OpenGL程序初步

华中科技大学
何云峰

CONTENTS

```
tial(void)

义图形对象的顶点数据
t  float triangle[] = {
.5f, -0.5f, 0.0f,
5f,  -0.5f, 0.0f,
0f,  0.5f, 0.0f


成并绑定VAO和VBO
nVertexArrays(1, &vertex_array_object);
dVertexArray(vertex_array_object);

nBuffers(1, &vertex_buffer_object);
dBuffer(GL_ARRAY_BUFFER, vertex_buffer_ob

点数据绑定至当前默认的缓冲中
ata(GL_ARRAY_BUFFER, sizeof(t
性指针
ointer(0, 3, GL_FLOA
ribArray(0);
```

PART 01

初识OpenGL

01 初识OpenGL

- OpenGL软件包
- OpenGL的绘制流程
- OpenGL的基本语法
- OpenGL环境配置

01 OpenGL软件包

- OpenGL是SGI（Silicon Graphics Inc.）公司对IRIS GL进行改进，扩展可移植性，形成的一个跨平台开放式图形编程接口
- OpenGL标准由1992年成立的独立财团OpenGL Architecture Review Board（ARB）以投票方式产生，并制成规范文档公布
- 2006年，Khronos集团，技术联合体

01 OpenGL软件包

OpenGL历程

- 2004，2.0，可编程的，基于着色器的API
- 2006，2.1，像素缓冲区对象
- 01年，1.3，多重纹理等
- 02年，1.4，纹理环境，深度纹理比较等
- 03年，1.5，增加对缓冲区对象的支持
- 2009年8月，3.2，核心和兼容性配置，增加几何着色器
- 2010年，曲面细分（Tessellation），增加曲面细分控制和曲面细分评估着色器
- 4.1 （2010年7月）；4.2（2011年8月）；4.3和4.4（2013年）；4.5（2014年）；4.6（2017年）

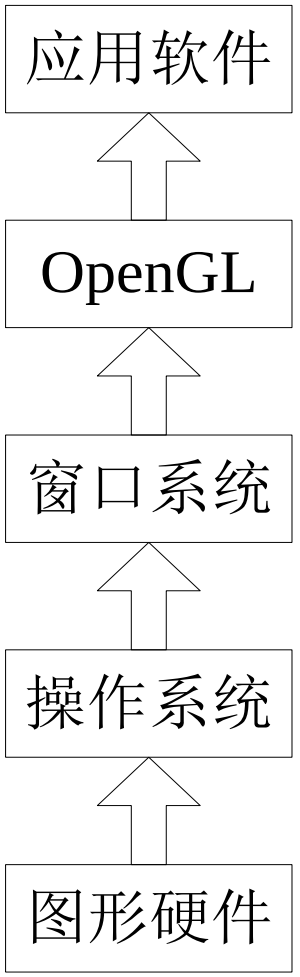
OpenGL4.X

01 初识OpenGL

- OpenGL软件包
- **OpenGL的绘制流程**
- OpenGL的基本语法
- OpenGL环境配置

01 OpenGL的绘制流程

- OpenGL的工作方式
  - 图形硬件
  - 操作系统
  - 窗口系统
  - OpenGL
  - 应用软件



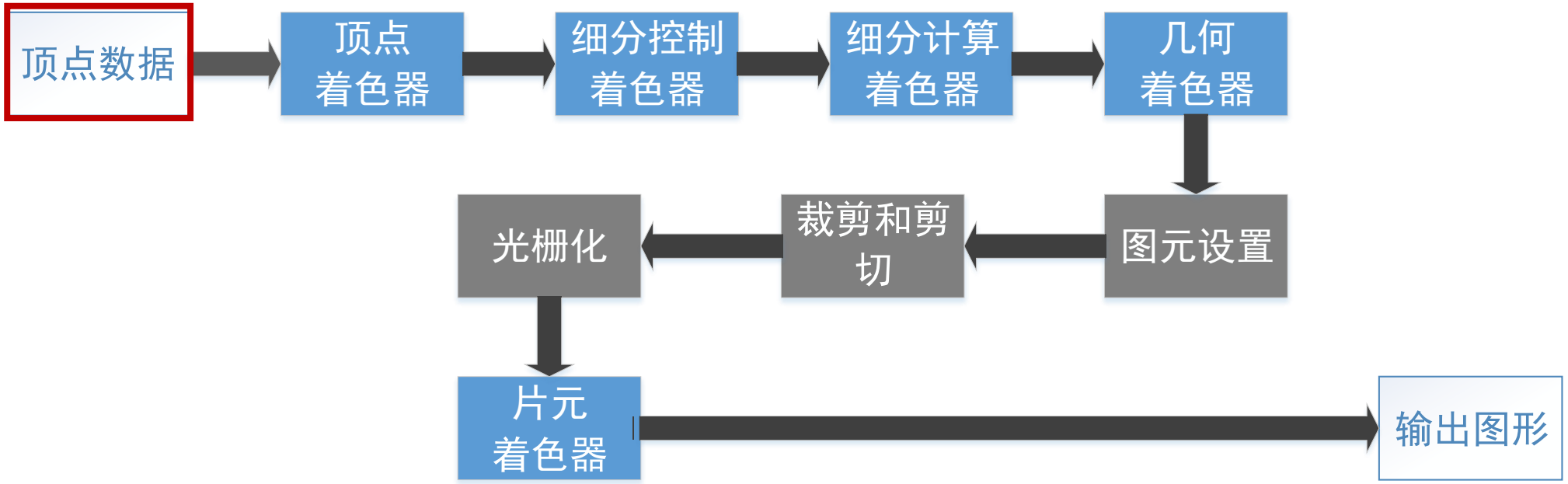
01 OpenGL的绘制流程

- OpenGL的主要操作
  - 从OpenGL的几何图元中设置数据，用于构建形状
  - 使用不同的着色器（Shader）对输入的图元进行计算操作，判断其位置，颜色以及其他渲染属性
  - 将输入图元的数学描述转换为与屏幕位置对应的像素片元（fragment），即光栅化
  - 针对光栅化过程产生的片元，执行片元着色器，决定其最终的颜色和位置
  - 针对片元的其他操作，如可见性判断，融合等等

