计算机图形学

Computer Graphics

陈中贵
chenzhonggui@xmu.edu.cn
http://graphics.xmu.edu.cn/~zgchen

Graphics@XMU

第二章

OpenGL图形管线

OpenGL简介

主要内容

- 图形 API的发展
- OpenGL的体系结构
  - 状态机 state machine
- 函数
  - 类型
  - 格式
- 安装编译

OpenGL

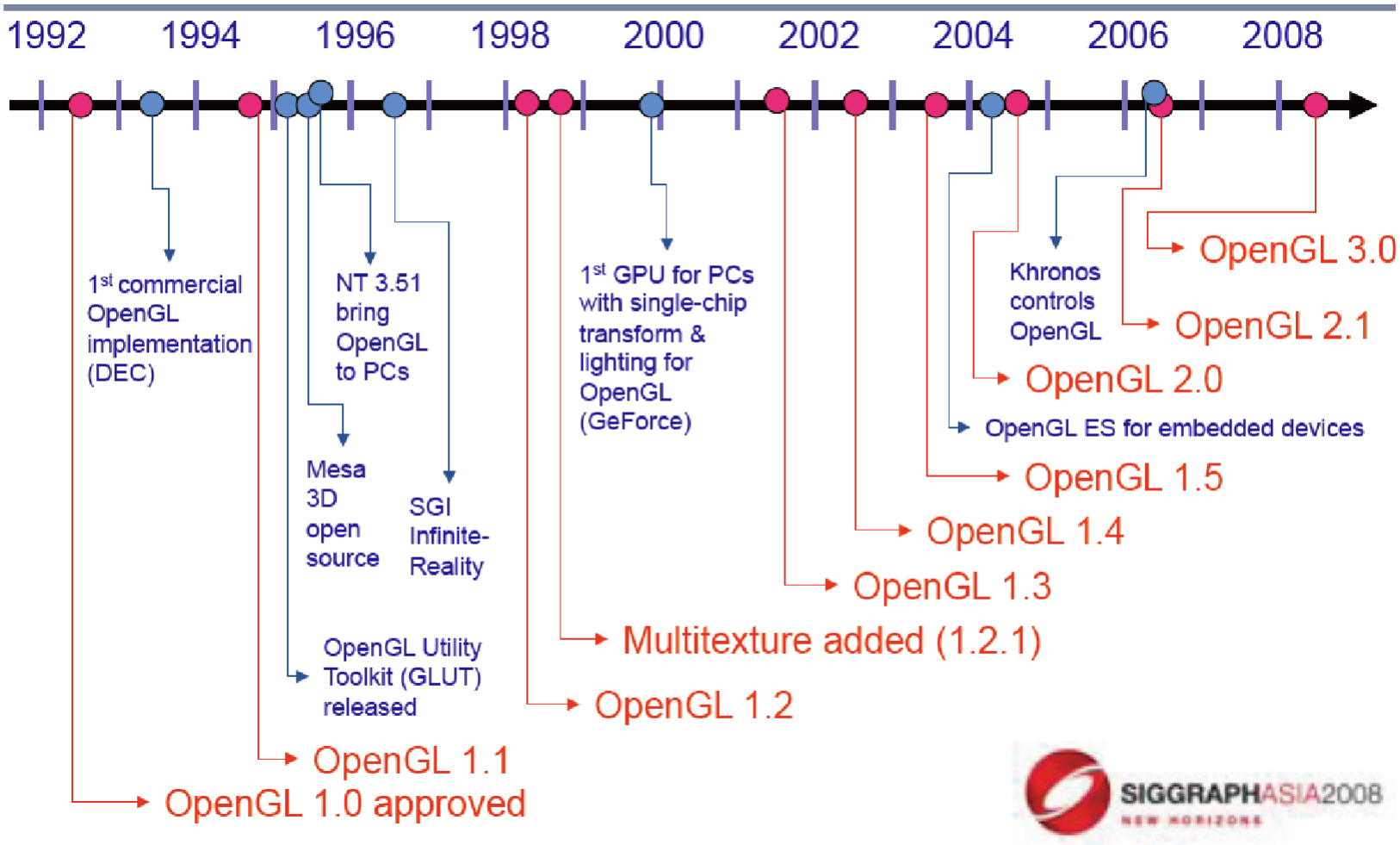
- 1992年 SGI领导的OpenGL Architectural Review Board(OpenGL ARB)发布1.0版
  - 平台无关的API：
  - 易于使用
  - 与硬件非常贴近，从而可以充分发挥其性能
  - 着重在于渲染 (rendering)
  - 没有提供窗口和输入接口，从而避免依赖于具体的窗口系统

OpenGL的发展

- 早期是由 ARB掌控其发展
  - 成员包括SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, ……
  - 相对稳定 2.1(2006.7)/3.3(2010.3)/4.5(2014.8)
  - 发展反映了新的硬件能力
    - 3D纹理映射和纹理对象(1.2, 1998)
    - 顶点着色器、片段着色器(2.0, 2004)
    - 几何着色器(3.2, 2009)
    - Tessellator(4.1, 2011)
    - Compute Shader(4.3, 2012)
- 通过扩展支持平台相关的特性
- 2006年，ARB被Khronos工作组取代

http://en.wikipedia.org/wiki/OpenGL

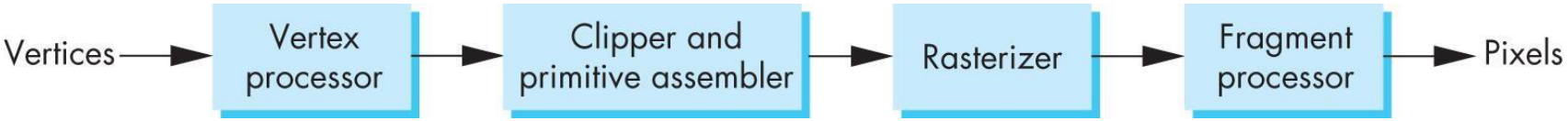
OpenGL的发展



SIGGRAPHASIA2008
NEW HORIZONS

现代OpenGL

- 利用GPU而不是CPU来获得性能提升
- 通过称为着色器(shader）的程序来控制
- 应用程序的工作就是把数据发送到GPU
- GPU执行所有的渲染任务



OpenGL 3.1

- 完全基于shader
  - 没有缺省的shader
  - 每个应用程序必须提供vertex shader和fragment shader
- 取消立即模式（immediate mode）
- 减少状态变量
- 大部分OpenGL 2.5版本增加的功能废弃
- 非向后兼容

其他版本

- OpenGL ES
  - 嵌入式系统
  - 1.0版：简化的OpenGL 2.1
  - 2.0版：简化的OpenGL 3.1
    - Shader based
- WebGL
  - OpenGL ES 2.0的Javascript实现
  - 被新版本的网络浏览器支持
- OpenGL 4.4 （2013）
- 支持geometry shaders、tessellator和compute shaders
- OpenGL 4.6 (2017)
- Vulkan 1.0 (2015): 下一代OpenGL

