中国科学技术大学
University of Science and Technology of China

计算机图形学

计算机学院 黄章进
zhuang@ustc.edu.cn

3.1 OpenGL简介

3.2 完整的程序

3.3 三维图形程序

3.1 OpenGL简介

中国科学技术大学
University of Science and Technology of China

- 3.1.1 图形 API的发展
- 3.1.2 OpenGL的体系结构
- 3.1.3 OpenGL的函数
- 3.1.4 一个简单例子
- 3.1.5 安装编译说明

GKS(1980s)

- IFIPS (1973) 组织了两个委员会建立图形API的标准
  - GKS(Graphical Kernel System)
    - 二维，同时包含很好的工作站模型
  - Core: 同时应用于二维和三维
  - GKS成为ISO标准，稍后成为ANSI标准(1980s)
- GKS很难推广到三维（GKS-3D)
  - 远远落后于硬件的发展

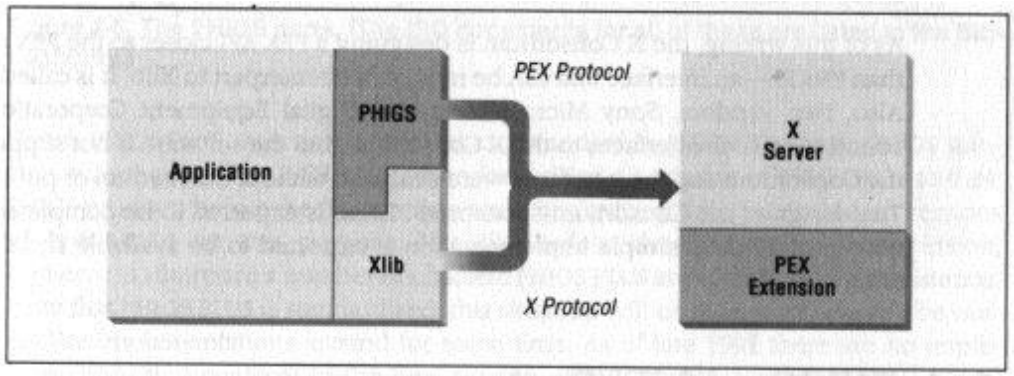
IFIPS: International Federation of Information Processing Societies

PHIGS（1990s）

- 程序员层次交互式图形系统 (Programmers Hierarchical Interactive Graphics System)
  - 来自于CAD团体的3D图形API
  - 保留模式：绘制前在数据库里保存场景的层次结构
  - 基本几何图元和网格模型
  - 不支持光照
- PHIGS+
  - NURBS曲面
  - 支持光照，但不支持纹理
- PHIGS和PHIGS+都是ANSI标准和ISO标准

PHIGS与X

- X Window系统（也常称为X11或X）R7.6
  - 1984年 DEC/MIT的雅典娜计划 Data Equipment Company 数据设备公司
  - 图形用户界面(GUI)环境底层框架，硬件无关
  - 客户端-服务器(client-server)架构模型
  - GNONE、KDE
- PEX（ PHIGS Extension to X ）把两者组合
  - 不易应用



IRIS GL

- Silicon Graphics(SGI: 1981-2006)
  - Stanford的Jim Clark博士带领7个研究生创办
  - Geometry Engine（几何引擎）：硬件(VLSI)实现几何流水线，极大改良了图形工作站
- IRIS GL(Integrated Raster Imaging System Graphics Library)
  - 立即模式绘制
  - 可非常简单地设计出三维交互图形应用程序

OpenGL

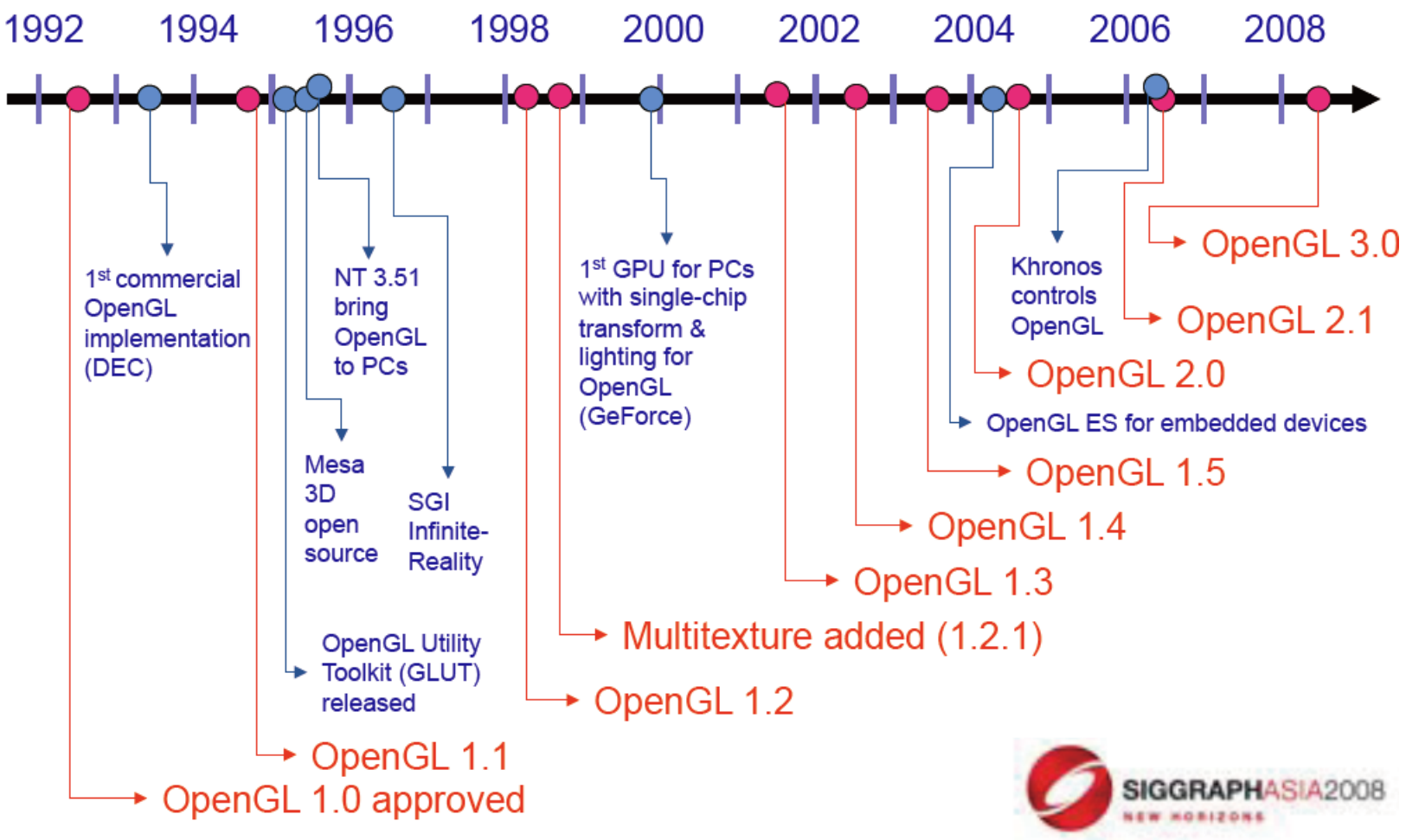
- 1992年 SGI领导的OpenGL Architectural Review Board(OpenGL ARB)发布1.0版
  - 平台无关的API
  - 易于使用
  - 与硬件非常贴近，从而可以充分发挥其性能
  - 着重在于渲染 (rendering)
  - 没有提供窗口和输入接口，从而避免依赖于具体的窗口系统

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)/4.4(2013.7)
  - 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

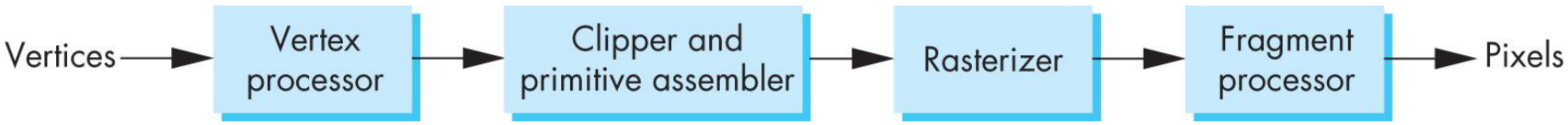
OpenGL的发展



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）
- 状态变量减少
- 函数废弃机制
- 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

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

3.1 OpenGL简介

中国科学技术大学
University of Science and Technology of China

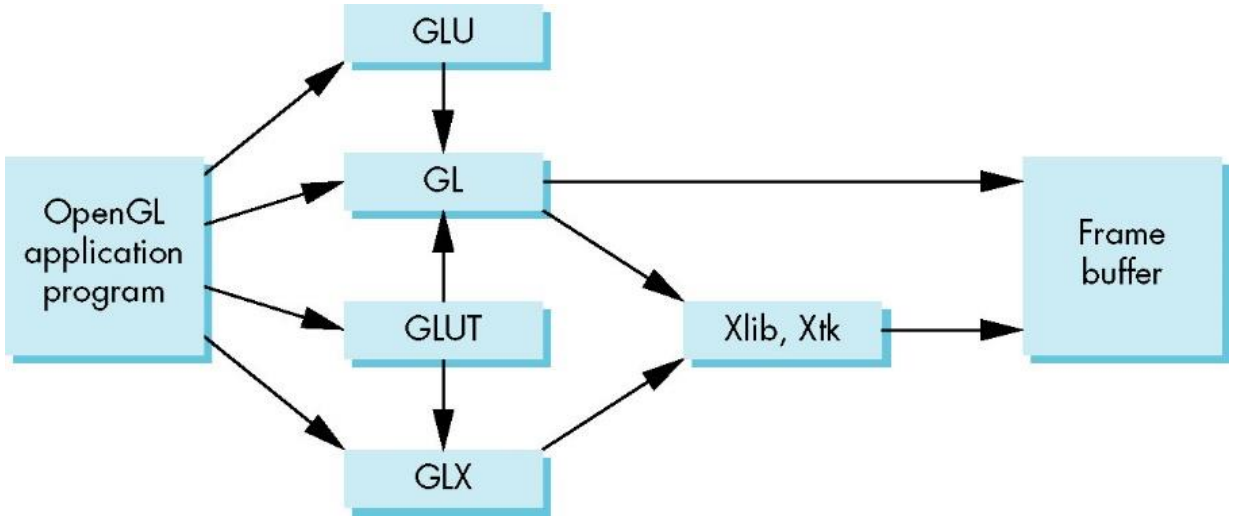
- 3.1.1 图形 API的发展
- 3.1.2 OpenGL的体系结构
- 3.1.3 OpenGL的函数
- 3.1.4 一个简单例子
- 3.1.5 安装编译说明

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核心库提供一些功能，避免重复编写代码
  - 二次曲面、NURBS、多边形网格化等

OpenGL库

- 与窗口系统的连接
  - X Window系统：GLX
  - Windows: WGL
  - Macintosh: AGL
  - 粘合OpenGL和窗口系统

