2018-2019年度第二学期 00106501

计算机图形学

童 伟 华　管理科研楼1205室

E-mail: tongwh@ustc.edu.cn

中国科学技术大学　数学科学学院

http://math.ustc.edu.cn/

第三节 GLSL(II)

基础知识

- GLSL的语法来自于C/C++
  - 字符集，记号（tokens），标识符（identifiers）
  - 分号， 大括号嵌套
  - 控制流：if-else, for, while, do-while
  - 关键字
  - 注释：//…和/*…*/
  - 入口函数：void main()

数据类型

- 标量
  - float：IEEE 32位浮点值
    - 单精度，字面浮点常数的后缀f或F是可选的
  - double：IEEE 64位浮点值
    - 双精度，字面浮点常熟的后缀lF或LF是强制的
  - int：有符号的32位整数
    - 十进制、八进制、十六进制：42, 052, 0x2A
    - 不支持长整型，不支持按位运算
  - uint：无符号的32位整数
  - bool 布尔型：true/false
    - 关系运算和逻辑运算产生布尔型
    - 条件跳转表达式（if, for, ?:, while, do-while）只接受布尔型

数据类型

- 向量
  - 二维到四维的float、double、int、uint、bool型
    - vec2、vec3、vec4
    - dvec2、dvec3、dvec4
    - ivec2、ivec3、ivec4
    - uvec2、uvec3、uvec4
    - bvec2、bvec3、bvec4
  - 分量访问
    - (x, y, z, w) 位置或方向
    - (r, g, b, a) 颜色
    - (s, t, p, q) 纹理坐标

数据类型

- 矩阵
  - 2×2, 2×3, 2×4, 3×2, 3×3, 3×4, 4×2, 4×3, 和 4×4浮点矩阵
    - mat2, mat3, mat4
    - mat2x2, mat2x3, mat2x4
    - mat3x2, mat3x3, mat3x4
    - mat4x2, mat4x3, mat4x4
  - 双精度的浮点矩阵前面加前缀d
  - 第一个数指定列数，第二个数指定行数
  - 列优先。例如4列3行矩阵：mat4x3 M;
    - M[2]是第3列，类型vec3；M[3][1]是第4列第2行元素

数据类型

- 采样器（sampler）
  - 着色器访问纹理的不透明句柄
    - sampler[1,2,3]D 访问1D,2D,3D纹理
    - samplerCube 访问立方图纹理
    - Sampler[xxxx]MS 多重采样版本
    - Sampler[xxxx]Array 数组版本
    - samplerRect 二维矩形纹理
    - samplerBuffer 一维纹理缓存
  - 例子：

```
uniform sampler2D Grass;
vec4 color = texture2D(Grass, coord);
```

数据类型-结构（struct）

- 结构成员可以是基本类型和数组
- 不支持位字段
- 不支持嵌套的匿名结构
- 不支持嵌入的结构定义

```
struct light {
float intensity;
vec3 position;
} lightVar;

light lightVar2;
```

```
struct S { float f; };

struct T {
S; // Error: 匿名结构
struct { ... }; // Error:
S s; // Okay: 嵌套结构
};
```

数据类型

数组

- 可以声明基本类型和结构的数组
- 除了作为函数参数外，声明时可以不指定大小
- GLSL 4.3之后的版本支持多维数组
- 可用length方法来查询数的大小

1、可通过再次声明指定数组大小，但之后不能再次声明

```
vec4 points[];
vec4 points[10];
vec4 points[20]; // error
vec4 points[];   // error
```

2、编译时根据最大下标确定数组大小

```
vec4 points[];
points[2]=vec4(1.0); // 3
points[7]=vec4(2.0); // 8
int size=points.length();
```

数据类型

- void
  - 声明函数不返回任何值
  - 没有缺省的函数返回值
  - 除了再用于空的函数形参列表外，不能用于其他声明

```
void main()
{
      …
}
```

隐式类型转换

- GLSL支持的隐式类型转换
  - int -> uint, float, double
  - uint -> float, double
  - float -> double
  - 标量、向量及矩阵
- 不支持数组和结构的隐式转换
  - int数组不能隐式转换为float数组

作用域

- GLSL的作用域规则和C++相似
- 在所有函数定义之外声明的变量具有全局作用域
- 花括号内声明的变量作用域在花括号内
- while测试和for语句中声明的变量作用域限于循环体内
- if语句不允许声明新变量