01 OpenGL的绘制流程

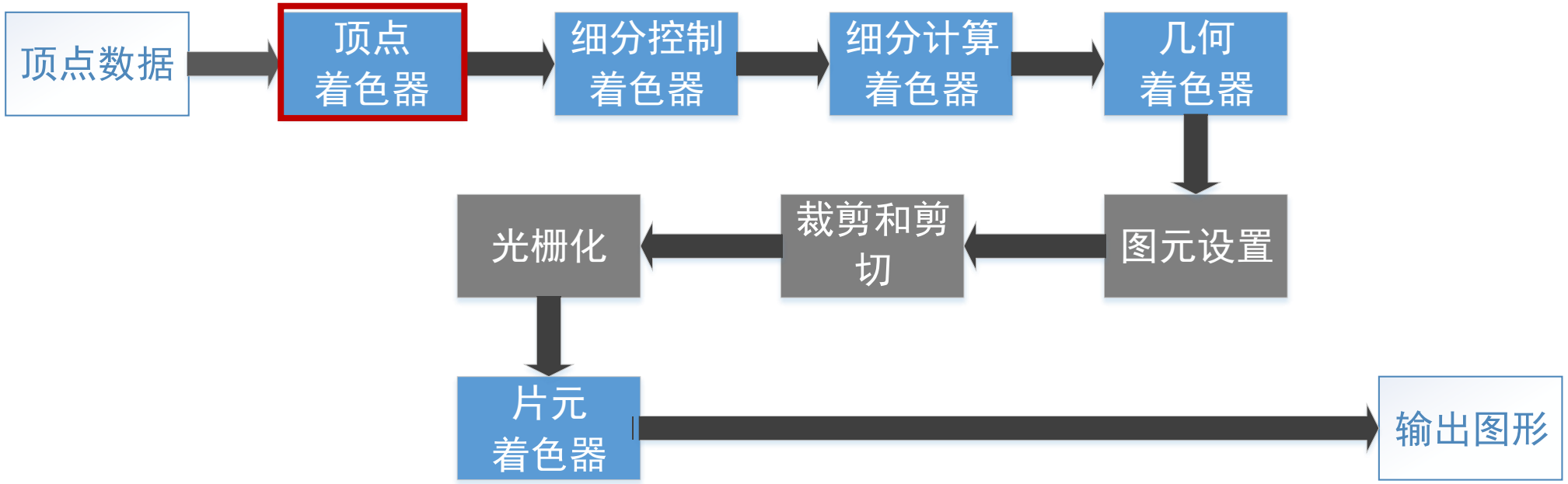
OpenGL的绘制流水线

- 顶点数据：将顶点数据保存在缓存对象中， OpenGL通过绘制命令传输顶点数据



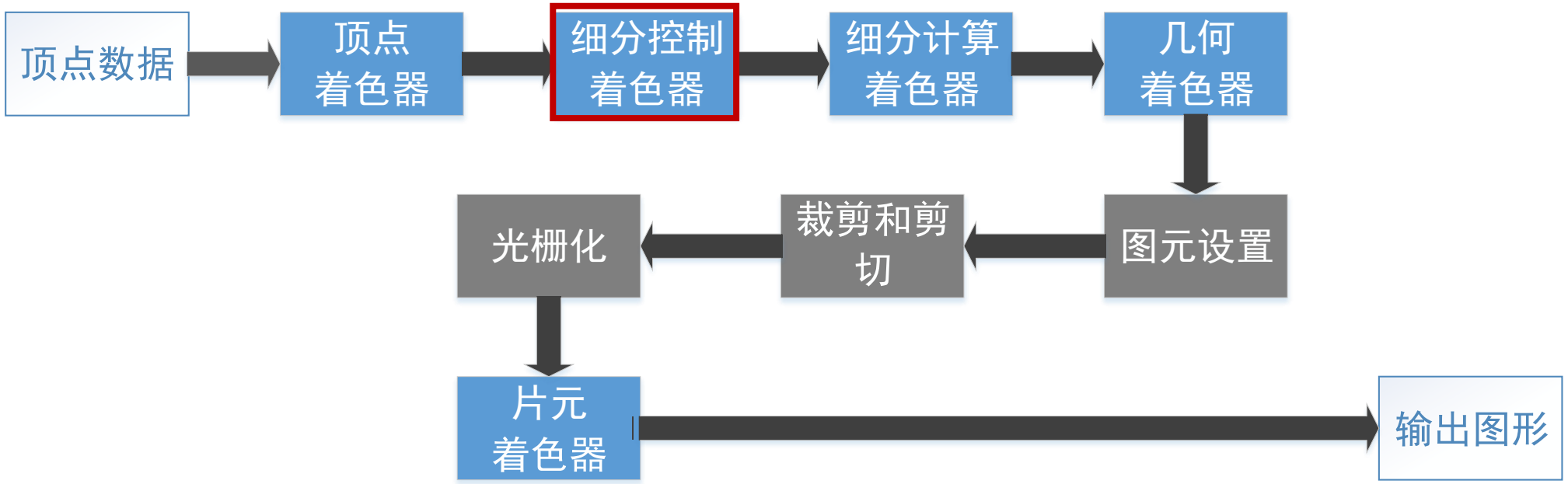
01 OpenGL的绘制流程

- OpenGL的绘制流水线
  - 顶点着色器：对顶点数据进行处理，包括变换，顶点着色（材质属性）



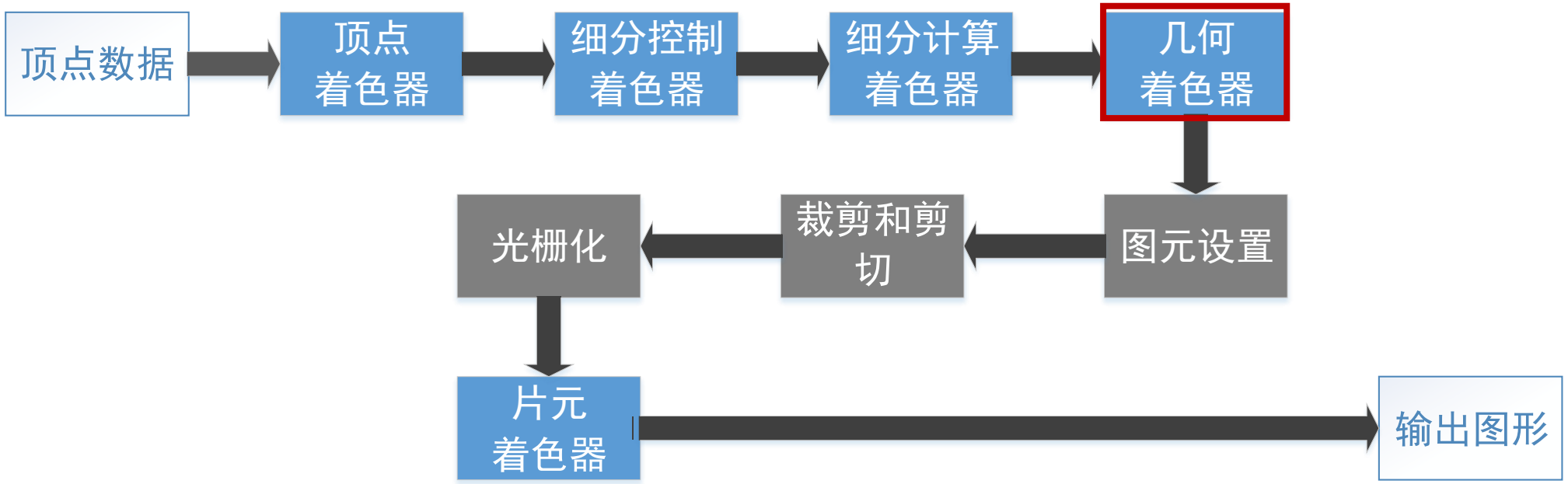
01 OpenGL的绘制流程

- OpenGL的绘制流水线
  - 细分着色器：使用面片（patch）描述物体的形状，细分可以使模型外观变平滑，但会增加点的数量



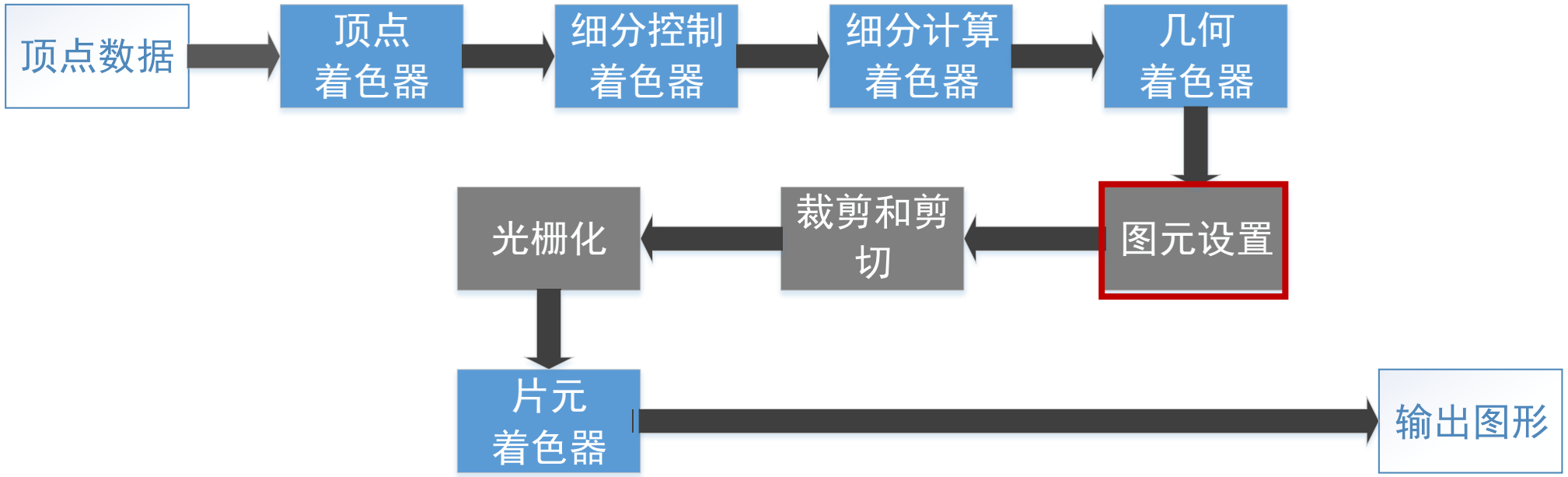
01 OpenGL的绘制流程

- OpenGL的绘制流水线
  - 几何着色：允许在光栅化之前对每个几何图元进行更进一步的处理，如创建新图元。



01 OpenGL的绘制流程

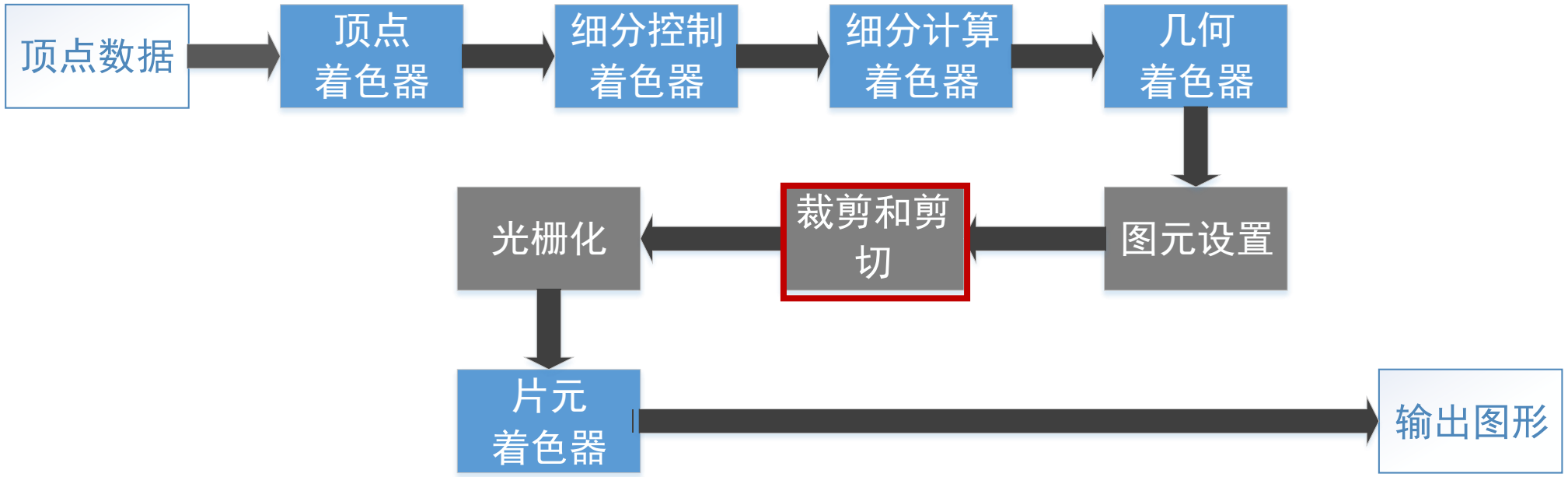
- OpenGL的绘制流水线
  - 图元装配：将前几步处理的顶点数据与相关的几何图元组织起来，准备下一步操作