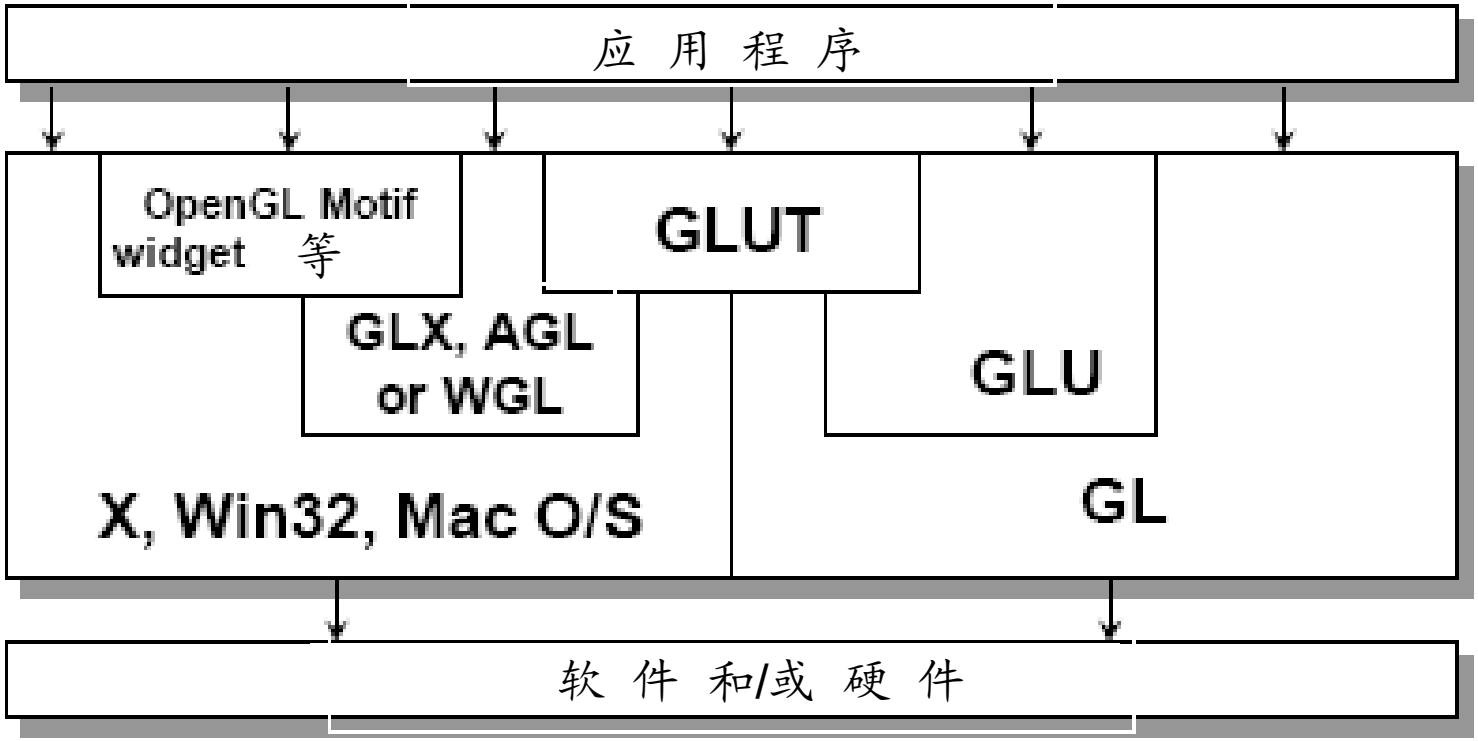


GLUT

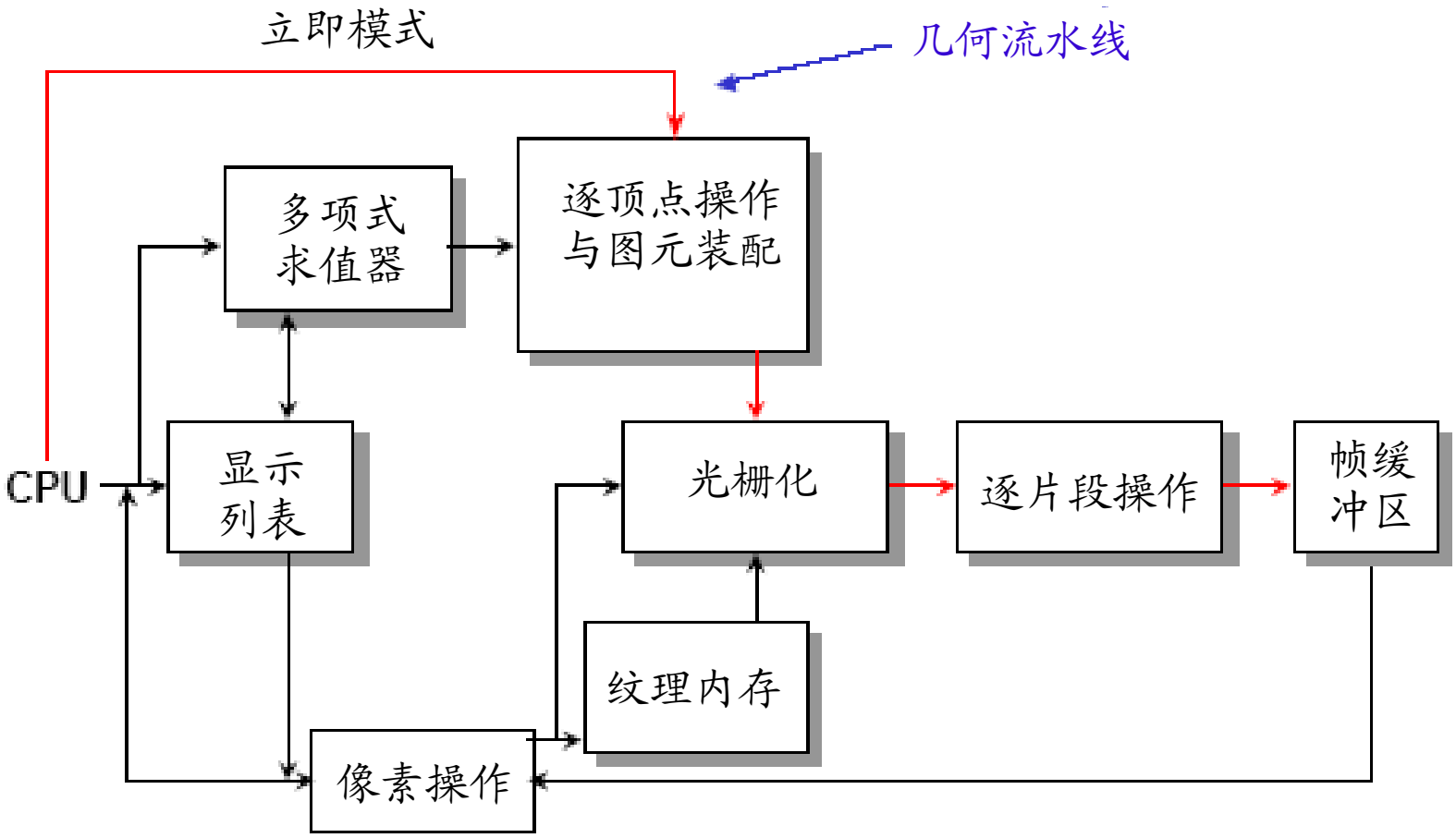
- OpenGL实用工具库（OpenGL Utility Toolkit Library, GLUT）
  - 提供所有窗口系统的共同功能
    - 创建窗口
    - 从鼠标和键盘获取输入
    - 菜单
    - 事件驱动
- 代码可以在平台间移植，但是GLUT缺乏一些现代GUI的控件和功能
  - 无滚动条
  - 可用FLTK、SDL

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

软件组织


应 用 程 序
OpenGL Motif widget 等
GLUT
GLX, AGL or WGL
GLU
X, Win32, Mac O/S
GL
软 件 和/或 硬 件

OpenGL流水线架构



3.1 OpenGL简介

- 3.1.1 图形 API的发展
- 3.1.2 OpenGL的体系结构
- 3.1.3 OpenGL的函数
- 3.1.4 一个简单例子
- 3.1.5 安装编译说明

OpenGL的函数

- 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) GLUT
  - 处理来自键盘、鼠标等设备的输入
- 控制函数(control) GLUT
  - 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
    - 顶点如何被处理以及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以后版本，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:

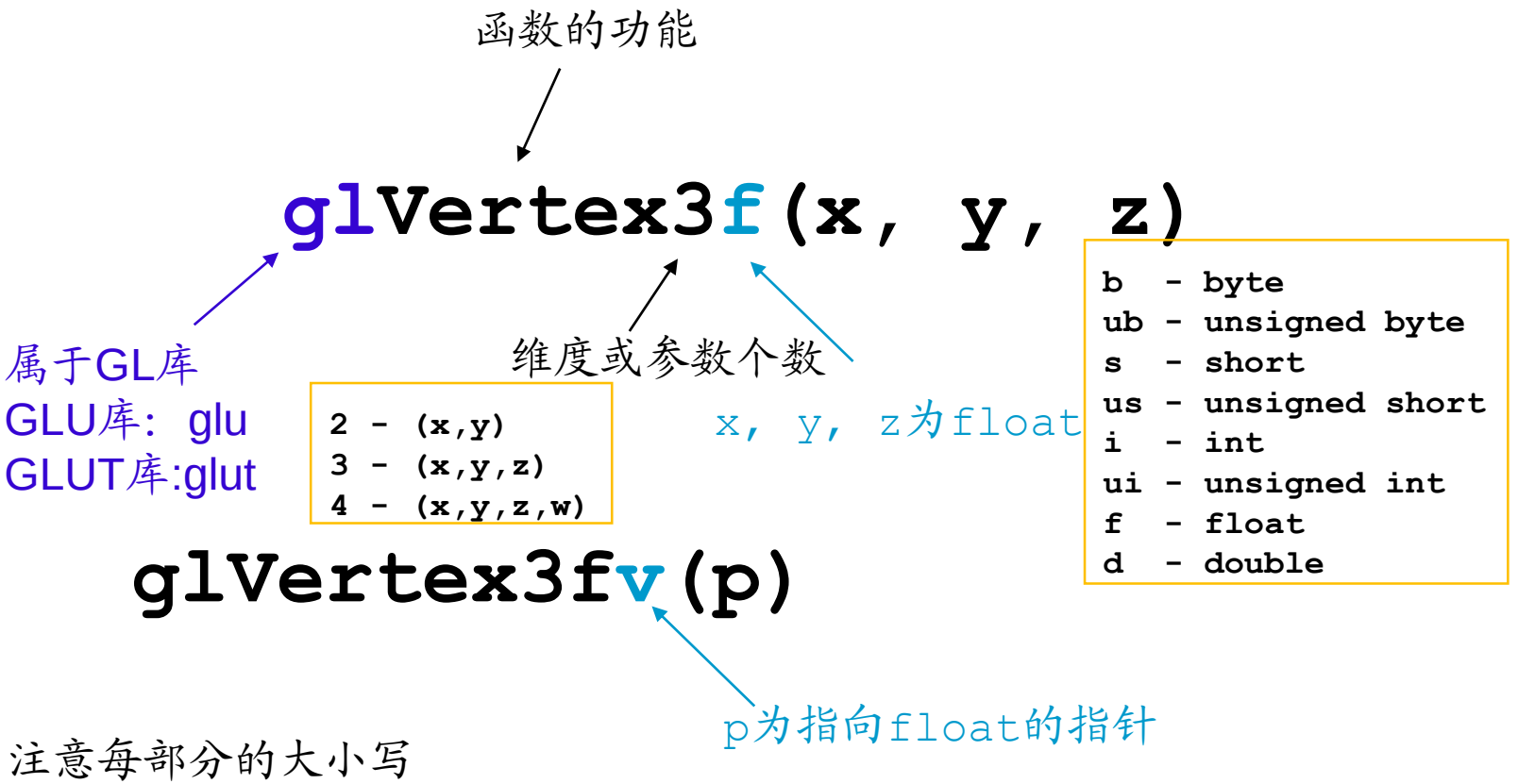
  glVertex3f

  glVertex2i

  glVertex3dv

- 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);
```

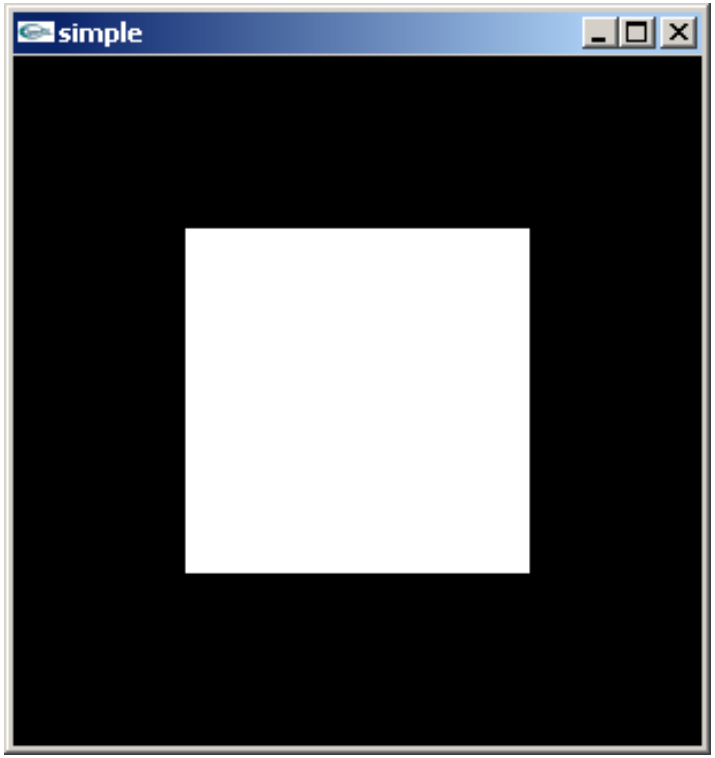
  - 注意：#include <GL/glut.h>自动将其他两个头文件包含到程序中
- 头文件中定义了OpenGL数据类型: GLfloat, GLdouble,…

3.1 OpenGL简介

- 3.1.1 图形 API的发展
- 3.1.2 OpenGL的体系结构
- 3.1.3 OpenGL的函数
- 3.1.4 一个简单例子
- 3.1.5 安装编译说明

一个简单程序

- 在黑色背景上画一个白色矩形



simple.c


```
#include <GL/glut.h>
void display(){
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_QUAD);
            glVertex2d(-0.5, -0.5);
            glVertex2d(-0.5, 0.5);
            glVertex2d(0.5, 0.5);
            glVertex2d(0.5, -0.5);
      glEnd();
      glFlush();
}
int main(int argc, char* argv[]) {
      glutInit(&argc, argv);
      glutCreateWindow("simple");
      glutDisplayFunc(display);
      glutMainLoop();
}
```

函数

- 初始化GLUT

```
void glutInit(int * argc, char ** argv)
```

- 创建窗口：窗口标题为title

```
int glutCreateWindow(char * title)
```

- 注册显示回调函数：窗口需要重绘时调用func

```
void glutDisplayFunc(void (*func)(void))
```

- 事件处理循环：main函数最后一条语句

```
void glutMainLoop()
```

- 清空帧缓冲区

```
void glClear(GLbitfield mask)
```

- 图元定义：mode可取GL_POINTS、GL_LINES、GL_POLYGON等

```
void glBegin(Glenum mode) // 开始mode型图元定义
void glEnd()              // 结束顶点序列
```

- 强制执行OpenGL命令