存储限定符

- 声明变量时可以在类型前指定存储限定符
  - < none: default > 局部读写或函数输入参数
  - const 只读的编译时常量或函数参数
  - in 着色器阶段的输入变量
  - out 着色器阶段的输出变量
  - uniform 应用程序传递给着色器的数据，它对于给定的图元而言是一个常量
  - buffer 应用程序共享的一块可读写内存
  - shared 本地工作组中共享的变量（仅用于计算着色器）

存储限定符

- 全局变量只能指定为限定符const、attribute、uniform和varying中的一个
- 局部变量只能使用const限定符
- 函数参数只能使用const限定符
- 函数返回值和结构字段不能使用限定符
- 缺省限定符
  - 没有限定符的全局或局部变量不能和应用程序和其他着色器交换信息

const限定符

- const变量必须在声明时初始化
  - const vec3 zAxis = vec3 (0.0, 0.0, 1.0);
- 结构变量可以声明为const，但结构的字段不能限定为const

in和out限定符

- 分别定义着色器阶段的输入与输出变量
- 逐顶点或逐片元的变量

uniform限定符

- unifrom指定一个在应用程序中设置好的变量，在图元的处理过程中不会发生变化
  - 不能在glBegin和glEnd间改变其值
  - 不能在glDrawElement等绘制命令改变其值
- 着色器无法写入uniform变量，也无法改变其值
- 可以修饰任何类型的变量，包括结构和数组

buffer修饰符

- buffer修饰符指定随后的块作为着色器与应用程序共享的一块内存缓存，这块缓存对着色器来说是可读的也是可写的
- 缓存的大小可以在着色器编译和程序链接完成后设置
- 常与buffer变量一同使用

初始化和构造函数

- 变量可以在声明的时候初始化
  - 整型常量可以用八进制、十进制和十六进制
  - 浮点值必须包括一个小数点，除非是用科学计数法，如3E-7，后缀f或F可选
  - 布尔型为true或false
- 聚合类型（向量、矩阵、数组和结构）必须用构造函数进行初始化

构造函数

```
vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba = vec4(1.0); // sets each component to 1.0
vec3 rgb = vec3(color); // drop the 4th component
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

$$\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$$

```
struct light {
float intensity;
vec3 position;
};
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));

const float c[3] = float[3](5.0, 7.2, 1.1);
const float d[3] = float[](5.0, 7.2, 1.1);
```

类型转换

- GLSL不支持C风格的强制类型转化，只能用构造函数进行显式类型转换
- true转换为1或1.0，false转换为0或0.0
- 0或0.0转换为false，非0转换为true

```
float f = 2.3;
bool b = bool(f);

float f = float(3); // convert integer 3 to floating-
point 3.0 float g = float(b); // convert Boolean b to
floating point vec4 v = vec4(2); // set all components
of v to 2.0
```

访问向量元素

- 可以用[]或者选择运算符(.)逐个索引向量的元素
  - x, y, z, w 位置或方向数据
  - r, g, b, a 颜色数据
  - s, t, p, q 纹理坐标， 注意OpenGL是s,t,r,q
  - `a[2], a.b, a.z, a.p` 是一样的
- 混合（Swizzling）运算符可以用来操纵每个分量

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

混合操作

- 各分量名称可打乱顺序，但只能使用同一组名称，且长度不能大于4
- 右值中名称可以重复，左值不可以

```
vec4 v4;
v4.rgb; // is a vec3
v4.xgba; // is illegal

vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 dup = pos.xxyy; // dup = (1.0, 1.0, 2.0, 2.0)
pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)
pos.xx = vec2(3.0, 4.0); // illegal - 'x' used twice
```

运算符

- GLSL中使用的运算符优先级和结合性和C是一致的
  - ++ --
  - - !
  - 算术运算符： * / + -
  - 关系运算符：< > <= >=
  - 逻辑运算符： == != && ^^(异或) ||
  - 选择运算符：?:
  - 赋值： = += -= *= /=

逐分量运算

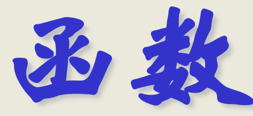
- 一般地，向量和矩阵运算是逐分量进行的
- 例外情形：矩阵乘向量、向量乘矩阵以及矩阵相乘按线性代数运算规则

```
vec4 v, u;  float f;
v = u + f; // v = (u.x + f, u.y + f, u.z + f, u.w + 
f);

mat4 m;
v * u; // 逐分量乘，不是内积
v * m; // 行向量乘矩阵
m * v; // 矩阵乘列向量
m * m; // 矩阵相乘
```

流控制

- GLSL的流控制和C++非常相似，着色器的入口点是main函数
- 循环结构：for、while和do-while，for和while语句中可以声明变量，作用域持续到子语句结束。break和continue同C
- 选择结构：if和if-else，switch语句
  - 限制：不允许在第一个case之前添加语句
- discard 仅用在片段着色器中丢弃片段



函数

- 函数的使用和C++类似，函数名可以通过参数个数和类型进行重载
- 函数声明：

