XIAMEN UNIVERSITY

# COMPUTER GRAPHICS

## 第四章 管理 3D 图形数据

陈中贵
厦门大学信息学院
http://graphics.xmu.edu.cn

# 从C++发送数据到shaders

- 两种方式
  - 通过顶点属性的缓冲区
  - 直接发送给统一变量

- 把顶点数据在 C++端放入一个缓冲区，并把这个缓冲区和着色器中声明的顶点属性相关联，具体步骤
  - 只做一次的步骤如下，它们一般包含在 init()中。

    （1）创建缓冲区。

    （2）将顶点数据复制到缓冲区。
  - 每帧都要做的步骤如下，它们一般包含在 display()中。

    （1）启用包含顶点数据的缓冲区。

    （2）将这个缓冲区和一个顶点属性相关联。

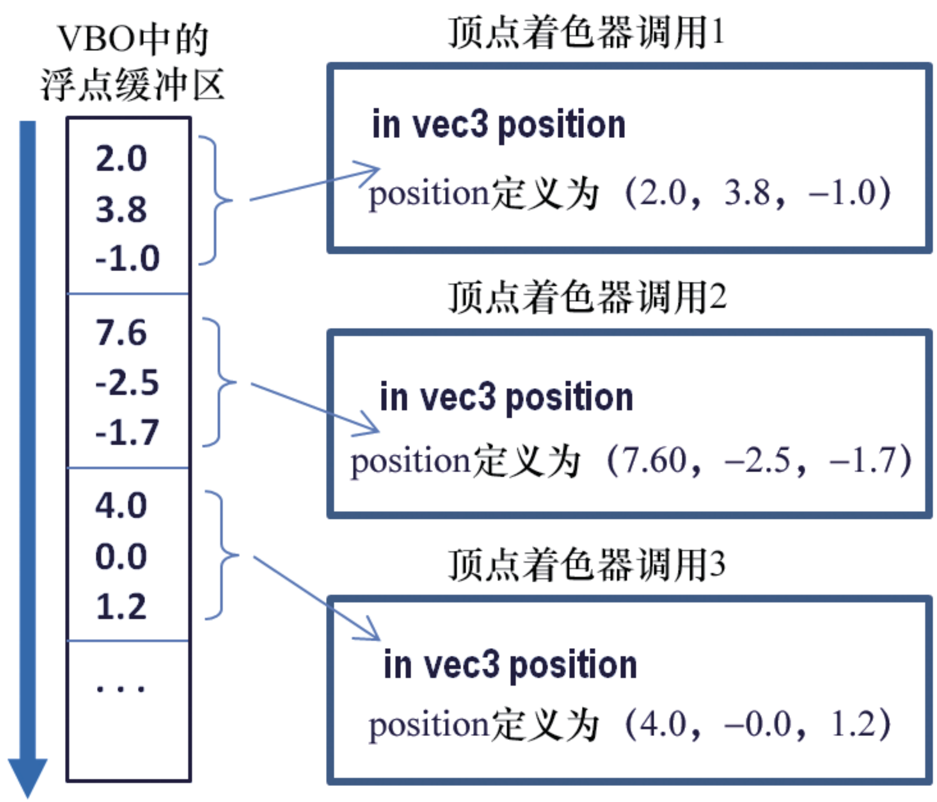
    （3）启用这个顶点属性。

    （4）使用 glDrawArrays()绘制对象。

# 从C++发送数据到shaders

- 在 OpenGL 中，缓冲区被包含在顶点缓冲对象（ Vertex Buffer Object， VBO） 中，VBO 在C++/OpenGL 程序中被声明和实例化

- 在 VBO 和顶点属性之间的数据传递



# 实例化VBO

```
GLuint vao[1]; // OpenGL 要求这些数值以数组的形式指定
GLuint vbo[2];
…
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);
glGenBuffers(2, vbo);
```

**注意：VAO是组织VBO所必需的结构，至少需要一个。**

- 每个缓冲区需要有在顶点着色器中声明的相应顶点属性变量。顶点属性通常是着色器中首先声明 的变量：

```
layout (location = 0) in vec3 position;
```

## 用数据填充VBO

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW);
```

*顶点数据存储在名为 vPositions 的浮点类型数组中*

## 将VBO与顶点属性相关联