```
void flush()
```

事件循环

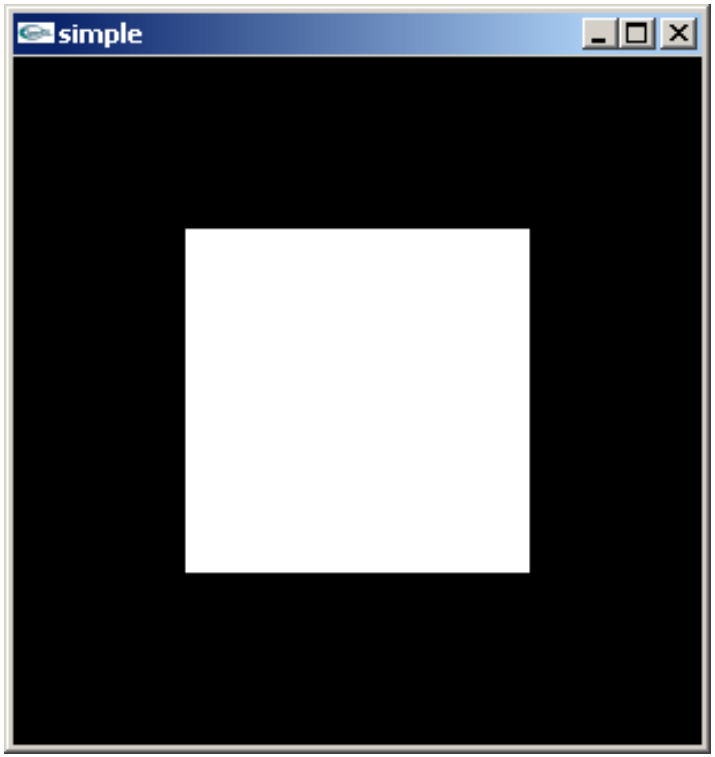
- 注意在程序中定义了一个显示回调函数(display callback):

  void display()

  – 每个GLUT程序都必须有一个显示回调函数
  – 只要OpenGL确定显示内容要被刷新时，显示回调函数就会被调用：例如，当窗口被打开的时候
  – main函数是以程序进入事件循环做为结束

默认值

- simple.c非常简单
- 大量使用状态变量的默认值
  - 视图
  - 颜色
  - 窗口参数
- 以后的程序将直接改变一些默认值



3.1 OpenGL简介

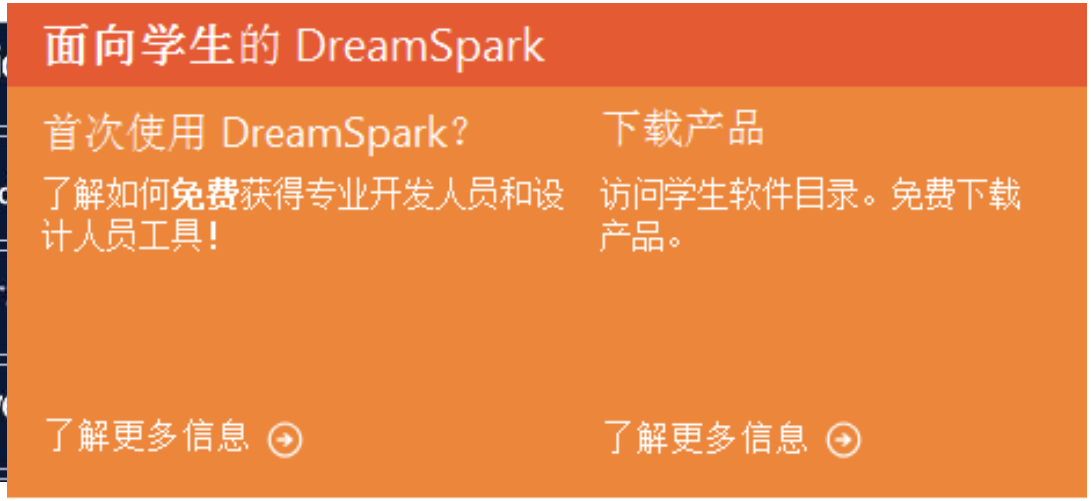
- 3.1.1 图形 API的发展
- 3.1.2 OpenGL的体系结构
- 3.1.3 OpenGL的函数
- 3.1.4 一个简单例子
- 3.1.5 安装编译说明

安装编译说明

- Windows 7 & 8，Office 2007/2010/2013下载 http://ms.ustc.edu.cn/zbh.php

安装编译说明

- Visual Studio 2005/2008/2010/2012 下载
  - http://dreamspark.eol.cn/
  - https://www.dreamspark.com/
- OpenGL+GLUT配置编译见课程主页



安装编译说明

- Windows 7 64位 + Visual Studio 2010 + GLUT：
  - 把GLUT头文件glut.h复制到Windows SDK的Include\gl子目录 C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\include\gl
  - 把GLUT库文件glut32.lib复制到Windows SDK的Lib子目录 C:\Program Files (x86) \Microsoft SDKs\Windows\v7.0A\Lib
  - 把GLUT的dll文件glut32.dll复制到系统目录 C:\Windows\SysWOW64

安装编译说明

- Visual Studio 2010 + GLUT :
  - 新建一个Win32控制台应用程序(Win32 Console Application)，指定目录位置(Location)和项目名称(Name)，点击确认(OK)
  - 在Win32应用程序向导(Win32 Application Wizard)中的应用程序设置(Application Settings)里选择空项目(Empty project)，点击完成(Finish)
  - 新建或添加C/C++文件
  - 更改项目属性，添加OpenGL库文件。在 配置属性 -> 链接器 -> 输入 -> 附加依赖项 (Configuration Properties -> Linker -> Input -> Additional Dependencies)里填入 opengl32.lib glu32.lib glut32.lib（这里用空格分隔）
  - 生成(Bulid)并执行程序

3.1 OpenGL简介

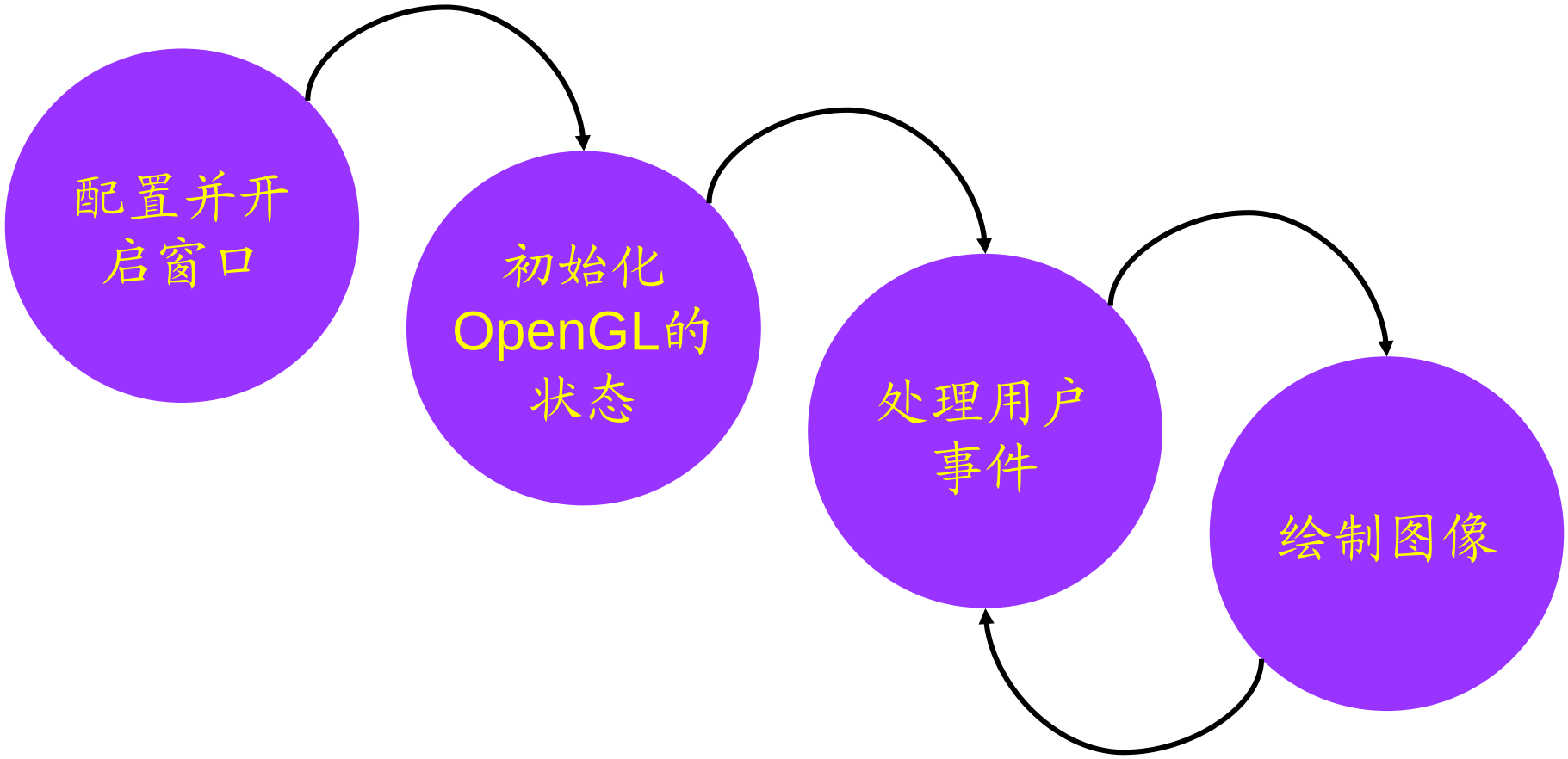
3.2 完整的程序

3.3 三维图形程序

3.2 完整的程序

- 3.2.1 程序结构
- 3.2.2 控制函数
- 3.2.3 视图
- 3.2.4 图元
- 3.2.5 属性

OpenGL程序的一般结构



程序结构

- 绝大多数OpenGL程序具有类似的结构，包含下述函数
  - main():
    - 定义回调函数
    - 打开一个或多个具有指定属性的窗口
    - 进入事件循环（最后一条可执行语句）
  - init(): 设置状态变量
    - 视图
    - 属性
  - 回调
    - 显示函数 display()
    - 输入和窗口函数

程序结构


```
#include <GL/glut.h>

void init() {…} // 设置OpenGL状态: 照相机参数等
void display() {…} // 显示回调函数
int main(int argc, char** argv)
{
        glutInit(&argc,argv);
        …
        //创建并初始化GLUT窗口;
        glutCreateWindow("OpenGL");
        glutDisplayFunc(display);
        …
        //注册其他回调函数;
        …
        init();
        glutMainLoop();
}
```

这自动包含了gl.h,glu.h

显示回调函数

设置OpenGL状态

进入事件循环

对simple.c进行修改

- 在新版本中，会得到同样的输出，但是常用的具有默认值的相应状态值都通过函数调用显式地指定
- 特别地，设置了
  - 颜色
  - 视图条件
  - 窗口属性

main.c


```
#include <GL/glut.h>

int main(int argc, char** argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(300,300);
        glutInitWindowPosition(0,0);
        glutCreateWindow("简单示例“);
        glutDisplayFunc(display);

        init();

        glutMainLoop();
}
```

这自动包含了gl.h

定义窗口属性

显示回调函数

设置OpenGL状态

进入事件循环

3.2 完整的程序

- 3.2.1 OpenGL程序结构
- 3.2.2 控制函数
- 3.2.3 视图
- 3.2.4 图元
- 3.2.5 属性

控制函数

- 控制函数
  - 与窗口系统通信
  - 初始化程序
  - 处理程序执行期间发生的错误
- GLUT库函数
  - 窗口管理
  - 事件处理循环
  - 回调函数机制

窗口

- 窗口：显示器上的一块矩形区域
- 窗口内的位置用窗口坐标来指定，单位是像素
  - 科学和工程中，左下角是原点(0,0)
  - 光栅显示器按照从上到下，从左到右的顺序进行扫描，所以左上角是原点
  - OpenGL命令假定原点在左下角
  - 窗口系统返回的信息（例如鼠标位置）假定原点在左上角

GLUT窗口管理

- 初始化GLUT，在调用其他GLUT函数前调用

```
void glutInit(int * argc, char ** argv)
```

- 设置窗口的初始宽度和高度，单位为像素，缺省300x300

```
void glutInitWindowSize(int width, int height)
```

- 窗口左上角相对于屏幕左上角的位置，单位为像素，缺省(0,0)

```
void glutInitWindowPosition(int x, int y)
```

- 设置窗口的显示模式

```
void glutInitDisplayMode(unsigned int mode)
```

  - 颜色模型：GL_RGB/GL_RGBA – RGB颜色(默认)；GL_INDEX – 索引颜色
  - 缓冲区类型：GL_SINGLE – 单缓冲区(默认)；GL_DOUBLE – 双缓冲
  - 属性选项按逻辑或组合在一起

- 创建窗口，标题为**title**。调用**glutMainLoop()**之前，窗口不会被显示

```
int glutCreateWindow(char * title)
```

GLUT事件处理

- GLUT事件处理循环

  void glutMainLoop()

  - 进入GLUT事件处理循环，当有事件发生时，调用相应的回调函数；否则，处于等待状态
  - **main()**函数是以程序进入事件循环做为结束

- 每个GLUT程序都必须有一个显示回调函数

  void glutDisplayFunc(void(*func)(void))

  - 窗口首次被打开
  - 窗口移动
  - **glutPostRedisplay**被显式地调用

GLUT回调函数

- GLUT使用回调函数机制来进行事件处理
  - 窗口重绘 **glutDisplayFunc()**
  - 窗口改变大小 **glutReshapeFunc()**
  - 键盘输入 **glutKeyboardFunc()**
  - 鼠标按键 **glutMouseFunc()**
  - 鼠标移动 **glutMotionFunc()**
  - 空闲函数 **glutIdleFunc()**

3.2 完整的程序

- 3.2.1 OpenGL程序结构
- 3.2.2 控制函数
- 3.2.3 视图
- 3.2.4 图元
- 3.2.5 属性

init()函数