01 OpenGL的绘制流程

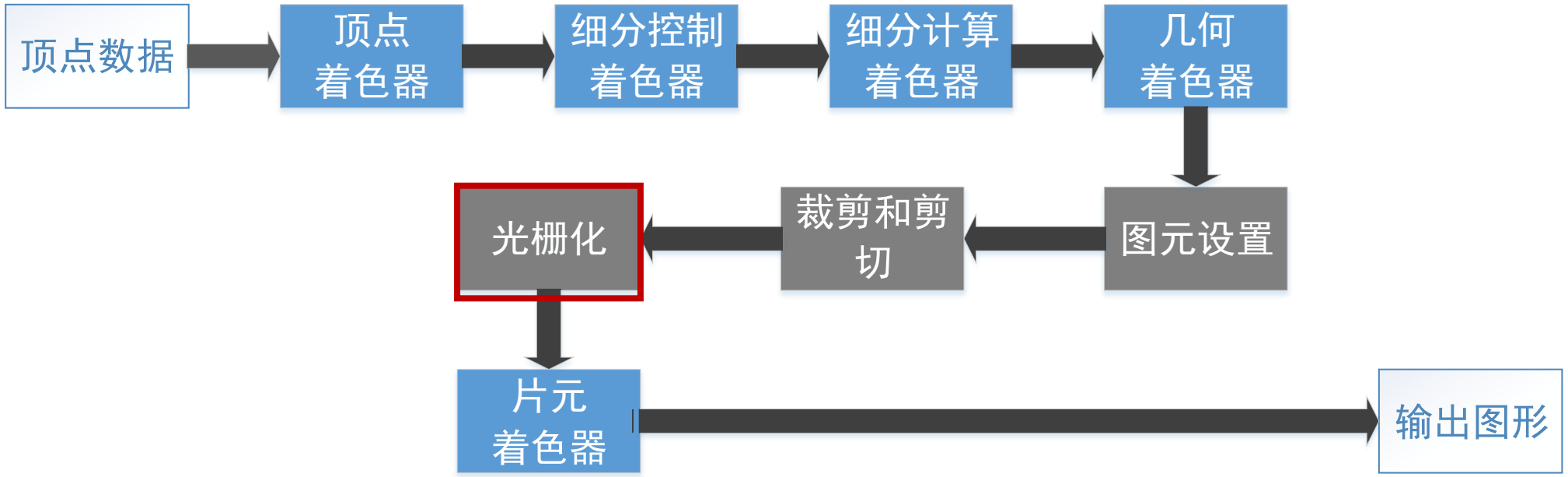
OpenGL的绘制流水线

- 裁剪和剪切：OpenGL根据指定的方式自动完成裁剪过程



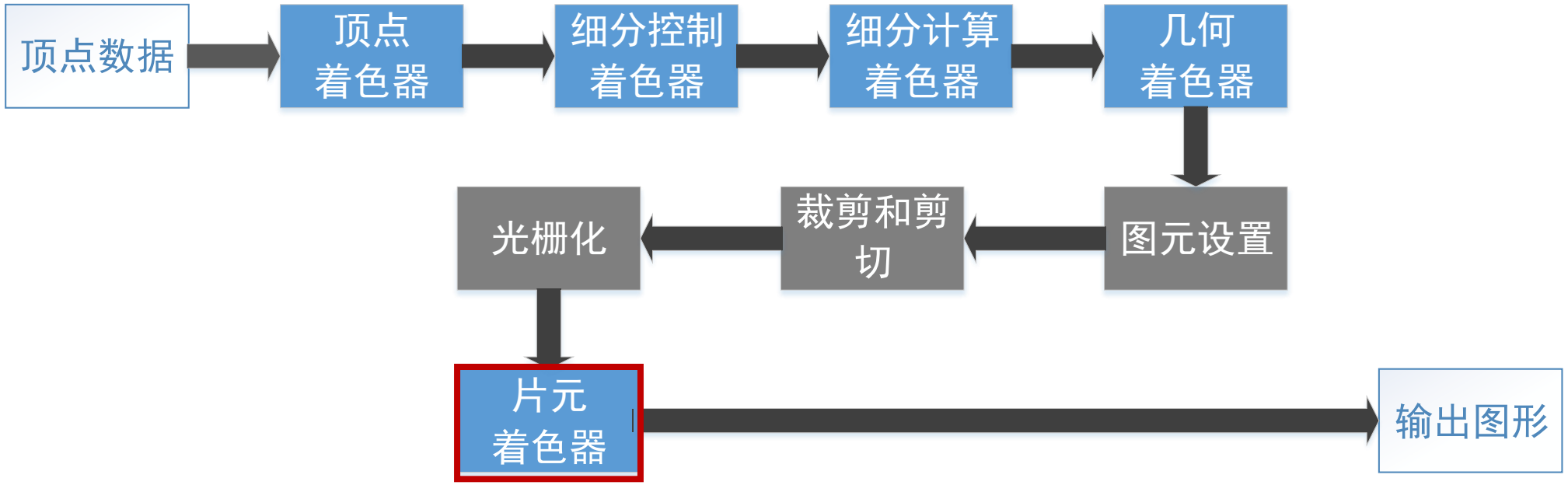
01 OpenGL的绘制流程

- OpenGL的绘制流水线
  - 光栅化：根据图元顶点坐标，生成片元
  - 片元是候选的像素点



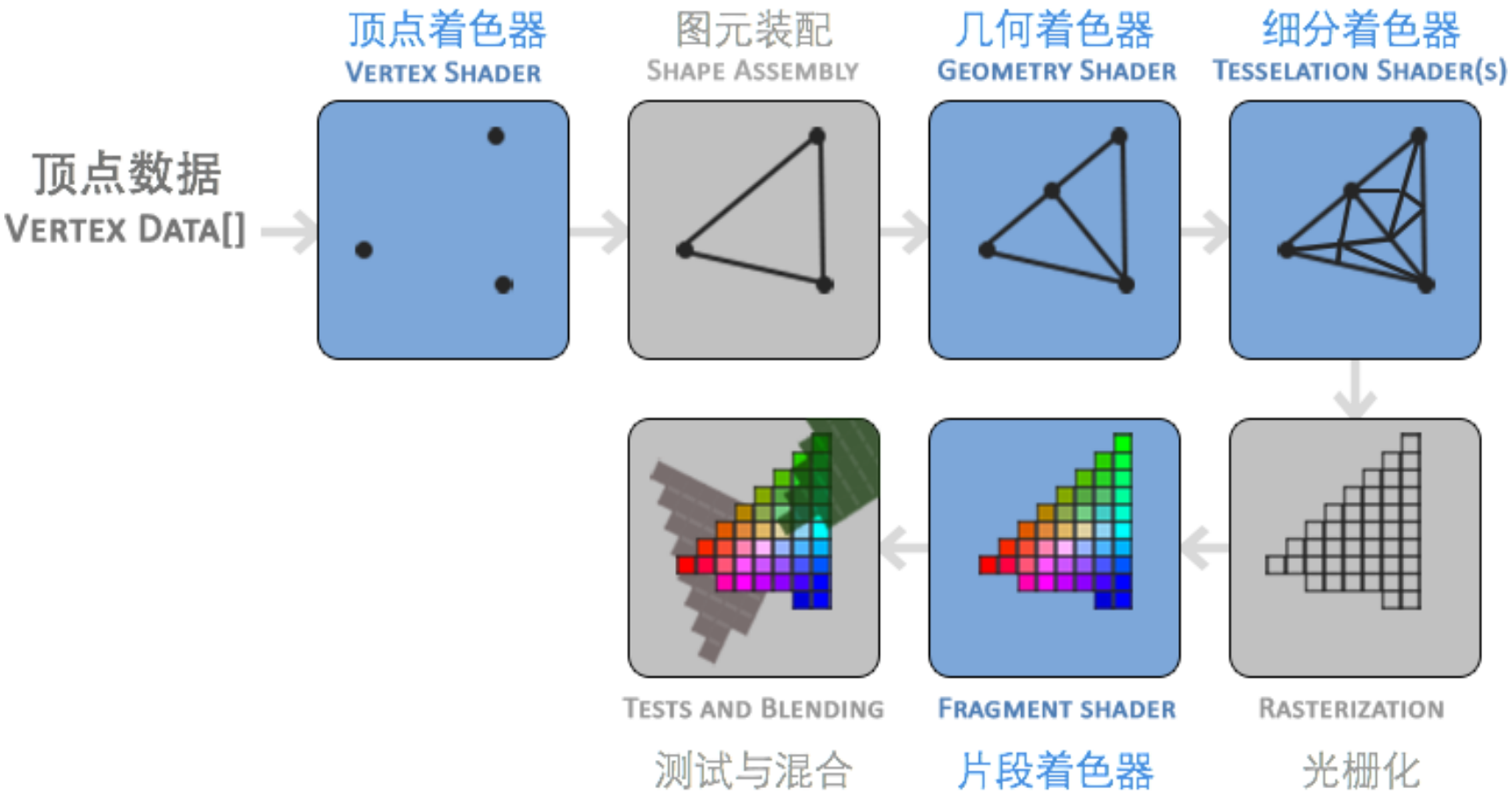
01 OpenGL的绘制流程

- OpenGL的绘制流水线
  - 片元着色器：可以决定片元最终的颜色，是否显示或者终止片元的处理



01 OpenGL的绘制流程

OpenGL的绘制流水线



01 初识OpenGL

- OpenGL软件包
- OpenGL的绘制流程
- **OpenGL的基本语法**
- OpenGL环境配置

01 OpenGL的基本语法

- OpenGL的库
  - 核心库gl
  - 实用程序库glu
  - 编程辅助库：aux
  - 实用程序工具包：glut
  - 实用工具库：glfw
  - 第三方库：glad
  - 跨平台扩展库：glew

01 OpenGL的基本语法

- 命名规则
  - OpenGL函数都遵循一个命名约定，即采用以下格式：

<库前缀><根命令><可选的参数个数><可选的参数类型>

01 OpenGL的基本语法

数据类型

OpenGL数据类型	内部表示法	定义为C类型	C字面值后缀
GLbyte	8位整数	signed char	B
GLshort	16位整数	short	S
GLint，GLsizei	32位整数	long	L
GLfloat，GLclampf	32位浮点数	float	F
GLdouble，GLclampd	64位浮点数	double	D
GLubyte，GLboolean	8位无符号整数	unsigned char	Ub
GLshort	16位无符号整数	unsigned short	Us
GLuint，GLenum，GLbitfield	32位无符号整数	unsigned long	Ui

01 初识OpenGL

- OpenGL软件包
- OpenGL的绘制流程
- OpenGL的基本语法
- **OpenGL环境配置**

01 OpenGL环境配置

- GLFW的环境
  - 访问https://www.glfw.org/download.html下载glfw源码；
  - 下载Cmake(https://cmake.org/download),建议下载win32-x86版本
  - 用Cmake编译源码，得到include和lib

01 OpenGL环境配置

- GLAD库
  - GLAD是一个开源的库，配置也与其他的库有些不同，GLAD使用了在线服务。https://glad.dav1d.de/
  - 选择3.3以上的OpenGL(gl)版本，Profile设置为Core，选中生成加载器(Generate a loader)，忽略拓展(Extensions)中的内容，生成库文件。

```
tial(void)

义图形对象的顶点数据
t  float triangle[] = {
.5f, -0.5f, 0.0f,
5f,  -0.5f, 0.0f,
0f,  0.5f, 0.0f


成并绑定VAO和VBO
nVertexArrays(1, &vertex_array_object);
dVertexArray(vertex_array_object);

nBuffers(1, &vertex_buffer_object);
dBuffer(GL_ARRAY_BUFFER, vertex_buffer_ob

点数据绑定至当前默认的缓冲中
ata(GL_ARRAY_BUFFER, sizeof(t
性指针
ointer(0, 3, GL_FLOA
ribArray(0);
```

PART 02

第一个OpenGL程序

02 第一个OpenGL程序

- OpenGL的程序框架（C语言）
  - 搭建窗口系统——**main**
  - OpenGL初始化——**initial**
  - 对窗口系统的响应操作
    - ✓ 绘图——**Draw**
    - ✓ 窗口改变大小——**reshape**
    - ✓ 键盘相应函数
    - ✓ ……

02 第一个OpenGL程序（基本框架）

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <iostream>

void initial(void)
{…}

void Draw(void)
{…}

void reshape(GLFWwindow* window, int width, int height)
{…}

int main()
{…}
```

第一个OpenGL程序

- 使用GLFW库搭建窗口系统
  - **GLFW库初始化**

02 第一个OpenGL程序（glfw初始化）

```
int main()
{
    //初始化glfw
    glfwInit();

    //OpenGL的版本号
