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

第四章 输入与交互

4.1 输入设备

4.2 事件驱动编程

4.3 弹出式菜单

4.4 更多的交互

4.1 输入设备

- 物理设备
- 逻辑设备
- 输入模式

图形输入设备

- 输入设备可以用下述两种属性中的一种进行描述
  - 物理属性
    - 鼠标 键盘 跟踪球
  - 逻辑属性
    - 设备通过API返回给程序的内容是什么？
      - 位置
      - 对象标识符（一个整数值）
- 模式
  - 如何以及何时获取输入？
    - 请求(主动)或者事件(被动)

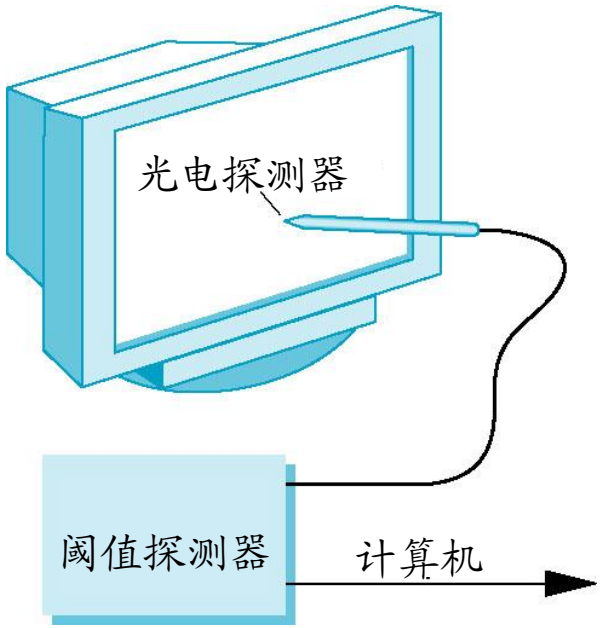
物理输入设备

- 指向设备：允许用户在屏幕上指定点的位置，有一个或多个按钮用来向计算机发送信号或中断请求

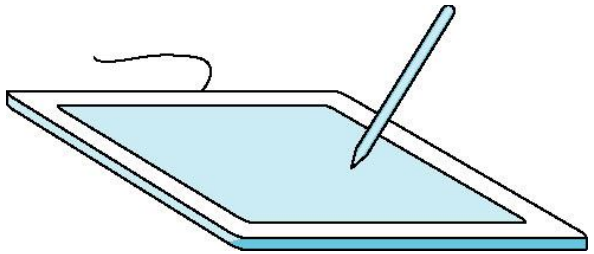
- 键盘设备：几乎总是指键盘，也可泛指任何能向计算机传送字符编码的设备（如，软键盘）

