第七章 OPENGL

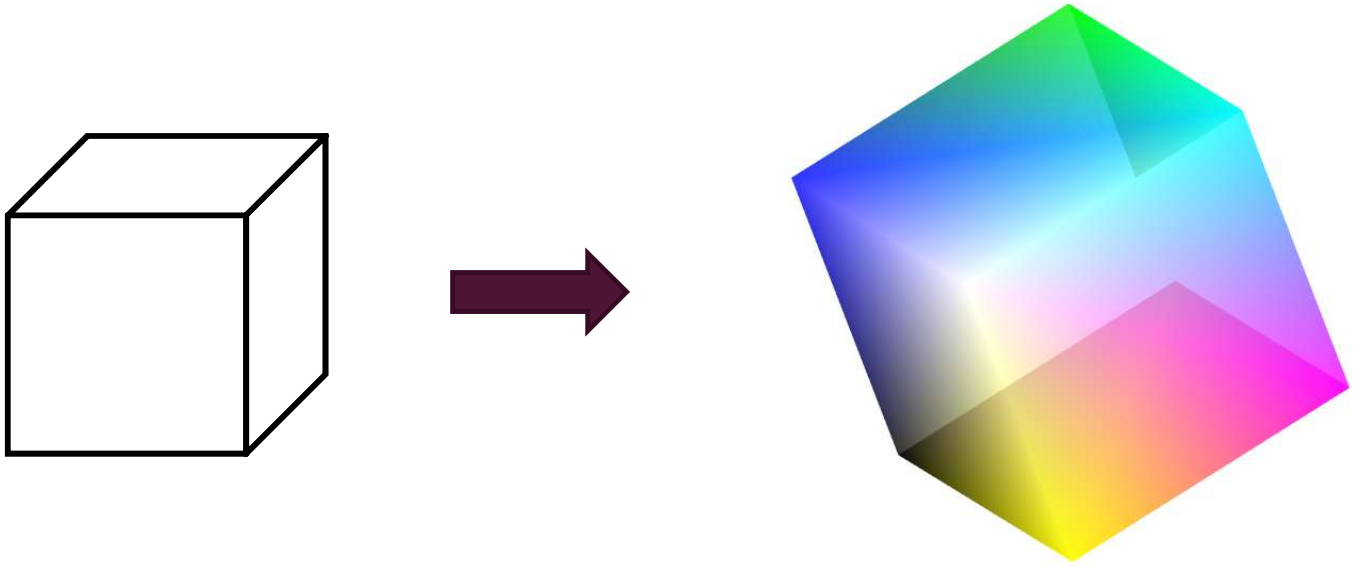
清华大学软件学院

徐枫 2019-10-08

3D图形绘制

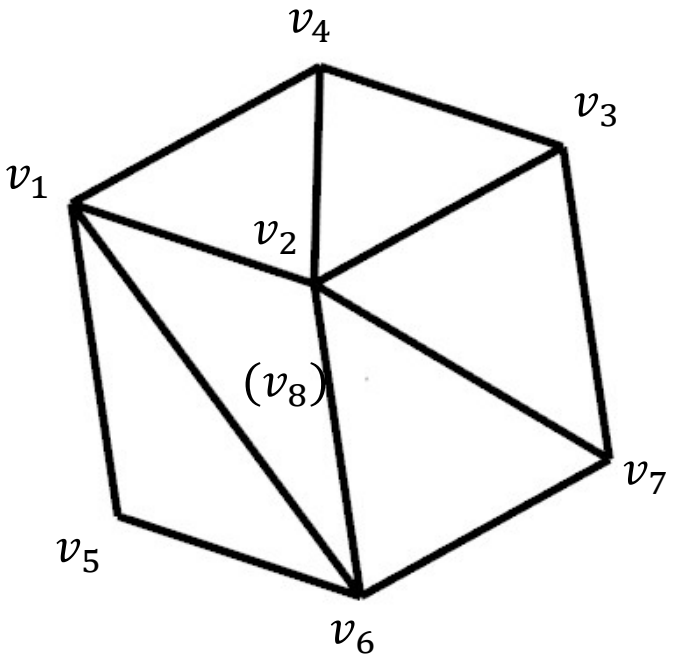
- 3D 场景
- 物体
  - 形状
  - 外观
- 观察者
- 投影

如何绘制一个立方体



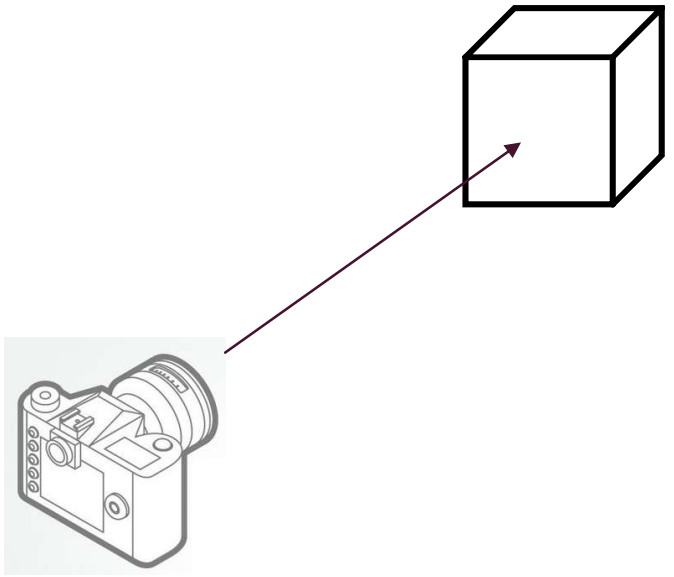
如何绘制一个立方体

- 在计算机中表示一个立方体
  - 8个顶点，6个面，12个三角形
  - 局部坐标系
  - 面的颜色、面的材质、面上的图案…
- 确定立方体在场景中的哪个位置
  - 桌上？床上？地上？
  - 世界坐标系



如何绘制一个立方体

- 观察者
  - 在场景中的哪个位置？
  - 看向什么方向？
  - 能够看到多大的范围？

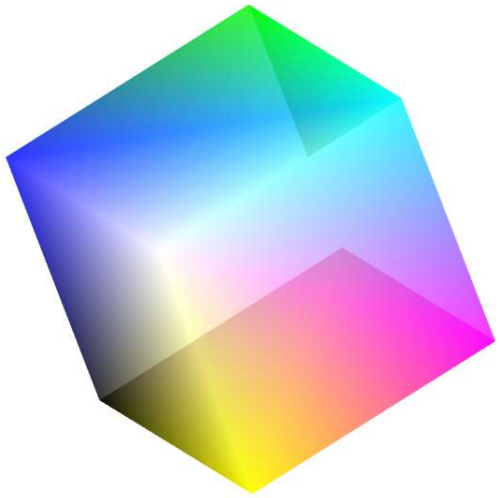


- 三维到二维
  - 三维场景如何映射成二维的？
  - 正交投影、透视投影

如何绘制一个立方体

- 映射到二维后，会发生什么变化？
  - 一些部分被遮挡了（背面）
  - 光照的影响（阴影、高光）

- 在显示器上显示



OPENGL

- OpenGL帮助我们更容易地实现上述过程
  - 以及更多更为复杂的绘制效果
- OpenGL是
  - 图形硬件的一种软件接口
  - 创建交互式的三维应用程序

- 特点
  - 可移植性
  - 高质量、高性能
  - 可扩展性
  - 开放性

OPENGL发展历史

- 1992年，SGI正式发布OpenGL 1.0标准
  - 最初的调用模式（后被称为**固定渲染管线**）
  - 易于理解
- 2004年，推出2.0版本，支持**着色器**、GLSL语言
  - 应用可编程渲染管线的概念
  - 灵活操作顶点运算和像素运算
- 06年以后，opengl 标准移交给 khronos 小组
- 2009年，推出3.1版本
  - 弃用了1.0标准的调用模式
  - 由**可编程渲染管线**替代
- 如今，4.6版本（2017年）
- 2016，Vulkan 1.0，下一代 OpenGL

OPENGL库

- 核心库——GL
  - OpenGL的核心函数库
  - 主要功能包括物体描述、几何变换、纹理、材质、光照、像素、位图、文字处理等。

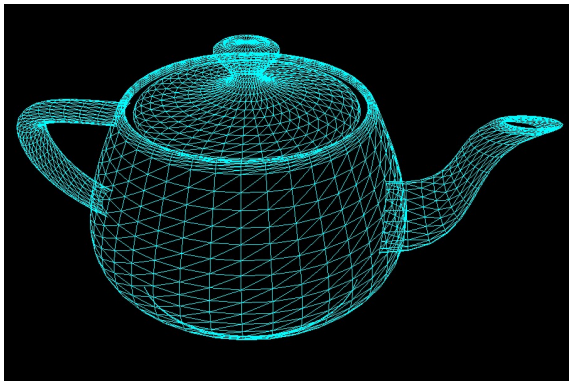
- 实用库——GLU
  - 利用gl库中的函数执行一些特定的任务
  - 主要功能包括绘制二次曲面、NURBS曲线曲面、辅助纹理贴图、坐标转换和投影变换、多边形分格化、简单三维物体（椭球）

其他库

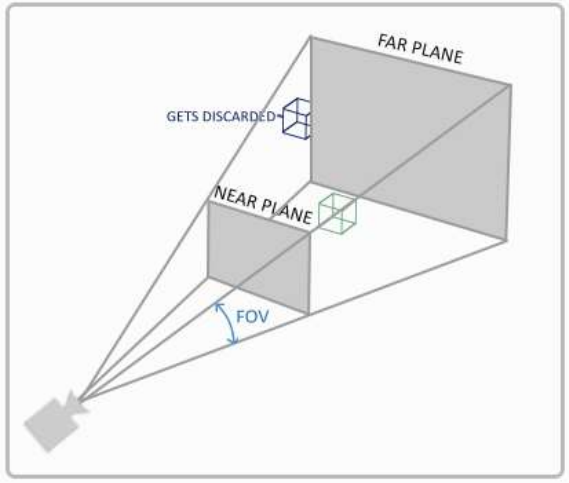
名称	功能	备注
辅助库 Glaux	窗口管理、输入输出处理、绘制一些简单三维物体	用于辅助学习OpenGL；被Glut库取代。
工具库 Glut	窗口操作、鼠标键盘事件、绘制长方体、球、茶壶	可移植（跨平台）；停止更新，freeglut替代。
Glext	适应不断变化的新特性	这些扩展太新，还没有添加到gl中
WGL	针对Windows的扩展	
GLX	针对Unix/Linux的扩展	
Glew	跨平台的C/C++扩展	管理平台支持的全部OpenGL扩展
Glee	与Glew类似	

