

简单的pipeline，vertex shader给vertices定位，然后光栅化//面积小格子，fragment shader上色

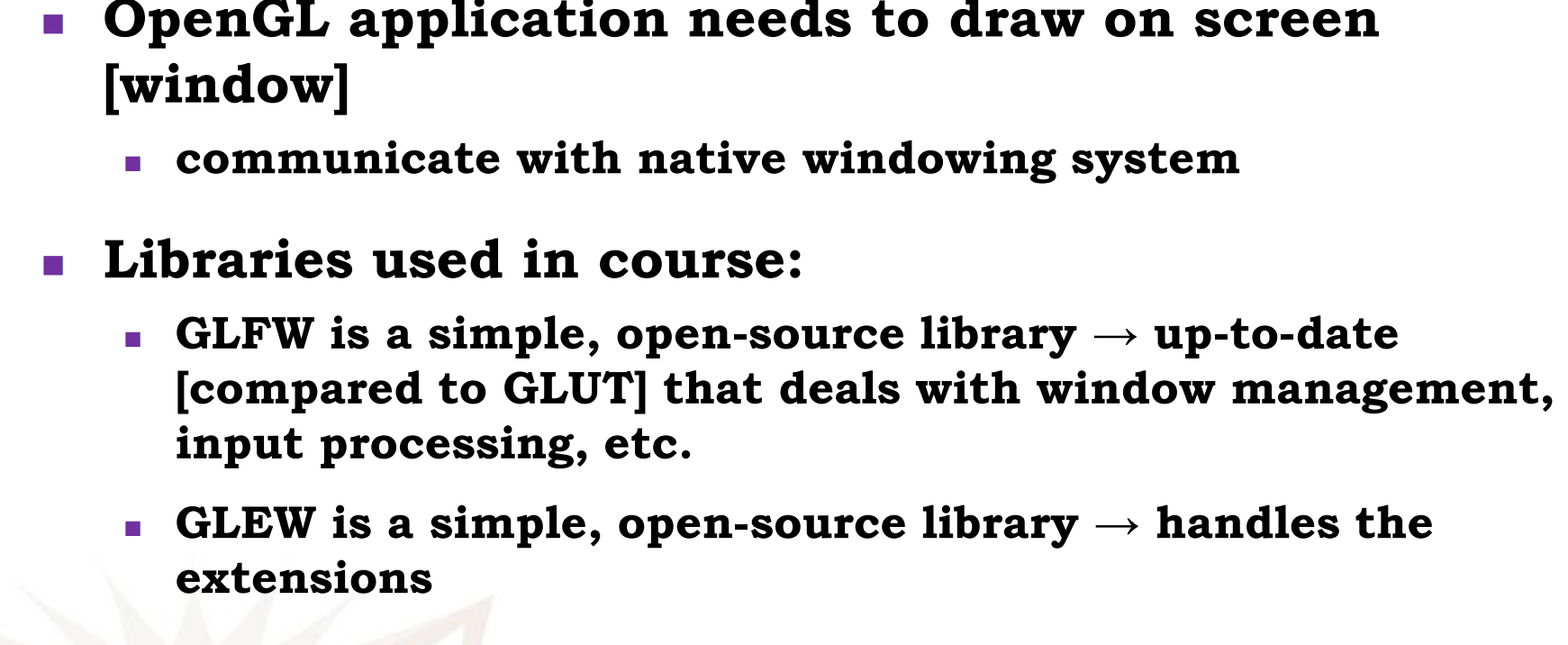


流程

创造shader program

创造buffer来存储数据，并且bind他们并传输到GPU

Shader读取这些数据，渲染



OPENGL需要创造个窗口来画图

Glfw管理window，输入

GLEW，管理extension

基本的data type:primitive GL\_POINTS:GL\_LINES….