Direct3D

- DirectX: 微软开发的多媒体编程接口
- Direct3D: DirectX的3D图形API
  - 1.0: 1995
  - 2.0: Windows 95 OSR2& NT 4.0, 1996
  - 9.0c: Windows Xp SP2, 2004
  - 10.1: Windows Vista SP1, 2008
  - 11: Windows 7&Vista, 2009
  - 11.1: Windows 8, 2011
  - 11.2 : Windows 8.1, 2013

OpenGL vs Direct3D

OpenGL

- 跨平台的开放式标准API
  - 可扩展机制
- 可硬件加速的3D渲染系统
  - 底层实现（驱动）管理硬件
- 专业图形应用、科研
  - 跨平台，可移植
- 适合图形学教学

Direct3D

- Windows平台的专利API
  - 一致性好
- 3D硬件接口
  - 应用程序管理硬件资源
- 计算机游戏
  - 高性能硬件存取能力
- Direct3D 7.0能匹敌，8.0（2001）开始胜出

OpenGL库

- OpenGL核心库(OpenGL Core Library)
  - 函数名gl开头
  - Windows: opengl32.dll (WINDOWS\SYSTEM32)
    - Windows Xp支持OpenGL 1.1，Vista支持1.4
    - Direct3D的封装，需安装驱动来实现硬件加速
  - 大多数Unix/Linux系统：GL库 (libGL.a)
- OpenGL实用库(OpenGL Utility Library, GLU)
  - OpenGL的一部分，函数名以glu开头
    - Windows: glu32.dll
  - 利用OpenGL实用库提供一些功能，避免重复编写代码
  - 二次曲面、NURBS、多边形网格化等

GLUT

- OpenGL实用工具库（OpenGL Utility Toolkit Library, GLUT）
  - 提供所有窗口系统的共同功能
    - 创建窗口
    - 从鼠标和键盘获取输入
    - 菜单
    - 事件驱动
- 代码可以在平台间移植，但是GLUT缺乏一些现代GUI的控件和功能
  - 无滚动条
  - 可用FLTK、SDL

http://www.opengl.org/resources/libraries/glut/

freeglut

- GLUT库已经很久没有更新
  - 可以和OpenGL 3.1一起使用
  - 有些功能不能使用，因为需要废弃的函数
- Freeglut是类似GLUT的开源扩展
  - 增加的功能
  - 上下文检查

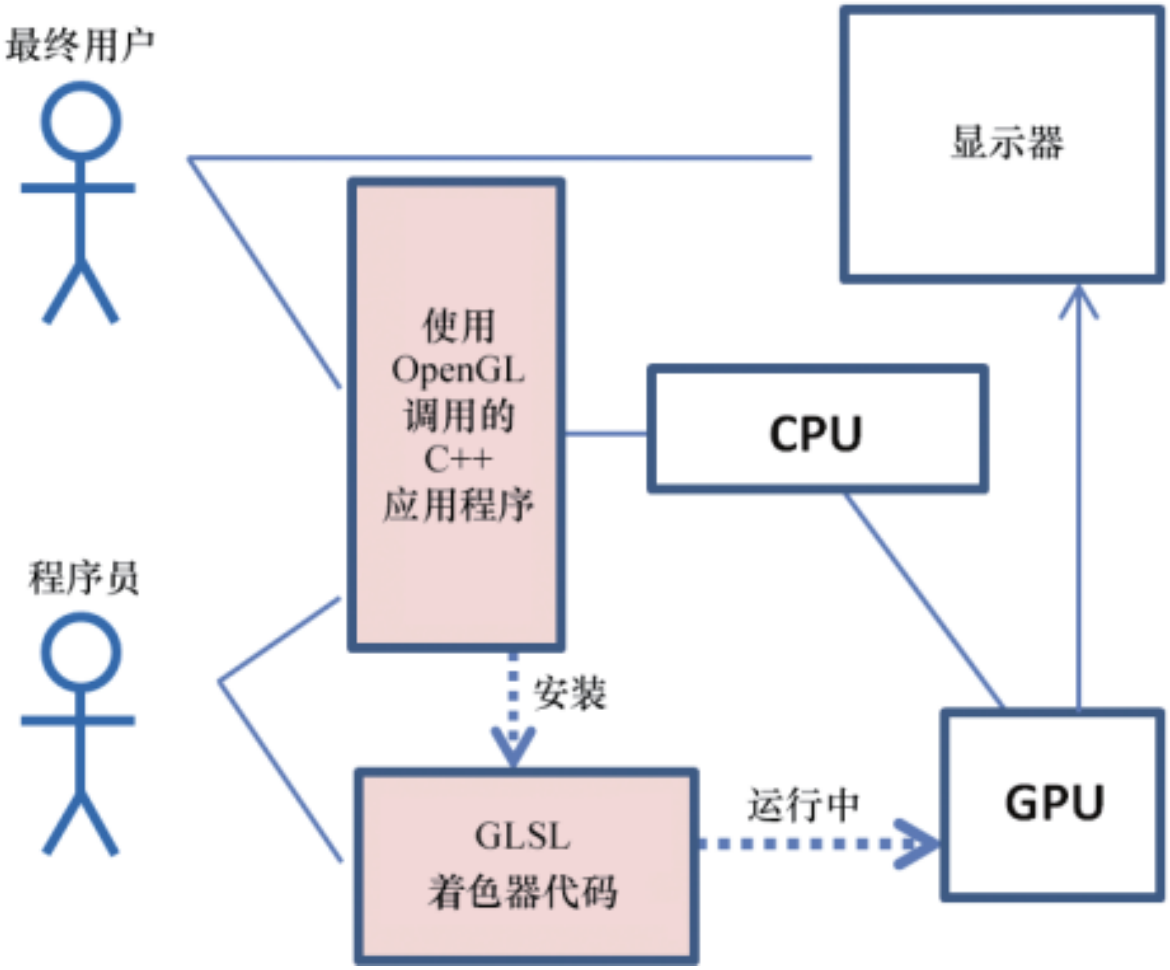
GLFW

- Graphics Library Framework
- 专门针对OpenGL的C语言库
- 提供一些渲染物体所需的最低限度的接口
- 允许用户创建OpenGL上下文
- 定义窗口参数
- 处理用户输入
- https://www.glfw.org/