```
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);                    // 标记第 0 个缓冲区为“活跃”
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);    // 将第 0 个属性关联到缓冲区
glEnableVertexAttribArray(0);                             // 启用第 0 个顶点属性
```

*将活跃的VBO与顶点着色器中的“第0个”顶点属性相关联。*
*请注意上一张幻灯片中“location=0”*

# 使用统一变量

- 在顶点和片段着色器中完成：
  - 声明统一的名称和类型

```
uniform mat4 mv_matrix;
```

- 在C++中完成 -- 通常在display()中完成：
  - 获取对统一变量的引用

```
mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
```

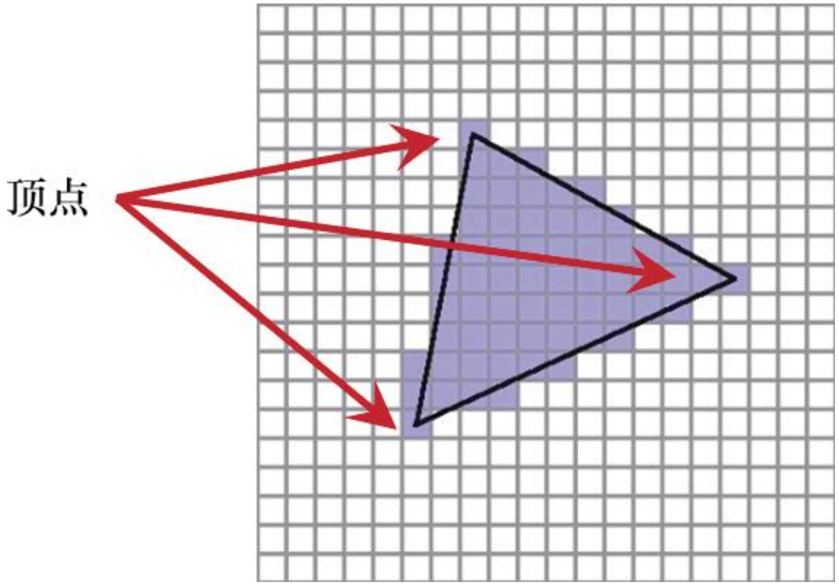
  - 将所需数据发送到统一变量

```
glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));
```

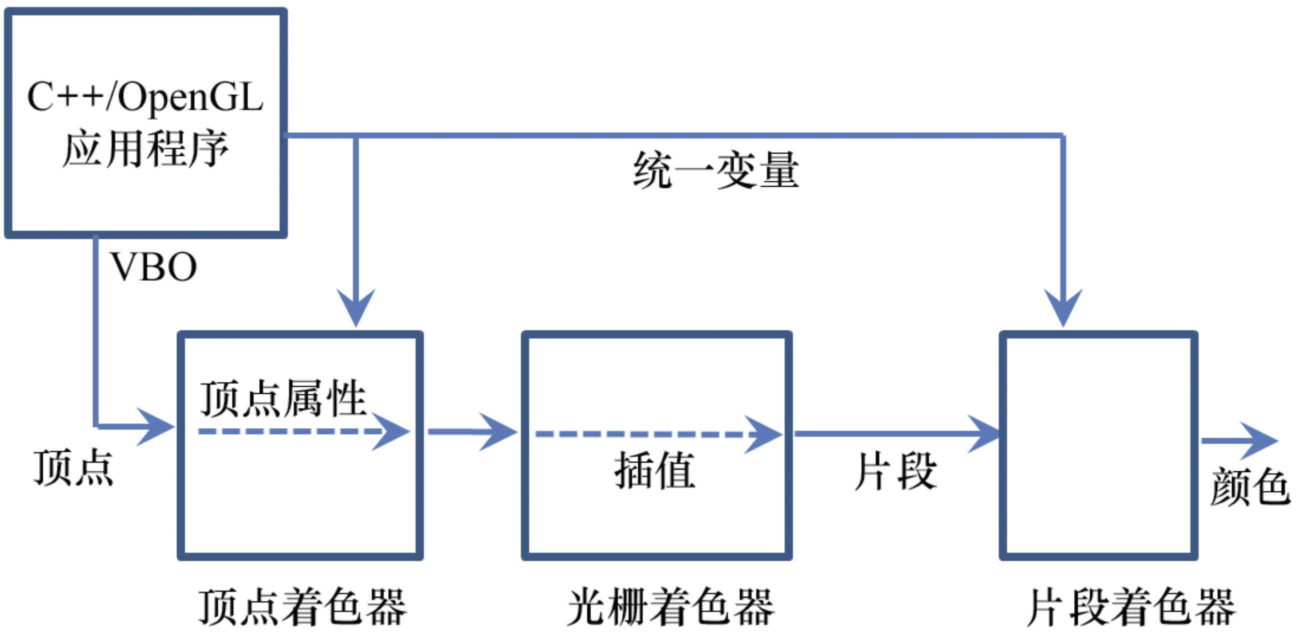
# 选择哪种数据发送模式?

- 对整个绘制对象的恒定值使用统一变量（如MV和P变换矩阵）

- 如果希望光栅化器对值进行插值（如绘制的模型中的顶点），使用顶点属性



# 数据流概况



# 第一个 3D 程序—— 一个 3D 立方体

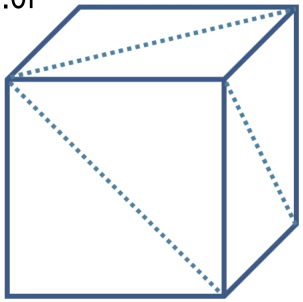
## *C++/OpenGL Application*