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);

    //OpenGL使用核心模式
    glfwWindowHint(GLFW_OPENGL_PROFILE,
                   GLFW_OPENGL_CORE_PROFILE);

…}
```

02 第一个OpenGL程序

- 使用GLFW库搭建窗口系统
  - GLFW库初始化
  - **创建一个窗口——glfwCreateWindow**
    - 窗口的大小
    - 窗口的名称
    - **NULL/glfwGetPrimaryMonitor()**
  - **指定当前的OpenGL环境——glfwMakeContextCurrent**

02 第一个OpenGL程序

```
int main()
{…
    // 创建窗口(宽、高、窗口名称)
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
                                          SCR_HEIGHT, "First Graphics", NULL, NULL);

    if (window == NULL){
        std::cout << "Failed to Create OpenGL Context" << std::endl;
        glfwTerminate();
        return -1;
    }

    // 将窗口的上下文设置为当前线程的主上下文
    glfwMakeContextCurrent(window);

…}
```

02 第一个OpenGL程序

- 使用GLFW库搭建窗口系统
  - GLFW库初始化
  - 创建一个窗口——**glfwCreateWindow**
  - 指定当前的OpenGL环境——**glfwMakeContextCurrent**
  - **GLAD库的初始化**

02 第一个OpenGL程序（加载glad）

```
int main()
{…
    // 初始化GLAD，加载OpenGL函数指针地址的函数
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
    }

    initial();
…}
```

02 第一个OpenGL程序

- 使用GLFW库搭建窗口系统
  - GLFW窗口管理——无限循环响应
    - 自定义绘制函数——**Draw**
    - 显示处理——**glfwSwapBuffers**
    - 检查消息——**glfwPollEvents**
  - 关闭窗口——**glfwDestroyWindow**
  - 关闭GLFW库——**glfwTerminate**

02 第一个OpenGL程序（设置glfw）

```
int main()
{…
    glfwSetFramebufferSizeCallback(window, reshape);

    while (!glfwWindowShouldClose(window))
    {
        Draw();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();
}
```

02 第一个OpenGL程序

- 初始化——initial
  - 只需要定义一次——图形的几何信息
  - 只需要操作一次——着色器
  - 只需要改变一次——图元的属性

第一个OpenGL程序（定义顶点）

```
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

int initial(void)
{
    // 定义图形对象的顶点数据
    const  float triangle[] = {
        -0.5f, -0.5f, 0.0f,
        0.5f,  -0.5f, 0.0f,
        0.0f,  0.5f, 0.0f
    };
…}
```

第一个OpenGL程序

- 顶点数组——存储所有的顶点数据信息
  - 坐标数据
  - 法矢量数据
  - 颜色数据
  - 纹理数据
  - .....

第一个OpenGL程序

- VAO：顶点数组对象（Vertex Arrary Object，VAO）
  - VAO保存了所有顶点数据的引用
  - VAO把顶点存储在一个对象中，每次绘制模型时，只需要绑定这个VAO对象就可以了
  - 简单理解：为一个图形对象起了一个名字，对应了图形对象顶点的所有信息
  - 注意VAO与顶点数组的区别

第一个OpenGL程序

- VBO：顶点缓冲对象（Vertex Buffer Objects）
  - VBO在显存中开辟出的一块内存缓存区，用于存储顶点的各类属性信息
  - 在渲染时，直接从VBO中取出顶点的各类属性数据，不需要从CPU传输数据，处理效率更高
  - VAO与VBO是关联的
  - 将数组存储（关联）到VBO中——**glBufferData**

02 第一个OpenGL程序（定义VAO和VBO）