指向设备－绝对定位

- 绝对定位设备：直接向操作系统返回屏幕位置
  - 数据板
  - 光笔
  - 触摸屏



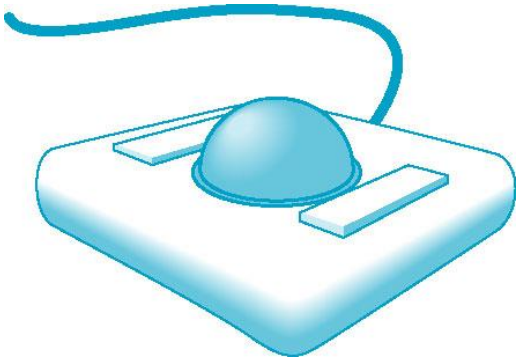
光笔



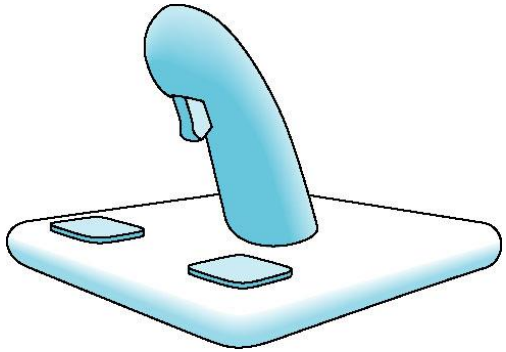
数据板

指向设备 – 相对定位

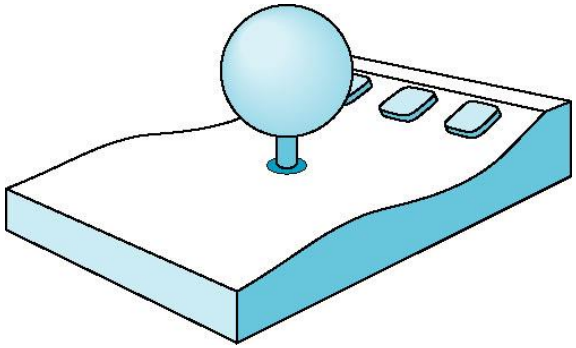
鼠标



跟踪球



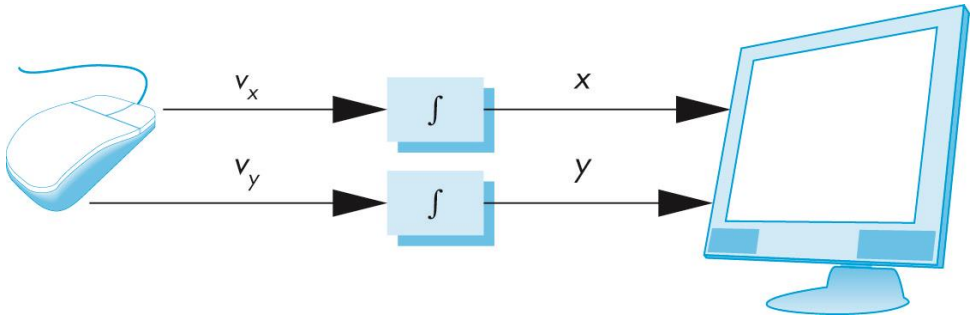
游戏操纵杆



三维空间球

相对定位设备

- 鼠标、跟踪球以及游戏操纵杆等设备向操作系统返回两个正交的速度分量值
  - 当鼠标移动时，对两个速度分量进行积分得到x和y方向上的偏移值，加到初始位置上转化为屏幕上某点的位置坐标
  - 利用球的位置改变来确定位置，并没有利用球的绝对位置



逻辑设备

- 考虑C与C++代码:
  - C++: cin >> x;
  - C: scanf (“%d”, &x);
- 输入设备是什么？
  - 代码中并没有指明
  - 可以是键盘、文件、或者其它程序的输出
- 上述代码要求的就是逻辑输入
  - 不管物理设备是什么，程序得到一个数(int)作为输入

图形逻辑设备

- 相对于非图形程序中的输入通常只是数字、字符或者字节而言，图形输入的形式更多
- 在GKS和PHIGS这两个老的API中，定义了六种类型的逻辑输入：
  - 定位(Locator)：返回一个位置
  - 拾取(Pick)：返回对象的标识ID
  - 键盘(Keyboard)：返回字符串
  - 笔划(Stroke)：返回一组位置数据
  - 定值(Valuator)：返回模拟量输入（浮点数）
  - 选择(Choice)：返回n项中的一项

输入模式

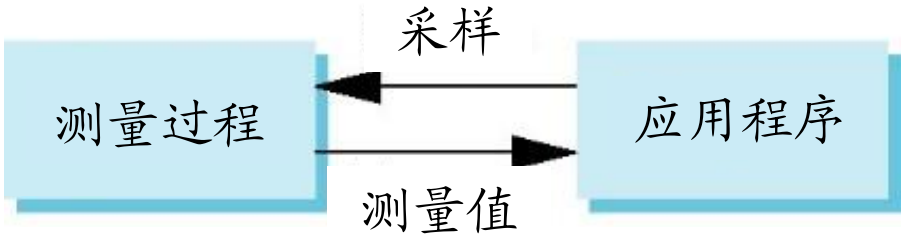
- 在输入设备中具有一个触发器 (trigger), 它可以向操作系统发送一个信号
  - 鼠标上的按钮
  - 按下/释放键盘上的键
- 当触发后，输入设备向系统返回信息，即相应的测量值(measure)
  - 鼠标返回位置信息
  - 键盘返回ASCII码
- 三种模式：采样模式、请求模式和事件模式

请求模式(Request Mode)

- 只有当用户触发了设备后，输入才提供给程序
- 键盘输入为典型例子
  - 在按回车键(触发器)之前，可以删除(回退)、编辑、修改输入的信息
- 用户必须指明哪个输入设备提供输入
  - 适合程序引导用户的情形，而不适合用户控制程序执行过程

采样模式(Sample Mode)