```
void init()
{
        glClearColor(0.0, 0.0, 0.0, 1.0);

        glColor3f(1.0, 1.0, 1.0);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

清除色（背景色）为黑色

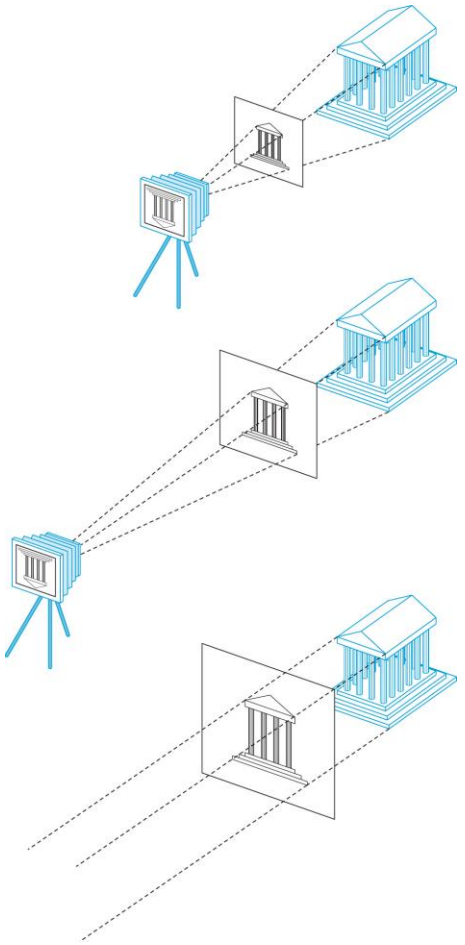
不透明窗口

矩形填充以白色

视景体

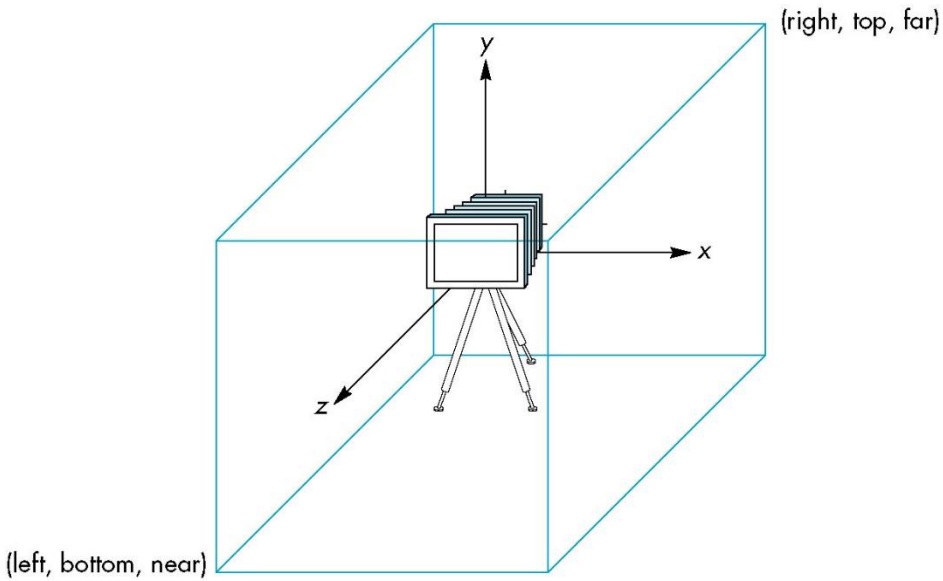
视图(Viewing)

- 虚拟照相机模型：对象与照相机独立
  - 应用程序只需关心对象和照相机的参数设置
- OpenGL默认视图：正交投影
  - 镜头焦距无限大，无限远离对象
  - 极限情形下，投影线彼此平行，投影中心变为投影方向



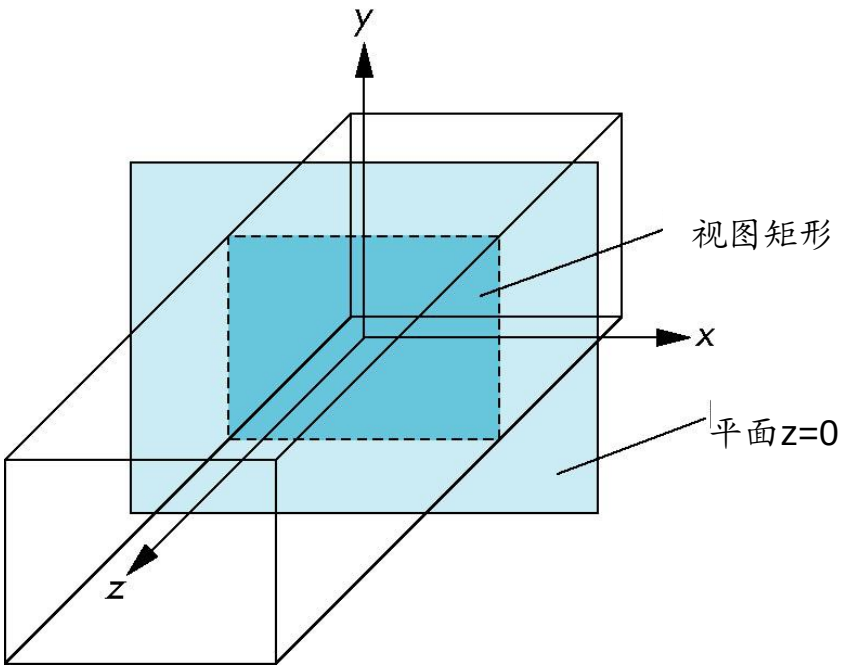
OpenGL中的照相机

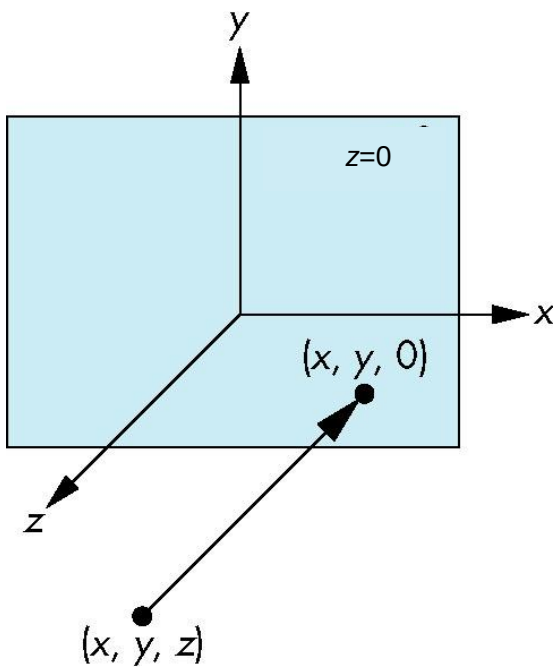
- 照相机被放置在对象坐标系的原点，指向z轴的负方向
- 默认的视景体是一个中心在原点，边长为2的立方体
- 正交投影可以对位于照相机后面的对象成像



正交视图

- 在正交视图（OpenGL默认视图）中，点沿着z轴投影到z=0的平面上





变换与视图

- 在OpenGL中投影是利用投影矩阵乘法(变换)进行的
- 由于只存在一个变换函数系列，因此必须先设置矩阵类型

```
glMatrixMode(GL_PROJECTION);
```

- 变换函数是累积在一起的，因此需要从单位阵开始，然后把它改变为一个定义正交投影视景体的投影矩阵

```
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

二维与三维视图

- 在glOrtho(左, 右, 底, 顶, 近, 远)中的近与远是相对于照相机的距离而言的

```
void glOrtho(GLdouble left, GLdouble right, GLdouble
    bottom, GLdouble top,GLdouble near, GLdouble far)
```

- 二维顶点命令把所有的顶点放在z=0的平面上
- 如果应用程序处于二维状态，那么可以使用下述函数设置正交视景体:

```
void gluOrtho2D(GLdouble left, GLdouble right,
    GLdouble bottom, GLdouble top)
```

- 对于二维情形，视景体或裁剪体退化为裁剪窗口或观察矩形

矩阵模式

- 变换矩阵是系统的状态变量
  - 模型-视图矩阵(model-view)
  - 投影矩阵(projection)
  - 初始值都是单位矩阵
- 每次只能对一个矩阵进行操作
- 通过设置矩阵模式来选择对哪个矩阵进行操作
  - 矩阵模式也是系统状态变量

矩阵模式

- 默认矩阵模式是对模-视矩阵进行操作
- 设置二维观察矩形：

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,50.0,0.0,50.0);
glMatrixMode(GL_MODELVIEW);
```

- 最好每次改变矩阵模式后都返回约定的矩阵模式，例如模-视模式

3.2 完整的程序

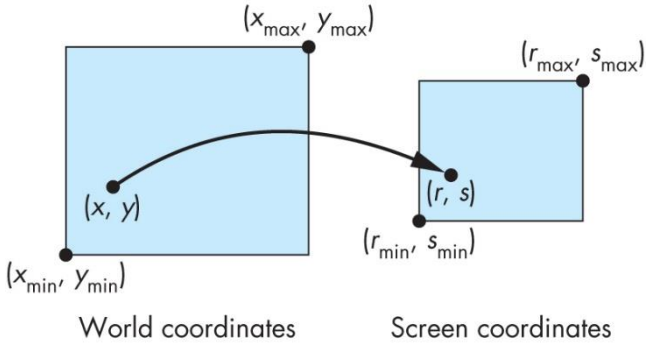
- 3.2.1 OpenGL程序结构
- 3.2.2 控制函数
- 3.2.3 视图
- 3.2.4 图元
- 3.2.5 属性

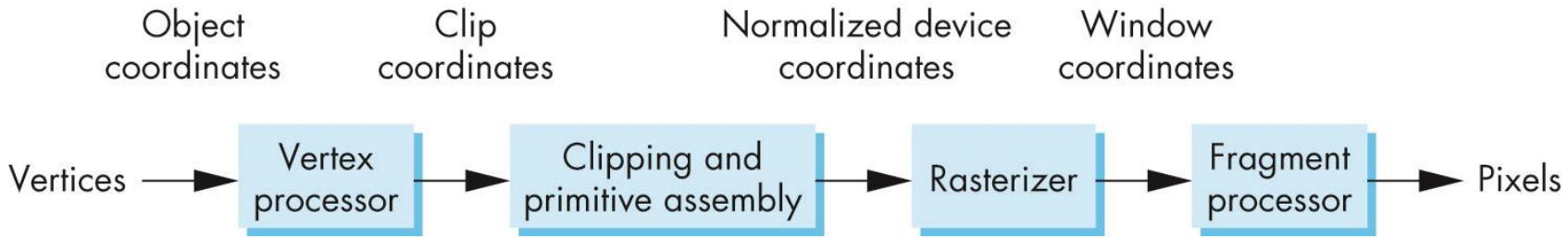
display()函数


```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_POLYGON);
            glVertex2d(-0.5,-0.5);
            glVertex2d(-0.5,0.5);
            glVertex2d(0.5,0.5);
            glVertex2d(0.5,-0.5);
      glEnd();
      glFlush();
}
```

坐标系

- 最初的图形系统要求用户直接按照显示设备的单位来确定所有的信息
- 设备无关图形学
- 顶点坐标在对象坐标系或世界坐标系中定义
- 顶点坐标最终映射成窗口坐标或屏幕坐标



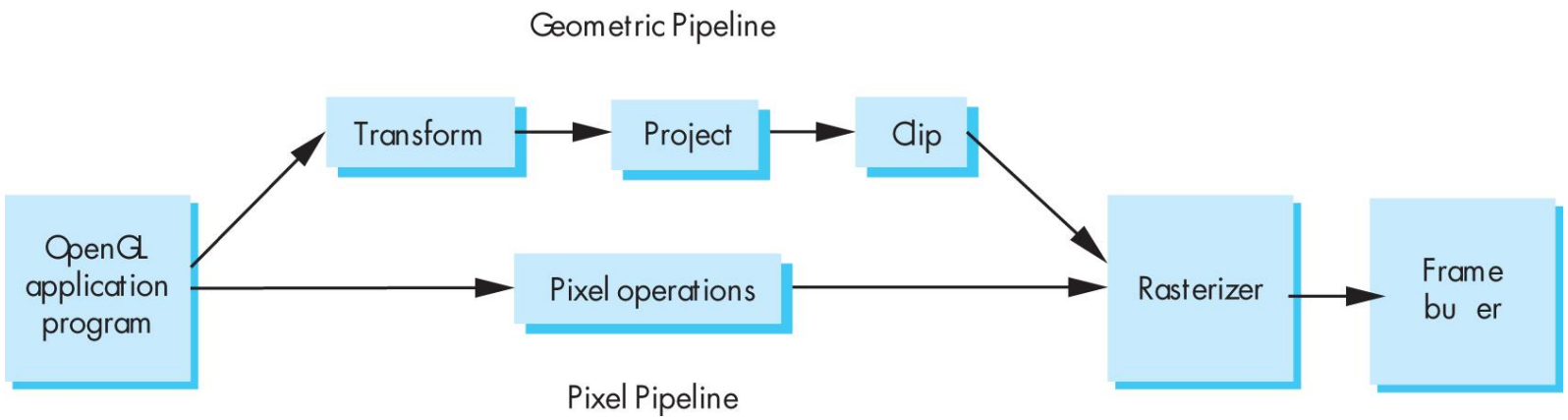


图元

- 最小系统观点
  - 线段、多边形和文本(字符串)，可硬件高效生成
- 复杂图元观点
  - 圆、曲线、曲面和实体等复杂图元
- OpenGL
  - 核心库：点、线、多边形
  - GLU：二次曲面、NURBS曲面
  - GLUT：笔划字符、点阵字符

图元

- 几何图元：点、线段、多边形、曲线和曲面
  - 几何流水线：变换、裁剪、光栅化
- 图像图元/光栅图元：像素阵列
  - 像素流水线



定义几何图元

- 图元由下面语句定义:

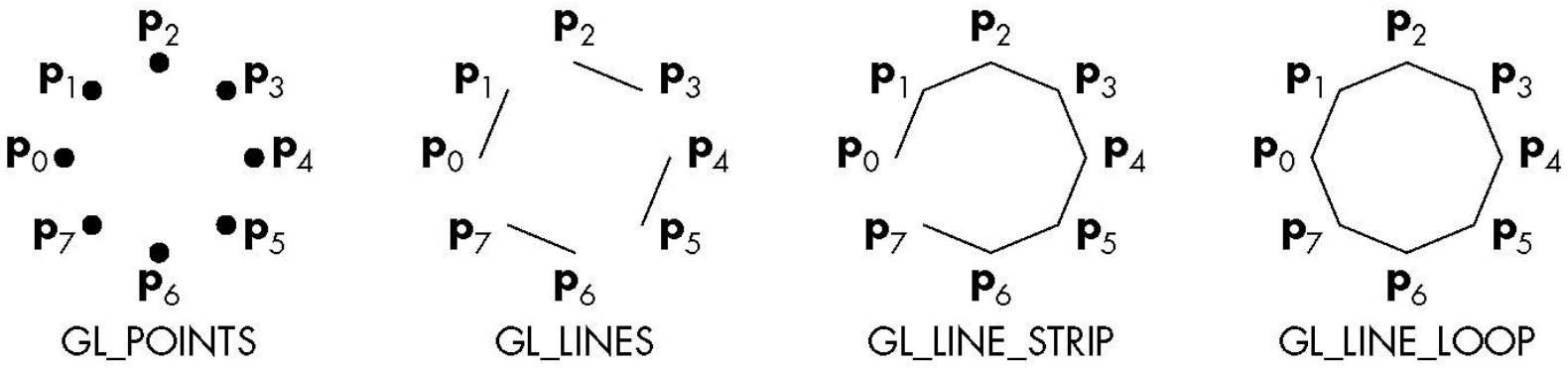
```
glBegin( primType );
glEnd();
```

  - *primType* 决定顶点如何组合成图元

```
glBegin( primType );
for ( i = 0; i < n; ++i ) {
  glColor3f( red[i], green[i], blue[i] );
  glVertex3fv( coords[i] );
}
glEnd();
```

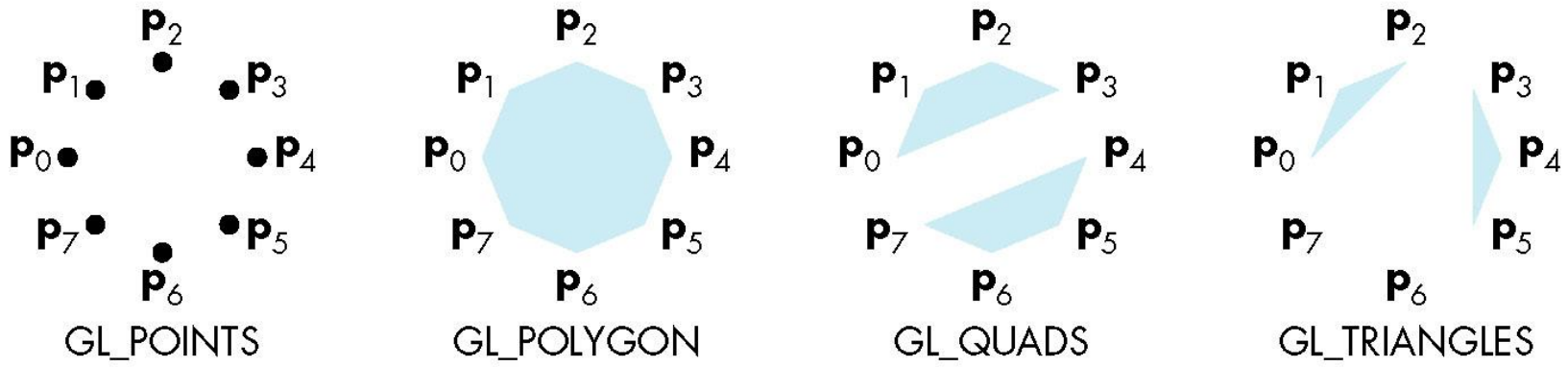
OpenGL的几何图元

- 点：GL_POINTS
- 线段：GL_LINES
- 开折线：GL_LINE_STRIP
- 封闭折线：GL_LINE_LOOP



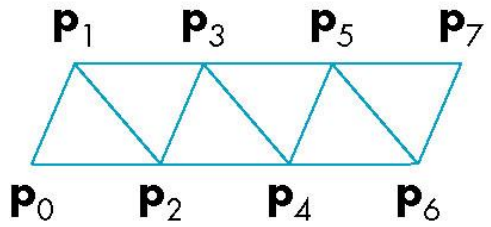
OpenGL的几何图元

- 多边形：GL_POLYGON
- 三角形：GL_TRIANGLES
- 四边形：GL_QUADS

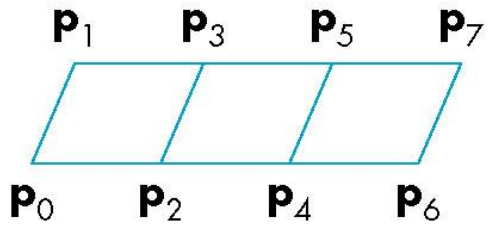


OpenGL的几何图元

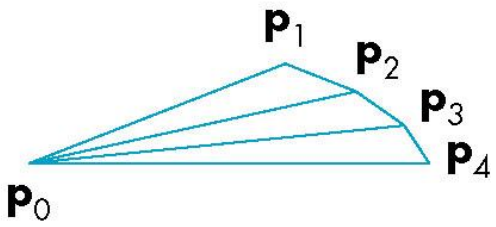
- 三角形带：GL_TRIANGLE_STRIP
- 四边形带：GL_QUAD_STRIP
- 三角形扇：GL_TRIANGLE_FAN



GL_TRIANGLE_STRIP



GL_QUAD_STRIP

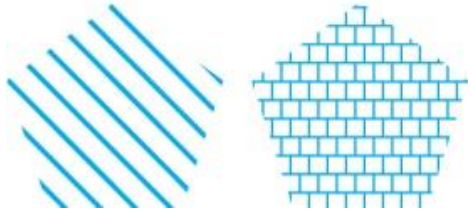


GL_TRIANGLE_FAN

多边形

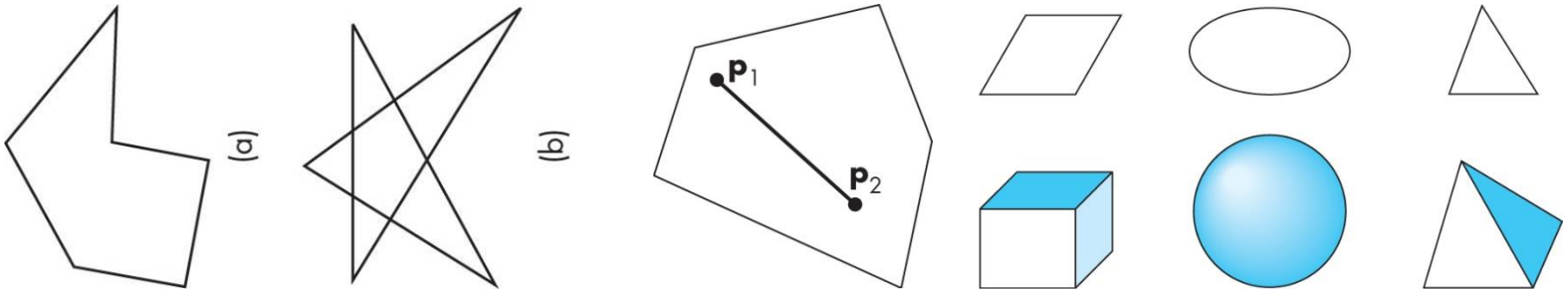
- 多边形的显示方式：绘制边、用单色或模式填充

- 确保正确显示的多边形
  - 简单的
  - 凸的
  - 平面的



定义多边形的限制条件

- 简单性：边除顶点外不相交
- 凸性：对于多边形中任意两点，连接这两点的线段完全在多边形内
- 平面性：所有顶点在同一平面上



- 三角形满足上述所有限制条件

多边形测试

- 用户自己确保上述条件满足
  - 如果不满足上述要求，OpenGL也会有输出，只是结果看起来与期望的不同
- 多边形的简单性和凸性概念上简单，但检测的代价很高
- 大部分图形系统要求应用程序来完成检验
- 新版的OpenGL只绘制三角形

球面的多边形近似

- 单位球面的参数表达:

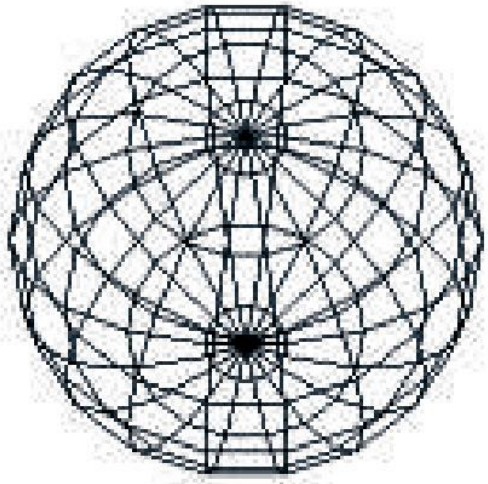
$$x(\theta,\varphi)=\sin\theta\cos\varphi$$

$$y(\theta,\varphi)=\cos\theta\cos\varphi$$

$$z(\theta,\varphi)=\sin\varphi$$

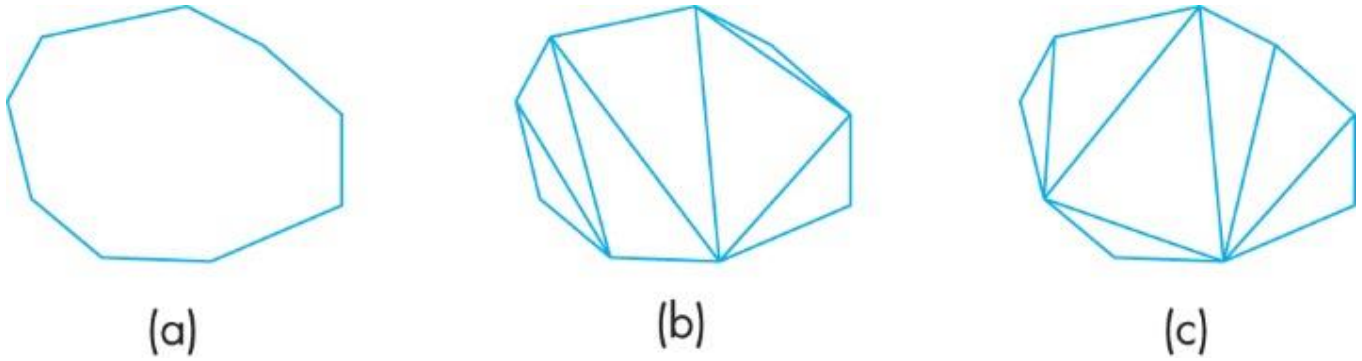
其中$0\le\theta\le 2\pi$, $0\le\varphi\le\pi$

- 中间四边形带->三角形带
- 两极处三角形扇



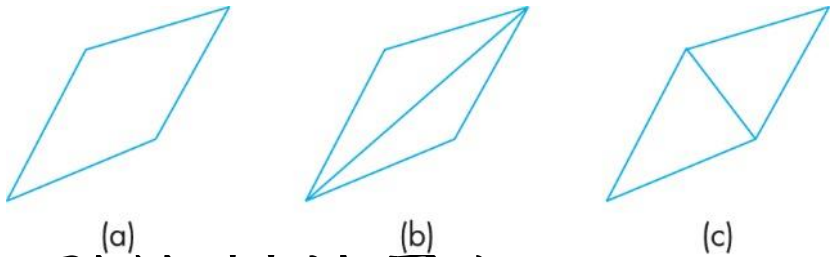
三角剖分

- 对一般多边形，应用程序必须把它剖分成一组三角形
- OpenGL 4.1引入tessellator



三角形的质量

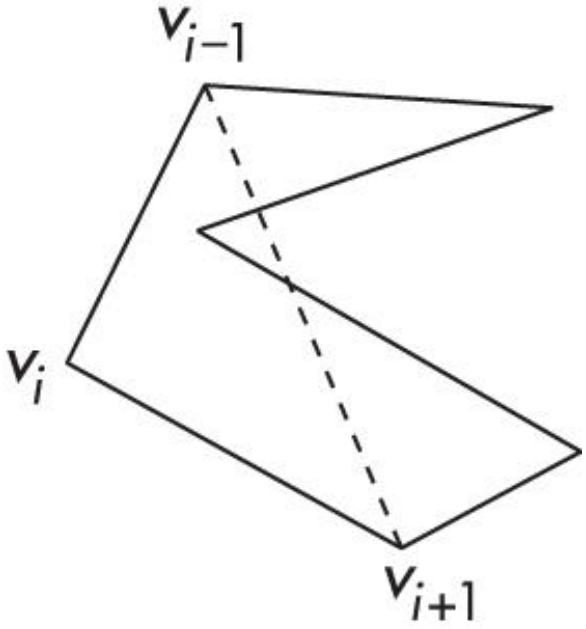
- 狭长的三角形会导致差的绘制效果

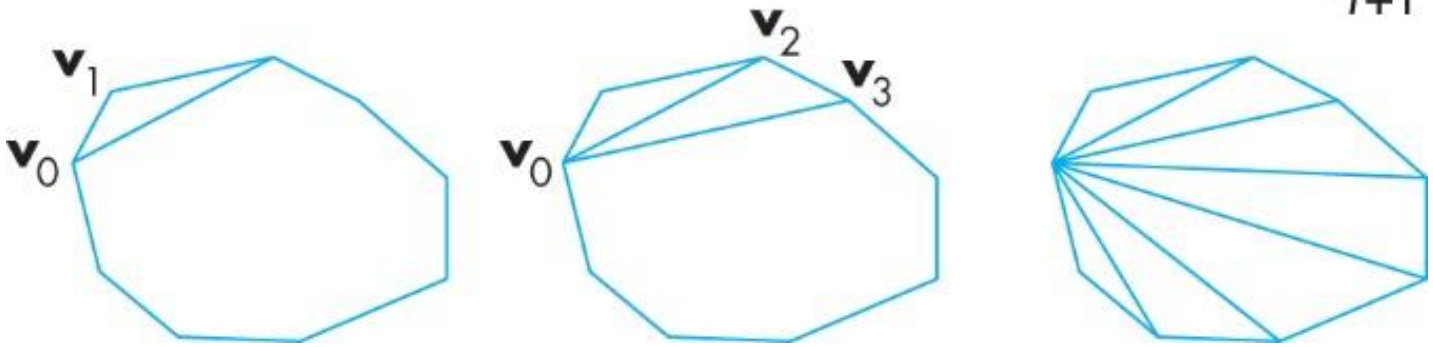


- 等边三角形绘制效果好
- 最大化最小角
- 对无结构点集，Delaunay三角剖分算法能得到一组最优的三角形

凸多边形的三角剖分

- 递归的方式进行剖分





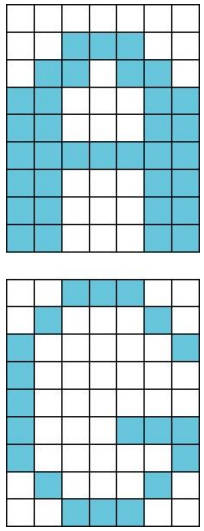
曲线与曲面

- 图元都是由顶点来定义的。
- 使用已有的图元来近似曲线和曲面
  - 正n边形来近似圆
  - 三角形和四边形来近似球面
  - 多边形网格来近似曲面
- 从数学定义出发，编写图形函数来生成
  - 二次曲面
  - NURBS曲线/曲面
- OpenGL中，GLU库和GLUT库提供了对常见曲面的一些近似