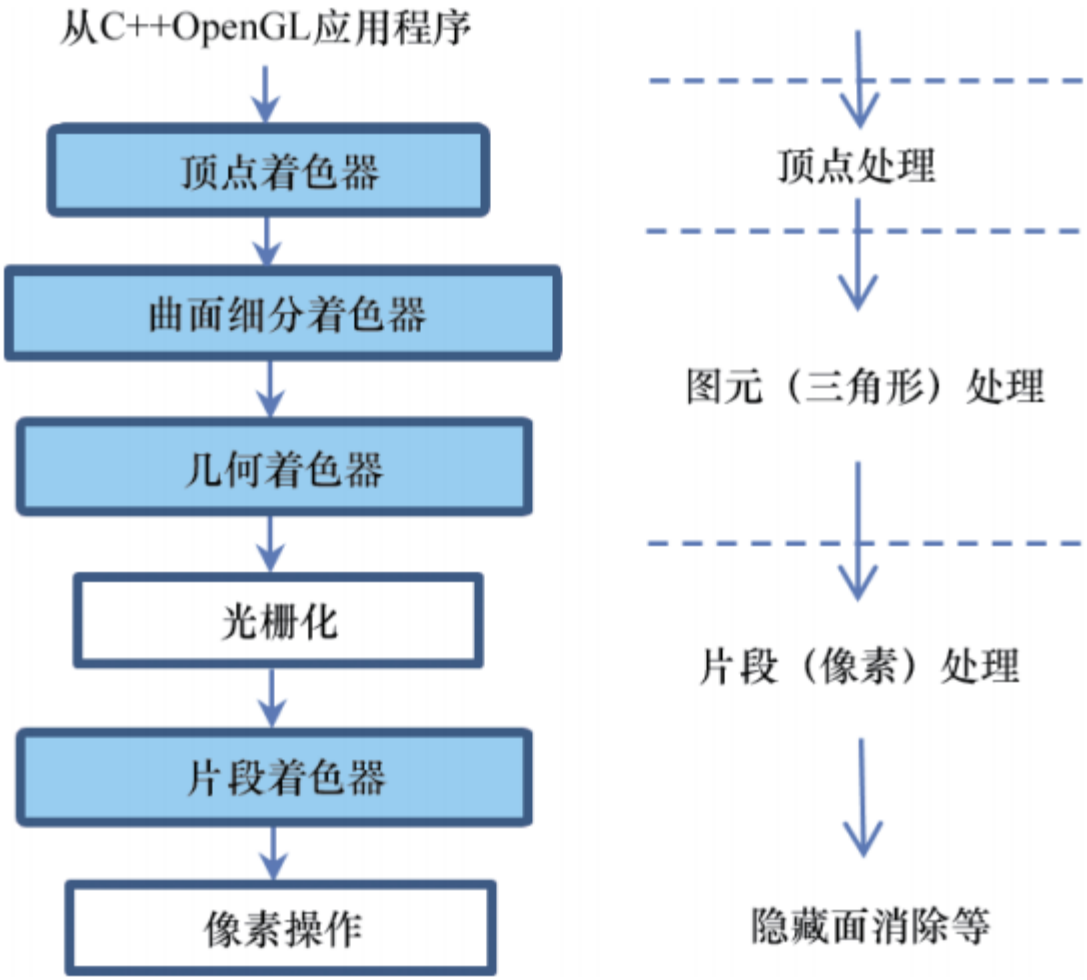
GLEW

- OpenGL Extension Wrangler Library：跨平台的开源OpenGL扩展加载库
- 使得调用特定系统支持的OpenGL扩展功能更简单
- 对于windows代码来说，避免直接调用实体入口
- 应用程序只需要包含glew.h头文件，并调用glewInit()即可

基于C++的图形应用概览



OpenGL管线概览


从C++OpenGL应用程序
顶点着色器
曲面细分着色器
几何着色器
光栅化
片段着色器
像素操作
顶点处理
图元（三角形）处理
片段（像素）处理
隐藏面消除等

图形函数

- 图元函数(primitive) what
  - 系统可以显示的低级对象或最基本的实体，包括点、线段、多边形、像素、文本和各种曲线/曲面等
- 属性函数(attribute) how
  - 控制图元在显示器上显示的方式：线段颜色、多边形填充模式、图标题文本的字体等
- 视图函数(viewing)
  - 设置虚拟照相机的位置、朝向和镜头参数等。
- 变换函数(transformation)
  - 对对象进行诸如平移、旋转和缩放等变换操作。
- 输入函数(input)
  - 处理来自键盘、鼠标等设备的输入
- 控制函数(control)
  - 与窗口系统通信，初始化程序，处理运行时的错误等
- 查询函数(query)
  - 确定特定系统或设备的性能参数，查询相机参数、帧缓冲区等API相关的信息

OpenGL的状态

- OpenGL是一个有限状态机(state machine)的黑盒
  - 状态：持续性参数，如颜色、线型、材质属性等
  - 来自应用程序的输入改变machine的状态或者产生可见的输出
- OpenGL函数有两种类型
  - 定义图元
    - 如果图元可见，则被输出
    - 顶点如何被处理，图元的外观由状态控制
  - 改变状态
    - 属性函数
    - 视图函数
    - 变换函数
    - 在3.1以下版本，大部分状态变量由应用程序定义并发送到着色器