OPENGL功能

- 物体描述（建模）：
  - 基本图元（点、线、多边形…）；
  - 曲线、曲面（二次曲面、NURBS）；
  - 三维物体（椭球、茶壶、网格）

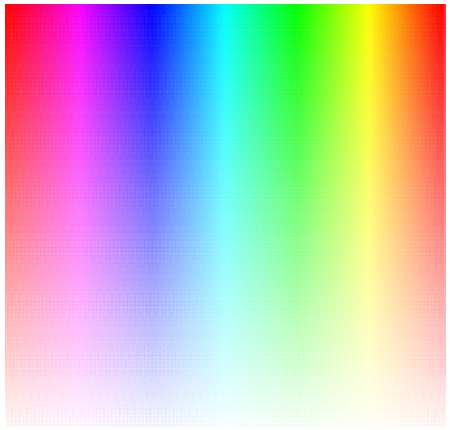


- 变换：
  - 几何变换（平移、旋转、缩放）；
  - 投影变换（正投影、透视投影）；
  - 视口变换



OPENGL功能

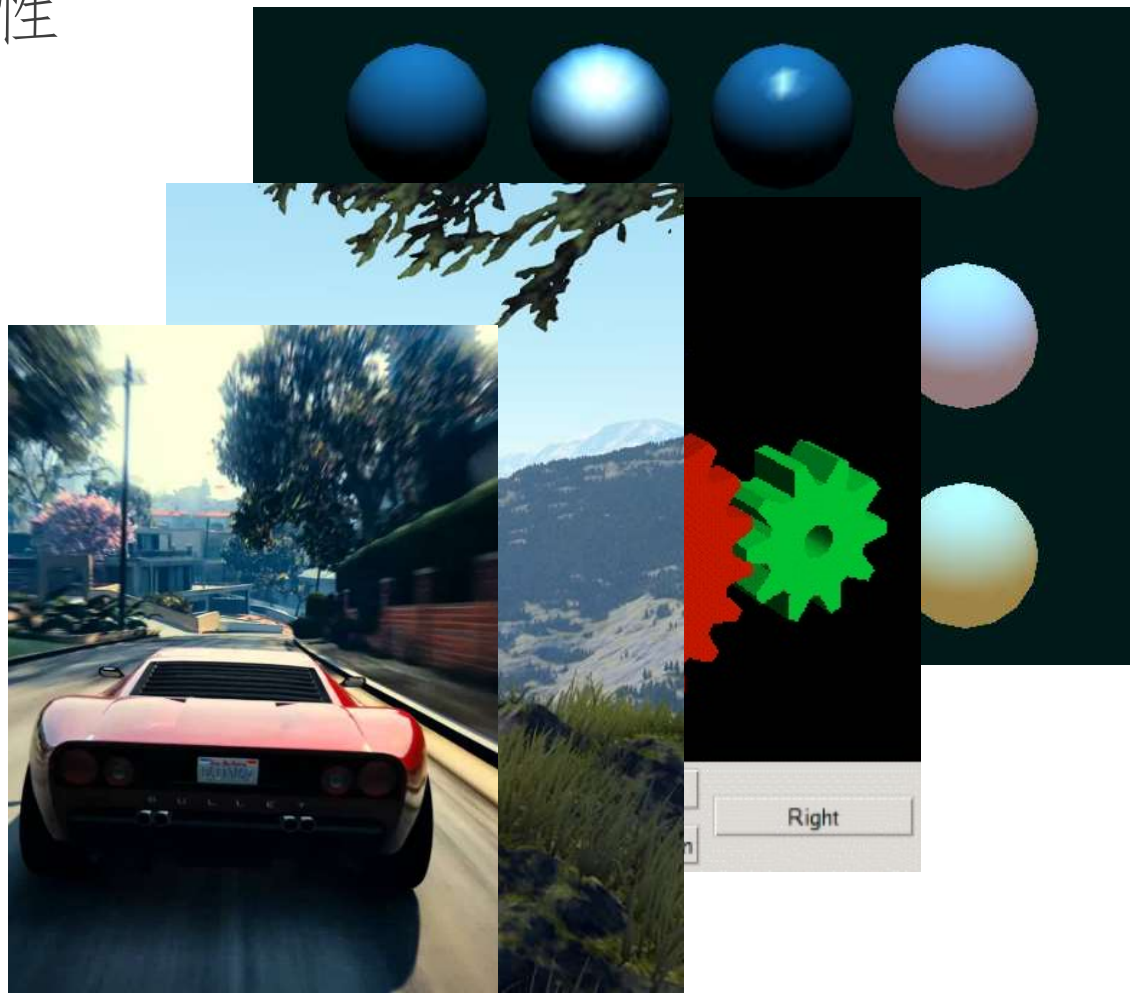
- 颜色模式：
  - RGBA模式，红、绿、蓝、alpha；
  - 索引模式，颜色表



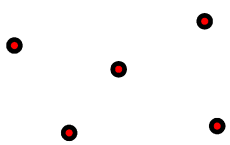
- 像素处理：
  - 像素、位图、字体和图像处理
  - 图像效果：混合、抗锯齿、雾
  - 纹理贴图

OPENGL功能

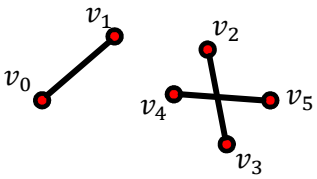
- 更多高级特性
  - 材质和光照
  - 双缓存动画
  - 深度暗示
  - 运动模糊
  - …



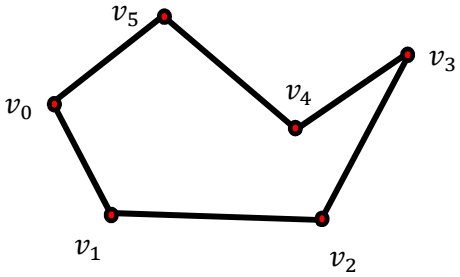
OPENGL物体描述 - 基本图元



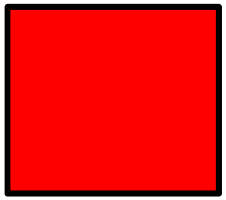
点
(GL_POINTS)



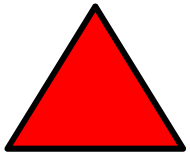
线段
(GL_LINES)



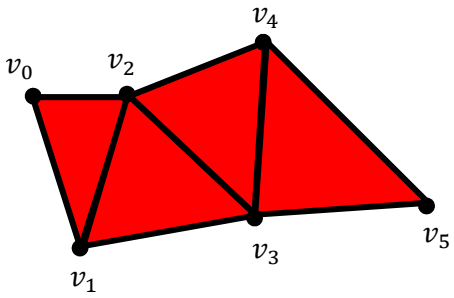
线段环
(GL_LINE_LOOP)



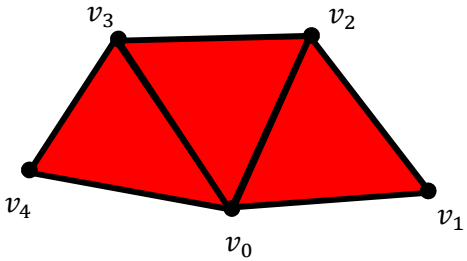
四边形
(GL_QUADS)



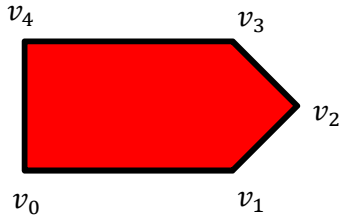
三角形
(GL_TRIANGLES)



三角形条带
(GL_TRIANGLE_STRIP)

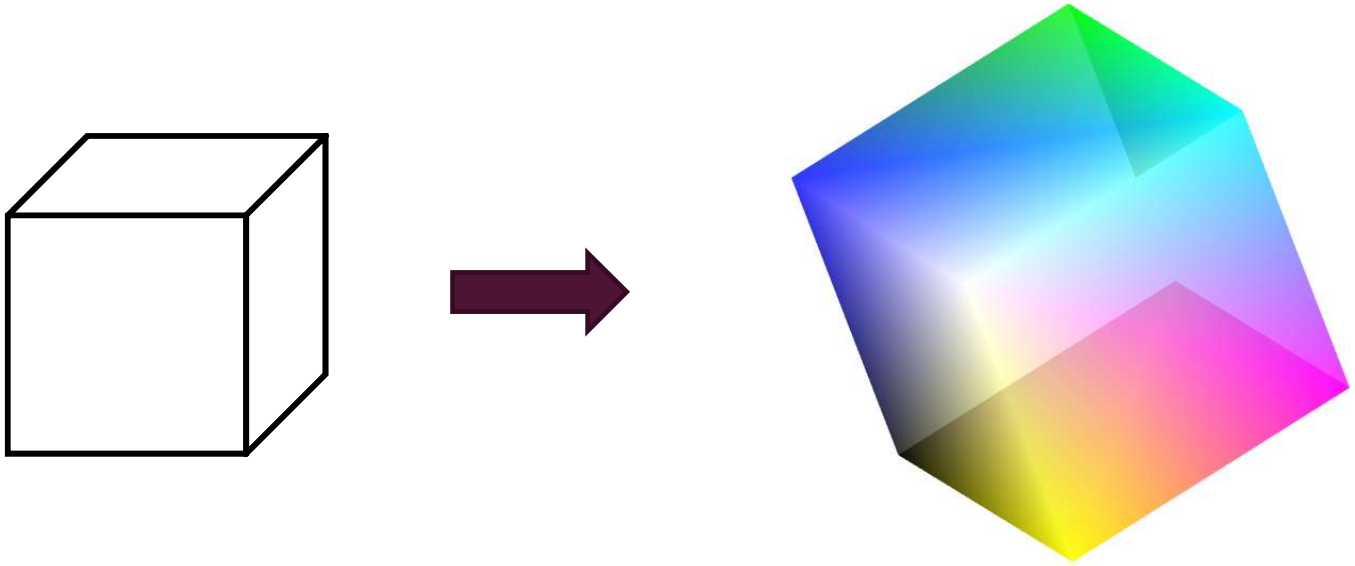


三角形扇
(GL_TRIANGLE_FAN)



简单凸多边形
(GL_POLYGON)

OPENGL绘制示例——立方体



OPENGL绘制示例——立方体

- 定义8个顶点的坐标

```
static const float verts[8][4] =
{
    { -1, -1, -1, 1 },
    { 1, -1, -1, 1 },
    { 1, 1, -1, 1 },
    { -1, 1, -1, 1 },
    { -1, -1, 1, 1 },
    { 1, -1, 1, 1 },
    { 1, 1, 1, 1 },
    { -1, 1, 1, 1 }
};
```

- 定义8个顶点的颜色

```
static const float vertColors[8][4] =
{
    { 0, 0, 0, 0.8f },
    { 1, 0, 0, 0.8f },
    { 0, 1, 0, 0.8f },
    { 0, 0, 1, 0.8f },
    { 1, 1, 0, 0.8f },
    { 1, 0, 1, 0.8f },
    { 0, 1, 1, 0.8f },
    { 1, 1, 1, 0.8f }
};
```

- 定义6个面的顶点顺序

```
static const uint quadFaces[6][4] =
{
    { 0, 1, 5, 4 },
    { 4, 5, 6, 7 },
    { 1, 2, 6, 5 },
    { 0, 4, 7, 3 },
    { 2, 3, 7, 6 },
    { 1, 0, 3, 2 }
};
```