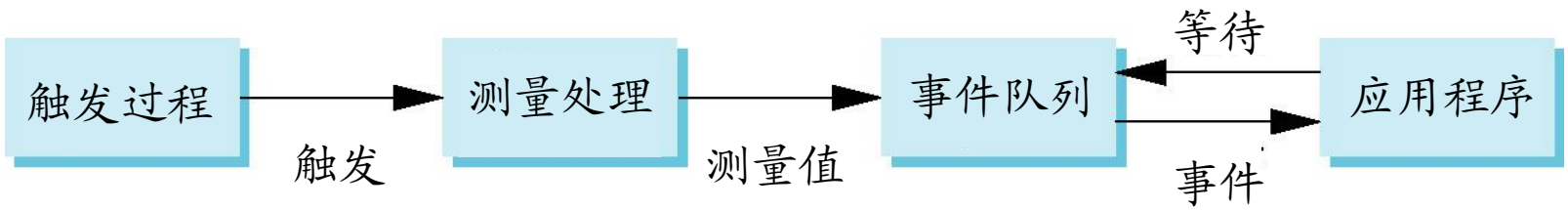
- 测量数据即时返回给程序，不需要触发器



- 同样地，用户必须指明哪个输入设备提供输入
  - 适合程序引导用户的情形，而不适合用户控制程序执行过程

事件模式(Event Mode)

- 绝大部分系统具有多个输入设备，每个设备都可能被用户在任意时间触发
- 每个触发生成一个事件，事件的测量值放到事件队列中，用户程序检查该队列，根据事件的类型采用相应的操作（回调函数）



第四章 输入与交互

4.1 输入设备

4.2 事件驱动编程

4.3 弹出式菜单

4.4 更多的交互

4.2 事件驱动编程

- 鼠标
- 键盘
- 窗口改变
- 空闲

事件类型

- 窗口：改变尺寸、重新显示、缩成图标
- 鼠标：点击一个或多个按钮，移动
- 键盘：按下或释放某个键
- 空闲：“没有事件”
  - 可以定义如果队列中没有其它事件就可以进行的某种操作

回调

- 回调是事件驱动输入方式的编程接口
- 为图形系统可以识别的每种类型事件定义一个回调函数
- 当相应的事件出现时，就会自动执行用户指定的函数
- GLUT示例

```
glutMouseFunc(mouse);
```

鼠标回调函数

GLUT中的回调函数

- GLUT识别在各种窗口系统(Windows, X, Macintosh)中都有的一组事件
  - **glutDisplayFunc**
  - **glutMouseFunc**
  - **glutReshapeFunc**
  - **glutKeyboardFunc**
  - **glutIdleFunc**
  - **glutMotionFunc**
  - **glutPassiveMotionFunc**

GLUT中的事件循环

- 在任何使用GLUT库的程序中，main()函数的最后一行实质语句必须是

  glutMainLoop();

  该语句使得程序进入一个无穷的事件循环
- 每经过事件循环一次，GLUT进行下述操作
  - 查看事件队列中的事件
  - 对于在队列中的每个事件，如果定义了相应的回调函数，GLUT就执行这个回调函数
  - 如果对该事件没有定义回调函数，那么就忽略该事件

显示回调函数

- 只要GLUT确定需要刷新窗口，那么就会执行显示回调函数，例如
  - 当第一次打开窗口的时候
  - 当改变了窗口形状的时候
  - 当重新露出了窗口的时候
  - 当用户程序决定需要改变显示内容的时候
- 在main()函数中，
  - glutDisplayFunc(display)注册要执行的显示回调函数display
  - 每个GLUT程序都必须有一个显示回调函数，必要时可以设置为空函数

标识重新显示

- 许多事件都会导致调用显示回调函数
  - 这会导致遍历一次事件循环的过程中多次执行显示回调函数
- 可以用下列方法避免这个问题

  glutPostRedisplay();

  这条语句设置一个标志
  - 当事件循环结束时，GLUT会检查是否设置了上述标志
  - 如果设置了标志，那么就会执行显示回调函数

使用定位设备

- 一般把定位设备假定成鼠标，不过定位设备也可以是跟踪球或数据板
- 有两类事件和定位设备相关联
  - 鼠标事件(mouse event)：当鼠标的一个按键被按下或释放。
    - 返回的信息包括：生成这个事件的鼠标按键、事件之后按键的状态（释放或按下）、窗口坐标系（原点位于左上角）中鼠标光标位置
  - 移动事件：如果鼠标在某个按键被按下时移动，就发生了移动事件(move event)；如果移动时没有按键被按下，这个事件称为被动移动事件(passive move event)