```
int initial(void)
{…
    // 生成并绑定VAO和VBO
    glGenVertexArrays(1, &vertex_array_object);
    glBindVertexArray(vertex_array_object);

    glGenBuffers(1, &vertex_buffer_object);
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object);
    // 将顶点数据绑定至当前默认的缓冲中
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
                 vertices,
                 GL_STATIC_DRAW);
    // 设置顶点属性指针
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                          3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
…}
```

Void glCreateVertexArrays(Glsizei n, Gluint *arrays);

OpenGL内部会将它作为当前对象，即所有后继的操作都会作用于这个被绑定的对象

02 第一个OpenGL程序（定义VAO和VBO）

```
int initial(void)
{…
    // 生成并绑定VAO和VBO
    glGenVertexArrays(1, &vertex_array_object);
    glBindVertexArray(vertex_array_object);

    glGenBuffers(1, &vertex_buffer_object);
    glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_object);
    // 将顶点数据绑定至当前默认的缓冲中
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle), triangle,
                 GL_STATIC_DRAW);
    // 设置顶点属性指针
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                          3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
…}
```

02 第一个OpenGL程序（定义VAO和VBO）

```
int initial(void)
{…
    // 生成并绑定VAO和VBO
    glGenVertexArrays(1, &vertex_array_object);
    glBindVertexArray(vertex_
    glGenBuffers(1, &vertex_
    glBindBuffer(GL_ARRAY_
    // 将顶点数据绑定至当前默认的缓冲中
    glBufferData(GL_ARRAY_
                 GL_STATIC_
    // 设置顶点属性指针
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                          3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
…}
```

设置index（着色器中的属性位置）位置对应的数据值。位置0，长度3，数据类型，归一化，偏移量

启用顶点数组属性

第一个OpenGL程序

- 着色器
  - 着色器定义：GLSL语言
  - **编写顶点着色器程序**

第一个OpenGL程序

- 着色器
  - 着色器定义：GLSL语言
  - **编写顶点着色器程序**

```
const char *vertex_shader_source =
"        #version 330 core                          \n"
"        layout (location = 0) in vec3 aPos;        \n"
"        void main()                                \n"
"        {                                          \n"
"            gl_Position = vec4(aPos, 1.0);         \n"
"         }                                         \n\0";
```

第一个OpenGL程序

- 着色器
  - 着色器定义：GLSL语言
  - 编写顶点着色器程序
  - **编写片段着色器程序**

```
const char *fragment_shader_source =
“    #version 330 core                                    \n"
“    out vec4 FragColor;                                  \n"
“    void main()                                          \n"
“    {                                                    \n"
"            FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);    \n"
“     }                                                   \n\0";
```

第一个OpenGL程序

- 着色器
  - 着色器定义：GLSL语言
  - 编写顶点着色器程序
  - 编写片段着色器程序
  - **编译顶点着色器和片段着色器**
    - **创建着色器**
    - **指定着色器的源代码**
    - **对着色器进行编译**

02 第一个OpenGL程序（编译着色器）

```
int initial(void)
{…
    int success;
    char info_log[512];

    // 生成并编译顶点着色器
    int vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, &vertex_shader_source, NULL);
    glCompileShader(vertex_shader);
    // 检查着色器是否成功编译，如果编译失败，打印错误信息
    glGetShaderiv(vertex_shader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
    glGetShaderInfoLog(vertex_shader, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << info_log << std::endl;
    }
…}
```

02 第一个OpenGL程序（编译着色器）

```
int initial(void)
{…
    // 生成并编译片段着色器
    int fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, &fragment_shader_source,
                                    NULL);
    glCompileShader(fragment_shader);
    // 检查着色器是否成功编译，如果编译失败，打印错误信息
    glGetShaderiv(fragment_shader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
    glGetShaderInfoLog(fragment_shader, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << info_log << std::endl;
    }
…}
```

02 第一个OpenGL程序

- 着色器
  - 着色器定义：GLSL语言
  - 编写顶点着色器程序
  - 编写片段着色器程序
  - 编译顶点着色器和片段着色器
  - **将着色器打包成一个着色器程序**
  - **指定当前画图使用的着色器程序**

02 第一个OpenGL程序（构建一个处理程序）

```
int initial(void)
{…
    // 链接顶点和片段着色器至一个着色器程序
    int shader_program = glCreateProgram();
    glAttachShader(shader_program, vertex_shader);
    glAttachShader(shader_program, fragment_shader);
    glLinkProgram(shader_program);
    // 检查着色器是否成功链接，如果链接失败，打印错误信息
    glGetProgramiv(shader_program, GL_LINK_STATUS, &success);
    if (!success) {
    glGetProgramInfoLog(shader_program, 512, NULL, info_log);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << info_log << std::endl;
    }
    glUseProgram(shader_program);
…}
```

第一个OpenGL程序

- 绘图——Draw
  - 清除颜色缓冲区的内容（用背景色填充颜色缓冲区）
  - 绘制对象
    - 指定当前需要绘制的对象（绑定VAO）
    - 按照指定的顶点绘制方式画出图形
    - 绘制其它对象

01 第一个OpenGL程序（绘图函数）

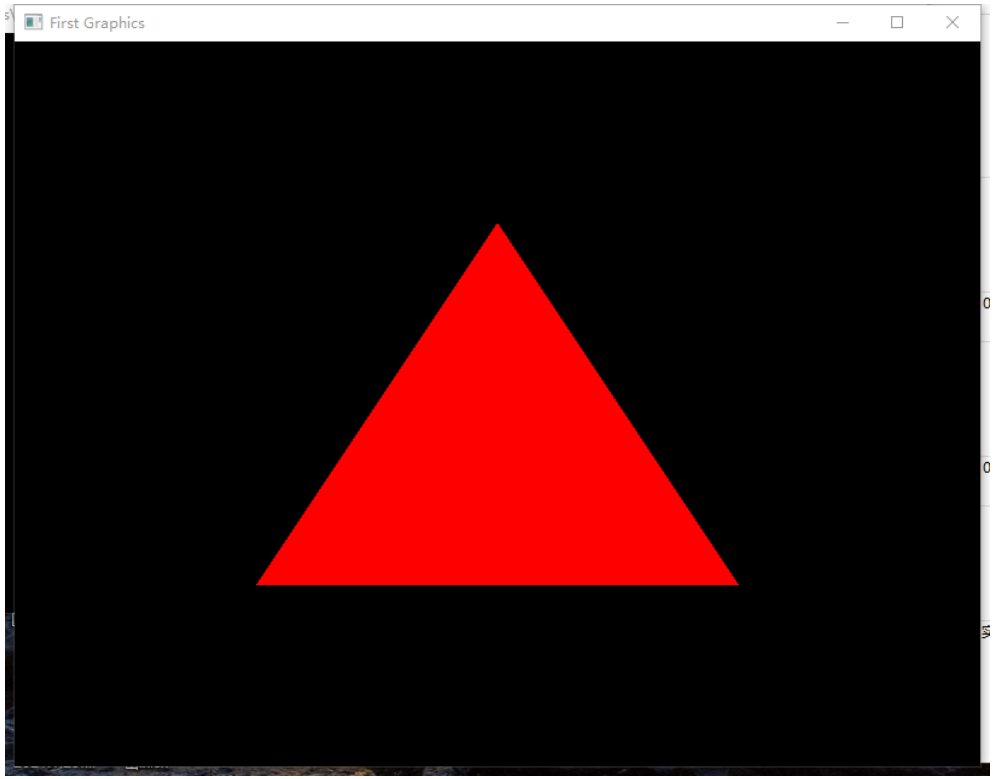
```
void Draw(void)
{
    // 清空颜色缓冲
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // 绘制三角形
    // 绑定VAO
    glBindVertexArray(vertex_array_object);
    // 绘制三角形
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // 解除绑定
    glBindVertexArray(0);
}
```

第一个OpenGL程序

- 窗口大小变化——**reshape**
  - 当窗口大小变化时，显示的分辨率（坐标）发生变化
  - 显示内容的分辨率（坐标）变化

02 第一个OpenGL程序（窗口整形函数）

```
void reshape(GLFWwindow* window, int width, int height)
{

// 指定当前视口尺寸(前两个参数为左下角位置，后两个参数是渲染窗口宽、高)
glViewport(0, 0, width, height);

}
```



```
tial(void)

义图形对象的顶点数据
t  float triangle[] = {
.5f, -0.5f, 0.0f,
5f,  -0.5f, 0.0f,
0f,  0.5f, 0.0f