OPENGL绘制示例——立方体

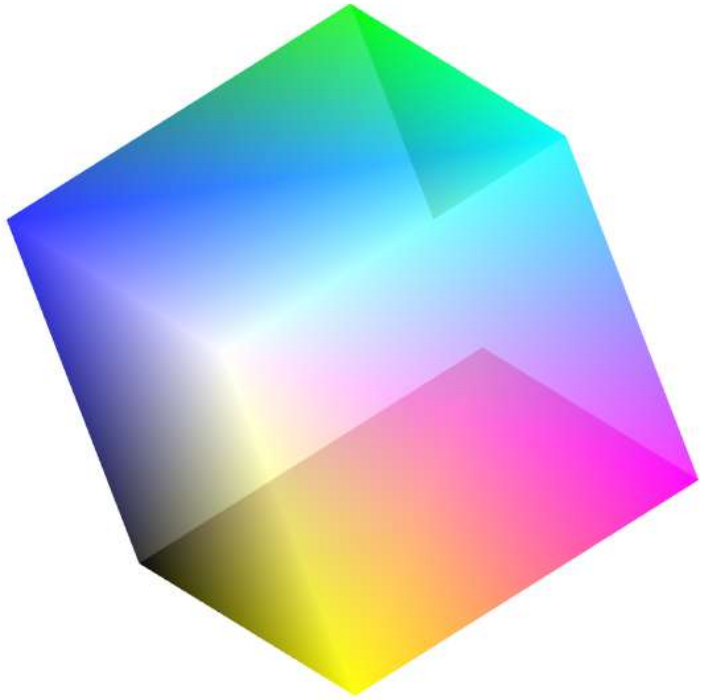
- 绘制
  - OpenGL是一个状态机
  - OpenGL 函数命名规则

```
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(modelViewMatrix.data());
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(projectionMatrix.data());

// draw a cube
glBegin(GL_QUADS);
for (int i = 0; i < 6; i++) {
    for (int j = 0; j < 4; j++) {
        uint vertId = quadFaces[i][j];
        glColor4fv(vertColors[vertId]);
        glVertex4fv(verts[vertId]);
    }
}
glEnd();
```

OPENGL绘制示例——立方体

- 运行结果



OPENGL绘制示例——立方体

- 上述绘制方法基于 OpenGL 固定功能渲染管线
- OpenGL 3.1 核心模式删除了这些功能

- 现代 OpenGL
  - 可编程管线 – 着色器
  - 更灵活
  - 高性能

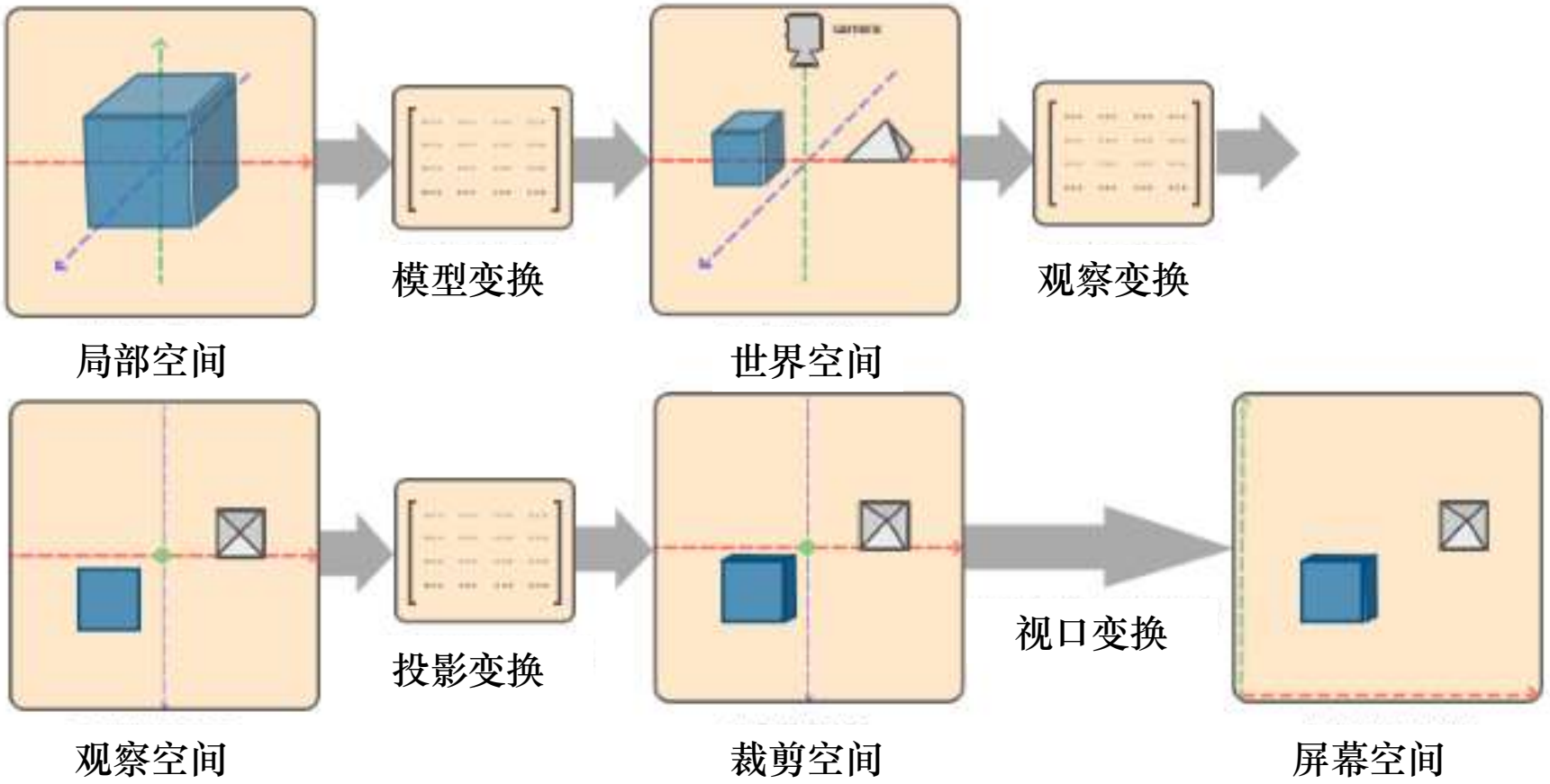
```
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(modelViewMatrix.data());
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(projectionMatrix.data());

// draw a cube
glBegin(GL_QUADS);
for (int i = 0; i < 6; i++) {
    for (int j = 0; j < 4; j++) {
        uint vertId = quadFaces[i][j];
        glColor4fv(vertColors[vertId]);
        glVertex4fv(verts[vertId]);
    }
}
glEnd();
```

几何变换

- 把物体的三维坐标变换为屏幕坐标
  - 变换：
    - 包括模型、视图和投影变换
    - 由矩阵乘法表示
  - 比喻：相机

几何变换



几何变换 - GLM 库

- OpenGL Mathematics
- 辅助 OpenGL 的 C++ header-only 库，方便矩阵运算
- 功能：矩阵变换，四元数，随机数等
- 使用：官网下载，包含有关头文件

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
```

几何变换 - GLM 示例

- 将向量（1，0，0）平移（1，1，0）

```
glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
glm::mat4 trans;
trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
vec = trans * vec;
```

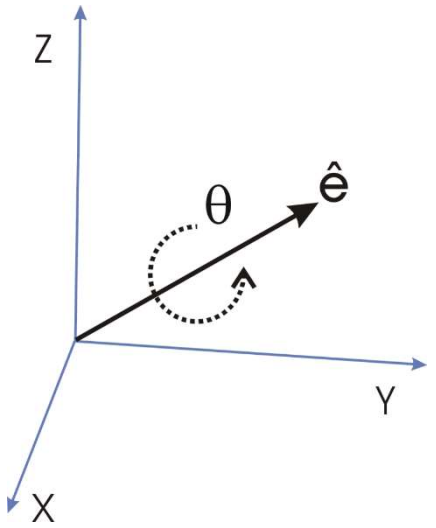
- translate 函数：对矩阵执行平移变换，相当于右乘平移变换矩阵

几何变换 - 模型变换

- 设置模型的位置和方向
- 平移
  - `mat4 translate(mat4 const & m, vec3 const & v);`
  - $P'_x = P_x + x; P'_y = P_y + y; P'_Z = P_z + z$
- 缩放、反射
  - `mat4 scale(mat4 const & m, vec3 const & v);`
  - $P'_x = P_x \cdot x; P'_y = P_y \cdot y; P'_Z = P_z \cdot z$

几何变换 - 模型变换

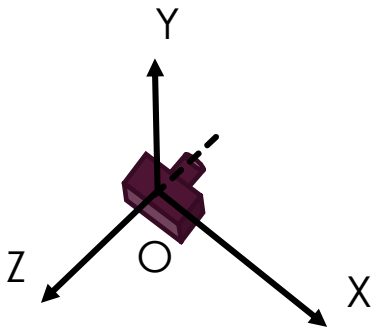
- 模型变换
  - 旋转
    - mat4 rotate(mat4 const & m, Type angle, vec3 const & axis);
    - 使用rodrigues旋转公式
    - 轴：$\hat{e} = (x, y, z)$
    - 角：$\theta = angle \cdot \frac{\pi}{180}$
    - *公式较为复杂，此处不再列出



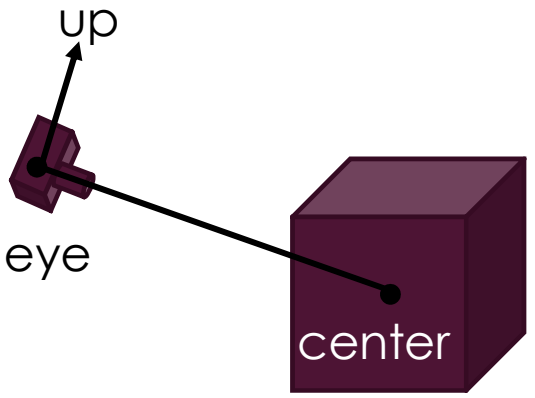
顶点操作 - 观察变换

- 观察（视图）变换
  - 世界坐标系→观察坐标系，