```
. . .
#include <glm\glm.hpp>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>
#include "Utils.h"
. . .
float cameraX, cameraY, cameraZ;          // locati
on of camera in scene
float cubeLocX, cubeLocY, cubeLocZ;     // location of cube object in scene

//  allocate variables used in display(), so they won't need to be allocated during rendering
glm::mat4 pMat;  // perspective matrix
glm::mat4 vMat;  // view matrix
glm::mat4 mMat; // model matrix
glm::mat4 mvMat;                    // model-view matrix
GLuint mvLoc, projLoc;
float aspect;

int main(void)  {
    //  same as before
}
```

*continued…*

# 绘制简单的红色立方体

```
void init(void) {
        renderingProgram = Utils::createShaderProgram("vertShader.glsl", "fragShader.glsl");
        cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
        cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f; // 沿 y 轴下移以展示透视效果
        setupVertices();
}
void setupVertices(void) {
    // 36 vertices of the 12 triangles making up a 2 x 2 x 2 cube centered at the origin
    float vertexPositions[108] = {
        -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
        1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f,  1.0f, 1.0f,  1.0f, -1.0f,
        1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f,  1.0f,
        -1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,
        -1.0f,  1.0f, -1.0f, 1.0f,  1.0f, -1.0f, 1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f, -1.0f
    };
```



```
    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);
    glGenBuffers(numVBOs, vbo);

    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions), vertexPositions, GL_STATIC_DRAW);
}
```

# 绘制简单的红色立方体

```
void display(GLFWwindow* window, double currentTime)  {
    . . .
    // Create a perspective matrix, this one has fovy=60, aspect ratio matches screen window.
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float) width() / (float) height();
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);     //  1.0472 radians = 60 degrees

    // build view, model, and model-view matrices
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
    mvMat = vMat * mMat;

    // prepare uniform variables
    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");
    gl.glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(pMat));
    gl.glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvMat));

    // prepare vertex attribute containing cube vertices
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    glEnableVertexAttribArray(0);
    . . .
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

# 绘制简单的红色立方体

## *Vertex shader*

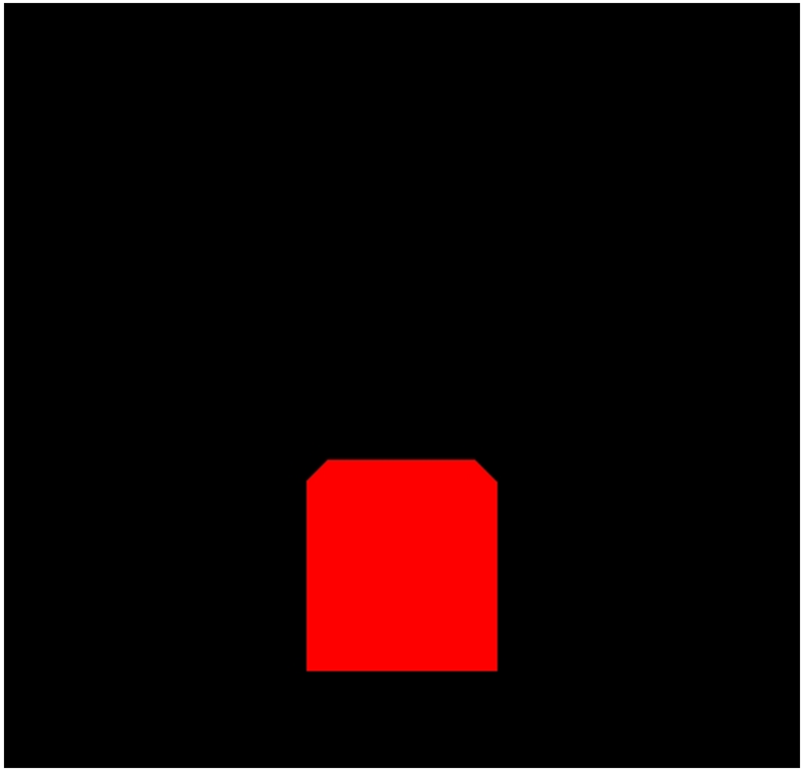
```
#version 430
layout (location=0) in vec3 position;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
void main(void)
{    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
}
```