他们由vertices构成



Vertices:有position等属性组成的端点

被存在vertex buffer objects 里vbo

Vbo被存在vao vertex array object里

Input and interaction 输入与交互

CLIENT-SERVER MODEL

Server为client服务

一个工作站，如果有着显示器键盘鼠标这种，叫做graphic server

Opengl 程序就是使用这个graphic server的client

Server为client提供了output/input device

Input device

Input device必须具备一个trigger,来给操作系统发出信号，证明他有Input

Mouse的button，键盘的按键

当triggered，input device return相对信息

例如鼠标提供位置信息，键盘提供ASCII码

OPENGL可以通过三种mode来得到数据

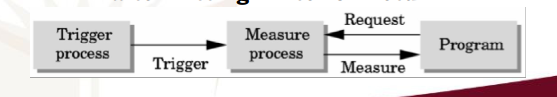
1. request mode
2. sample mode
3. event mode

request mode 请求方式

只有用户trigger的时候才有input

例如java里的键盘输入

你可以一直输入，直到按下enter键,，input设备会把信息传递给program



Sample Mode (Polling)

Input时及时的

不需要trigger

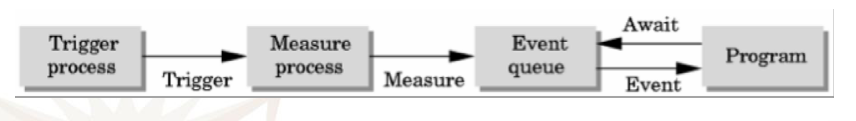
有一点像脚本

User必须指定哪个device提供Input,图形应用很难使用

Event mode

绝大部分程序都有多个input device，他们都可以任意时间被trigger

每一个trigger生成一个event，并被放到event queue里，由User program检验



例如，改变window的大小，鼠标点击，鼠标移动，键盘按任意键

总结，绝大多数都是event mode，一小部分需要用enter等特殊关键字的是request mode

Callbacks 回收

Callbacks是对event-driven 的input的接口

程序员需要对每种event type描述一个callback function

然后这个event发生时，这个user-supplied function就会被运行

例如定义keyCallback

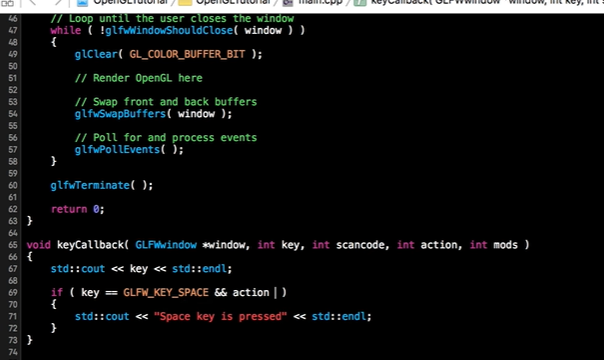
先在window之后使用默认的keycallbackfunction来激活这个东西