成并绑定VAO和VBO
nVertexArrays(1, &vertex_array_object);
dVertexArray(vertex_array_object);

nBuffers(1, &vertex_buffer_object);
dBuffer(GL_ARRAY_BUFFER, vertex_buffer_ob

点数据绑定至当前默认的缓冲中
ata(GL_ARRAY_BUFFER, sizeof(t
性指针
ointer(0, 3, GL_FLOA
ribArray(0);
```

PART 03

图元属性

03 图元属性

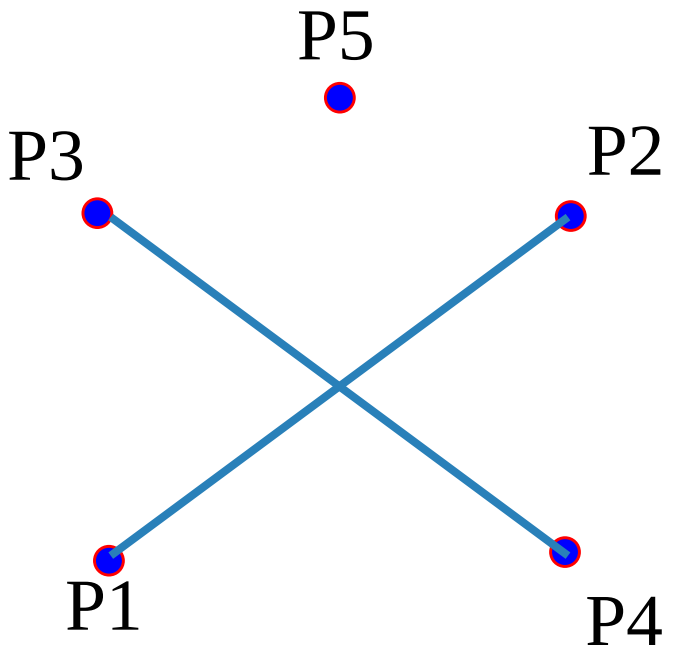
- 点的属性
- 线的属性
- 面的属性
- 让图形动起来（旋转）

03 点与线的属性

- 点的大小
  - void glPointSize(float size);
- 线的宽度
  - void glLineWidth(float width)

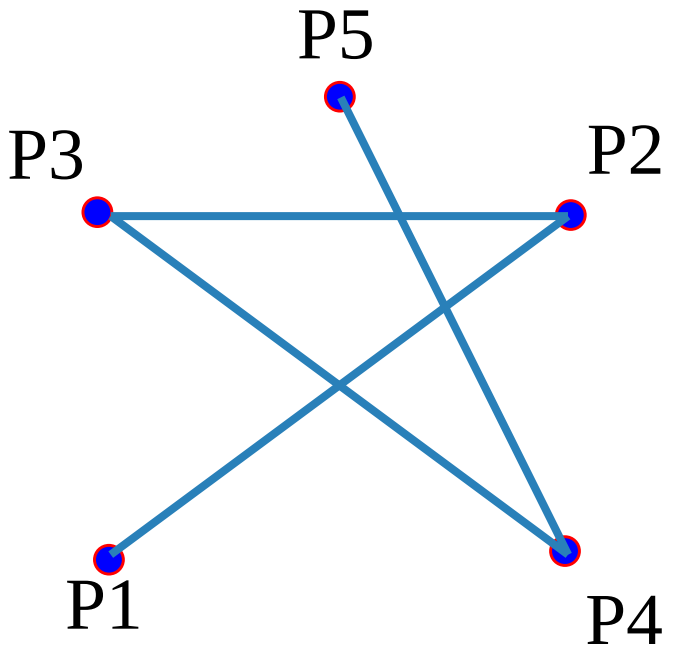
03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - **GL_LINES**



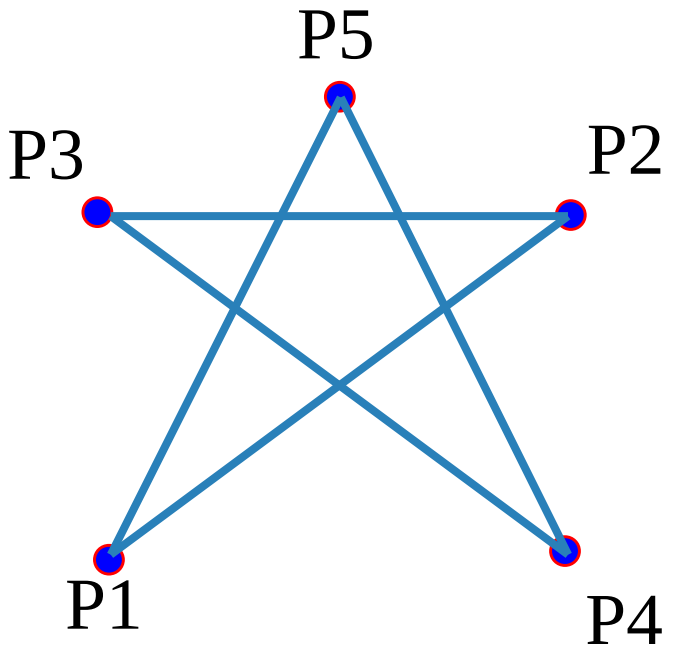
03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - GL_LINES
  - **GL_LINE_STRIP**



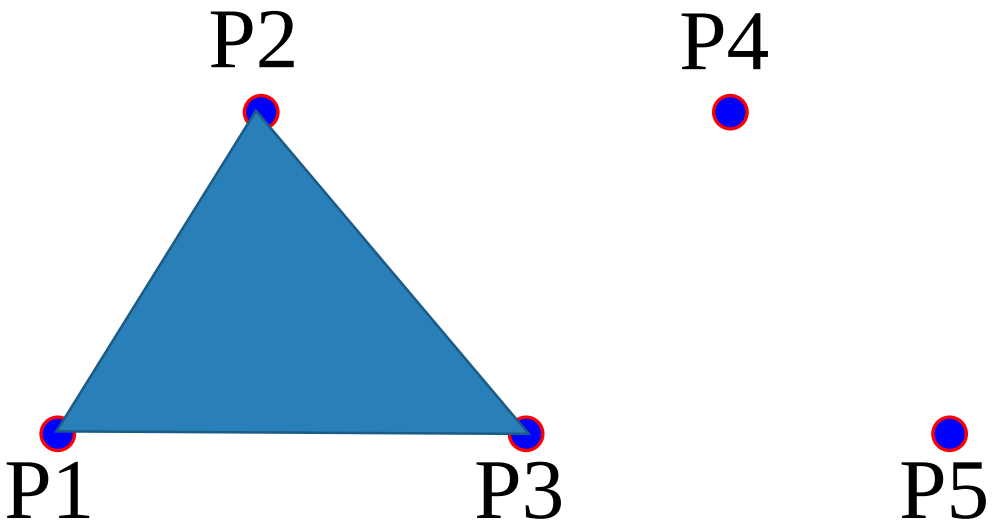
03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - GL_LINES
  - GL_LINE_STRIP
  - **GL_LINE_LOOP**



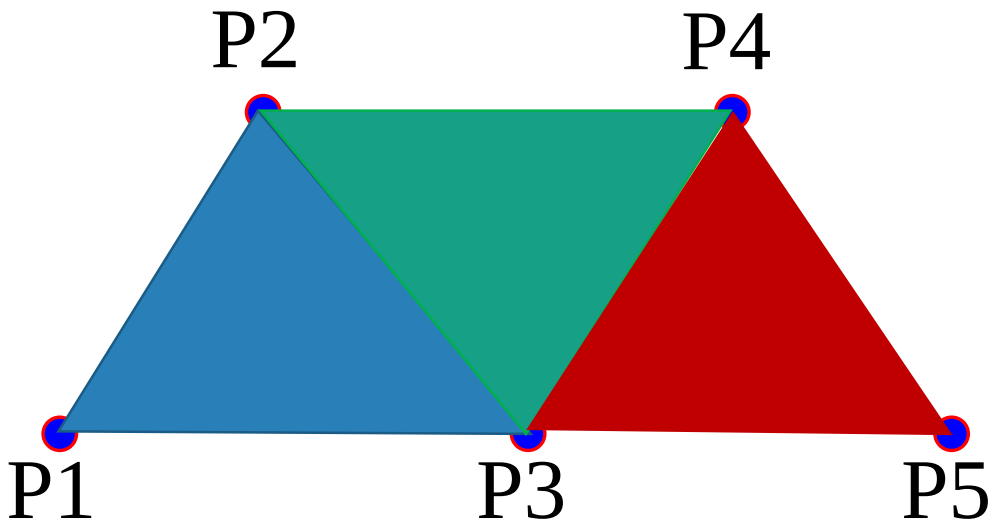
03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - GL_LINES
  - GL_LINE_STRIP
  - GL_LINE_LOOP
  - **GL_TRIANGLES**



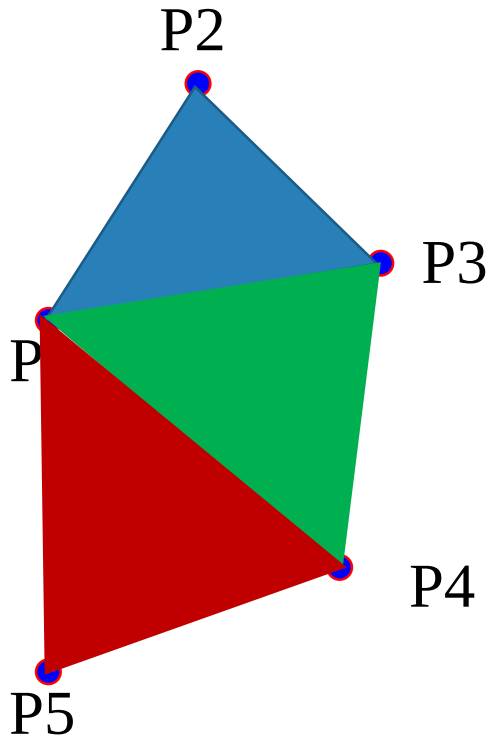
03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - GL_LINES
  - GL_LINE_STRIP
  - GL_LINE_LOOP
  - GL_TRIANGLES
  - **GL_TRIANGLE_STRIP**



03 点线面的绘制

- 绘制模式glDrawArrays(GL_TRIANGLES, 0, 3);
  - GL_POINTS
  - GL_LINES
  - GL_LINE_STRIP
  - GL_LINE_LOOP
  - GL_TRIANGLES
  - GL_TRIANGLE_STRIP
  - **GL_TRIANGLE_FAN**



03 点线面的绘制

- 多边形面的绘制规则
  - 所有多边形都必须是平面的
  - 多边形的边缘决不能相交，而且多边形必须是凸的
- 多边形面的正反属性（绕法）
  - 指定顶点时顺序和方向的组合称为“绕法”。绕法是任何多边形图元的一个重要特性。一般默认情况下，OpenGL认为逆时针绕法的多边形是正对着的。