```
mat4 lookAt( vec3 & eye,
  vec3 & center,
  vec3 & up);
```

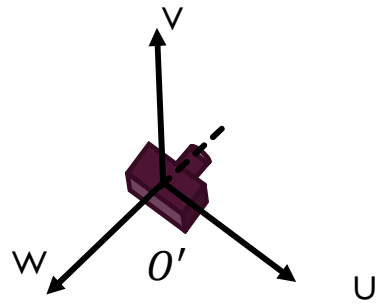


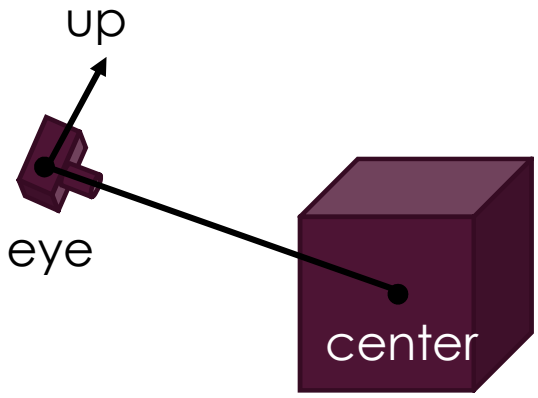
  - 默认情况下
    - 相机位于原点
    - 指向z轴负方向
    - 朝上（up）向量为$(0,1,0)$



几何变换 - 观察变换

- 建立视觉（观察）坐标系$O'(u,v,w)$
  - $w = \frac{eye-center}{\|eye-cente\ \|}$
  - $u = \frac{up \times w}{\|up \times w\|}$
  - $v = w \times u$
  - $v \neq up$





几何变换 - 观察变换

- $O \rightarrow O'$，先平移后旋转，$M = RT$

- 平移矩阵$\mathrm{T} = \begin{bmatrix} 1 & 0 & 0 & -eyex \\ 0 & 1 & 0 & -eyey \\ 0 & 0 & 1 & -eyez \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- 旋转矩阵$R = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

几何变换 - 投影变换

- 定义一个视景体

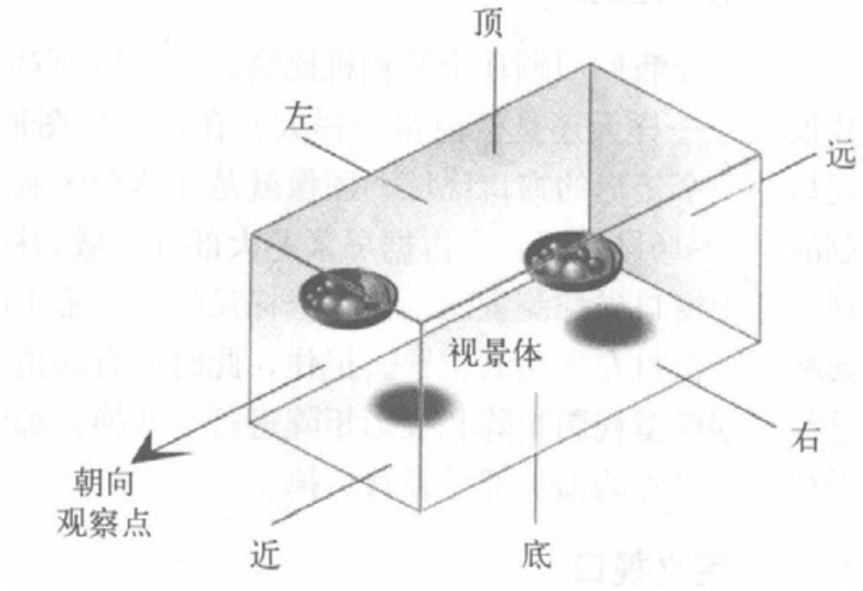
- 视景体的作用：
  - 决定物体如何映射到屏幕上（正投影 or 透视投影）
  - 决定哪些物体被裁剪

投影变换 - 正交投影

- 视景体是一个长方体；保大小、保角度