文本

- 笔划文本：矢量字体，字符轮廓是线段或曲线
  - 几何图元，变换和观察操作
- 光栅文本：点阵字体，字符定义为0和1的矩形阵列
  - 光栅图元
- GLUT库提供预定义好的笔划字符集和光栅字符集

Computer Graphics

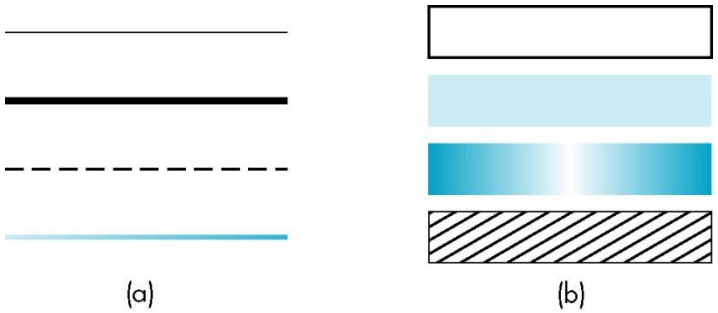


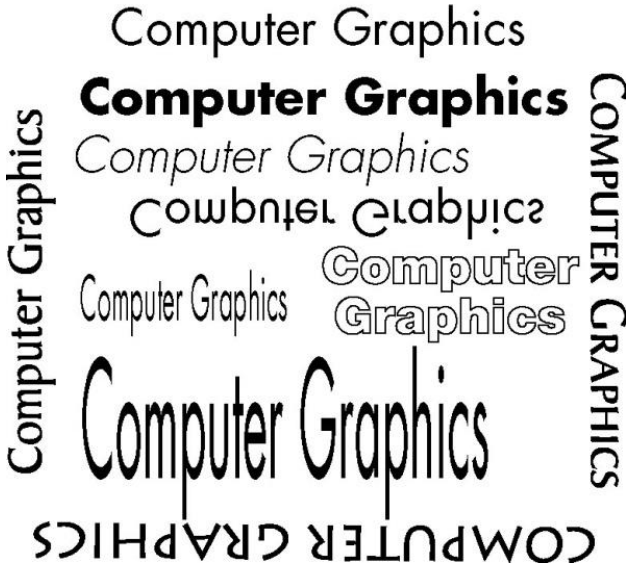
3.2 完整的程序

- 3.2.1 OpenGL程序结构
- 3.2.2 控制函数
- 3.2.3 视图
- 3.2.4 图元
- 3.2.5 属性

属性

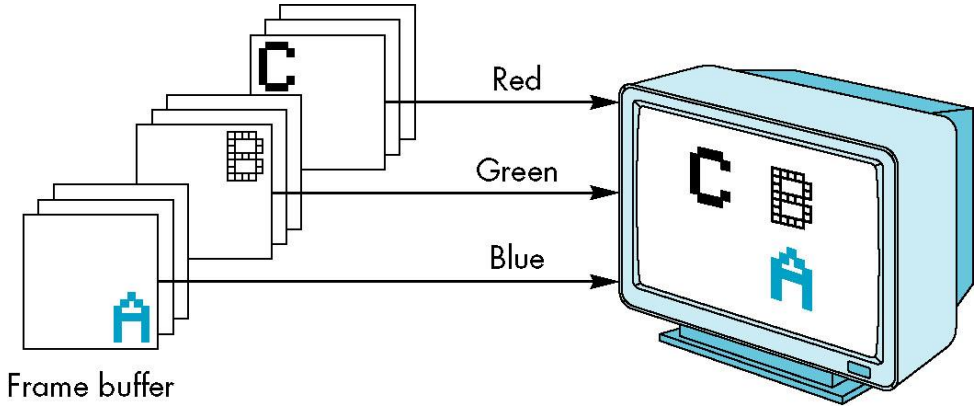
- 属性(attribute)决定图元将如何绘制，OpenGL状态的一部分
  - 点：颜色、大小
  - 线段：颜色、宽度、模式（实线、虚线）
  - 多边形：绘制模式、填充模式
  - 笔划文本：字体、字号、方向





RGB颜色

- 颜色的每个分量在帧缓冲区中是分开存储的
- 在缓冲区中通常每个分量占用8位字节
- 颜色值用float表示的变化范围是从0.0(无)到1.0(全部), 而用unsigned byte表示的变化范围是从0到255



RGBA颜色

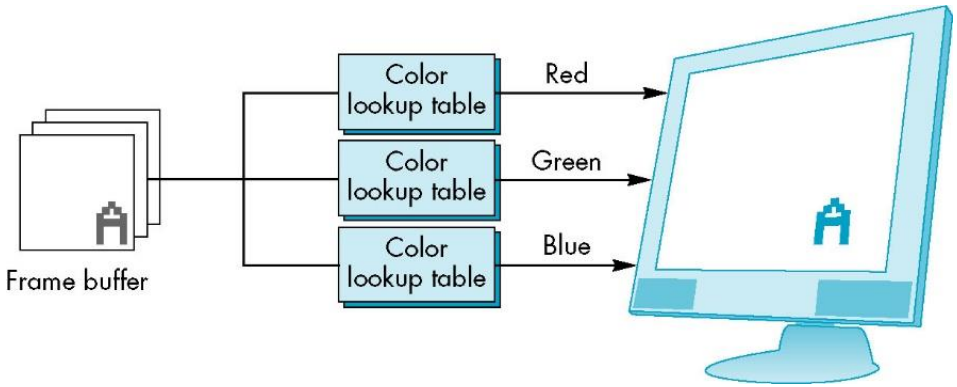
- 第四个分量（A或者α）是不透明度或透明度
  - 完全透明：A=0.0
  - 完全不透明：A=1.0
- 和RGB值一样存储在帧缓冲区中
- 用于图像融合
- 设置清屏颜色（背景色），默认黑色：

```
glClearColor(0.0, 0.0, 0.0, 1.0);
```

索引颜色

- 由一组RGB值构成一张表，“颜色”是表中项的索引
- 需要更少的帧缓存
  - 索引通常只有8位
  - 现在重要性下降
    - 内存价格下降
    - 需要更多的颜色

Input	Red	Green	Blue
0	0	0	0
1	$2^m - 1$	0	0
.	0	$2^m - 1$	0
.	.	.	.
.	.	.	.
$2^k - 1$	.	.	.
	m bits	*m* bits	*m* bits



颜色与状态

- 由glColor*设置的颜色成为状态的一部分，后续构造过程将使用这一颜色，直至它被修改为止
  - 颜色与其它属性不是对象的一部分，但是在渲染对象时要把这些属性赋给对象
- 可以按下述过程创建具有不同颜色的顶点

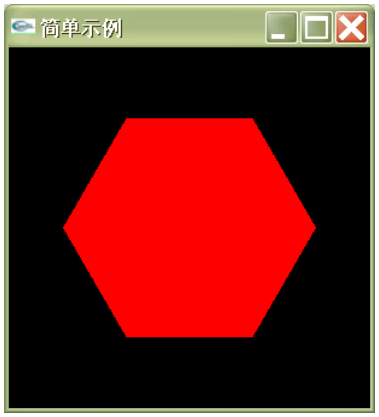
```
glColor(…);
glVertex(…);
glColor(…);
glVertex(…);
```

着色模型

- 默认状态是平滑着色
  - OpenGL根据多边形顶点的颜色插值出来内部的颜色
- 另外一种状态是平面着色
  - 第一个顶点的颜色确定填充颜色
- 设置着色模型：

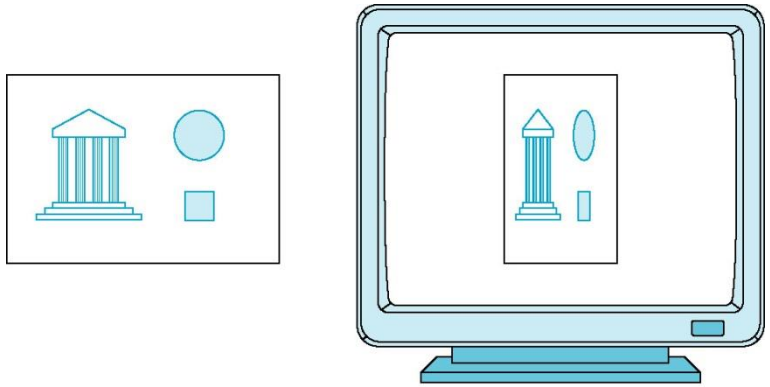
```
void glShadeModel(Glenum mode)
```

  - 平滑模式：GL_SMOOTH
  - 平面模式：GL_FLAT



宽高比

- 宽高比(aspect ratio)：矩形的宽度与高度之比
- 如果glOrtho和glutInitWindowSize设置矩形的宽高比不同，引起对象变形
  - 保证宽高比



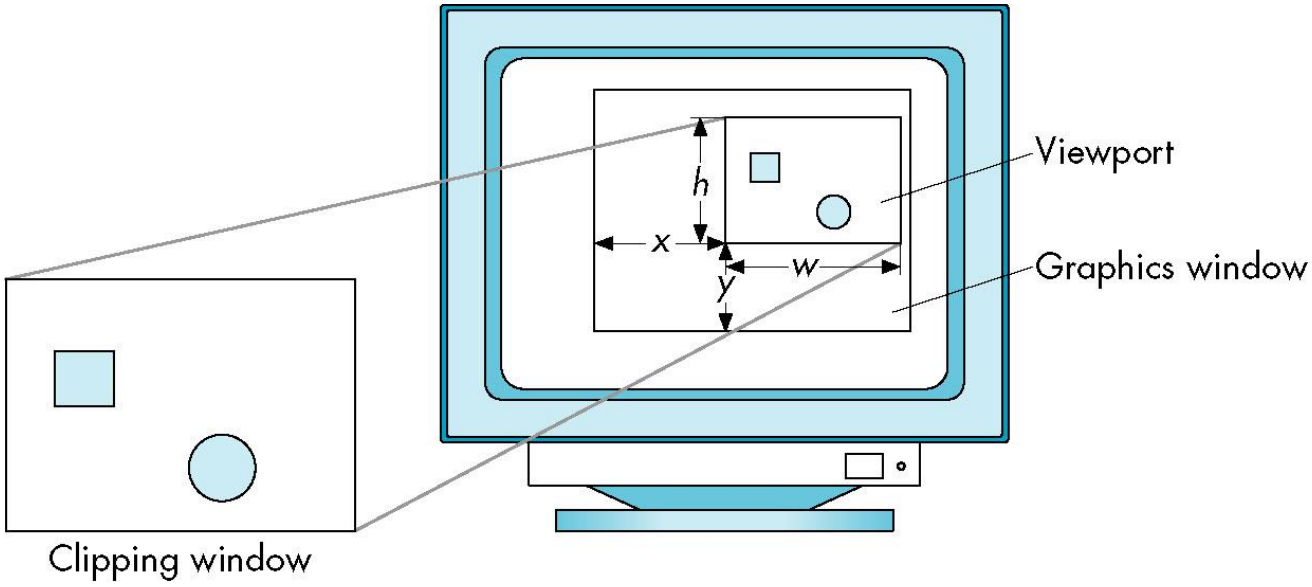
(a) (b)