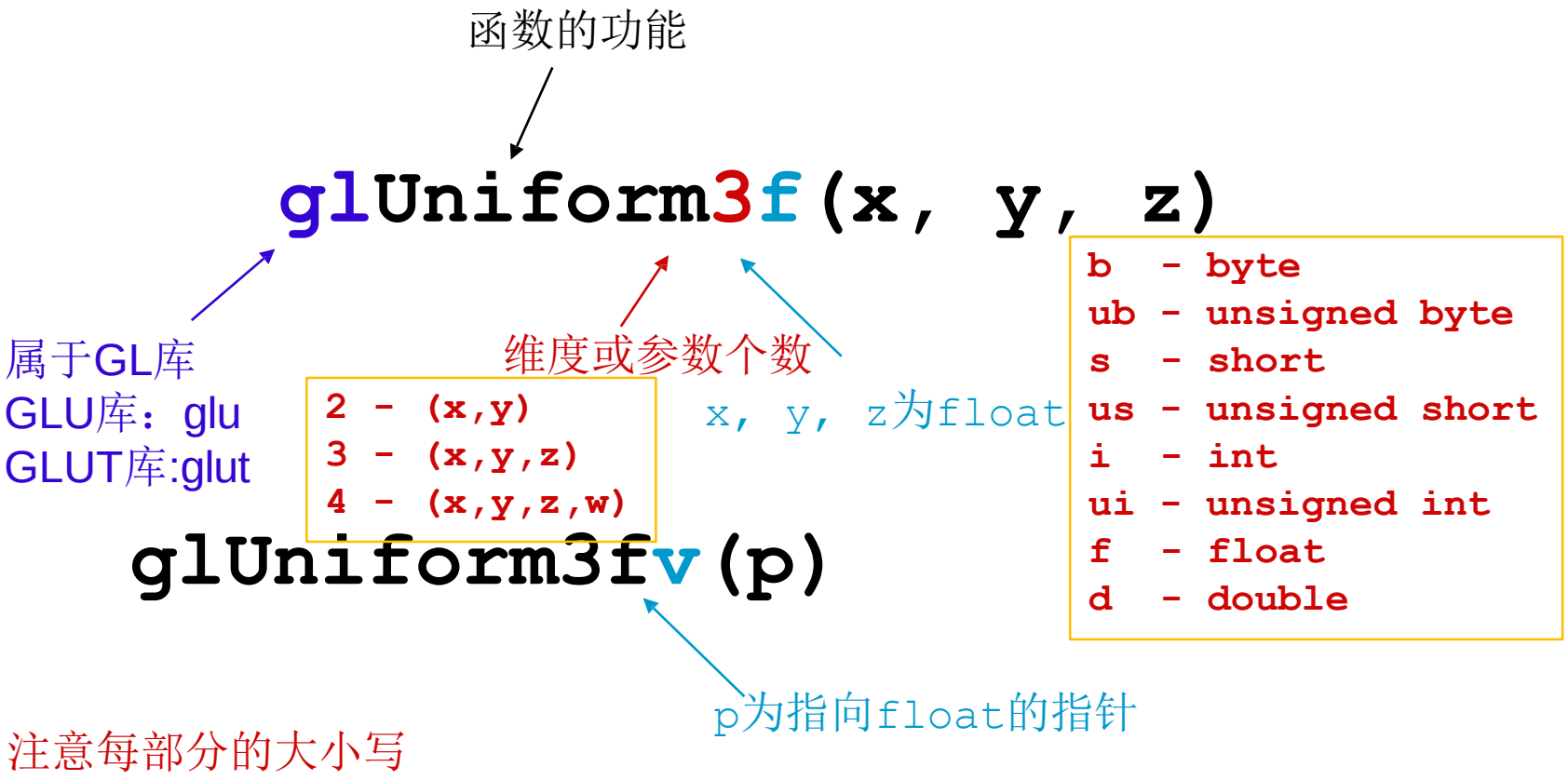
面向对象方面的缺陷

- OpenGL不是面向对象的，因此逻辑上的一个函数却对应着多个OpenGL函数:

  ```
  glUniform3f
  glUniform2i
  glUniform3dv
  ```

- 内在存储模式是相同的
- 在C++中很容易创建重载函数，但效率却成为主要问题

OpenGL函数名称的格式



OpenGL常量和数据类型

- 头文件gl.h, glu.h和glut.h中定义大量的常量
  - 例如：**`glEnable(GL_DEPTH_TEST)`**
  - **`glClear(GL_COLOR_BUFFER_BIT)`**
  - 注意：#include <GL/glut.h>自动将其他两个头文件包含到程序中
- 头文件中定义了OpenGL数据类型：Glfloat, Gldouble,…

OpenGL和GLSL

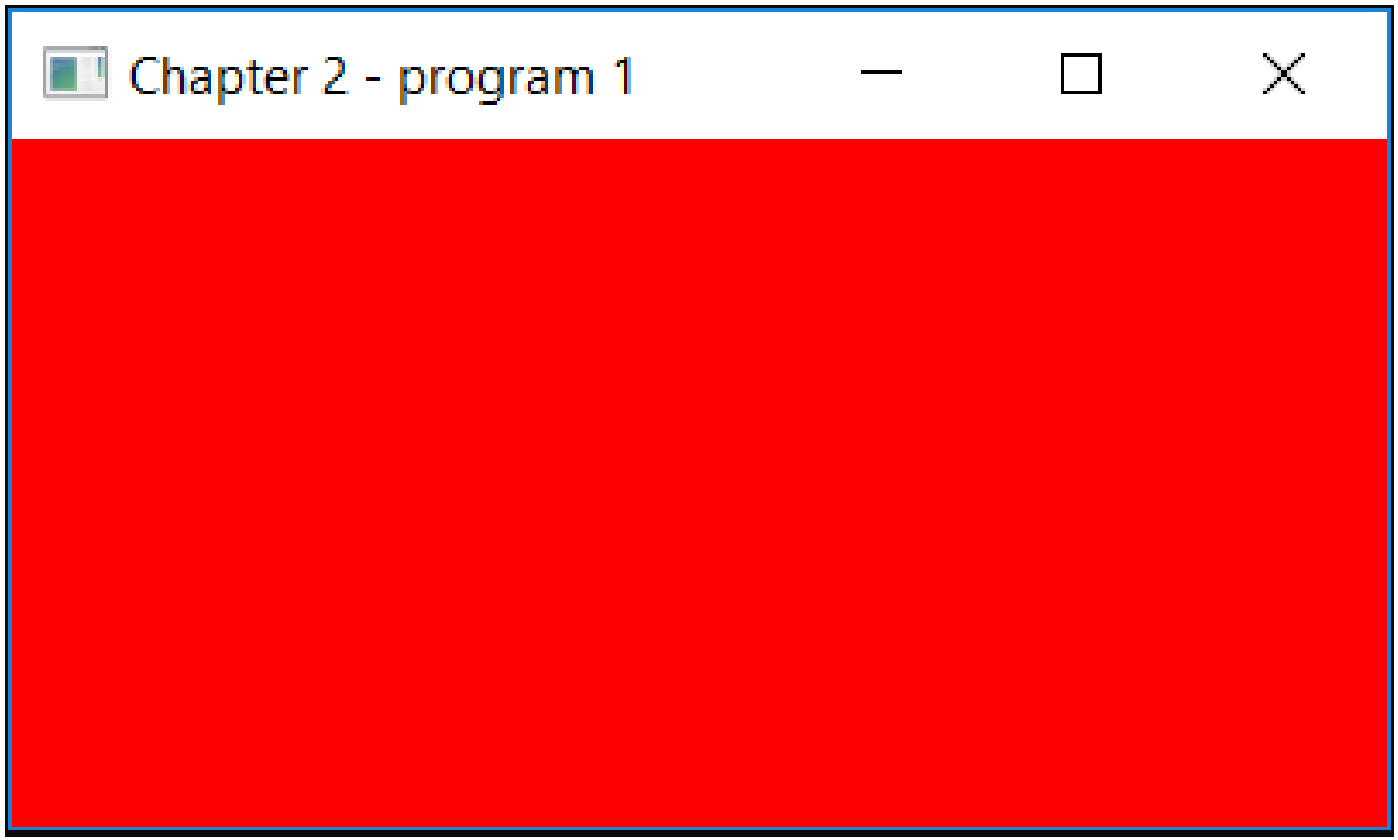
- 基于着色器的OpenGL与其说是状态机模型，不如说是数据流模型
- 大部分状态变量、属性和相关的3.1前的OpenGL函数被弃用
- 由着色器负责渲染任务
- 应用程序只需把数据传输到GPU

GLSL

- OpenGL Shading Language： OpenGL着色语言
- 类C语言
  - 2到4维的矩阵和向量类型
  - 重载的运算符
  - 类C++的构造函数
- 类似于Nvidia的Cg和Microsoft的HLSL
- 代码以源代码的形式发送到着色器
- 通过OpenGL函数编译、链接和发送信息到着色器