```
returnType functionName (type0 arg0, type1 arg1, ...,
typen argn);
```

- 函数定义：

```
returnType functionName (type0 arg0, type1 arg1, ...,
typen argn)
{
// do some computation
return returnValue;
}
```

参数限定符

- GLSL按值-返回（value-return）调用函数
- 参数类型type*n*必须指定类型和可选的参数限定符
  - < none: default > 同in
  - in 复制到函数中，函数内可写
  - const in 复制到函数中，函数内只读
  - out 返回时从函数中复制出来
  - inout 复制进函数并在返回时复制出来

参数和返回值类型

- 参数和返回值可以是任意类型
  - 数组必须显式指定大小
  - 数组传入或返回时只需使用变量名，且大小和函数声明中匹配
- 函数必须声明返回值类型，无返回值时声明为void
- 输入参数从左往右求值
- 函数不能递归调用

内置函数

- 角度和三角函数：逐分量
- 指数函数：逐分量
- 常用函数：逐分量
- 几何函数：向量
- 矩阵函数
- 向量关系函数
- 整函数
- 纹理函数
- 原子计数器函数
- 原子内存函数
- 图形函数
- 片元处理函数
- 噪声函数
- 几何着色器函数
- …

顶点着色器

- 顶点着色器必须负责如下OpenGL固定功能的逐顶点操作：
  - 模型-视图矩阵和投影矩阵没有应用于顶点坐标
  - 纹理矩阵没有应用于纹理坐标
  - 法向量没有变换到观察坐标，且没有重新缩放或规范化
  - 没有对启用GL_AUTO_NORMAL执行法向规范化
  - 没有自动生成纹理坐标
  - 没有执行每顶点的光照计算
  - 没有执行彩色材料计算（glColorMaterial）
  - 没有执行彩色索引光照计算
  - 当设置当前光栅位置时，将应用上述所有操作

顶点着色器

- 下述操作将应用到顶点着色器所得到的顶点值
  - 颜色钳位（clamping）和掩模（masking）
  - 裁剪坐标的透视除法
  - 包括深度范围缩放的视口映射
  - 包括用户定义裁剪平面的裁剪
  - 前向面确定
  - 平面明暗处理
  - 颜色、纹理坐标、雾、点大小等一般属性裁剪
  - 最终颜色处理

顶点着色器的输入变量

- 应用程序提供顶点信息：位置、法向、颜色、纹理坐标等，应用程序通过in变量传入
- 内置顶点属性：

```
in int gl_VertexID;
in int gl_InstanceID;
in int gl_DrawID; // Requires GLSL 4.60
in int gl_BaseVertex; // Requires GLSL 4.60
in int gl_BaseInstance; // Requires GLSL 4.60
```

顶点着色器的输出变量

- 输出变量使用out修饰符，前面还可以加插值修饰符
- 内置的输出变量：

```
out vec4 gl_Position;
out float gl_PointSize;
out float gl_ClipDistance[];
```

  - gl_Position：顶点着色器必须写入的齐次裁剪坐标，将用于图元装配、裁剪、剔除
  - gl_PointSize：将被光栅化的点大小
  - Gl_ClipDistance[]：顶点到用户定义的裁剪平面距离

顶点着色器示意图

Built-in attribute variables

gl_Color
gl_Normal
gl_Vertex
gl_MultiTexCoord0
etc.

User-defined attribute variables

StartColor
Velocity
Elevation
Tangent
etc.

Provided directly by application
Provided indirectly by application
Produced by the vertex processor

User-defined uniform variables

ModelScaleFactor, EyePos, Epsilon,
LightPosition, WeightingFactor1, etc.

Texture maps

Vertex Processor

Built-in uniform variables

gl_ModelViewMatrix, gl_FrontMaterial,
gl_LightSource[0..n], gl_Fog, etc.

Built-in varying variables

gl_FrontColor
gl_BackColor
gl_FogFragCoord
etc.

Special output variables

gl_Position
gl_PointSize
gl_ClipVertex

User-defined varying variables

Normal
ModelCoord
RefractionIndex
Density
etc.

片段着色器

- 片段着色器取代固定功能的纹理、颜色求和、雾等片段操作
  - 没有应用纹理环境和纹理函数
  - 没有执行纹理应用程序
  - 没有应用颜色求和
  - 没用应用雾化

片元着色器的输入变量

- 顶点着色器（或者细分着色器、几何着色器）的输出变量
- 内置片元属性：