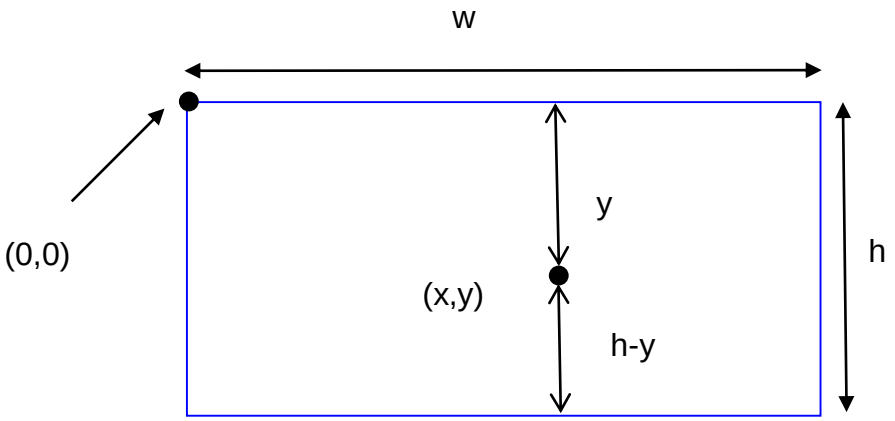
鼠标回调函数


```
glutMouseFunc(mouse)
void mouse(int button, int state, int x, int y)
```

- 其中button的值可能是
  GLUT_LEFT_BUTTON,
  GLUT_MIDDLE_BUTTON,
  GLUT_RIGHT_BUTTON
  表示哪个按钮导致了事件发生
- state表示相应按钮的状态:
  GLUT_UP, GLUT_DOWN
- x, y表示在窗口中的位置

定位

- 在屏幕上的位置通常是以像素为单位的，原点在左上角
  - 因为显示器自顶向下刷新显示内容
- 在OpenGL中使用的世界坐标系，其原点在左下角
- 在这个坐标系中的y坐标需要从窗口高度中减去回调函数返回的y值：y = h - y



获取窗口尺寸

- 为了完成y坐标的转换，需要知道窗口的尺寸
  - 在程序执行过程中，高度可能发生改变
  - 需要利用一个全局变量跟踪其变化
  - 新高度值返回给形状改变回调函数(见后)
  - 也可以用查询函数glGetIntegerv()和glGetFloatv() 获取，因为高度是状态的一部分

结束程序

- 在以前的程序中没有办法通过OpenGL结束当前程序
- 可以利用简单的鼠标回调函数做到这一点

```
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
        exit(0);
}
```

鼠标位置的应用

- 在要构造的例子中，每单击一次鼠标左按钮，就会在当前鼠标位置处画一个小方框
- 在这个例子中并没有用到显示回调函数，但是由于GLUT要求必须有一个显示回调函数，因此要定义一个空函数

```
display(){}
```

在指针处画方框


```
void mouse(int btn, int state, int x, int y) {
    if(btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(0);
    if(btn == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        drawSquare(x, y);
}

void drawSquare(GLint x, GLint y) {
    y = h-y; //转化y坐标
    glColor3ub( (char)rand()%256, (char)rand()%256,
                (char)rand()%256 );  // 随机颜色
    glBegin(GL_POLYGON);
        glVertex2f(x - size, y - size);
        glVertex2f(x + size, y - size);
        glVertex2f(x + size, y + size);
        glVertex2f(x - size, y + size);
    glEnd();
}
```

鼠标移动回调函数的应用

- 通过利用移动回调函数可以在不释放鼠标按钮的情况下，连续画一系列方框(square.c)
  - **glutMotionFunc(drawSquare)**
- 应用被动移动回调函数，可以不用按鼠标按钮就可以连续画方框
  - **glutPassiveMotionFunc(drawSquare)**

- **void glutMotionFunc(void (*f)(int x,int y))**
- **void glutPassiveMotionFunc(void (*f)(int x,int y))**

键盘的使用

当鼠标位于窗口内，并且键盘有某个键被按下或释放，就会产生键盘事件

```
glutKeyboardFunc(keyboard)
void keyboard(unsigned char key,
              int x, int y)
```

- 返回键盘上被按下键的ASCII码和鼠标位置
- 注意在GLUT中并不把释放键做为一个事件

```
void keyboard(unsigned char key, int x, int y)
{       // 按下Q、q或ESC键时，终止程序
        if(key == 'Q' || key == 'q' || key == '\27')
                exit(0);
}
```