## *Fragment shader*

```
#version 430
out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
void main(void)
{    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

# 绘制简单的红色立方体

- 程序 4.1 的输出，从(0,0,8)看位于(0,–2,0)的红色立方体



# 顶点属性插值

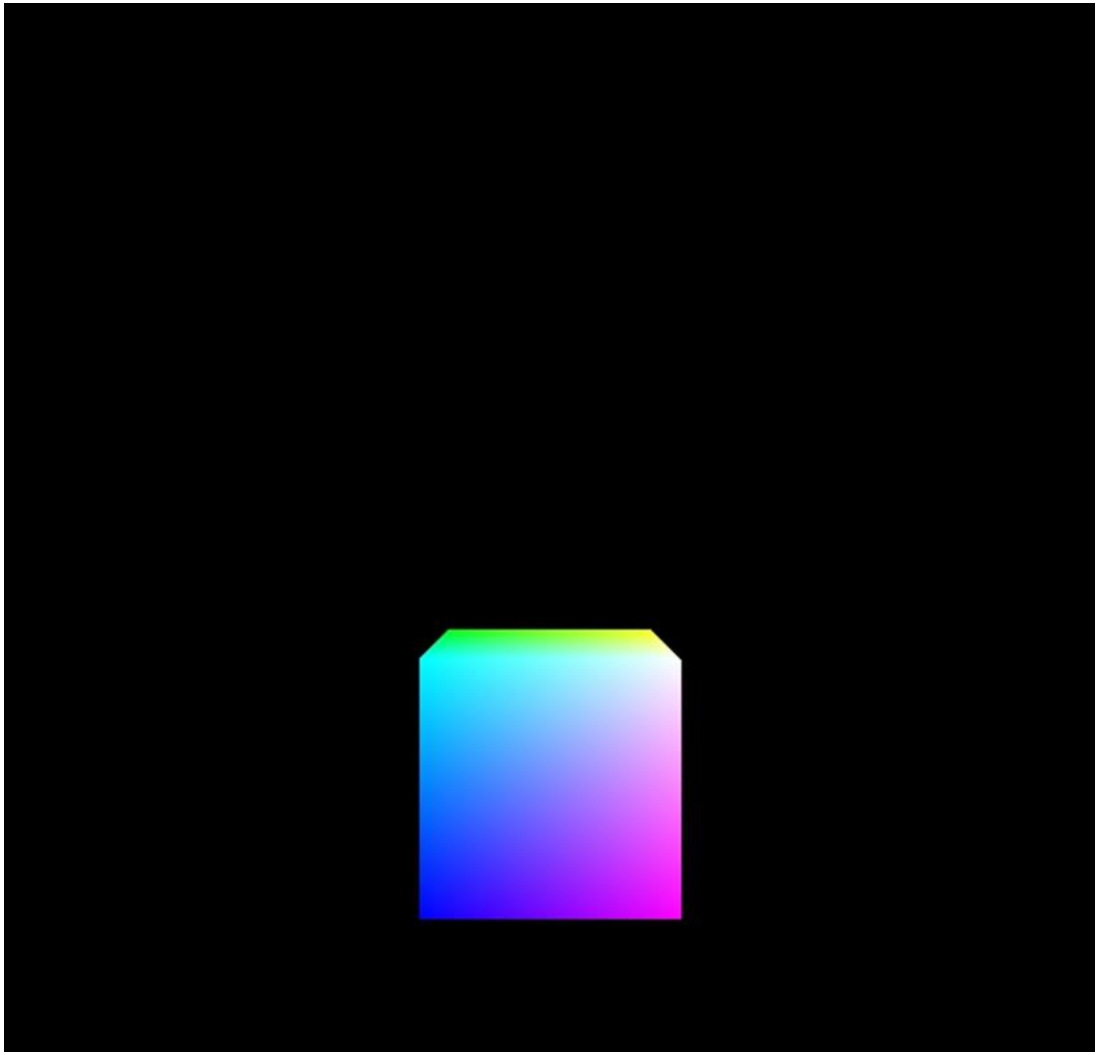
*Revised vertex shader:*