```
mat4 ortho(Type left, Type right,
     Type bottom, Type top, Type near, Type far);
```

- 近裁剪面左下角
  - $(left, bottom, -near)$
- 远裁剪面右上角
  - $(right, top, -far)$
- $near \neq far$

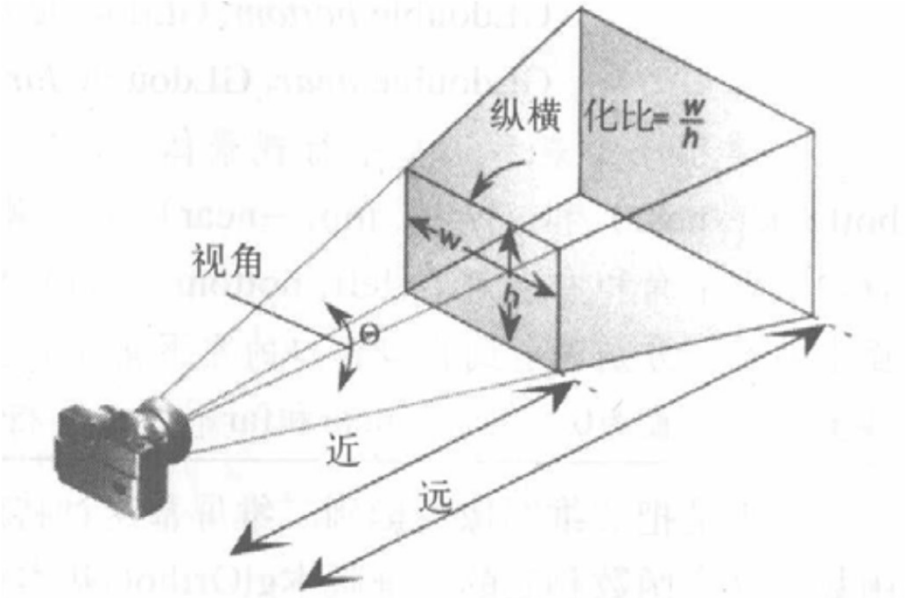


投影变换 – 透视投影

- 视景体是一个金字塔的平截头体；距相机越远，看起来越小

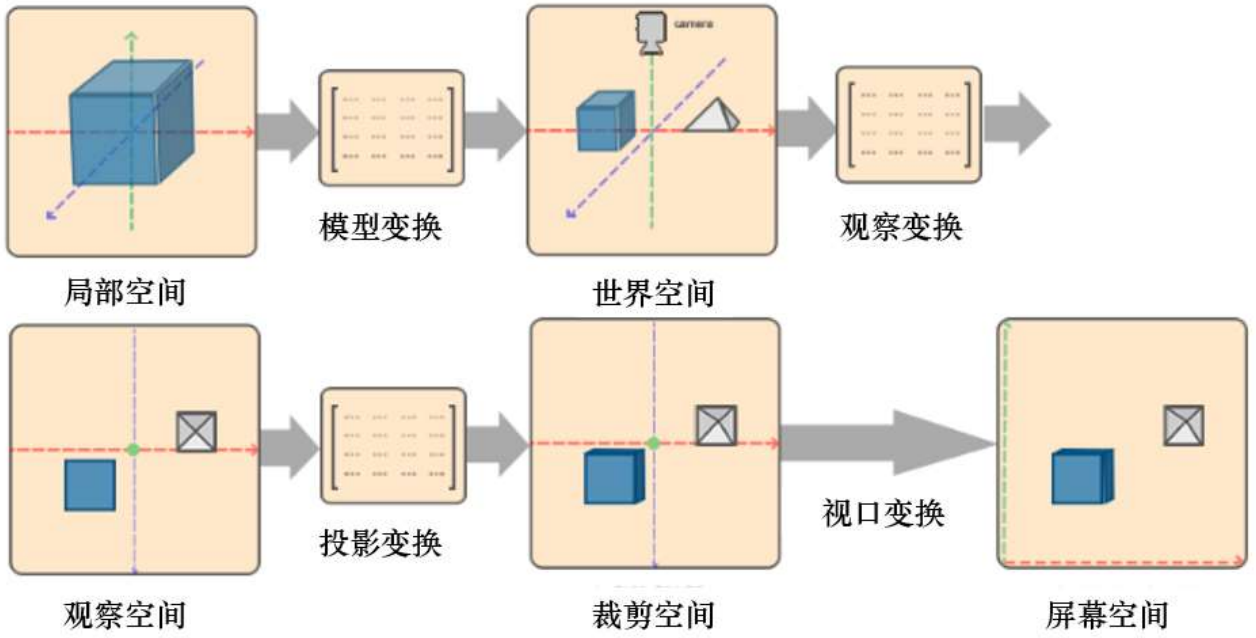
```
mat4 perspective(Type fovy, Type aspect,
    Type near, Type far);
```

- fovy：y方向上的视角θ
- aspect $= \frac{w}{h}$
- near和far为正值



几何变换 - 组合变换

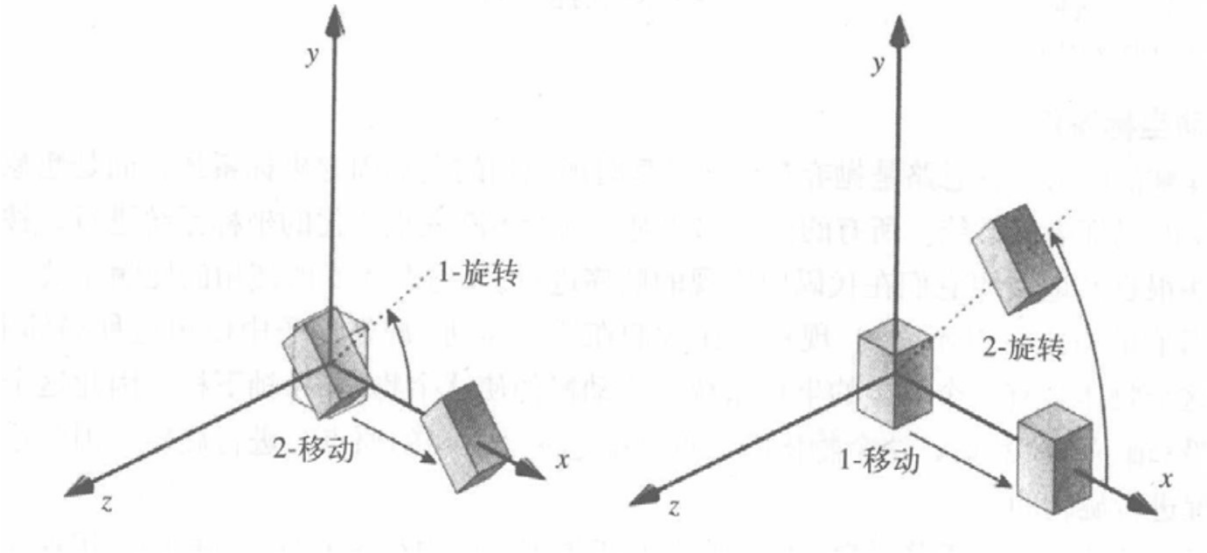
- 将模型变换，观察变换，投影变换组合起来



$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

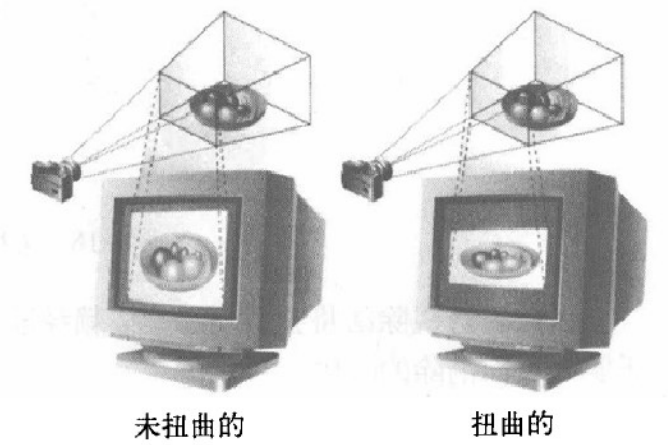
顶点操作 - 组合变换

- 变换的不同执行顺序会得到不同的结果



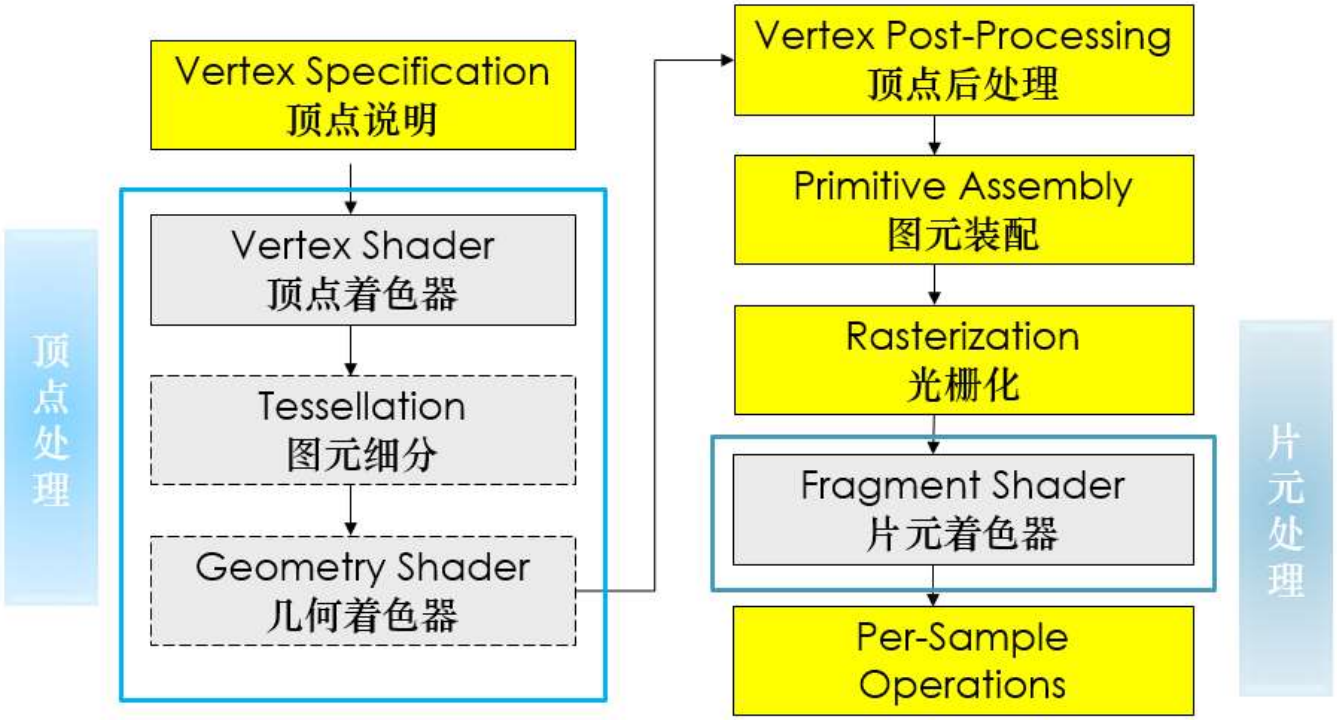
视口变换

- 视口变换
  - 视口是一个矩形窗口区域，图像在其中绘制
- `void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);`
  - 定义一个像素矩形
  - 最终图像映射到这个矩形中
  - 左下角：(x, y)
  - 宽：width；高：height
- 视口与视景体的纵横比一般相同
  - 否则图像会变形



OPENGL 渲染管线

- 渲染管线
  - OpenGL 进行渲染的一系列处理阶段



OPENGL渲染管线

- 固定功能管线
  - OpenGL 1.0中，对顶点和片元进行的处理是固定的
  - 这种模式称为“固定功能的管线”
    - glVertex*, glColor* → 频繁的CPU/GPU数据传输
- 可编程管线
  - 现代OpenGL则使用“可编程着色管线”
    - 使用可大规模并行的GPU程序（着色器）来处理顶点和片元数据，效率更高
    - 开放更底层的渲染管线，灵活性更强
    - 在 OpenGL 4中，固定管线API已被移除

OPENGL渲染管线

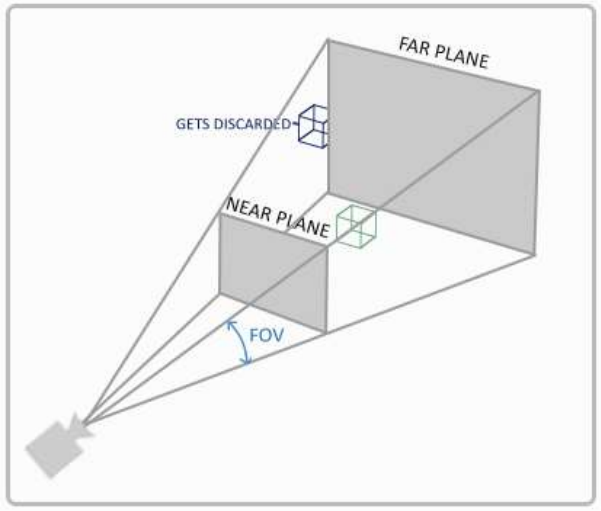
- 顶点说明（Vertex Specification）
  - 程序向渲染管线传递一系列的顶点
  - 顶点组成一个个图元
  - 每个顶点可以有多个属性
    - 位置、法向、颜色、…
  - 解决两个问题
    - 发送顶点数据，并在显卡中存储
    - 声明顶点数据格式：顶点属性，数据类型等

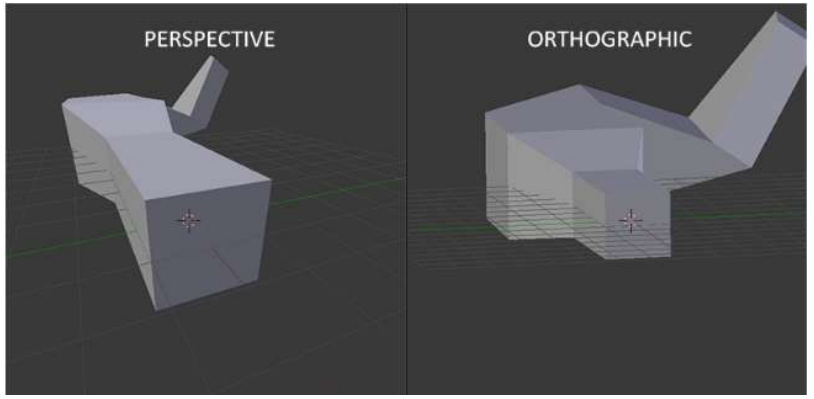
OPENGL渲染管线

- 顶点处理
  - 顶点着色器（Vertex Shader）
    - 可编程的
    - 对每一个输入顶点独立进行处理
    - 输出的顶点数 = 输入的顶点数
    - 将顶点变换到裁剪空间（clip-space）下
  - 图元细分（Tessellation）
  - 几何着色器（Geometry Shader）

OPENGL渲染管线

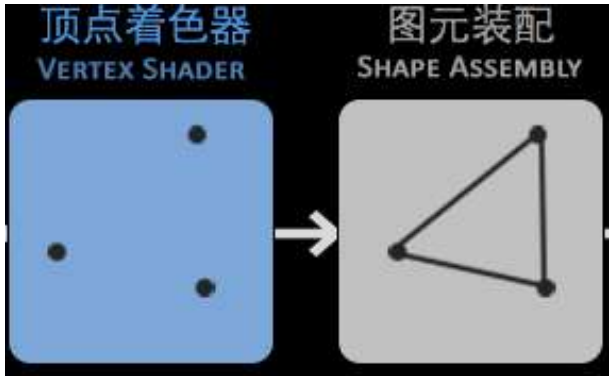
- 顶点后处理（Vertex Post-Processing）
  - 顶点处理之后的固定功能的处理阶段
  - 裁剪（clipping）
  - 裁剪空间（clip-space） → 屏幕空间（window space）
    - 透视除法
    - 视口变换



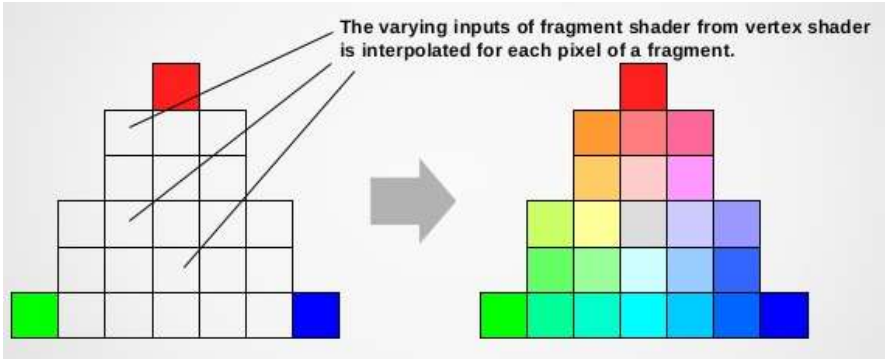


OPENGL渲染管线

- 图元装配（Primitive Assembly）
  - 顶点组合成图元
  - 图元剔除（Face Culling）
    - 根据图元的朝向决定是否继续渲染



- 光栅化（Rasterization）
  - 图元→片元（fragments），“候选像素”
  - 片元是一个状态的集合，用于计算一个像素的最终数据



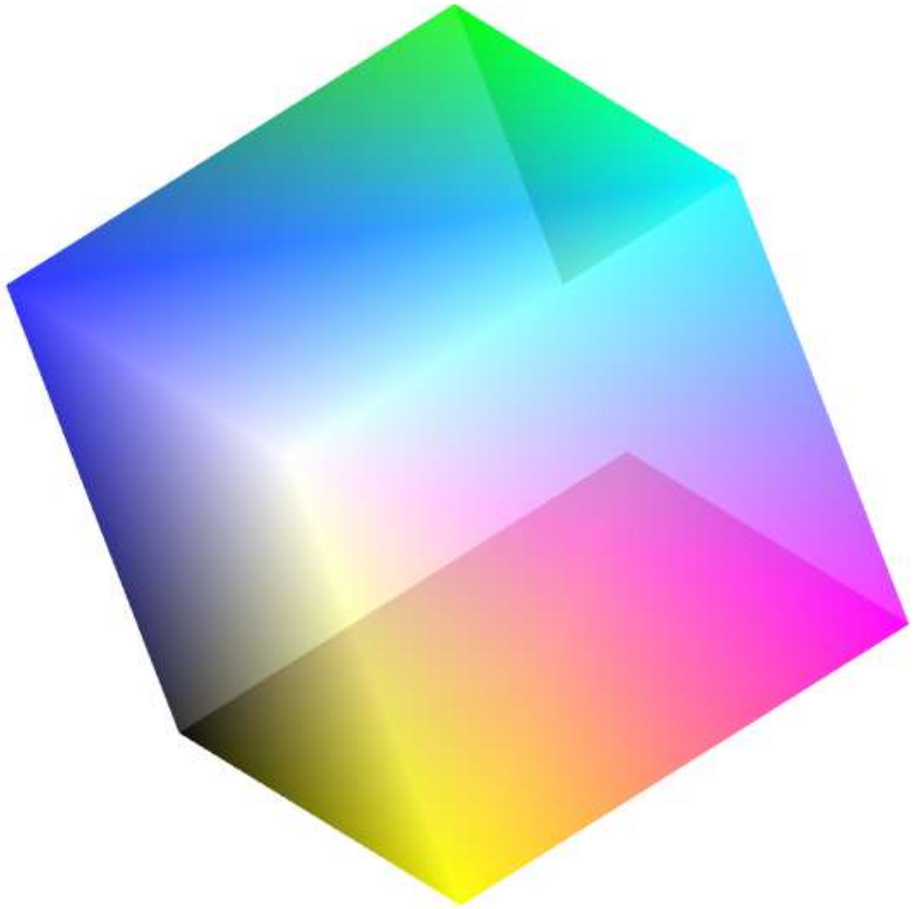
OPENGL渲染管线

- 片元着色器（Fragment Shader）
  - 可编程的
  - 处理每一个片元（fragment）
  - 输出包括一组颜色值，一个深度值和一个模板缓冲值
- Per-Sample Operations
  - 根据用户配置，对片元执行一系列测试
    - 剪裁(Scissor)测试、模板(Stencil)测试、深度(Depth)测试、
  - 颜色混合
- 片元数据写入帧缓冲区

OPENGL渲染管线

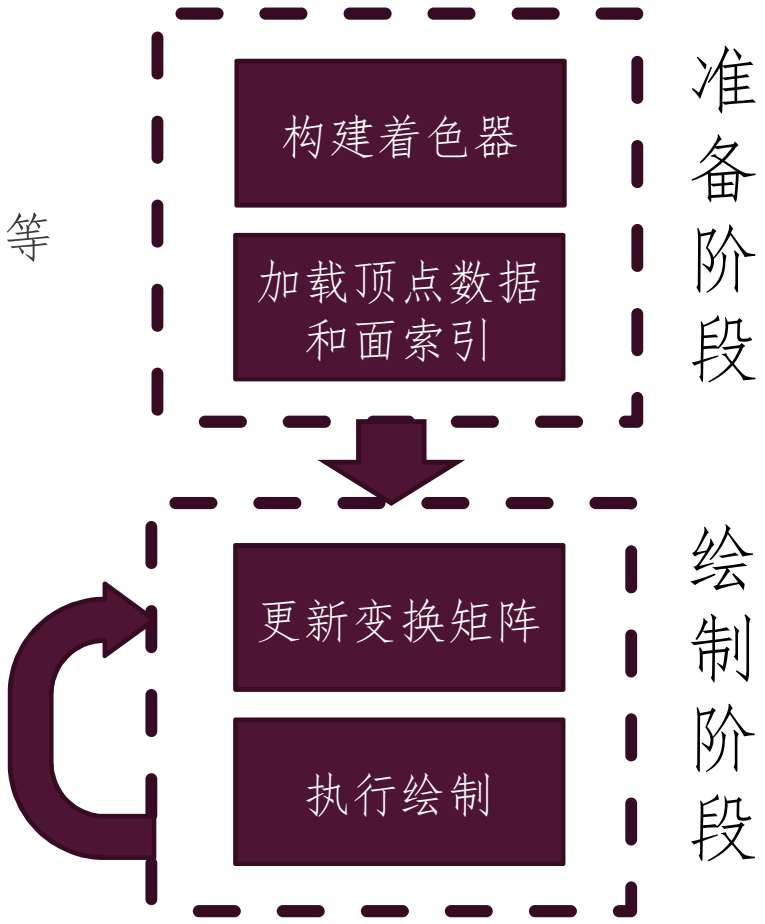
- 最重要的两个环节
  - 顶点着色器（Vertex Shader）
    - 对单个顶点数据进行操作
    - 三维 → 二维 —— 模型、视图、投影变换
    - 确定顶点在屏幕上的二维坐标以及深度缓冲
  - 片元着色器（Fragment Shader）
    - 片元：“候选像素”
    - 对单个片元进行处理 —— 纹理、材质、光照
    - 确定片元的最终颜色

目标：绘制一个彩色立方体



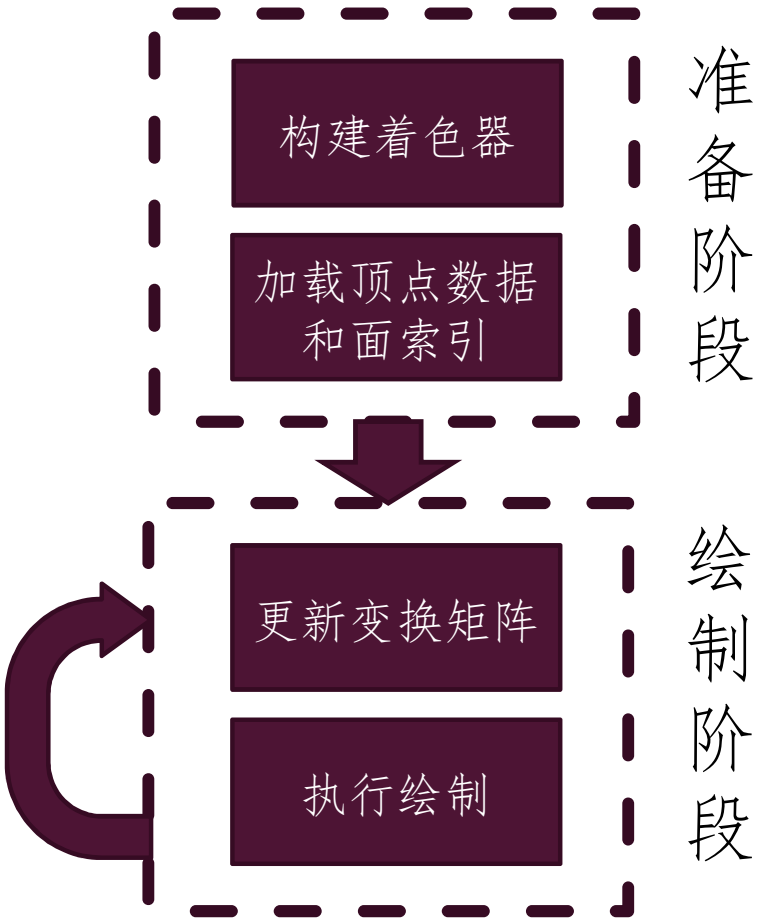
OPENGL 程序结构

- 两种类型的输入
  - 不随时间变化
    - 网格顶点的位置、法向、颜色等
    - 网格面环的索引顺序
    - 绘制逻辑
  - 随时改变
    - 模型矩阵、视图矩阵等等
- 如何提高绘制效率？
  - 不随时间变化→提前加载
  - 随时改变→绘制时设置



准备阶段

- 定义顶点数据与面索引
- 构建着色器
- 从着色器程序获取变量位置（地址）
- 加载顶点数据至缓冲区对象



定义顶点数据与面索引

- 定义8个顶点的坐标

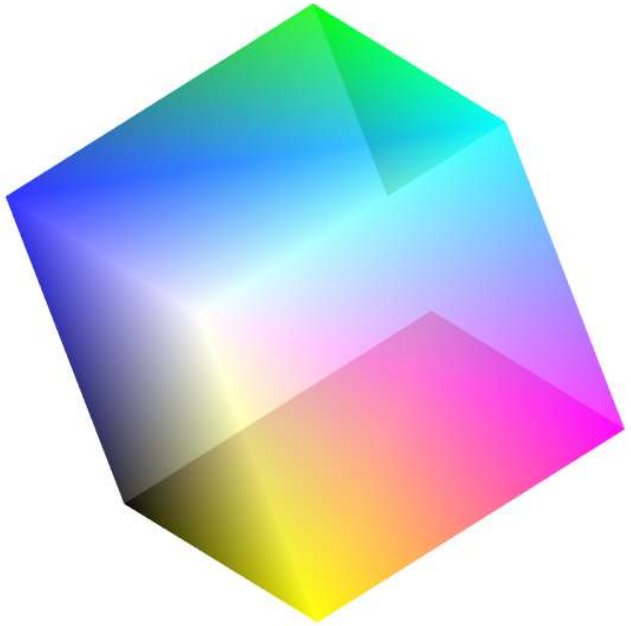
```
static const float verts[8][4] =
{
    { -1, -1, -1, 1 },
    { 1, -1, -1, 1 },
    { 1, 1, -1, 1 },
    { -1, 1, -1, 1 },
    { -1, -1, 1, 1 },
    { 1, -1, 1, 1 },
    { 1, 1, 1, 1 },
    { -1, 1, 1, 1 }
};
```

- 定义8个顶点的颜色