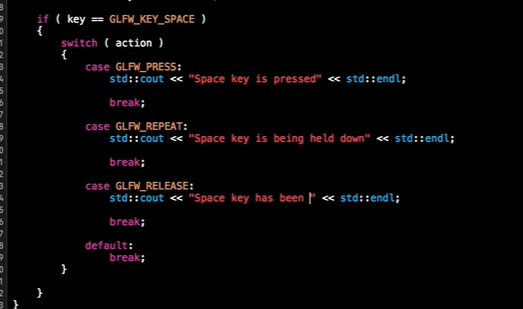
在while之后定义这个keycallback



最好多写一个这个

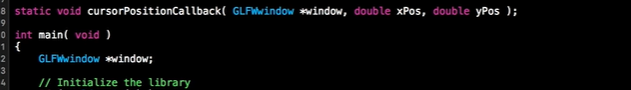


对一个key的按下，不放，release做分别动作



鼠标callback

在main之前定义一个static function





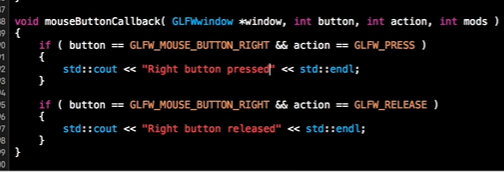
然后window后用这个东西



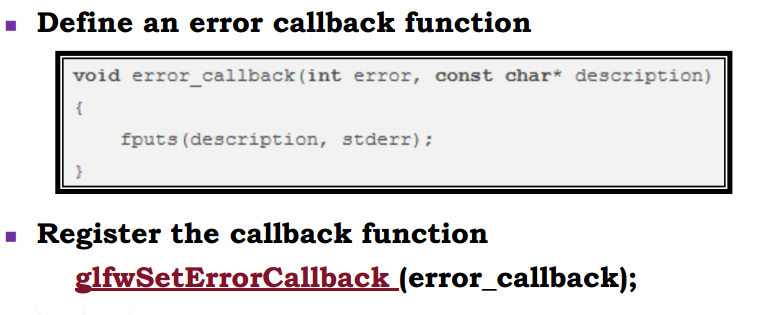
还是在while return0之后描述

鼠标点击version





控制error callback



类型set，不用window

GLFW MAIN EVENT LOOP

就是那个while的东西，刷一次等于1FPS，

Main loop会处理event queue里的events

处理events会让window与Input callback 和那些输入event连接起来

Void glfwPollEvents（void）只处理那些已经在event queue内的那些event并且瞬间return

Double Buffering

当我们不停的在显示屏上重复drawing的时候我们必须让刷新频率很快，这样才能自然

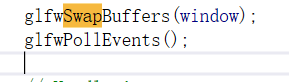
解决方法就是用两个不同的frame buffer//帧缓冲器

Front:总是显示的buffer

Back:我们正在画的buffer

这个buffer只有外部function call的时候才生效





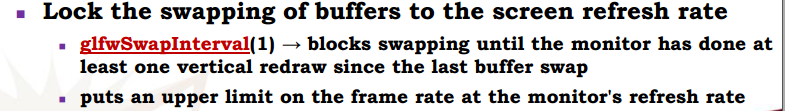
实际上是必备的策略

GPU虽然能高速渲染，但是只有一个framebuffer会导致中断，再画，中断，再画，而double buffering会让前一个buffer显示，第二个buffer缓存，然后交换，显示第二个buffer改好的内容，第一个buffer成了back buffer，无限循环