简单的C++/OpenGL应用程序

- 红色的背景，没有画任何图形



简单的C++/OpenGL应用程序

```
void init(GLFWwindow* window) { }

void display(GLFWwindow* window, double currentTime)  {
    glClearColor(1.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}

int main(void)  {
    if (!glfwInit()) { exit(EXIT_FAILURE); }
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    GLFWwindow* window = glfwCreateWindow(600, 600, "Chapter2 - program1", NULL, NULL);
    glfwMakeContextCurrent(window);
    if (glewInit() != GLEW_OK) { exit(EXIT_FAILURE); }
    glfwSwapInterval(1);

    init(window);

    while (!glfwWindowShouldClose(window)) {
        display(window, glfwGetTime());
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

(#includes and namespace not shown)

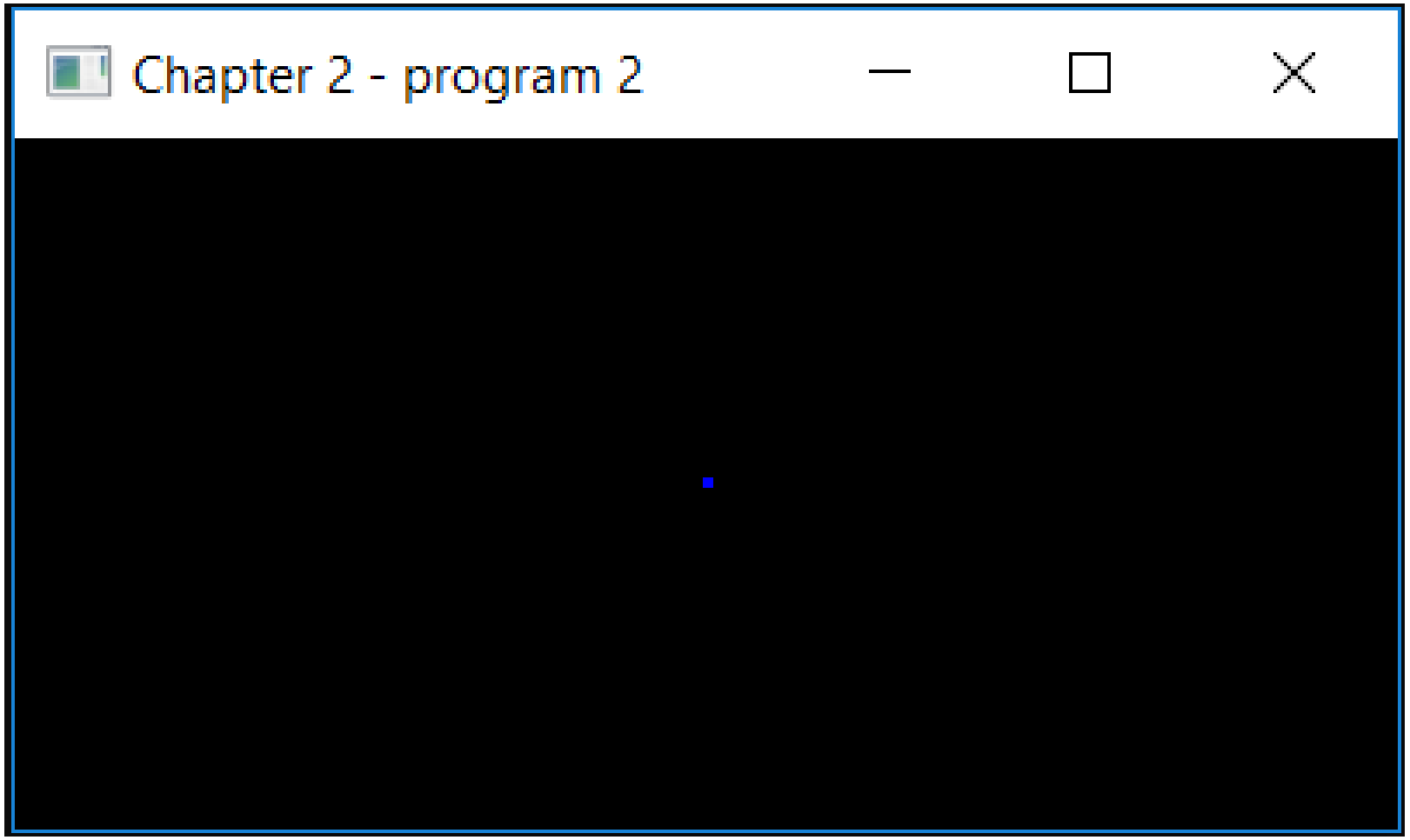
添加顶点着色器和片段着色器

```
#define numVAOs 1
GLuint renderingProgram;
GLuint vao[numVAOs];

void display(GLFWwindow* window, double currentTime)  {
      glUseProgram(renderingProgram);
      glDrawArrays(GL_POINTS, 0, 1);
}

