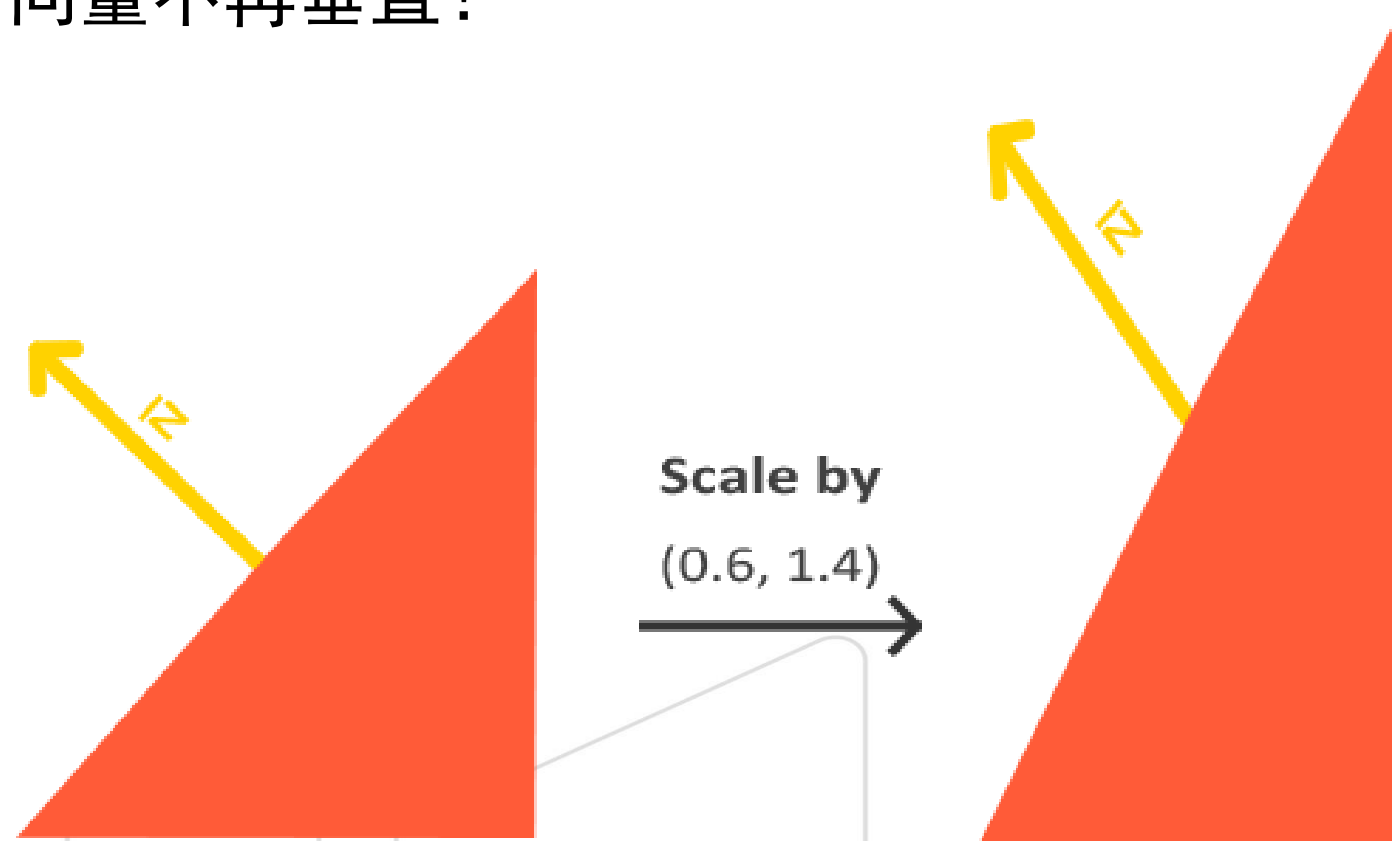
Normal matrix

– 将向量 v 视为两个顶点的差 $p - q$ 则其在camera space中为

- $\mathbf{M} \cdot p - \mathbf{M} \cdot q = \mathbf{M} \cdot (p - q) = \mathbf{M} \cdot v$
- 可适用于切向量，但不适用于法向量
- Nonuniform scaling下，法向量与切向量不再垂直！

– 计算normal matrix

- $n^T \cdot t = 0$
- $(N \cdot n)^T \cdot (M \cdot t) = 0$
- $n^T (N^T M) t = 0$
- $N = (M^{-1})^T$