```
. . .
out vec4 varyingColor;          // (added)
void main(void)
{       gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
        varyingColor = vec4(position,1.0) * 0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

*Revised fragment shader:*

```
in vec4 varyingColor;   // added
. . .
void main(void)
{     color = varyingColor;
}
```

# 有插值颜色的立方体



# 渲染多个不同模型

- 将每个对象的顶点放置在单独的VBO中
- 每个对象都需要自己的模型矩阵（M）
- 对象将共享相同的视图和投影矩阵
- 为正在绘制的每个对象调用glDrawArrays（）

- 例子：立方体和四棱锥
  - 让我们用立方体和四棱锥渲染一个场景
  - 立方体的顶点被放置在VBO中（如前所述）
  - 四棱锥的顶点被放置在第二个VBO中
  - 每个对象都有自己不同的模型（M）矩阵
  - 相同的顶点着色器和片段着色器可以用于两个对象

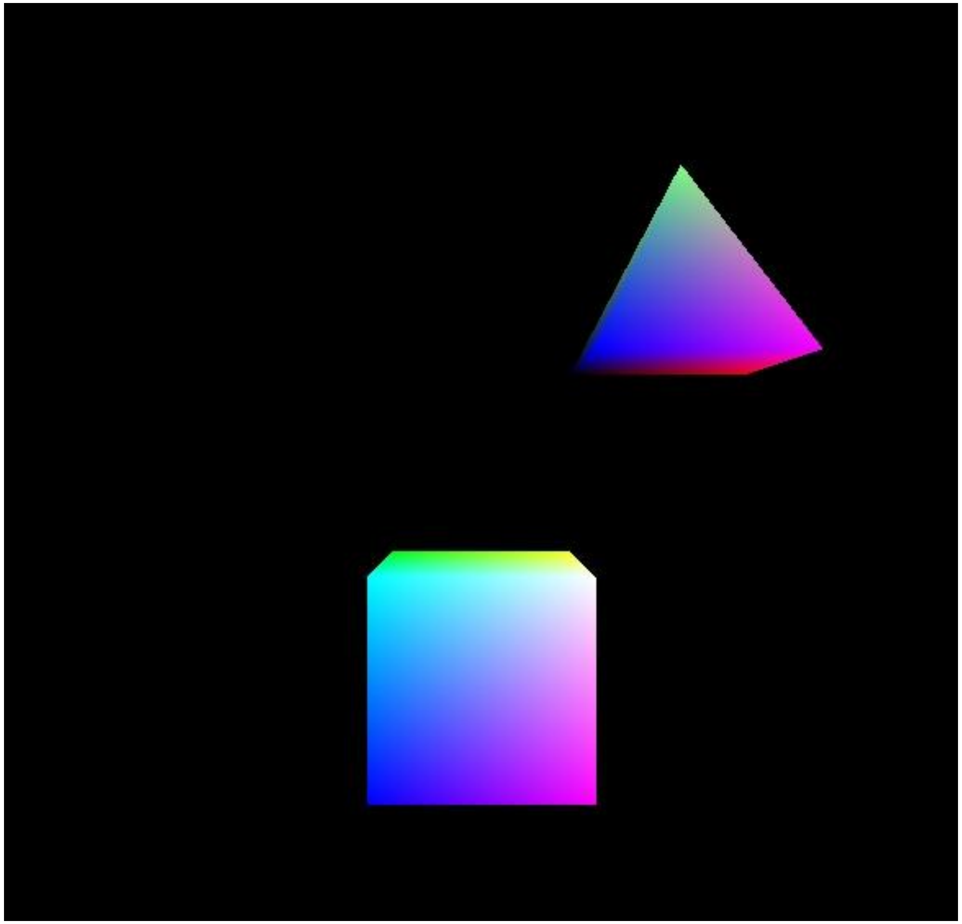
# 例子：立方体和四棱锥

```
void setupVertices()  {
    float[ cubePositions[108] = {
        -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
        1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
                        … same as before, for the rest of the cube vertices
    };
    //  pyramid with 18 vertices, comprising 6 triangles (four sides, and two on the bottom)
    float pyrPositions[54] = {
        -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,         // front face
        1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,         // right face
        1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 0.0f, 1.0f, 0.0f,       // back face
        -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, 0.0f, 1.0f, 0.0f,       // left face
        -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f,      // base – left front
        1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f       // base – right back
    };
    //  initialize VAO and VBOs as before
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);                 // 0th VBO for cube
    glBufferData(GL_ARRAY_BUFFER, sizeof(cubePositions), cubePositions,GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);                 // 1th VBO for pyramid
    glBufferData(GL_ARRAY_BUFFER, sizeof(pyrPositions), pyrPositions,GL_STATIC_DRAW);
}
```