```
glFrontFace(GL_CW);
```

03 点线面的绘制

- 多边形面的显示模式

```
glPolygonMode(GLenum face，GLenum mode);
```

  - 参数face用于指定多边形的哪一个面受到模式改变的影响。

GL_FRONT；GL_BACK；GL_FRONT_AND_BACK

  - 参数mode用于指定新的绘图模式

GL_POINT；GL_LINE；GL_FILL

03 面的属性

- 多边形面的剔除

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
glDisable(GL_CULL_FACE);
```

03 图元属性

- 点线面的属性
- **让图形动起来（旋转）**

03 图形的旋转

```
//增加旋转参数
    static GLfloat xRot = 20.0f;
    static GLfloat yRot = 20.0f;

int main()
{…
     //窗口键盘事件调用函数key_callback
    glfwSetKeyCallback(window, key_callback);

    //显示操作说明
    std::cout << "方向键可以控制图形的旋转。" << std::endl;
…}
```

03 图形的旋转

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    switch (key)
    {
        case GLFW_KEY_UP:
            xRot -= 5.0f;          break;
        case GLFW_KEY_DOWN:
            xRot += 5.0f;          break;
        case GLFW_KEY_LEFT:
            yRot -= 5.0f;          break;
        case GLFW_KEY_RIGHT:
            yRot += 5.0f;          break;
    }
}
```

03 图形的旋转

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    switch (key)
    {
        case GLFW_KEY_UP:
            xRot -= 5.0f;          break;
        case GLFW_KEY_DOWN:
            xRot += 5.0f;          break;
        case GLFW_KEY_LEFT:
            yRot -= 5.0f;          break;
        case GLFW_KEY_RIGHT:
            yRot += 5.0f;          break;
    }
}
```

03 图形的旋转

```
void initial(void)
{…
    // 定义图形对象的顶点数据
    const  float triangle[] = {
        -0.5f, -0.5f, 0.0f,
        0.5f,  -0.5f, 0.0f,
        0.0f,  0.5f, 0.0f
    };
    // 定义图形对象的颜色数据，与顶点匹配
    const  float color[] = {
        1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f
    };
…}
```

03 图形的旋转

- 定义VAO和VBO
  - 处理多类顶点数据——**glBufferSubData**

03 图形的旋转

```
void initial(void)
{…
    // 将顶点数据绑定至当前默认的缓冲中
    glBufferData(GL_ARRAY_BUFFER, sizeof(triangle) + sizeof(color),
                 NULL, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(triangle), triangle);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(triangle), sizeof(color),
                    color);
    // 设置顶点属性指针
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float),
                          (void*)sizeof(triangle));
    glEnableVertexAttribArray(1);
…}
```

03 图形的旋转

- 顶点着色器
  - 接收多组数据
  - 处理顶点的变换

03 图形的旋转

```
void initial(void)
{…
    const char *vertex_shader_source =
    "#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "layout (location = 1) in vec4 vColor;\n"
    "out vec4 color;\n"
    "uniform mat4 transform;\n"
    "void main()\n"
    "{\n"
    "    gl_Position = transform*vec4(aPos, 1.0);\n"
    "    color = vColor;\n"
    "}\n\0";
…}
```

图形的旋转

```
void initial(void)
{…
    const char *fragment_shader_source =
    "#version 330 core\n"
    "in vec4 color;\n"                    // 输入的颜色向量
    "out vec4 FragColor;\n"               // 输出的颜色向量
    "void main()\n"
    "{\n"
    "    FragColor = color;\n"
    "}\n\0";
…}
```

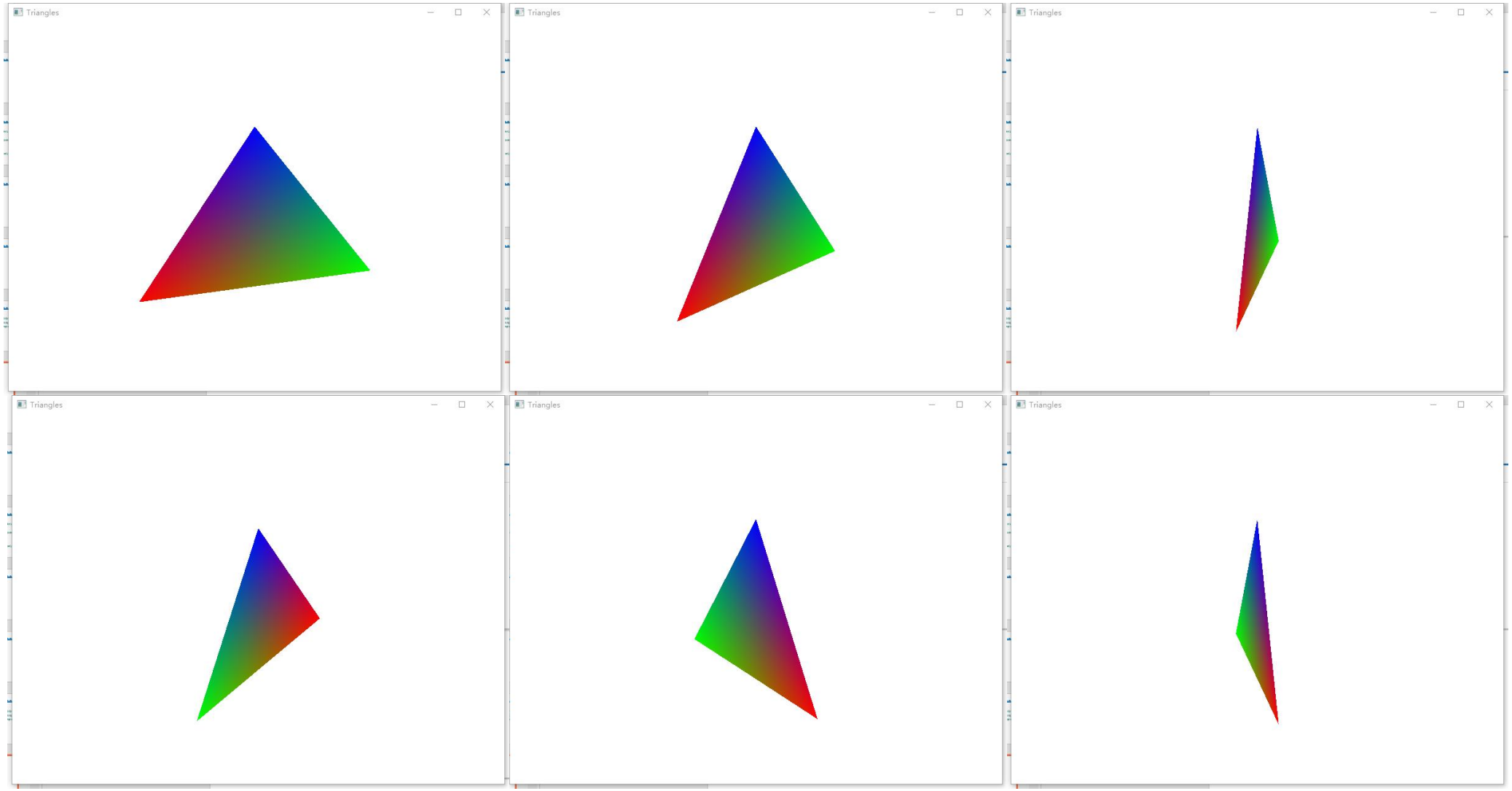
03 图形的旋转

- 矩阵参数的传递
  - 定义变换矩阵参数
  - 指定处理变换的着色器程序，以及顶点着色器中接收矩阵参数的变量名称
  - 将矩阵数据传输到着色器程序

03 图形的旋转

```
void Draw(void)
{…
    //处理图形的旋转
    vmath::mat4 trans = vmath::rotate(xRot, vmath::vec3(1.0, 0.0, 0.0)) *
                        vmath::rotate(yRot, vmath::vec3(0.0, 1.0, 0.0));
    unsigned int transformLoc = glGetUniformLocation(shader_program,
                                "transform");
    glUniformMatrix4fv(transformLoc, 1, GL_FALSE, trans);
    // 绘制三角形
    glBindVertexArray(vertex_array_object);          // 绑定VAO
    glDrawArrays(GL_TRIANGLES, 0, 3);                // 绘制三角形
    glBindVertexArray(0);                            // 解除绑定
…}
```