- Fragment shader: 计算片元颜色
 - 计算过程中可能使用顶点属性插值后得到的片元属性

```
in vec3 normal;  
in vec3 vertex_to_light_vector;  
uniform vec4 ambient_color;
```

```
void main()  
{  
    // Calculating The Final Color  
    gl_FragColor = ambient_color;  
}
```



◉ GLSL中的数据类型

- 向量: `vec2`, `vec3`, `vec4`, `ivec2`, `ivec3`, `ivec4`, `bvec2`, `bvec3`, `bvec4`
- 矩阵: `mat2`, `mat3`, `mat4`
- 纹理: `sampler1D`, `sampler2D`, `sampler3D`, `samplerCube`, `sampler1Dshadow`, `sampler2Dshadow`

◉ GLSL中的数据修饰词

- `uniform`: 对所有顶点而言为常量，不因顶点而异（如光源位置）
- `attribute`: 因顶点而异，只读，只能在vertex shader中使用
- `varying`: vertex shader的输出，fragment shader的输入，传输过程中进行插值
- `in`, `out`: 表明变量为输入或输出

• GLSL中的内置变量

– Vertex shader中的内置attribute

- `gl_Vertex`, `gl_Normal`, `gl_Color`, `gl_MultiTexCoordX`

– 内置uniform

- `gl_ModelViewMatrix`, `gl_ModelViewProjectionMatrix`,
`gl_NormalMatrix`

– Shader输出

- vertex shader: `gl_Position`
- fragment shader: `gl_FragColor`, `gl_FragDepth`

● 编译GLSL着色器程序

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);  
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

```
glShaderSource(vertexShader, 1, &vsource, 0);  
glShaderSource(fragmentShader, 1, &fsource, 0);
```

```
glCompileShader(vertexShader);  
glCompileShader(fragmentShader);
```

```
GLuint program = glCreateProgram();
```

```
glAttachShader(program, vertexShader);  
glAttachShader(program, fragmentShader);
```

```
glLinkProgram(program);
```

● 使用GLSL着色器程序

- **glUseProgram**(program): 指明在接下来的绘制中使用program所代表的着色器程序
- **glUseProgram**(0): 使用默认着色器

● 设置程序中uniform变量的值

- 获取uniform变量在程序中的位置
 - **glGetUniformLocation**(program, "variable_name")
- 设置uniform变量在程序中的值
 - **glUniform{a}{b}{c}**(location, value);
 - {a}: 1, 2, 3, 4
 - {b}: f, i, ui
 - {c}: /, v

- 此前，我们的绘制方式是

- 使用 **glVertex**, **glColor**, **glNormal** 指明顶点位置，颜色，法向量等信息
 - 慢：对每个顶点的每个属性都需要调用一次相应的函数，将数据传至 OpenGL 的 buffer 中，此后在绘制时，将 buffer 中内容传至显存进行绘制
 - 是否必要？
- 使用 GLSL 中的内置变量 **gl_Vertex**, **gl_Normal** 等引用这些信息

```
out vec3 normal;

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    normal = gl_NormalMatrix * gl_Normal;
}
```

- 当前趋势使用 generic attributes，而非 **gl_Vertex** 等内置变量

- 使用VBO将需要绘制的内容放在编程人员创建的buffer中
 - 创建VBO: `void glGenBuffers(GLsizei n, GLuint* ids)`
 - 绑定VBO: `void glBindBuffer(GLenum target, GLuint id)`
 - 拷贝数据至VBO: `void glBufferData(GLenum target, GLsizei size, const void* data, GLenum usage)`
 - `usage`: GL_STATIC_DRAW, GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, GL_DYNAMIC_COPY, GL_STREAM_DRAW, GL_STREAM_READ, GL_STREAM_COPY
 - STATIC: 拷贝一次后不再改变
 - DYNAMIC: 程序运行过程中可能发生改变
 - STREAM: 绘制过程中的每帧都发生改变
 - 删除VBO: `void glDeleteBuffers(GLsizei n, const GLuint* ids)`

- 使用VBO将需要绘制的内容放在编程人员创建的buffer中
 - 完整示例

```
GLuint vbold; // ID of VBO
GLfloat* vertices = new GLfloat[vCount*3]; // create vertex array

// generate a new VBO and get the associated ID
glGenBuffers(1, &vbold);
// bind VBO in order to use
glBindBuffer(GL_ARRAY_BUFFER, vbold);
// upload data to VBO
glBufferData(GL_ARRAY_BUFFER, dataSize, vertices, GL_STATIC_DRAW);
// it is safe to delete after copying data to VBO
delete [] vertices;
...
// delete VBO when program terminated
glDeleteBuffers(1, &vbold);
```