# 例子：立方体和四棱锥

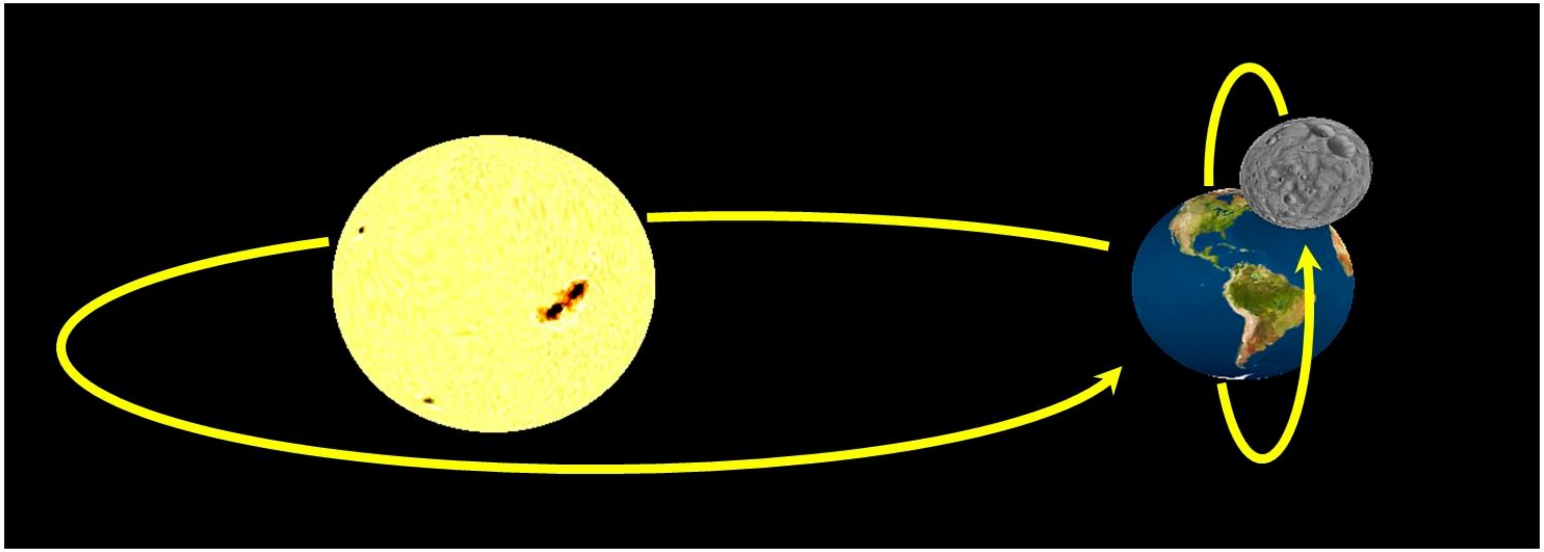
```
void display(GLFWwindow* window, double currentTime)  {
    //  set up the view matrix and rendering program, and pass uniforms
    //    to the shaders as before
    . . .
    // draw the cube using buffer #0
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(cubeLocX, cubeLocY, cubeLocZ));
    mvMat = vMat * mMat;
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    . . .
    glDrawArrays(GL_TRIANGLES, 0, 36);
    // draw the pyramid using buffer #1
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(pyrLocX, pyrLocY, pyrLocZ));
    mvMat = vMat * mMat;  //两个模型使用相同的视图矩阵vMat，不同的模型矩阵mMat
    . . .
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);
    . . .
    glDrawArrays(GL_TRIANGLES, 0, 18);
}
```