```
in vec4 gl_FragCoord;
in bool gl_FrontFacing;
in float gl_ClipDistance[];
in vec2 gl_PointCoord;
in int gl_SampleMaskIn;
```

  - Gl_FragCoord：片元的窗口坐标
  - Gl_FrongFacing：片元的朝向
  - Gl_ClipDistance[]：顶点到用户定义的裁剪平面距离
  - gl_PointCoord：用于点块纹理（point sprites），片元在点（光栅化后形成像素块）中的位置
  - gl_SampleMaskIn：多重采样的掩码

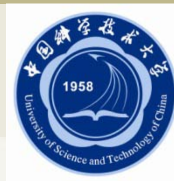
片元着色器的输出变量

- 片元着色器可以利用out修饰符向帧缓存（frame buffer）输出一系列的“颜色”
- 内置的输出变量：

```
out float gl_FragDepth;
out int gl_SampleMask[];
```

  - Gl_FragDepth：片元的深度值，缺省取gl_FragCoord.z
  - Gl_SampleMask：多重采样的掩码，缺省取gl_SampleMaskIn

片段着色器示意图



内置常量

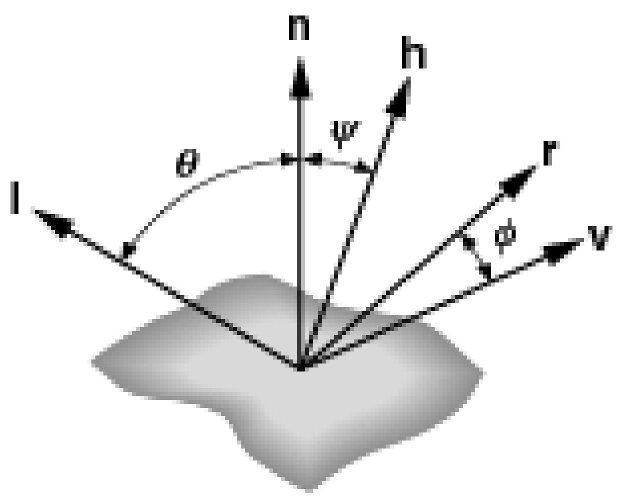
```
const ivec3 gl_MaxComputeWorkGroupCount = {65535,
65535,65535};
const ivec3 gl_MaxComputeWorkGroupSize = {1024,1024,64 };
const int gl_MaxComputeUniformComponents = 1024;
const int gl_MaxComputeTextureImageUnits = 16;

const int gl_MaxComputeImageUniforms = 8;
const int gl_MaxComputeAtomicCounters = 8;
const int gl_MaxComputeAtomicCounterBuffers = 1;
const int gl_MaxVertexAttribs = 16;
const int gl_MaxVertexUniformComponents = 1024;

const int gl_MaxVertexOutputComponents = 64;
const int gl_MaxGeometryInputComponents = 64;
const int gl_MaxGeometryOutputComponents = 128;
const int gl_MaxFragmentInputComponents = 128;
const int gl_MaxVertexTextureImageUnits = 16;
…
```

光照模型

$$I = k_d I_d \, \mathbf{l} \cdot \mathbf{n} + k_s I_s (\mathbf{h} \cdot \mathbf{n})^{\alpha} + k_a I_a$$



光照计算的坐标系

- 思考：光照计数选择哪个坐标系？
- 视点坐标系
- 下述代码把顶点位置变换为视点坐标

```
layout(location = 0) in vec4 aPosition;
uniform mat4 ModelViewMatrix;

// Transform vertex to eye coordinates
vec4 vPosition = ModelViewMatrix * aPosition;
```

法向变换

- 在视点空间计算光照时，法向也必须变换到视点空间

```
normal = NormalMatrix * aNormal;
```

- 法向规范化：

```
normal = normalize(normal);
```

- 思考：变换矩阵M与法向变换矩阵N之间的关系？

$$N = (M^{-1})^{T}$$

RenderMonkey & ShaderGen

- 着色器集成开发环境：包括编辑、快速原型验证、编译、预览、错误调试等
  - https://gpuopen.com/archive/gamescgi/rendermonkey-toolsuite/
- 自动生成模拟固定流水线功能着色器源代码的程序
  - https://github.com/mellinoe/ShaderGen

Thanks for your attention!