```
static const float vertColors[8][4] =
{
    { 0, 0, 0, 0.8f },
    { 1, 0, 0, 0.8f },
    { 0, 1, 0, 0.8f },
    { 0, 0, 1, 0.8f },
    { 1, 1, 0, 0.8f },
    { 1, 0, 1, 0.8f },
    { 0, 1, 1, 0.8f },
    { 1, 1, 1, 0.8f }
};
```

- 定义6个面的顶点顺序

```
static const uint quadFaces[6][4] =
{
    { 0, 1, 5, 4 },
    { 4, 5, 6, 7 },
    { 1, 2, 6, 5 },
    { 0, 4, 7, 3 },
    { 2, 3, 7, 6 },
    { 1, 0, 3, 2 }
};
```

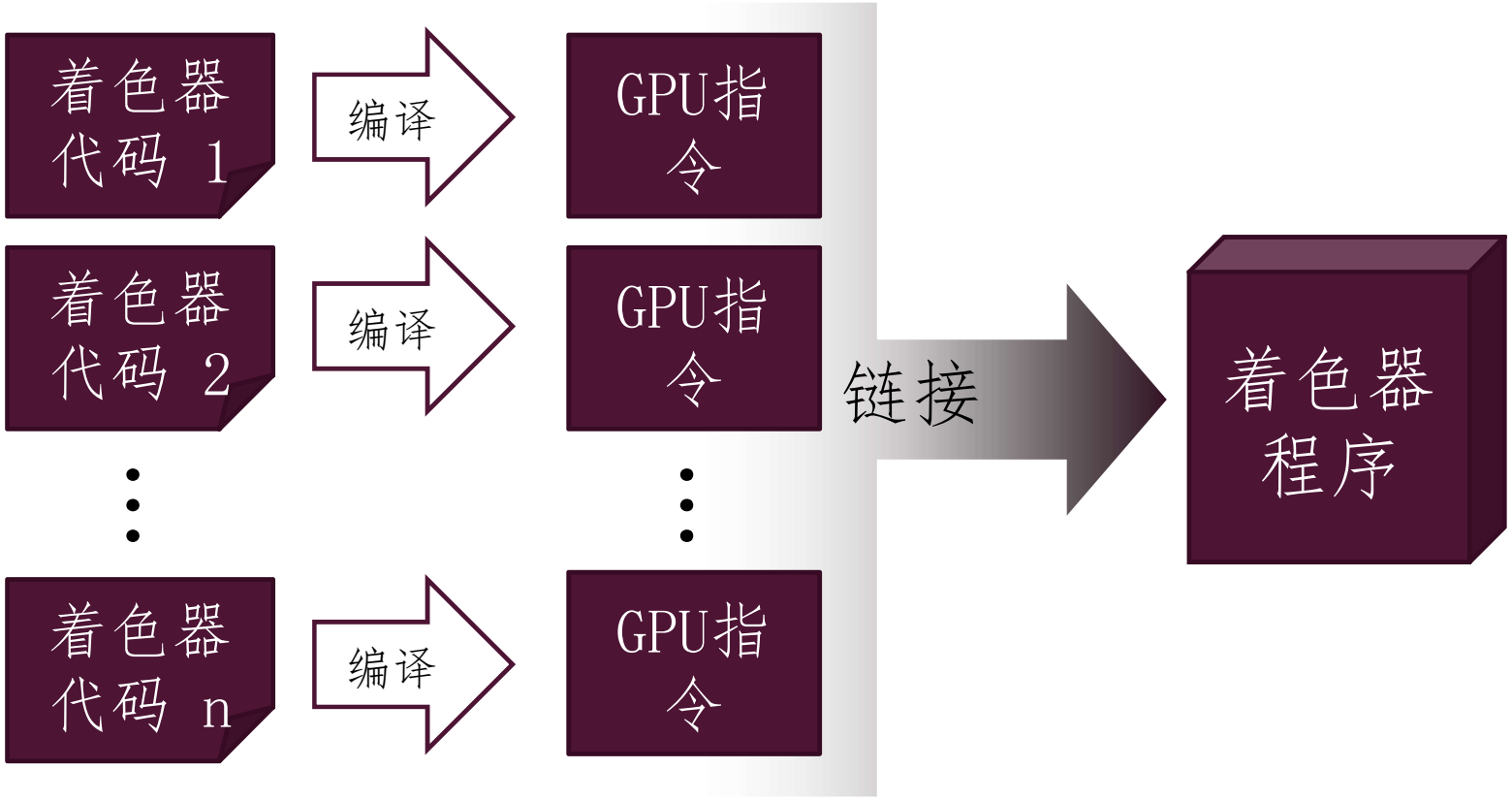


构建着色器

- GPU端：如何设计着色器逻辑？
  - GLSL：OpenGL着色器语言，语法与C类似
  - 语法详见官方手册 https://www.khronos.org/files/opengl43-quick-reference-card.pdf
- CPU端：如何加载着色器？如何传输数据？
  - 调用相关 OpenGL API 来创建、编译和链接着色器
  - 使用 OpenGL 缓冲区对象传递数据

构建着色器

- 着色器是GPU端的可执行程序



GLSL语法