视口

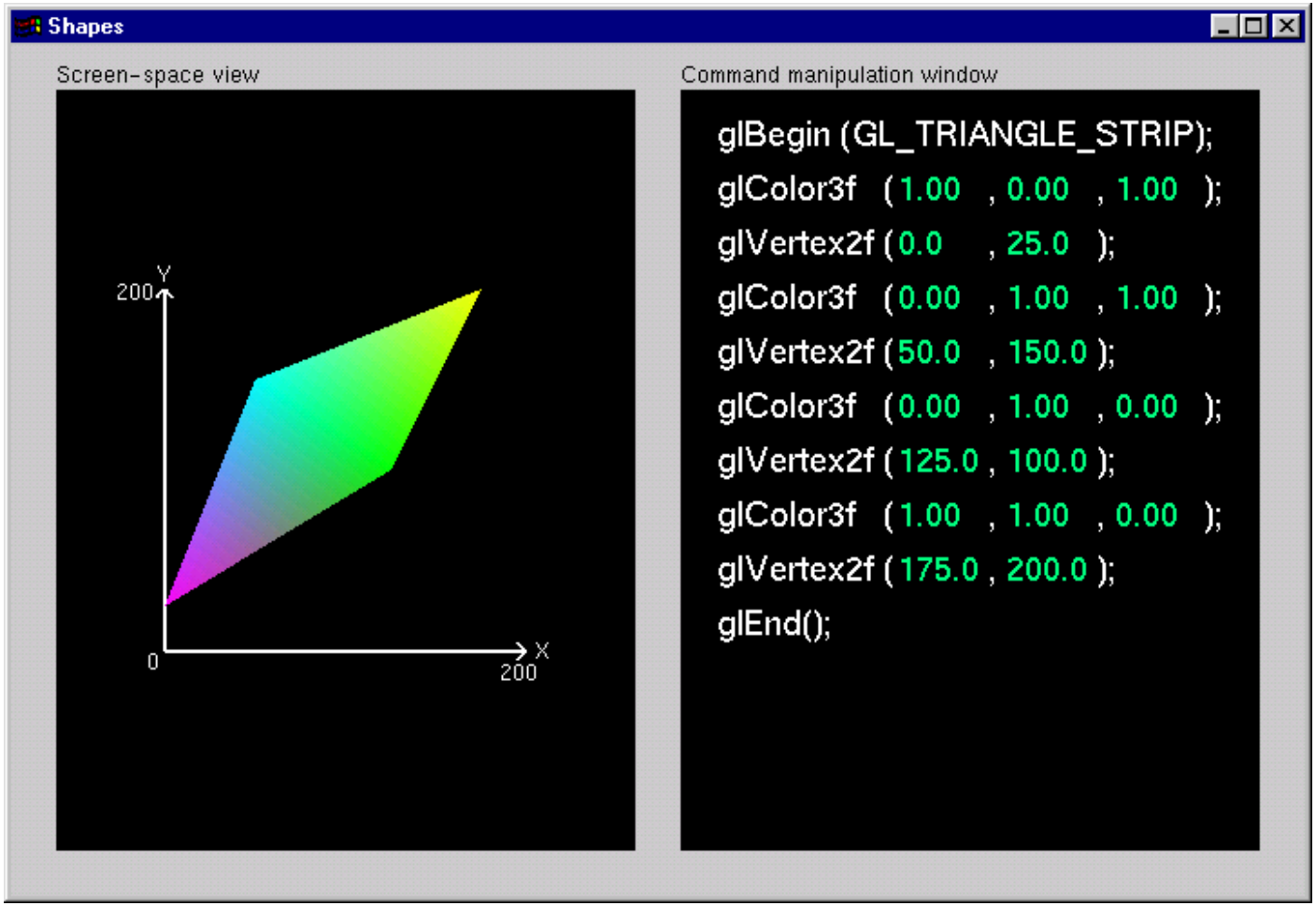
- 可在当前显示窗口中定义视口用来显示图像:

  `void glViewport(GLint x, GLint y, GLint w, GLint h)`

  – 左下角为（x,y），以像素为单位（窗口坐标）
- 通过调节视口的高度和宽度来匹配裁剪矩形的宽高比
- 视口也是状态的一部分



Shapes


Shapes
Screen-space view
Command manipulation window
Y
200
0
X
200
glBegin (GL_TRIANGLE_STRIP);
glColor3f (1.00 , 0.00 , 1.00);
glVertex2f (0.0 , 25.0);
glColor3f (0.00 , 1.00 , 1.00);
glVertex2f (50.0 , 150.0);
glColor3f (0.00 , 1.00 , 0.00);
glVertex2f (125.0 , 100.0);
glColor3f (1.00 , 1.00 , 0.00);
glVertex2f (175.0 , 200.0);
glEnd();

3.1 OpenGL简介

3.2 完整的程序

3.3 三维图形程序

3.3 三维图形程序

- 3.3.1 Sierpinski镂垫
- 3.3.2 隐藏面消除
- 3.3.3 面细分与体细分

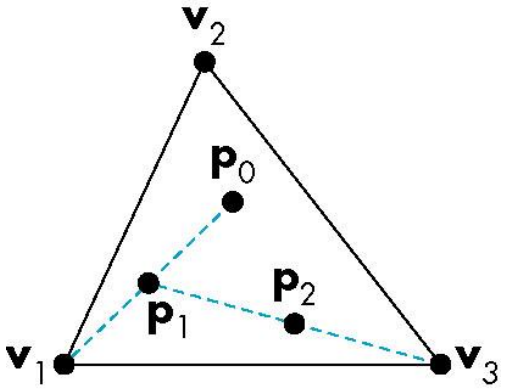
三维图形应用程序

- 在OpenGL中二维应用程序是三维应用程序的特殊情形
- 要得到三维图形应用程序
  - 只需对前面程序进行很小的修改
    - 使用glVertex3*( )
  - 必须考虑多边形绘制的顺序或者启用隐藏面消除功能
  - 只考虑平面的简单凸多边形

Sierpinski镂垫(2D)

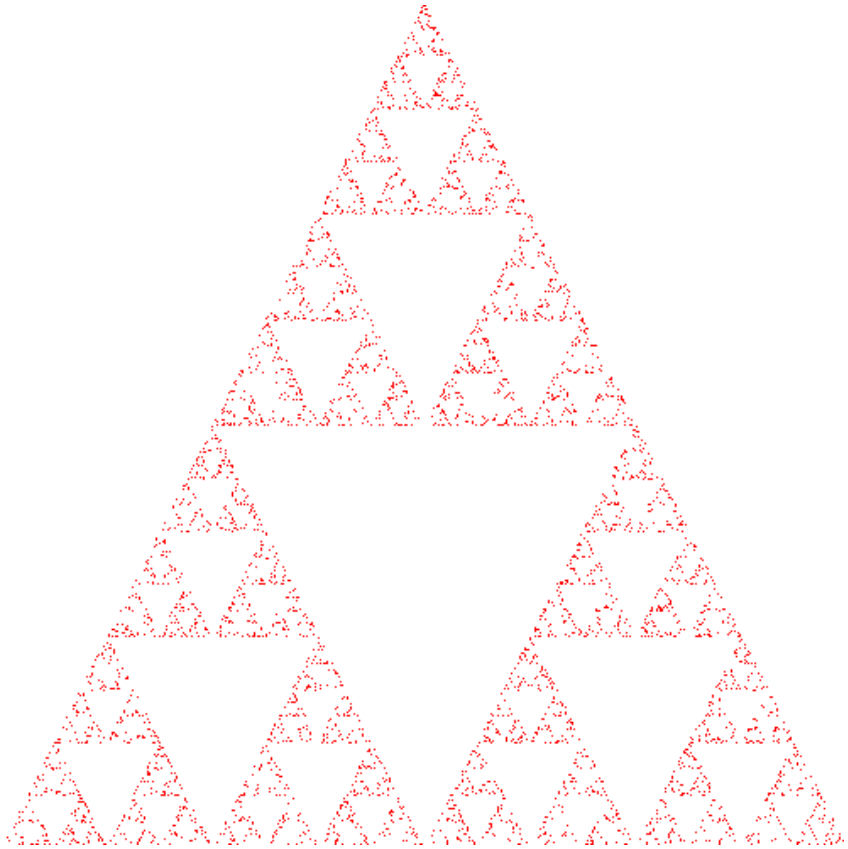
- 给定一个三角形，构造过程如下：
  - 1. 在三角形内随机选择一个初始点p(x,y,0)
  - 2. 随机选择三个顶点之一
  - 3. 找出p和随机选择的顶点连线的中点q
  - 4. 把中点q显示出来
  - 5. 用中点q替换p
  - 6. 转到步骤2

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



绘制程序

- 在白色背景上绘制红色的点
- 设置一个二维坐标系，要绘制的点落在50x50的正方形内，正方形的左下角位于原点



init()函数

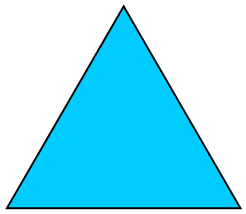

```
void init()
{
   /* attributes */
   glClearColor(1.0, 1.0, 1.0, 1.0); /* white background */
   glColor3f(1.0, 0.0, 0.0); /* draw in red */

   /* set up viewing */
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0, 50.0, 0.0, 50.0);
   glMatrixMode(GL_MODELVIEW);
}
```

display()函数

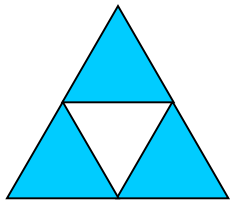

```
void display( void )
{   /* A triangle */
    GLfloat vertices[3][2]={{0.0, 0.0}, {25.0, 50.0}, {50.0, 0.0}};
    int j, k;
    int rand();       /* standard random number generator */
    GLfloat p[2] = {7.5, 5.0};  /* An arbitrary initial point inside traingle */
    glClear(GL_COLOR_BUFFER_BIT);  /*clear the window */

    glBegin(GL_POINTS);      /* compute and plots 5000 new points */
    for (k=0; k<5000; k++)
    {
        j = rand()%3; /* pick a vertex at random */
        /* Compute point halfway between selected vertex and old point */
        p[0] = (p[0]+vertices[j][0])/2.0;
        p[1] = (p[1]+vertices[j][1])/2.0;

        glVertex2fv(p);  /* plot new point */
    }
    glEnd();
    glFlush();
}
```

Sierpinski镂垫(2D)

- 递归方法
  - 从一个三角形开始

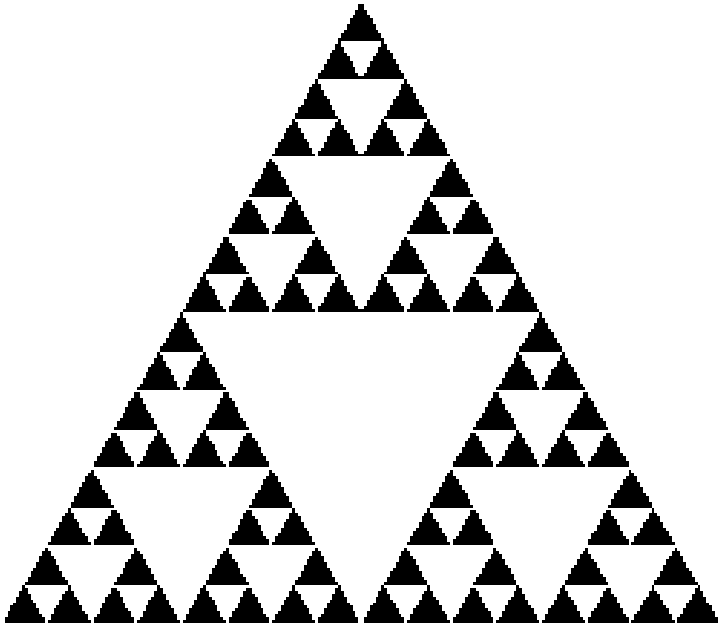


  - 连接三边的中点并去掉中间的三角形



  - 重复上述过程

四次细分后的结果



分形

- 考虑黑色填充区域的面积与周长（即包含填充区域的所有线段总长）
- 当持续细分时
  - 面积趋向于零
  - 但周长趋向于无穷
- 因此无穷细分后的结果不是通常的几何形状
  - 它的维数既不是一维的，也不是二维的
- 我们称之为分形（分数维1.585）

绘制三角形


```
#include <GL/glut.h>


// initial triangle
GLfloat v[3][2] = {{-1.0, -0.58}, {1.0, -0.58},
        {0.0, 1.15}};
int n; // number of recursive steps


void triangle( GLfloat *a, GLfloat *b, GLfloat *c)
// display one triangle
{
        //glBegin(GL_TRIANGLES);
        glVertex2fv(a);
        glVertex2fv(b);
        glVertex2fv(c);
        //glEnd();
}
```

三角形细分


```
void divide_triangle(GLfloat *a, Glfloat *b,
Glfloat *c, int m)
{
      // triangle subdivision using vertex numbers
      GLfloat v0[2], v1[2], v2[2];
      int j;
      if (m>0) {
            for(j=0; j<2; j++) v0[j]=(a[j]+b[j])/2;
            for(j=0; j<2; j++) v1[j]=(a[j]+c[j])/2;
            for(j=0; j<2; j++) v2[j]=(b[j]+c[j])/2;
            divide_triangle(a, v0, v1, m-1);
            divide_triangle(c, v1, v2, m-1);
            divide_triangle(b, v2, v0, m-1);
      }
      // draw triangle at end of recursion
      else   triangle(a,b,c);
}
```

显示与初始化函数


```
void display(void) {
      glClear(GL_COLOR_BUFFER_BIT);
      glBegin(GL_TRIANGLES);
          divide_triangle(v[0], v[1], v[2], n);
      glEnd();
      glFlush();
}

void init() {
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(-2.0, 2.0, -2.0, 2.0);
      glMatrixMode(GL_MODELVIEW);
      glClearColor(1.0, 1.0, 1.0, 1.0); //背景白色
      glColor3f(0.0,0.0,0.0);
}
```

main()函数


```
int main(int argc, char **argv)
{
      n=4;
      glutInit(&argc, argv);
      glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
      glutInitWindowSize(500, 500);
      glutCreateWindow("2D Gasket");
      glutDisplayFunc(display);
      init();
      glutMainLoop();
}
```

效率的考虑

- 通过把glBegin和glEnd放在显示回调函数中，而不是triangle函数中，把glBegin的参数写为GL_TRIANGLES，而不是GL_POLYGON。
- 这样，绘制整个图形只需要调用一次glBegin和glEnd，而不是每个三角形都调用一次

改变到三维的情形

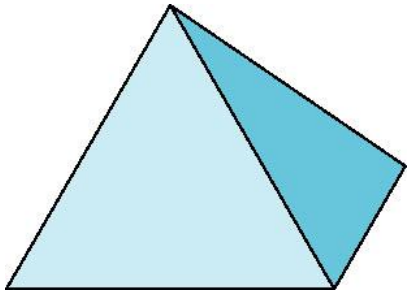
- 通过下述修改，可以很容易地把程序生成三维图形:

```
GLfloat v[3][3]
glVertex3f
glOrtho
```

- 但这并不没有多大实质性改变（2D!）
- 下面从四面体开始迭代

三维镂垫-随机方法

- 随机方法:
  - 4个顶点定义四面体
  - 显示与某顶点连线的中点



```
GLfloat vertices[4][3] = {{0.0, 0,0, 0.0}, {25.0, 50.0, 10.0},
                          {50.0,25.0,25.0}, {25.0,10.0,25.0}};

GLfloat p[3] = {25.0,10.0,25.0};
```