void init(GLFWwindow* window)  {
      renderingProgram = createShaderProgram();
      glGenVertexArrays(numVAOs, vao);
      glBindVertexArray(vao[0]);
}
```

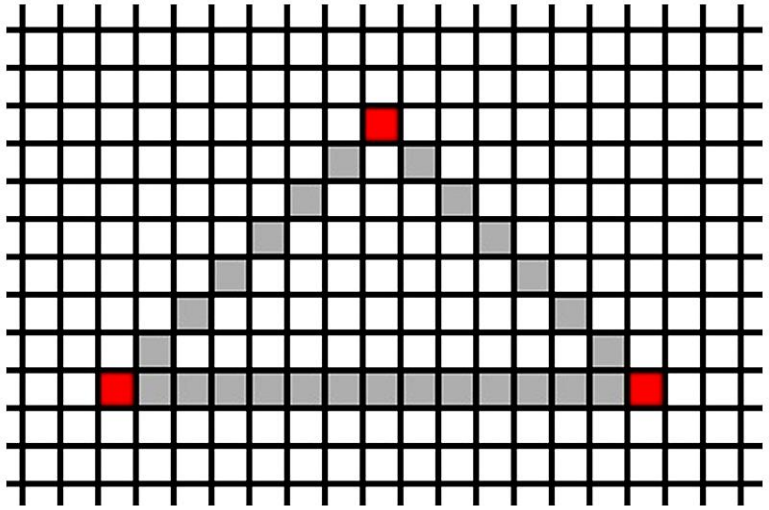
(continued)

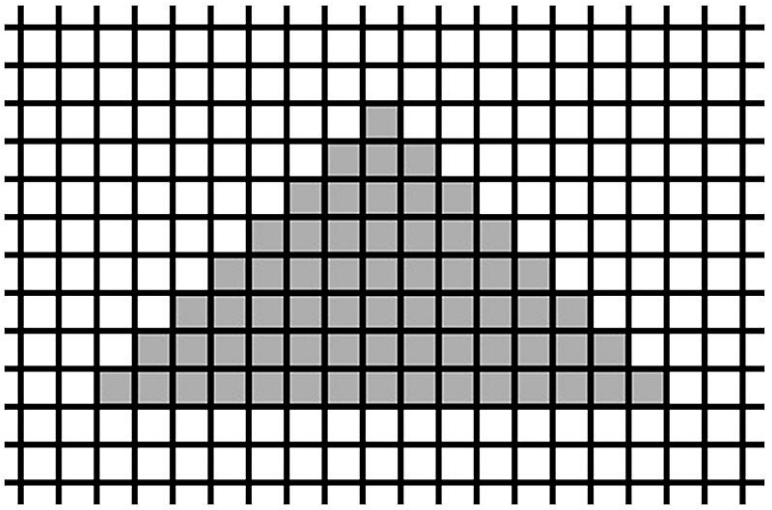
```
GLuint createShaderProgram()  {
    const char *vshaderSource =
        "#version 430    \n"
        "void main(void) \n"
        "{ gl_Position = vec4(0.0, 0.0, 0.0, 1.0); }";
    const char *fshaderSource =
        "#version 430    \n"
        "out vec4 color; \n"
        "void main(void) \n"
        "{ color = vec4(0.0, 0.0, 1.0, 1.0); }";
    GLuint vShader = glCreateShader(GL_VERTEX_SHADER);
    GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(vShader, 1, &vshaderSource, NULL);