```
static const char* vshaderSource =
    "#version 410\n"                                                    OpenGL版本声明
    "uniform mat4 modelViewMatrix;\n"
    "uniform mat4 projectionMatrix;\n"                                  Uniform 变量
    "in vec4 position;\n"
    "in vec4 color;\n"                                                  变量声明 输入
    "out vec4 pixelColor;\n"                                            输出
    "void main () {\n"
    "    gl_Position = projectionMatrix * modelViewMatrix * position;\n"
    "    pixelColor = color;\n"
    "}";                                                                函数定义

static const char* fshaderSource =
    "#version 410\n"                                                    OpenGL版本声明
    "in vec4 pixelColor;\n"
    "out vec4 fragColor;\n"                                             变量声明
    "void main () {\n"
    "    fragColor = pixelColor;\n"                                     函数定义
    "}";
```

GLSL语法

- 着色器中常用的基本数据类型：

类型	含义
bool	布尔值
int/uint	有符号/无符号32位整数
float	单精度浮点数
vec2/3/4	长为2/3/4的浮点数向量
bvec2/3/4	长为2/3/4的布尔向量
ivec2/3/4	长为2/3/4的有符号整数向量
mat2/3/4	2x2/3x3/4x4的浮点数矩阵
sampler2D	一个可以访问二位纹理的句柄

GLSL语法

- 变量的存储类型

类型	含义
无修饰符（默认）	定义本地变量，或者用于定义函数的一般输入参数
const	编译时常量，或者用于修饰一个函数的输入是只读的
in	代表着色器的输入
out	代表着色器的输出
uniform	在着色器内只读，而可被外部程序修改的变量
attribute (removed in OpenGL 4.0)	等价于顶点着色器的 in ，对应输入的每个顶点的数据
varying (removed in OpenGL 4.0)	等价于顶点着色器的 out 或片元着色器的 in ，表示被插值的数据

GLSL语法

- 变量声明举例
  - uniform mat4 view, projection;
  - in vec4 position;
  - in vec3 normal;
  - in vec2 texCoord;
  - struct light { float intensity; vec3 position; } lightVar;
  - const int numLights = 2;
  - uniform light lights[numLights];

GLSL语法

- 内置变量

Vertex Language

Inputs	in int gl_VertexID; in int gl_InstanceID;

Outputs	out gl_PerVertex {
	vec4 gl_Position;
	float gl_PointSize;
	float gl_ClipDistance[];
	};

Fragment Language

Inputs	in vec4 gl_FragCoord; in bool gl_FrontFacing; in float gl_ClipDistance[]; in vec2 gl_PointCoord; in int gl_PrimitiveID; in int gl_SampleID; in vec2 gl_SamplePosition; in int gl_SampleMask[]; in int gl_Layer; in int gl_ViewportIndex;
Outputs	out float gl_FragDepth; out int gl_SampleMask[];

构建着色器程序

创建着色器程序

创建顶点着色器
加载顶点着色器代码
编译顶点着色器

创建片元着色器
加载片元着色器代码
编译片元着色器

关联顶点着色器
关联片元着色器

链接程序

```
_program = glCreateProgram();

int vshader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vshader, 1, &vshaderSource, 0);
glCompileShader(vshader);

{ ... }

// create a fragment shader
int fshader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fshader, 1, &fshaderSource, 0);
glCompileShader(fshader);
{ ... }

// attach shaders to the program
glAttachShader(_program, vshader);
glAttachShader(_program, fshader);

// link the program
glLinkProgram(_program);
{ ... }
```

构建着色器程序

- 创建程序对象
  - GLuint glCreateProgram(void)
- 创建着色器对象
  - GLuint glCreateShader(GLenum shaderType)
- 设置着色器对象的源代码
  - void glShaderSource(GLuint shader, GLsizei count, const GLchar **string, const GLint *length)

构建着色器程序

- 编译着色器对象
  - void glCompileShader(GLuint shader)
- 将着色器对象关联到程序对象
  - void glAttachShader(GLuint program, GLuint shader)
- 链接程序对象
  - void glLinkProgram(GLuint program)

获取变量的位置

- 获取每个变量在程序对象中的位置（地址），用于数据传输

```
// get uniform location
_modelMatrixLocation = glGetUniformLocation(_program, "modelViewMatrix");
_projectionMatrixLocation = glGetUniformLocation(_program, "projectionMatrix");

// get attribute(in) location
_positionLocation = glGetAttribLocation(_program, "position");
_colorLocation = glGetAttribLocation(_program, "color");
```

获取变量的位置

- 获取uniform变量的位置
  - GLint glGetUniformLocation (GLuint program, const GLchar *name);
- 获取attribute变量的位置
  - GLint glGetAttribLocation (GLuint program, const GLchar *name);
- 显式将attribute变量绑定到某位置
  - void glBindAttribLocation (GLuint program, GLuint index, const GLchar *name);

使用缓冲区对象加载顶点数据

- 为什么需要缓冲区对象？
  - OpenGL按照客户机-服务器模式设计
  - 需要数据时，必须从客户机传输到服务器
    - 缓慢、冗余
- 缓冲区对象
  - 显式地制定把哪些数据存储在图形服务器中。
  - 标识符：一个非零无符号整数

使用缓冲区对象加载顶点数据

```
// vertex positions buffer
glGenBuffers(1, &_vertexPositionsBuffer);
glBindBuffer(GL_ARRAY_BUFFER, _vertexPositionsBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_STATIC_DRAW);

// vertex colors buffer
glGenBuffers(1, &_vertexColorsBuffer);
glBindBuffer(GL_ARRAY_BUFFER, _vertexColorsBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertColors), vertColors, GL_STATIC_DRAW);

// quad face indices buffer
glGenBuffers(1, &_quadFacesIndexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _quadFacesIndexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(quadFaces), quadFaces, GL_STATIC_DRAW);

// unbind buffers
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

将顶点位置数据载入**数组缓冲区对象** _vertexPositionBuffer中

将顶点颜色数据载入**数组缓冲区对象** _vertexColorBuffer中

将面片索引数组载入**索引缓冲区对象**_quadFacesIndexBuffer中

缓冲区对象API

- 创建缓冲区对象
  - void glGenBuffers(GLsizei n, GLuint *buffers);
  - 在buffers中返回n个当前未使用的对象标识符
- 判断缓冲区
  - GLboolean glIsBuffer(GLuint buffer);
  - 判断以buffer为标识符的缓冲区对象当前是否被使用

缓冲区对象API

- 激活缓冲区对象
  - void glBindBuffer(GLenum target, GLuint buffer);
  - target的可选值：

值	含义
GL_ARRAY_BUFFER	表示顶点数据
GL_ELEMENT_ARRAY_BUFFER	表示索引数据
GL_PIXEL_PACK_BUFFER	表示传递给OpenGL的像素数据
GL_PIXEL_UNPACK_BUFFER	表示从OpenGL获取的像素数据
GL_COPY_READ_BUFFER	表示在缓冲区之间复制数据
GL_COPY_WRITE_BUFFER	表示在缓冲区之间复制数据
GL_TRANSFORM_FEEDBACK_BUFFER	表示执行一个变换反馈着色器的结果
GL_UNIFORM_BUFFER	表示统一变量值

缓冲区对象API

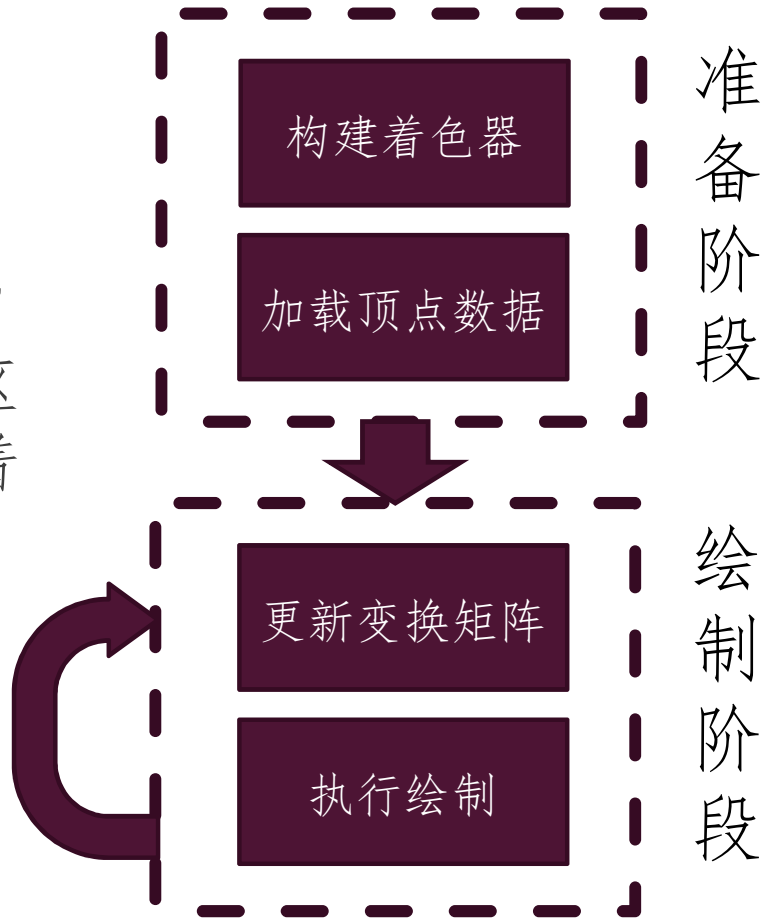
- 加载数据至缓冲区对象
  - void glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage);
  - target与glBindBuffer时对应
  - size：需要的内存数量
  - data：指向客户机内存的指针，可以为NULL
  - usage：提供提示，之后将如何进行读取和写入
    - 3种类型的操作：绘图（DRAW）、读取（READ）、复制（COPY）
    - 3种模式：流模式（STREAM）、静态模式（STATIC）、动态模式（DYNAMIC）
    - 例：GL_STATIC_DRAW：数据只指定1次，多次绘制