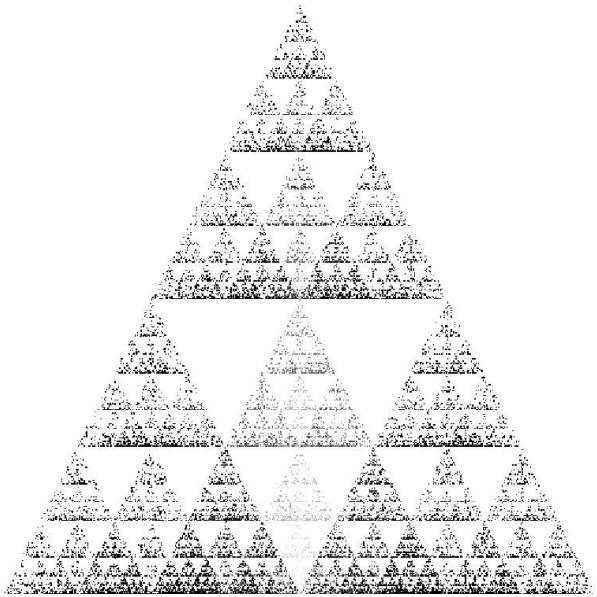
产生新点


```
glBegin(GL_POINTS);
   for( k = 0; k < 5000; k++)
   {
       j = rand() % 4; /* pick a vertex at random */

      /* Compute point halfway between selected vertex and old point */
      p[0] = (p[0]+vertices[j][0])/2.0;
      p[1] = (p[1]+vertices[j][1])/2.0;
      p[2] = (p[2]+vertices[j][2])/2.0;

      /* plot new point */
      glColor3f(p[0]/250.0, p[1]/250.0, p[2]/250.0);
      glVertex3fv(p);
   }
glEnd();
```

三维观察

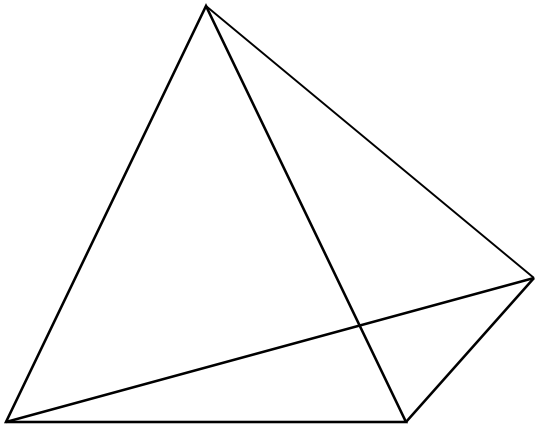
- 定义三维裁剪体

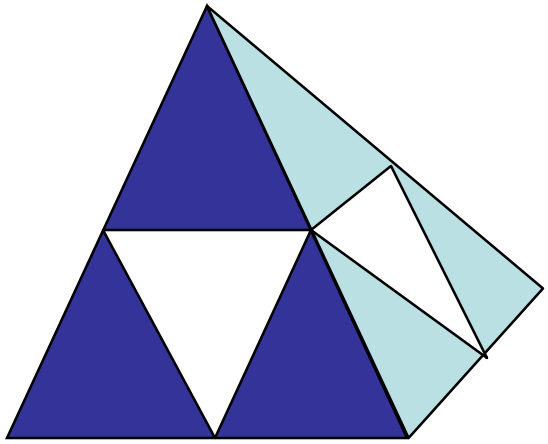
```
glOrtho(-50.0,50.0,-50.0,50.0,-50.0,50.0);
```



三维镂垫-迭代方法

- 细分四面体的四个面





- 看起来好像是把四面体的中心移走，留下四个小四面体

细分代码

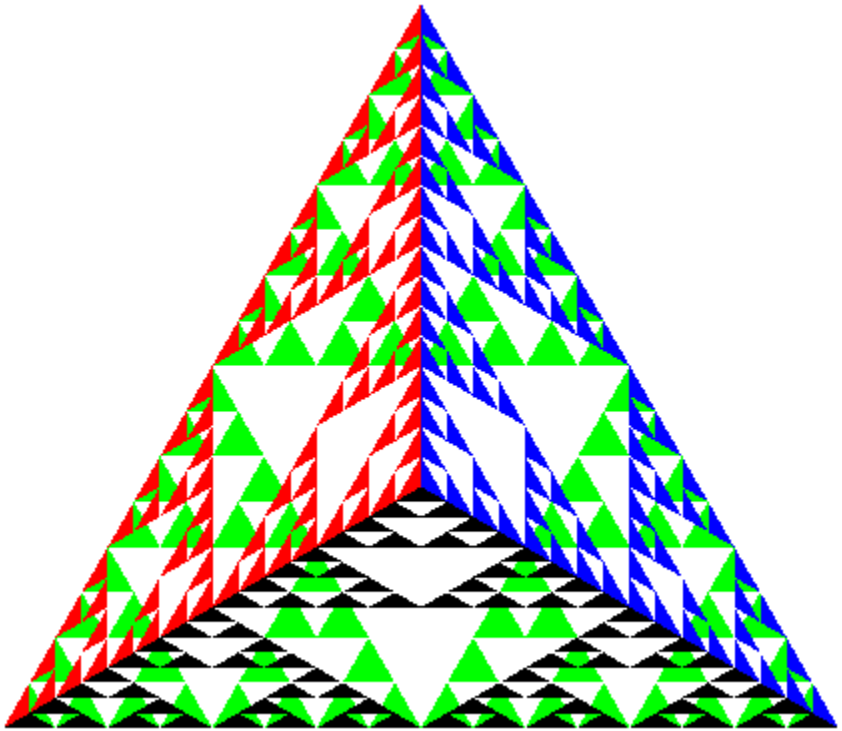

```
void triangle( GLfloat *a, GLfloat *b, GLfloat *c)
{
        glVertex3fv(a);
        glVertex3fv(b);
        glVertex3fv(c);
}

void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *c, int m)
{
        GLfloat v1[3], v2[3], v3[3];
        int j;
        if (m>0) {
                for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;
                for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;
                for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;
                divide_triangle(a, v1, v2, m-1);
                divide_triangle(c, v2, v3, m-1);
                divide_triangle(b, v3, v1, m-1);
        }
        else  triangle(a,b,c);
}
```

生成四面体的代码


```
void tetrahedron( int m)
{
      glColor3f(1.0, 0.0, 0.0);
      divide_triangle(v[0], v[1], v[2], m);
      glColor3f(0.0, 1.0, 0.0);
      divide_triangle(v[3], v[2], v[1], m);
      glColor3f(0.0, 0.0, 1.0);
      divide_triangle(v[0], v[3], v[1], m);
      glColor3f(0.0, 0.0, 0.0);
      divide_triangle(v[0], v[2], v[3], m);
}
```

结果

- 四次迭代后

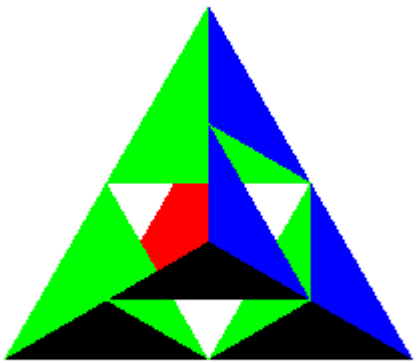


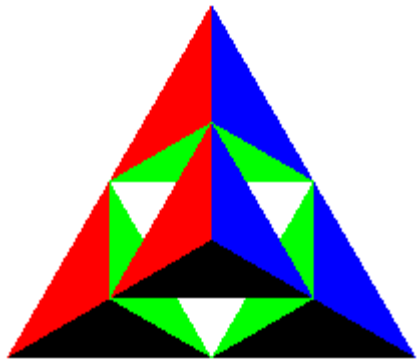
3.3 三维图形程序

- 3.3.1 Sierpinski镂垫
- 3.3.2 隐藏面消除
- 3.3.3 面细分与体细分

问题

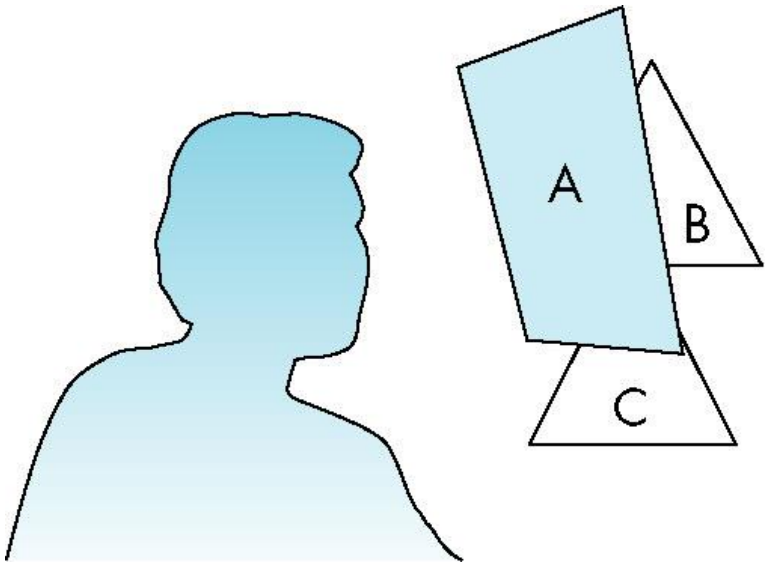
- 由于三角形是按照在程序中定义的顺序画出的，本来在前面的三角形并不是显示在位于它后面的三角形的前面





隐藏面消除

- 只想见到那些位于其它面前面的曲面或平面片
- OpenGL采用称为z缓冲区算法进行隐藏面消除，在z缓冲区中存贮着对象的深度信息，从而只有前面的对象出现在图像中



Z缓冲区算法

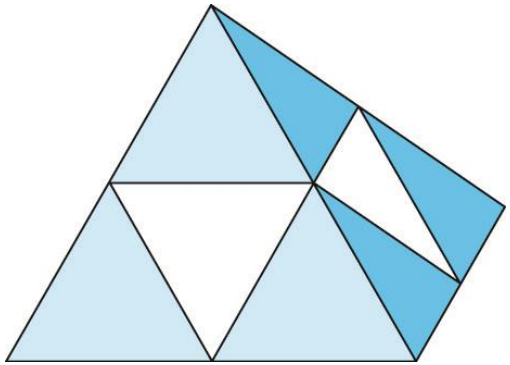
- 该算法创建专门的缓冲区(称为Z缓冲区)，当几何体经过流水线各步骤时，存储着该几何体的深度信息
- 启用该算法的要素
  - 在main()中，显示模式初始化语句改为
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
  - 在init()中激活Z-Buffer隐藏面消除算法
    glEnable(GL_DEPTH_TEST);
  - 在显示回调函数中清空深度缓冲区
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

3.3 三维图形程序

- 3.3.1 Sierpinski镂垫
- 3.3.2 隐藏面消除
- 3.3.3 面细分与体细分

面细分vs体细分

- 在前面的例子中，我们对每个表面进行细分
- 也可以利用相同的中点对体进行细分
- 各边中点以及四面体的中点定义了四个小四面体，每个小四面体与大四面体的一个顶点对应
- 保留这四个小面体，把其它的去掉



初始四面体


```
GLfloat v[4][3]={{0.0, 0.0, 1.0},
                 {0.0, 0.942809, -0.33333},
                 {-0.816497, -0.471405, -0.333333},
                 {0.816497, -0.471405, -0.333333}};

GLfloat colors[4][3] = {{1.0, 0.0, 0.0},
                 {0.0, 1.0, 0.0},
                {0.0, 0.0, 1.0},
                 {0.0, 0.0, 0.0}};

int n;
```

绘制四面体


```
void triangle(GLfloat *va, GLfloat *vb, GLfloat *vc)
{
        glVertex3fv(va);
        glVertex3fv(vb);
        glVertex3fv(vc);
}
void tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d)
{
    glColor3fv(colors[0]);
    triangle(a, b, c);
    glColor3fv(colors[1]);
    triangle(a, c, d);
    glColor3fv(colors[2]);
    triangle(a, d, b);
    glColor3fv(colors[3]);
    triangle(b, d, c);
}
```

细分


```
void divide_tetra(GLfloat *a, GLfloat *b, GLfloat *c, GLfloat *d, int m)
{
    GLfloat mid[6][3];
    int j;
    if(m>0)    {
        /* compute six midpoints */
        for(j=0; j<3; j++) mid[0][j]=(a[j]+b[j])/2;
        for(j=0; j<3; j++) mid[1][j]=(a[j]+c[j])/2;
        for(j=0; j<3; j++) mid[2][j]=(a[j]+d[j])/2;
        for(j=0; j<3; j++) mid[3][j]=(b[j]+c[j])/2;
        for(j=0; j<3; j++) mid[4][j]=(c[j]+d[j])/2;
        for(j=0; j<3; j++) mid[5][j]=(b[j]+d[j])/2;
        /* create 4 tetrahedrons by subdivision */
        divide_tetra(a, mid[0], mid[1], mid[2], m-1);
        divide_tetra(mid[0], b, mid[3], mid[5], m-1);
        divide_tetra(mid[1], mid[3], c, mid[4], m-1);
        divide_tetra(mid[2], mid[4], d, mid[5], m-1);
    }
    else(tetra(a,b,c,d)); /* draw tetrahedron at end of recursion */
}
```

显示回调函数


```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
       divide_tetra(v[0], v[1], v[2], v[3], n);
    glEnd();
    glFlush();
}
```

窗口改变回调函数


```
void reshape(int w, int h)
{
    glViewport(0, 0, w, h); // 设置视口
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
         2.0*(GLfloat) w/(GLfloat) h, -2.0, 2.0,-10.0,10.0);
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();     // 调用显示回调函数
}
```

main()函数


```
int main(int argc, char **argv)
{
    n=3;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("3D Gasket");
    glutReshapeFunc(reshape);      // 窗口改变回调函数
    glutDisplayFunc(display);
    glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

体细分