03 图形的旋转


Triangles
Triangles
Triangles
Triangles
Triangles
Triangles

```
tial(void)

义图形对象的顶点数据
t  float triangle[] = {
.5f, -0.5f, 0.0f,
5f,  -0.5f, 0.0f,
0f,  0.5f, 0.0f


成并绑定VAO和VBO
nVertexArrays(1, &vertex_array_object);
dVertexArray(vertex_array_object);

nBuffers(1, &vertex_buffer_object);
dBuffer(GL_ARRAY_BUFFER, vertex_buffer_ob

点数据绑定至当前默认的缓冲中
ata(GL_ARRAY_BUFFER, sizeof(t
性指针
ointer(0, 3, GL_FLOA
ribArray(0);
```

PART 04

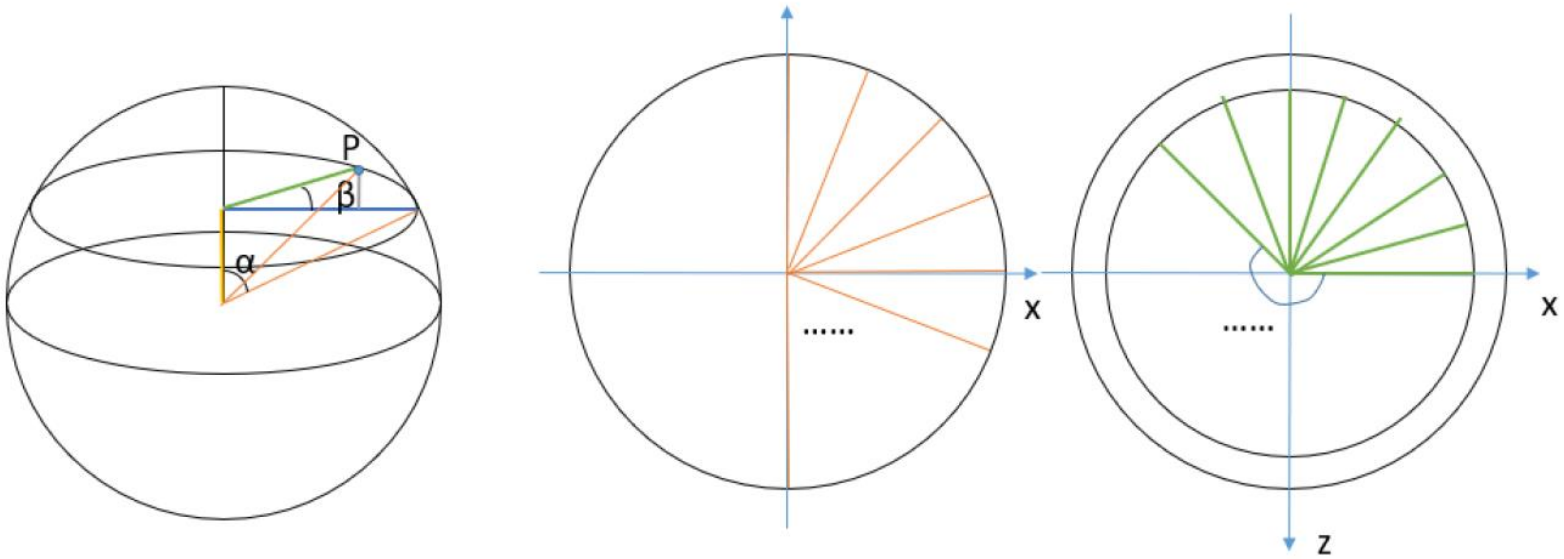
简单曲面绘制

04 简单曲面绘制

球的绘制

04 球的绘制

- 求出球面上点的坐标
  - 球的位置和半径
  - 球的分割：经线和纬线
  - 球面上的每个点的坐标



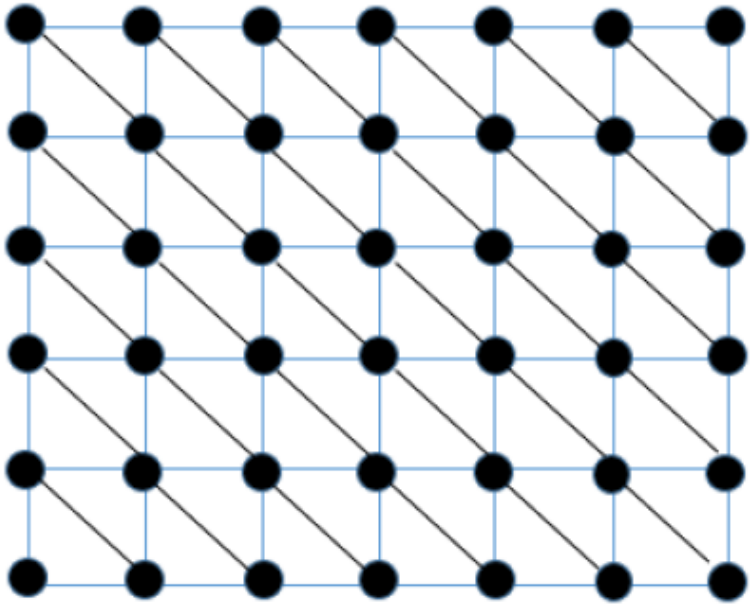
04 图形的旋转

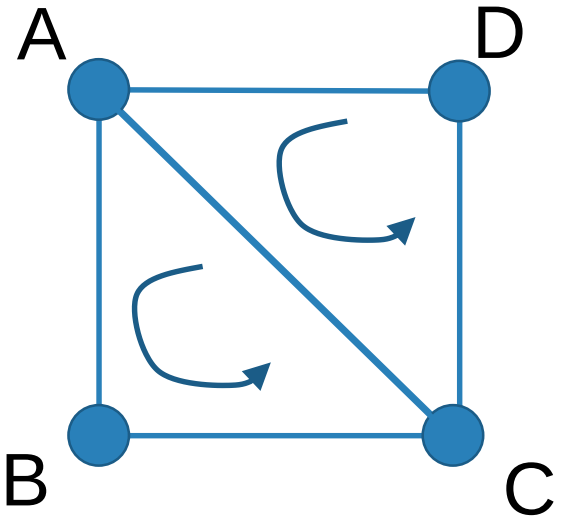
```
for (int y = 0; y <= Y_SEGMENTS; y++)
{
    for (int x = 0; x <= X_SEGMENTS; x++)
    {
        float xSegment = (float)x / (float)X_SEGMENTS;
        float ySegment = (float)y / (float)Y_SEGMENTS;
        float xPos = std::cos(xSegment * Radio * PI) * std::sin(ySegment * PI);
        float yPos = std::cos(ySegment * PI);
        float zPos = std::sin(xSegment * Radio * PI) * std::sin(ySegment * PI);

        sphereVertices.push_back(xPos);
        sphereVertices.push_back(yPos);
        sphereVertices.push_back(zPos);
    }
}
```

04 球的绘制

构造球面

利用球面点坐标构造三角形面





{A，B，C}

{A，C，D}

04 球的绘制

- 构造球面
  - 利用球面点坐标构造三角形面
  - 使用EBO
    - ✓ EBO：索引缓冲对象（Element Buffer Object，EBO）
    - ✓ EBO跟VBO类似，也是在显存中的一块内存缓冲器，只不过EBO保存的是顶点的索引
    - ✓ 解决同一个顶点多次重复调用的问题，减少内存空间浪费，提高执行效率

04 图形的旋转

```
for (int i = 0; i < Y_SEGMENTS; i++)
{
    for (int j = 0; j < X_SEGMENTS; j++)
    {
        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j);
        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j);
        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j + 1);

        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j);
        sphereIndices.push_back((i + 1) * (X_SEGMENTS + 1) + j + 1);
        sphereIndices.push_back(i * (X_SEGMENTS + 1) + j + 1);
    }
}
```

04 第一个OpenGL程序

```
void initial(void)
{…
    glGenBuffers(1, &element_buffer_object);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, element_buffer_object);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sphereIndices.size() *
                        sizeof(int), &sphereIndices[0], GL_STATIC_DRAW);
…}
```

感谢您的观看