缓冲区对象API

- 清除缓冲区对象
  - void glDeleteBuffers(GLsizei n, const GLuint *buffers);
  - 删除buffers给定的n个缓冲区。

绘制阶段

- 启用着色器程序
- 更新变换矩阵
- 指定顶点数据的额外信息
  - 顶点数据虽已加载到缓冲区对象，但尚未指定如何与着色器中的变量相互关联
- 绘制



启用着色器对象

```
glUseProgram(_program);
```

- 启用程序对象
  - void glUseProgram(GLuint program);
  - program 是已经完成链接的着色器程序对象

更新变换矩阵

```
//// set uniform values
// setup model matrix
glUniformMatrix4fv(_modelMatrixLocation, 1, GL_FALSE, _modelMatrix.data());
// setup projection matrix
QMatrix4x4 projectionMatrix;
projectionMatrix.setToIdentity();
projectionMatrix.ortho(-width() / 2.0, width() / 2.0, -height() / 2.0, height() / 2.0, -1e3, 1e3);
glUniformMatrix4fv(_projectionMatrixLocation, 1, GL_FALSE, projectionMatrix.data());
```

- 在准备阶段，我们已经得到了uniform变量在程序中的位置（地址）
- 可根据变量位置直接更新数据

指定顶点数据的额外信息

- 使用OpenGL缓冲区对象设置顶点属性变量
  - 根据存有顶点位置信息的缓冲区对象_vertexPositionsBuffer来设置“position”属性
  - 根据存有顶点颜色信息的缓冲区对象_vertexColorsBuffer来设置“color”属性

```
//// set attributes data
glBindBuffer(GL_ARRAY_BUFFER, _vertexPositionsBuffer);
// enable vertex attribute "position"
glEnableVertexAttribArray(_positionLocation);
// set the data of vertex attribute "position" using current ArrayBuffer
glVertexAttribPointer(_positionLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, _vertexColorsBuffer);
// enable vertex attribute "color"
glEnableVertexAttribArray(_colorLocation);
// set the data of vertex attribute "color" using current ArrayBuffer
glVertexAttribPointer(_colorLocation, 4, GL_FLOAT, GL_FALSE, 0, 0);
```

指定顶点数据的额外信息

- 启用/禁用一个顶点属性数组
  - void glEnableVertexAttribArray (GLuint index)
  - void glDisableVertexAttribArray (GLuint index)

  - index 为顶点属性的位置

指定顶点数据的额外信息

- 设置顶点属性与数组缓冲区对象的绑定方式
  - void glVertexAttribPointer (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer)
  - void glVertexAttribIPointer (GLuint index, GLint size, GLenum type, GLsizei stride, const GLvoid * pointer)
  - index 为顶点属性的位置
  - size 指定每个顶点属性的维度，必须是1,2,3,4中的一个

指定顶点数据的额外信息

- type 指定数据类型。
  - 可接受的值包括 GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT
- stride 用于指定相邻的顶点属性数据之间的比特偏移量。如果设为0，则认为这些数据是紧致连续存放的
- pointer 指定顶点属性数组第一个元素的地址，若当前 GL_ARRAY_BUFFER 绑定到一个缓冲区对象，那么 pointer 则用于指定该缓冲区中顶点属性数组第一个元素的偏移值

绘制

- 设置索引缓冲区
- 按索引对已启用的顶点属性数组进行绘制

```
// bind ElementArrayBuffer to _quadFacesIndexBuffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _quadFacesIndexBuffer);
// draw mesh using the indices stored in ElementArrayBuffer
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_INT, 0);
```

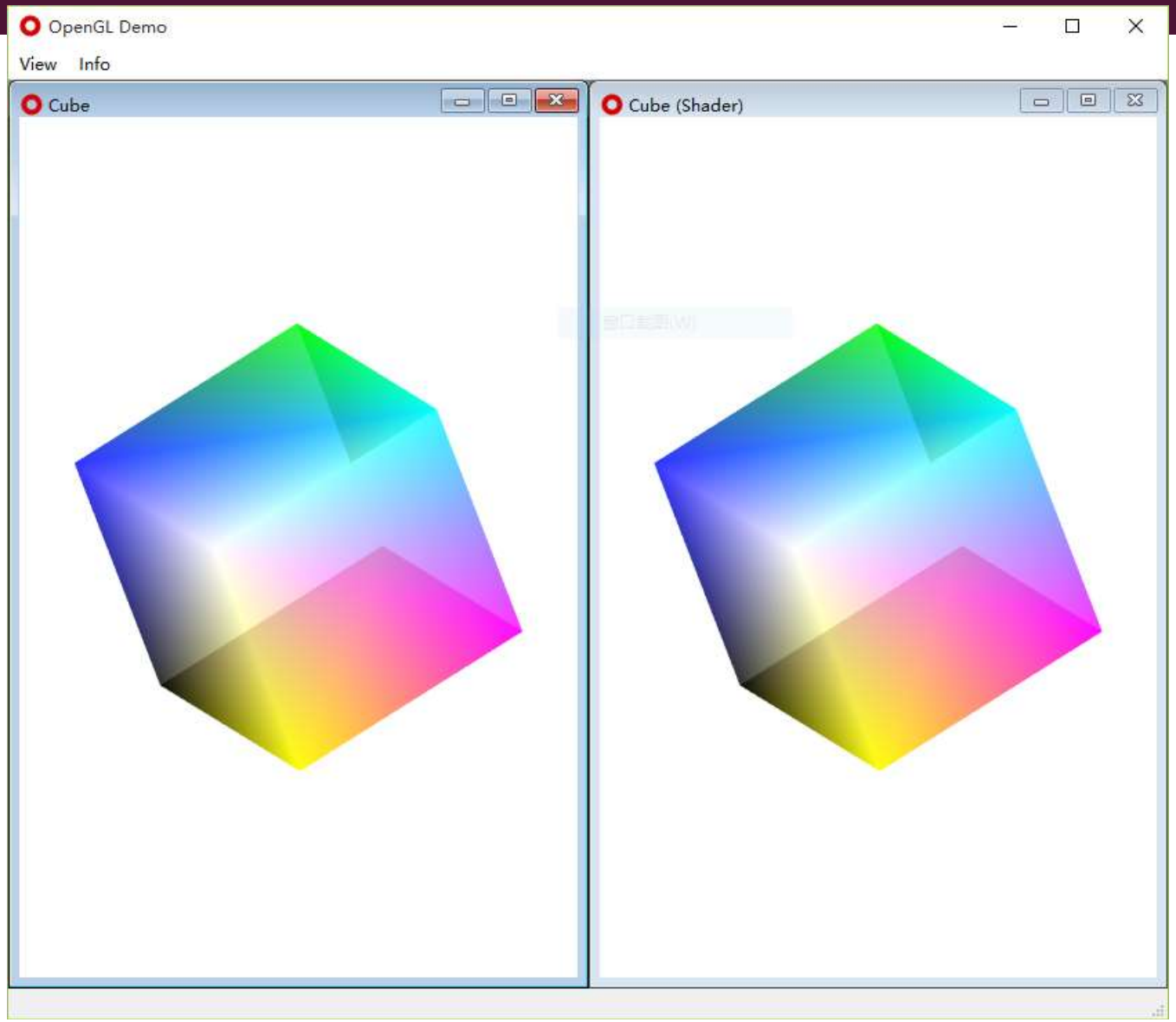
使用索引进行绘制

- void glDrawElements (GLenum mode, GLsizei count, GLenum type, const GLvoid * indices)
  - mode 用于指定绘制模式（GL_POINTS, GL_TRIANGLES …）
  - count 用于指定需要绘制的索引数组的长度
  - type 指定索引类型，只能是 GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT 或 GL_UNSIGNED_INT 中的一个
  - indices 若当前 GL_ELEMENT_ARRAY_BUFFER 非空，则用于指定索引数组起始位置的比特偏移值；若当前 GL_ELEMENT_ARRAY_BUFFER 为空，则用于指定存储索引数组的内存地址

直接绘制

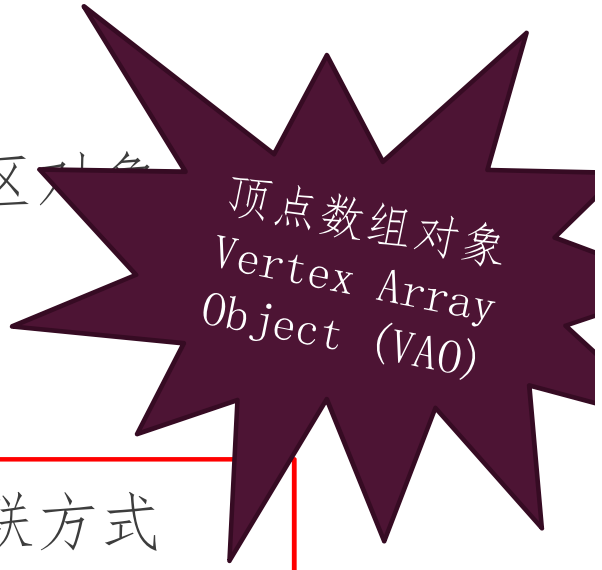
- void glDrawArrays (GLenum mode, GLint first, GLsizei count)
  - mode 用于指定绘制模式（GL_POINTS, GL_TRIANGLES …）
  - first 用于指定从数组中的第几个顶点数据开始进行绘制
  - count 用于指定需要对多少个顶点的数据进行绘制

固定管线 VS 着色器



总结

- 准备阶段
  - 构建着色器：写GLSL、编译、链接
  - 获取着色器中变量的位置（地址）
  - 将顶点数据和面索引数据加载至缓冲区对象
- 绘制阶段
  - 根据变量位置直接为uniform变量赋值
  - 设置顶点属性变量与缓冲区对象的关联方式
  - 根据面索引数据进行绘制



顶点数组对象 VERTEX ARRAY OBJECT (VAO)

- 等价于一个GPU端的结构体，成员包括：
  - 一个数组缓冲区对象
  - 一个索引缓冲区对象
  - 顶点属性变量与数组缓冲区对象的关联方式
- 在准备阶段构建好顶点数组对象，在绘制时激活此对象即可

OPENGL在线资源

- http://learnopengl.com/
  - 简单全面的OpenGL教程
- https://www.opengl.org/wiki
  - 官方Wiki
- http://www.realtech-vr.com/glview/
  - 一个小程序，用于查看本地设备对OpenGL各个版本的支持程度
- https://www.khronos.org/vulkan/
  - Vulkan，下一代OpenGL
- http://ogldev.atspace.co.uk/index.html
  - OpenGL历程