# 例子：立方体和四棱锥



# 使用矩阵栈构造层次场景

- 例子：太阳系



地球围绕太阳旋转和月球围绕地球旋转

# C++标准模板库（STL） stack 类

- push()：在栈顶部创建一个新的条目。我们通常会把目前在栈顶部的矩阵复制一份，并和其他的变换结合，然后利用这个命令把新的矩阵副本压入栈。
- pop()：移除（并返回）最顶部的矩阵。
- top()：在不移除的情况下，返回栈最顶部矩阵的引用。
- <stack>.top()*= rotate(构建旋转矩阵的参数)
- <stack>.top()*= scale(构建缩放矩阵的参数)
- <stack>.top()*= translate(构建平移矩阵的参数)

直接对栈顶部的矩阵应用变换

- 基本方法：

（1）声明栈，stack<glm::mat4> mvStack

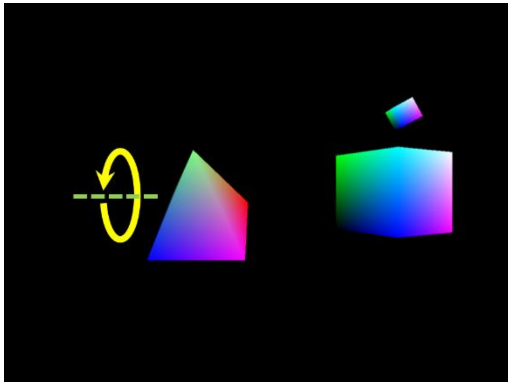
（2）当相对于父对象创建新对象时，调用mvStack.push(mvStack.top())

（3）应用新对象所需的变换，也就是与所需的变换矩阵相乘

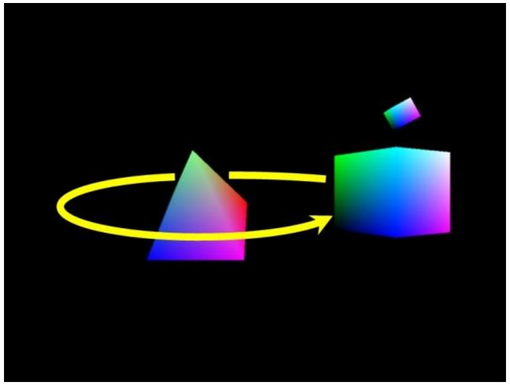
（4）完成对象或子对象的绘制后，调用 mvStack.pop()从矩阵栈顶部移除其 MV 矩阵。

# 使用立方体和四棱锥模拟太阳系

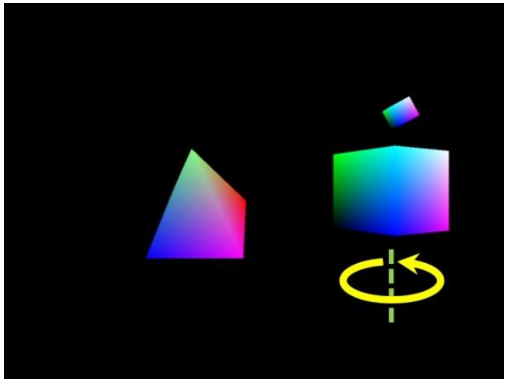
- 先将视图矩阵V压入堆栈，然后：
  - 太阳的自转被压入堆栈，
    并在绘制后弹出：