GPU虽然能高速渲染，但如果我们描述一个高速旋转的立方体，会一片糊，因为没法在一个buffer上不停渲染

因此我们要用double frame buffer

并且我们还要用timing mechanisms来人为制造延迟



锁定刷新帧率，只有monitor重新画好一个vertice才交换

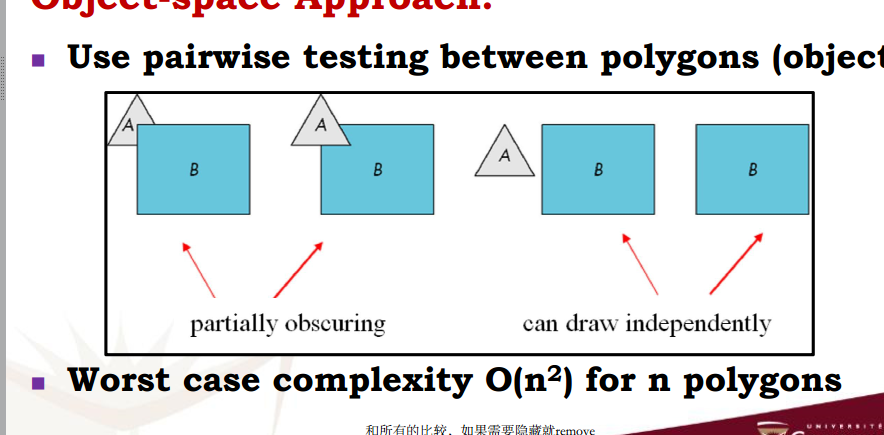
Hidden Surface Removal

目标：去除那些viewer看不到的画面

分为两大类

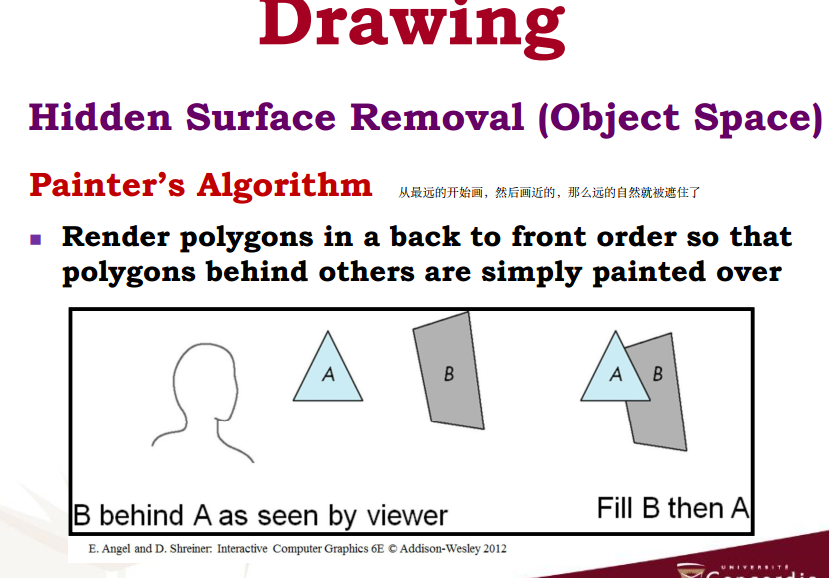
Object-space algorithm与Image-space algorithm