特殊按键

- GLUT在glut.h中定义了特殊按键:
  - 功能键1: GLUT_KEY_F1
  - 向上方向键: GLUT_KEY_UP

```
void glutSpecialFunc(void (*func)(int key, int x, int y))
```

  - 回调函数内

```
if(key == GLUT_KEY_F1) ……
if(key == GLUT_KEY_UP) ……
```

修饰键 – Ctrl/Alt/Shift


```
int glutGetModifiers()
```

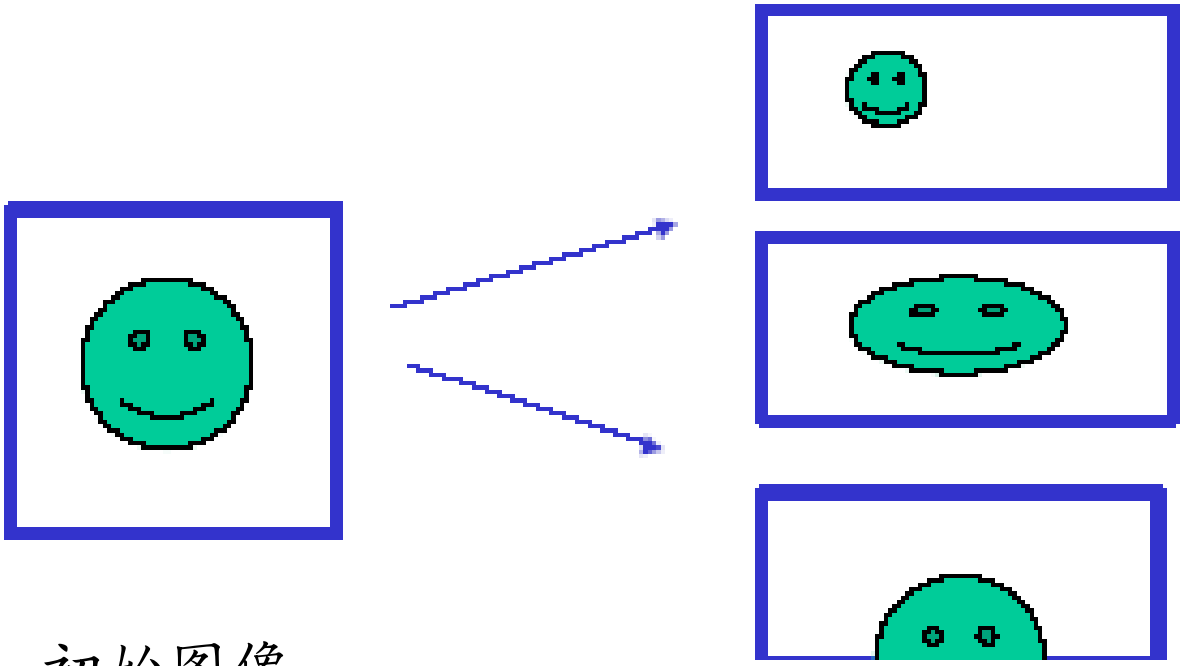
- 如果在鼠标或键盘事件产生时修饰键被按下，该函数返回GLUT_ACTIVE_SHIFT，GLUT_ACTIVE_CTRL，GLUT_ACTIVE_ALT的逻辑与结果
  - 可以利用单键或双键鼠标模拟三键鼠标
- 让Ctrl+c或Ctrl+C终止程序，在键盘回调函数中

```
if ((glutGetModifiers() == GLUT_ACTIVE_CTRL)
   &&(key == 'c' || key == 'C'))
     exit(0);
```

窗口形状的改变

- 通过拖动窗口的角点可以改变窗口的形状和尺寸
- 那么其中的显示内容该如何处理？
  - 必须由应用程序重新绘制
  - 有两种可能性
    - 显示原来内容的一部分
    - 通过强迫适应新窗口来显示所有内容
      - 可能会改变了显示长宽比率

形状改变的可能性



初始图像

改变后的图像

形状改变回调函数

glutReshapeFunc(reshape)

void reshape(int w, int h)

- 传入新窗口的宽度与高度（单位：像素）
- 回调函数执行后自动发送刷新显示事件，触发显示回调
- GLUT有一个缺省的形状改变的回调函数，调用glViewport(0,0,w,h)把视口设置为新窗口