  - 地球围绕太阳的公转被压入堆栈
    （但没有弹出）：

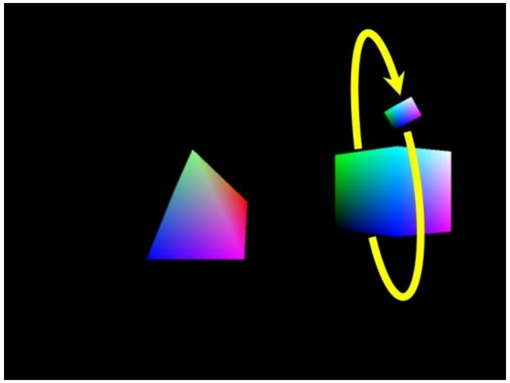


  - 地球的旋转被压入堆栈上，
    并在绘制后弹出：

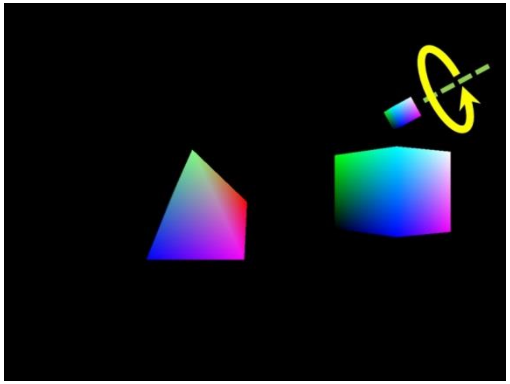


# 使用立方体和四棱锥模拟太阳系

- 月球绕地球公转被推到堆栈上（但没有弹出）：



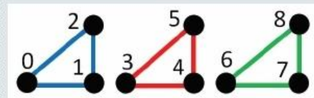
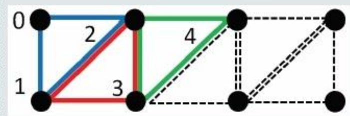
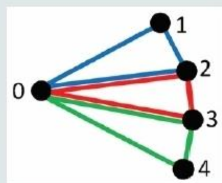
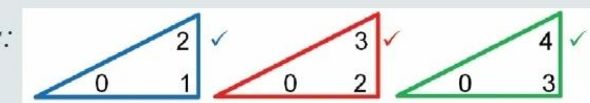
- 月亮的自转被推到堆栈上:



- 在每个绘制步骤中，堆栈顶部的矩阵用作MV矩阵
- 在display()的末尾，剩余的矩阵从堆栈中弹出

```
stack<glm::mat4> mvStack;
void display(GLFWwindow* window, double currentTime )  {
    . . .
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(-cameraX, -cameraY, -cameraZ));
    mvStack.push(vMat);        // push view matrix onto the stack
    . . .
    // ---------------------  pyramid == sun  -----------------------------------------
    mvStack.push(mvStack.top());
    mvStack.top() *= glm::translate(glm::mat4(1.0f), glm::vec3(0,0,0));    // sun position
    mvStack.push(mvStack.top());
    mvStack.top() *= glm::translate(glm::mat4(1.0f), glm::vec3(0,0,0));    // sun rotation
    glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::value_ptr(mvStack.top()));   // pass MV to shader
    glDrawArrays(GL_TRIANGLES, 0, 18);             // draw the sun
    mvStack.pop();                  // remove the sun's axial rotation from the stack (but not its translation)
    //----------------------  cube == planet  -----------------------------------------
    mvStack.push(mvStack.top());
    . . .    // planet's revolution around sun goes here.  Also push a matrix for planet's axis rotation
    . . .    //  then glUniformMatrix4fv to pass MV matrix to shaders, etc.
    glDrawArrays(GL_TRIANGLES, 0, 36);             // draw the planet
    mvStack.pop();                  // remove the planet's axial rotation from the stack (but not the translation)
    //----------------------  smaller cube == moon  ----------------------------------
    . . .     // moon's revolution around planet, and rotation on its axis, go here, similar as for planet and sun
    glDrawArrays(GL_TRIANGLES, 0, 36);             // draw the moon
    //  remove moon scale/rotation/position, planet position, sun position, and view matrices from stack
    mvStack.pop();  mvStack.pop();  mvStack.pop();  mvStack.pop();
}
```

# 其他图元

| | |
|---|---|
| GL_TRIANGLES<br> | The most common primitive type in this book. Vertices that pass through the pipeline form distinct triangles:<br>*vertices:* 0 1 2 \| 3 4 5 \| 6 7 8 etc.<br>*triangles:* ✓ ✓ ✓ |
| GL_TRIANGLE_STRIP<br> | Each vertex that passes through the pipeline efficiently forms a triangle with the previous two vertices:<br>*vertices:* 0 1 2 3 4 etc.<br>*triangles:* ✓ ✓ ✓ |
| GL_TRIANGLE_FAN<br> | Each pair of vertices that passes through the pipe-line forms a triangle with the very first vertex:<br>*vertices:* 0 1 2 3 4 etc.<br>*triangles:*<br> |
| GL_LINES<br> | Vertices that pass through the pipeline form distinct lines:<br>*vertices:* 0 1 \| 2 3 \| 4 5 etc.<br>*lines:* ✓ ✓ ✓ |
| GL_LINE_STRIP<br> | Each vertex that passes through the pipeline efficiently forms a line with the previous vertex:<br>*vertices:* 0 1 2 3 etc.<br>*lines:* ✓ ✓ ✓ |