- 使用VBO将需要绘制的内容放在编程人员创建的buffer中
 - 拷贝部分数据至VBO
 - `void glBufferSubData(GLenum target, GLint offset, GLsizei size, void* data)`
 - 在VBO之间拷贝数据
 - `void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr readoffset, GLintptr writeoffset, GLsizeiptr size)`
 - 修改VBO内的数据

```
float data[] = {0.5f, 1.0f, -0.35f, [...]};  
glBindBuffer(GL_ARRAY_BUFFER, buffer);  
// get pointer  
void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);  
// now copy data into memory  
memcpy(ptr, data, sizeof(data));  
// make sure to tell OpenGL we're done with the pointer  
glUnmapBuffer(GL_ARRAY_BUFFER);
```

- 使用VBO进行绘制

- 指明每个属性在VBO中的位置

- `glVertexPointer`, `glColorPointer`, `glNormalPointer`, `glTexCoordPointer`
 - Generic: `glVertexAttribPointer`

- 开启/关闭相应属性的使用

- `glEnableClientState`/`glDisableClientState`
 - `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, `GL_TEX_COORD_ARRAY`
 - Generic: `glEnableVertexAttribArray`/`glDisableVertexAttribArray`

- 绘制函数

- `glDrawArrays`, `glMultiDrawArrays`, `glDrawElements`, `glMultiDrawElements`, `glDrawRangeElements`



● 使用VBO进行绘制（使用内置变量）

```
// bind VBOs for vertex array and index array
glBindBuffer(GL_ARRAY_BUFFER, vboId1); // for vertex attributes
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboId2); // for indices

glEnableClientState(GL_VERTEX_ARRAY); // activate vertex position array
glEnableClientState(GL_NORMAL_ARRAY); // activate vertex normal array
glEnableClientState(GL_TEXTURE_COORD_ARRAY); // activate texture coord array

// do same as vertex array except pointer
glVertexPointer(3, GL_FLOAT, stride, offset1); // last param is offset, not ptr
glNormalPointer(GL_FLOAT, stride, offset2);
glTexCoordPointer(2, GL_FLOAT, stride, offset3);

// draw 6 faces using offset of index array
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, 0);

glDisableClientState(GL_VERTEX_ARRAY); // deactivate vertex position array
glDisableClientState(GL_NORMAL_ARRAY); // deactivate vertex normal array
glDisableClientState(GL_TEXTURE_COORD_ARRAY); // deactivate vertex tex coord array

// bind with 0, so, switch back to normal pointer operation
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

- 使用VBO进行绘制（使用generic属性）

```
// bind VBOs for vertex array and index array
glBindBuffer(GL_ARRAY_BUFFER, vboId1); // for vertex coordinates
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboId2); // for indices

glEnableVertexAttribArray(attribVertex); // activate vertex position array
glEnableVertexAttribArray(attribNormal); // activate vertex normal array
glEnableVertexAttribArray(attribTexCoord); // activate texture coords array

// set vertex arrays with generic API
glVertexAttribPointer(attribVertex, 3, GL_FLOAT, false, stride, offset1);
glVertexAttribPointer(attribNormal, 3, GL_FLOAT, false, stride, offset2);
glVertexAttribPointer(attribTexCoord, 2, GL_FLOAT, false, stride, offset3);

// draw 6 faces using offset of index array
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_BYTE, 0);

glDisableVertexAttribArray(attribVertex); // deactivate vertex position
glDisableVertexAttribArray(attribNormal); // deactivate vertex normal
glDisableVertexAttribArray(attribTexCoord); // deactivate texture coords

// bind with 0, so, switch back to normal pointer operation
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

• 使用内置变量 vs 使用generic属性

– 内置变量

```
out vec3 normal;

void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    normal = gl_NormalMatrix * gl_Normal;
}
```

– generic属性

```
layout (location=0) in vec3 vertex_attrib;
layout (location=1) in vec3 normal_attrib;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
out vec3 normal;

void main()
{
    gl_Position = projection*view*model*vertex_attrib;
    normal = mat3(transpose(inverse(view*model))*normal_attrib);
}
```

Questions?