    glShaderSource(fShader, 1, &fshaderSource, NULL);
    glCompileShader(vShader);
    glCompileShader(fShader);
    GLuint vfProgram = glCreateProgram();
    glAttachShader(vfProgram, vShader);
    glAttachShader(vfProgram, fShader);
    glLinkProgram(vfProgram);
    return vfProgram;
}
```

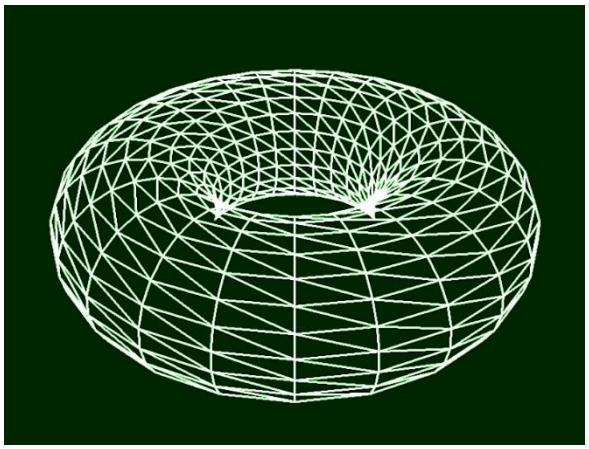
运行效果

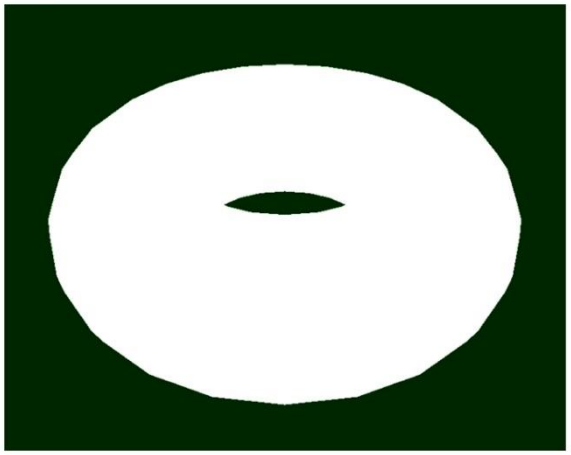


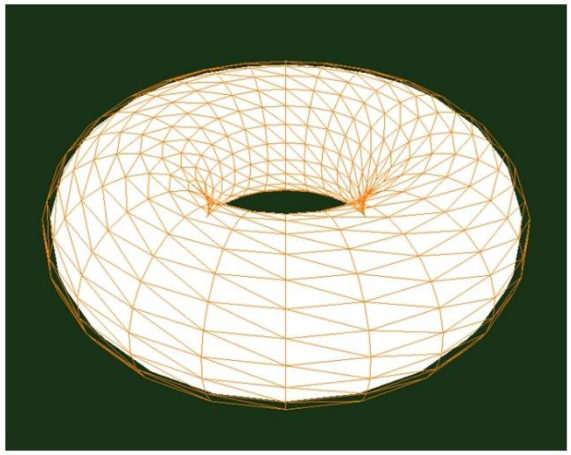
光栅化





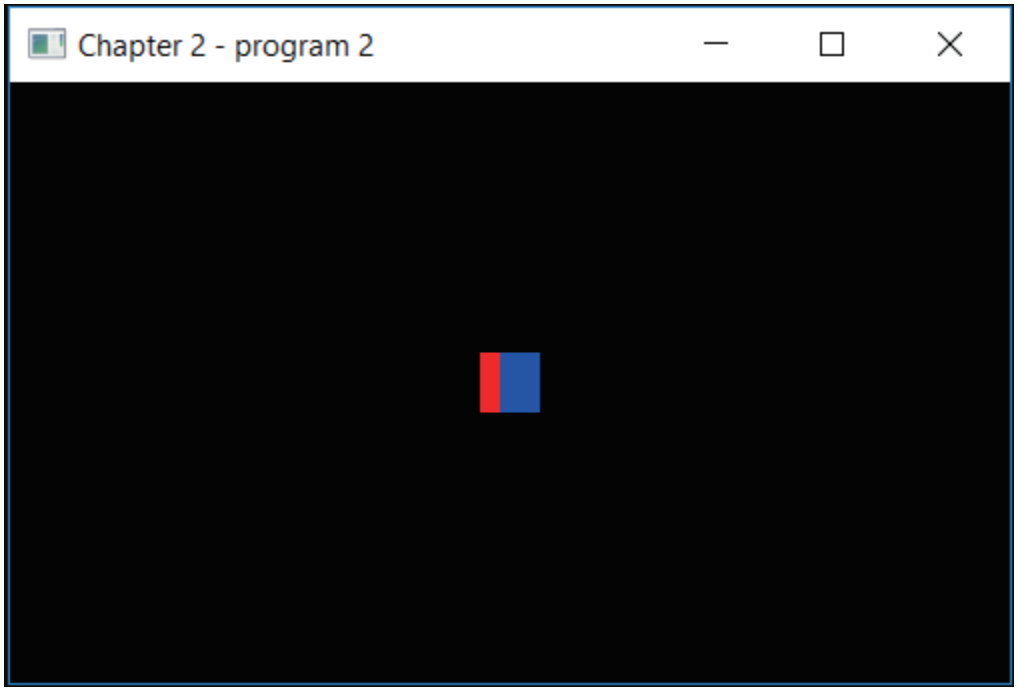






片段着色器

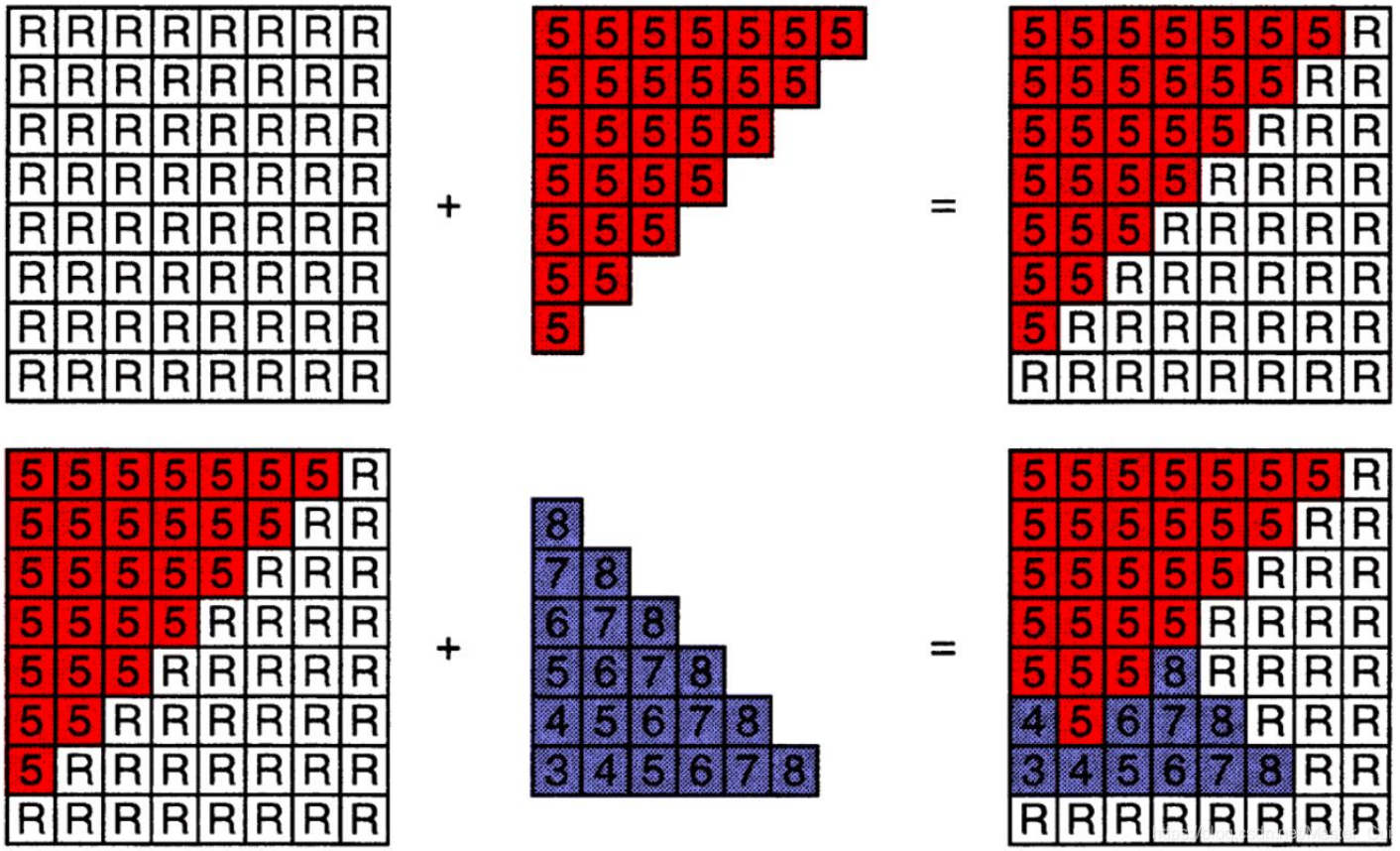
```
#version 430
out vec4 color;
void main(void)
{ if (gl_FragCoord.x < 295) color = vec4(1.0, 0.0, 0.0, 1.0);
  else color = vec4(0.0, 0.0, 1.0, 1.0);
}
```



隐藏面消除（Z缓冲区）