1. pairwise testing,把每一个polygon一对对分别比较，然后排哪个在前面哪个在后面，这个是最差的，复杂度是On^2



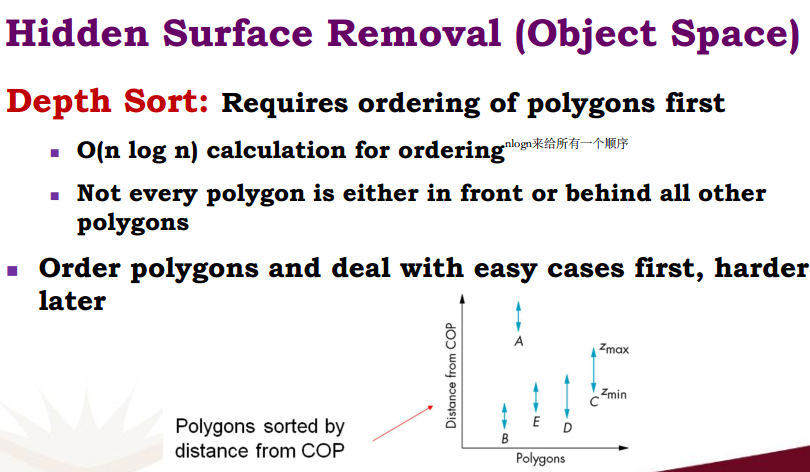
­­­­­­­­Painter algorithm,

直接从最远的画到最近的，最近的覆盖



Painter algorithm建立在depth sort的基础上

因为你要对所有的polygon先排序

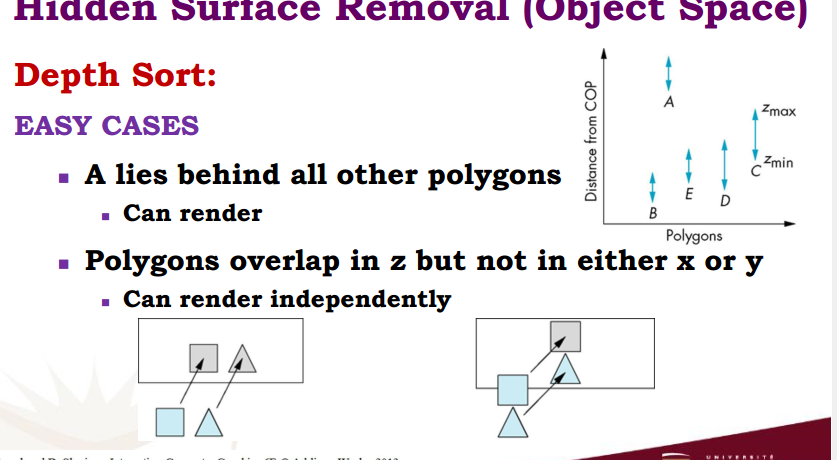


复杂度是nlogn

而且问题在于不是每一个polygon都正好就完全整个在另一个Polygon前面

解决方法，先处理简单的，在处理复杂状况

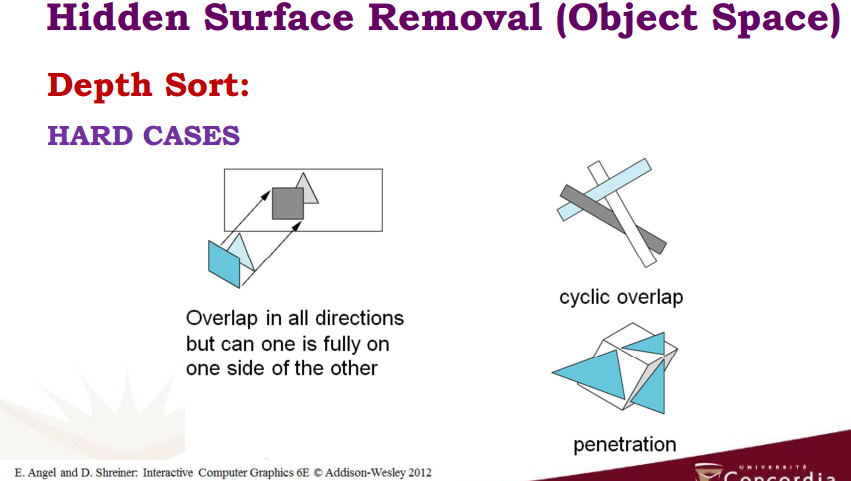
简单的，



在最后面的直接渲染，或者那些z轴上有重叠但是XY没重叠的，就可以直接画

Opengl里xy就是二维坐标轴，z是斜着的，

复杂的，



整个都盖住了，但另一角度还能看到

圆柱overlap，

Penetration渗透

Painter算法的好处在于

简单

可以很好的处理transparency透明度

特殊情况下你都不用排序（高度场height field）

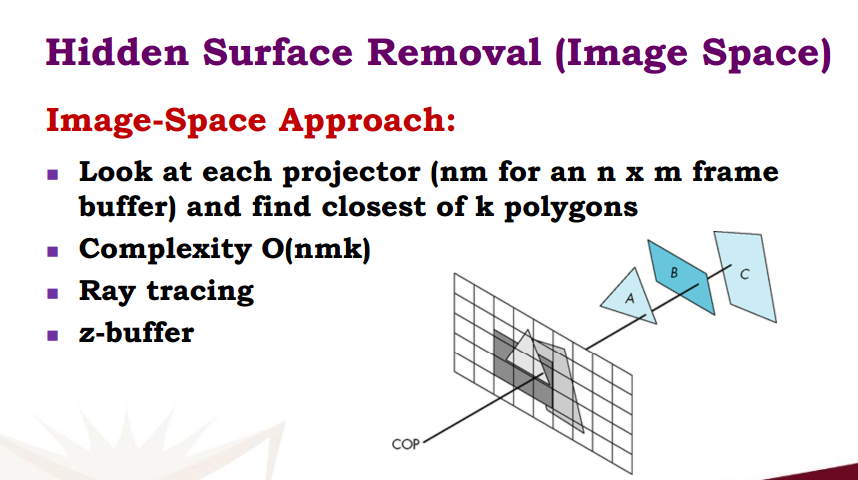
缺陷

无法处理复杂图形

没法sort

Opengl没有Implement,必须要user自己Implement

Image space:z buffer algorithm



把所有投影都投到一个视角平面上，然后再找到最近的polygon

复杂度Nmk

用一个叫做depth buffer(z buffer)来存储最近的（格子上画出来的pixel）的depth

在渲染每个polygon

好处

简单，不用sort

独立于图形

坏处

要大量memory

处理不了透明

Opengl里有zbuffer

当我们创造window的时候申请一个depth buffer,这一步实际可以跳过（默认的）



在Initialization routine,并要在内容创建之后，（换句话说要在while前面一行）

每一次你都要clear他