```
Color [ ] [ ] colorBuf = new Color [pixelRows][pixelCols];
double [ ] [ ] depthBuf = new double [pixelRows][pixelCols];

for (each row and column)    // initialize color and depth buffers
{       colorBuf [row][col] = backgroundColor;
        depthBuf [row][col] = far away;
}

for (each shape)       // update buffers when new pixel is closer
{       for (each pixel in the shape)
        {       if (depth at pixel < depthBuf value)
                {       depthBuf [pixel.row][pixel.col] = depth at pixel;
                        colorBuf [pixel.row][pixel.col] = color at pixel;
}       }       }

return colorBuf;
```

隐藏面消除（Z缓冲区）


R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
+
5 5 5 5 5 5 5
5 5 5 5 5 5
5 5 5 5 5
5 5 5 5
5 5 5
5 5
5
=
5 5 5 5 5 5 5 R
5 5 5 5 5 5 R R
5 5 5 5 5 R R R
5 5 5 5 R R R R
5 5 5 R R R R R
5 5 R R R R R R
5 R R R R R R R
R R R R R R R R
5 5 5 5 5 5 5 R
5 5 5 5 5 5 R R
5 5 5 5 5 R R R
5 5 5 5 R R R R
5 5 5 R R R R R
5 5 R R R R R R
5 R R R R R R R
R R R R R R R R
+
8
7 8
6 7 8
5 6 7 8
4 5 6 7 8
3 4 5 6 7 8
=
5 5 5 5 5 5 5 R
5 5 5 5 5 5 R R
5 5 5 5 5 R R R
5 5 5 5 R R R R
5 5 5 8 R R R R
4 5 6 7 8 R R R
3 4 5 6 7 8 R R
R R R R R R R R

从顶点构建对象

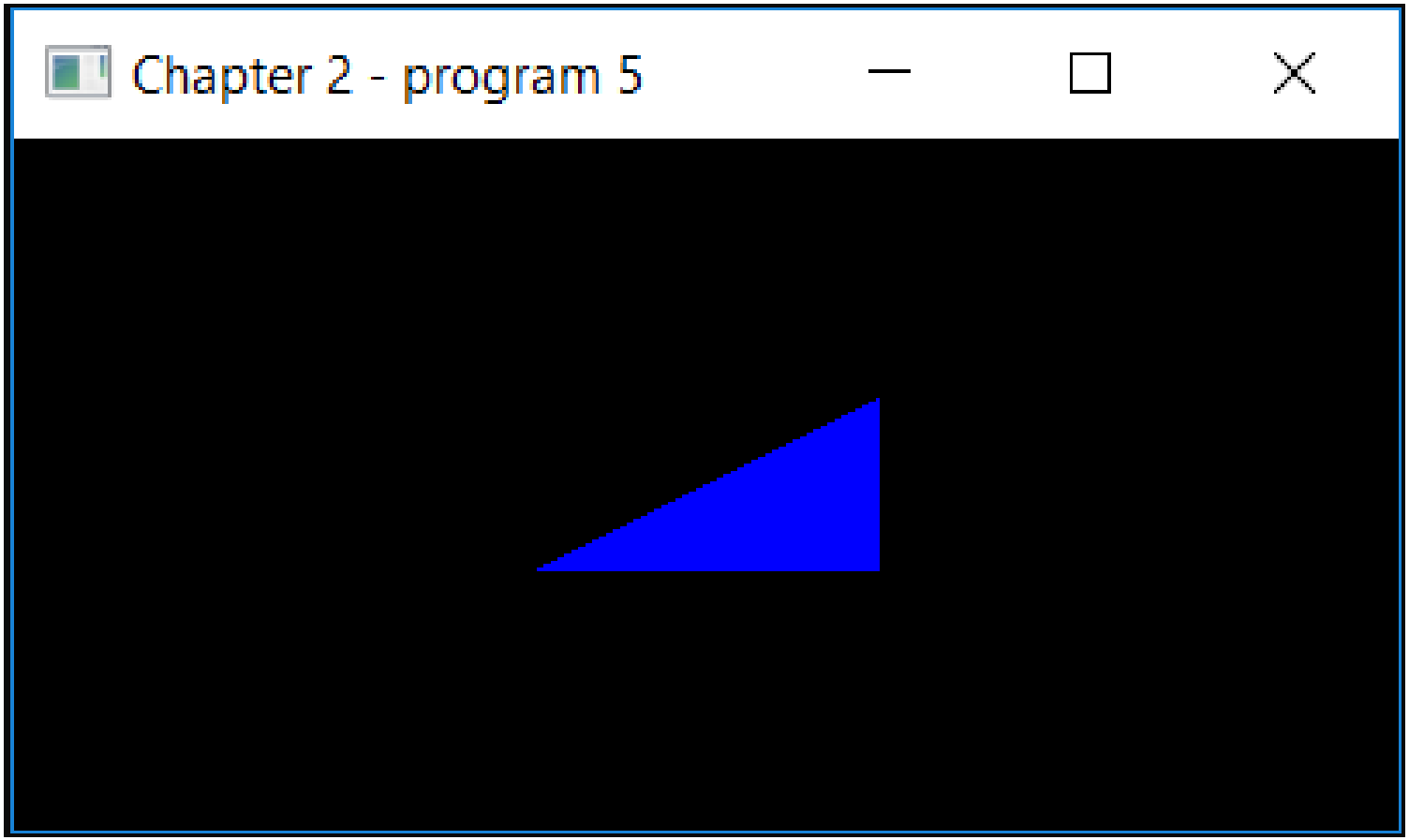
Vertex Shader

```
#version 430
void main(void)
{       if (gl_VertexID == 0) gl_Position = vec4( 0.25, -0.25, 0.0, 1.0);
        else if (gl_VertexID == 1) gl_Position = vec4(-0.25, -0.25, 0.0, 1.0);
        else gl_Position = vec4( 0.25, 0.25, 0.0, 1.0);
}
```

C++/OpenGL application -- in display()

```
. . .
glDrawArrays(GL_TRIANGLES, 0, 3);
```

从顶点构建对象

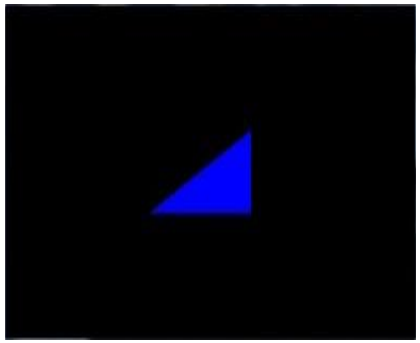
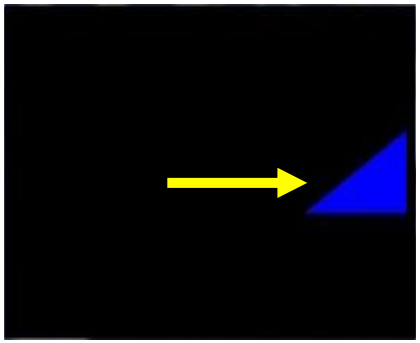
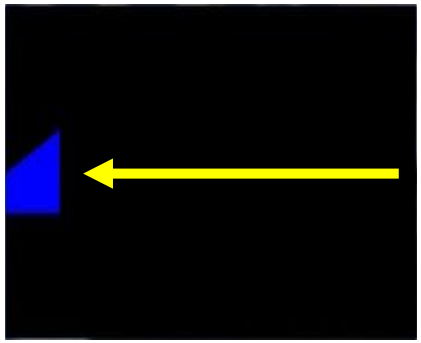


场景动画

- 设计 display()函数来随时间改变要绘制的内容
- 场景的每一次绘制都叫作一帧，调用 display()的频率叫作帧率
-

in Vertex shader:

```
#version 430
uniform float offset;
void main(void)
{    if (gl_VertexID == 0) gl_Position = vec4( 0.25 + offset, -0.25, 0.0, 1.0);
     else if (gl_VertexID == 1) gl_Position = vec4( -0.25 + offset, -0.25, 0.0, 1.0);
     else gl_Position = vec4( 0.25 + offset,  0.25, 0.0, 1.0);
}
```

场景动画

in C++/OpenGL application:

```
. . .
float x = 0.0f;          // location of triangle on x axis
float inc = 0.01f;       // offset for moving the triangle

void display(GLFWwindow* window, double currentTime)  {
        glClear(GL_DEPTH_BUFFER_BIT);
        glClearColor(0.0, 0.0, 0.0, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);       // clear the background to black, each time
        glUseProgram(renderingProgram);
        x += inc;     // move the triangle along x axis
        if (x > 1.0f) inc = -0.01f;         // switch to moving the triangle to the left
        if (x < -1.0f) inc = 0.01f;         // switch to moving the triangle to the right
        // retrieve pointer to "offset"
        GLuint offsetLoc = glGetUniformLocation(renderingProgram, "offset");
        glProgramUniform1f(renderingProgram, offsetLoc, x);      // send value in "x" to "offset"
        glDrawArrays(GL_TRIANGLES,0,3);
}
```

C++代码文件结构

- GLSL着色器代码放在 vertShader.glsl 和 fragShader.glsl
- 主要的 C++/OpenGL 文件总是叫作 main.cpp
- 程序模块化，比如在单独的文件（ Sphere.cpp 和 Sphere.h）中定义Shpere类
- 需要重复使用的函数放在Utils.cpp（与 Utils.h 关联）
- Utils.cpp 文件中的函数都以静态函数实现，因此我们不需要实例化 Utils 类