- 这个回调函数是放置照相机函数的恰当地方，因为当窗口首次创建时就会调用它。
  - 当窗口形状尺寸发生改变时，可在回调函数里对观察条件进行修改

例子

通过保持视口和世界窗口的长宽比一样，使得物体不变形

```
void reshape(int w, int h) {
   glViewport(0, 0, w, h);  // 设置视口为整个窗口
   glMatrixMode(GL_PROJECTION); // 调整裁剪窗口
   glLoadIdentity();
   if(w<=h) // 宽小于高，裁剪矩形宽度置为4
       gluOrtho2D(-2.0, 2.0, -2.0*(GLfloat)h/(GLfloat)w,
                  2.0*(GLfloat)h/(GLfloat)w);
   else      // 高小于宽，裁剪矩形高度置为4
       gluOrtho2D(-2.0*(GLfloat)w/(GLfloat)h,
                  2.0*(GLfloat)w/(GLfloat)h, -2.0, 2.0);
   glMatrixMode(GL_MODELVIEW); /*return to modelview mode*/
}
```

动画

- 当通过显示回调函数重新绘制显示结果时，我们通常会首先清除整个窗口：

  glClear(GL_COLOR_BUFFER_BIT|…)

  然后再绘制已发生了变化的显示结果
- 问题：帧缓冲区中的信息在显示器上的显示结果出现了错位
  - 屏幕刷新速率与绘制速度不匹配屏幕
  - 图形系统可以同时向显存中写入内容和从中读出内容
- 从而我们会看到部分显示内容
  - 例：single_double.c 旋转的正方形
  - 显卡若启动了硬件加速功能，不会发现上述问题

双缓冲区

- 不只用一个颜色缓冲区，而是应用两个缓冲区
  - 前缓冲区：显示它的内容，但不向它写入内容
  - 后缓冲区：写入内容，不显示
- 程序在main()函数中请求使用双缓冲区
  - **glutInitDisplayMode(GL_RGB | GL_DOUBLE)**
  - 在显示回调函数结束之前，交换两个缓冲区

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT|…);
    // 绘制图形
   glutSwapBuffers();
}
```

双缓冲区

- 可用函数glDrawBuffer(GLenum mode)指定要写入的颜色缓冲区
  - 默认是GL_FRONT（单缓冲区）或GL_BACK（双缓冲区）
  - glDrawBuffer(GL_FRONT_AND_BACK)，同时绘制到前后缓冲区

空闲回调的应用

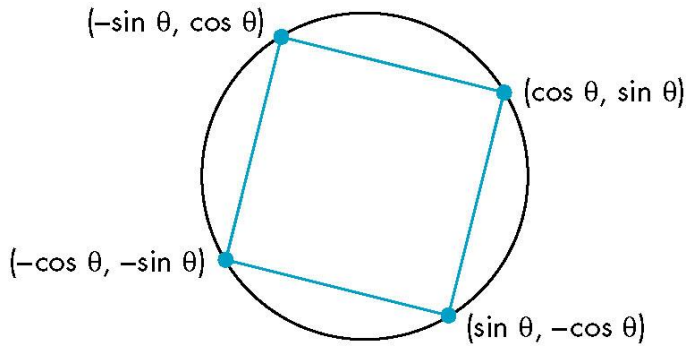
- 当在事件队列中没有事件时，就执行该回调函数
  - **glutIdleFunc(idle)**
  - 在动画中非常有用

```
void idle() {
    // 改变一些内容
    t += dt;
    glutPostRedisplay();
}
void display() {
    glClear(…);
    // 根据t绘制图形
    glutSwapBuffers();
}
```

旋转的正方形

- 将正方形的4个顶点定义为圆周上的4个等间隔点

```
GLfloat theta = 0.0; // 全局变量
void display() {
  glclear(GL_COLOR_BUFFER_BIT);
  Glfloat t1 = theta / (3.14159 / 180.0);//度转换为弧度
  glBegin(GL_POLYGON);
    glVertex2f(cos(t1),sin(t1));
    glVertex2f(-sin(t1),cos(t1));
    glVertex2f(-cos(t1),-sin(t1));
    glVertex2f(sin(t1),-cos(t1));
  glEnd();
  glFlush();
}
```



旋转的正方形

- 空闲回调函数：当事件队列为空时被执行

```
void glutIdleFunc(void (* f)(void))
```

- 利用空闲回调函数改变theta值，使正方形绕原点旋转
  - 在main()函数里，注册空闲回调函数：

```
glutIdleFunc(idle);
```

  - 空闲回调函数的定义：

```
void idle() {
    theta += 2.0;
    if (theta > 360.0) theta -= 360.0;
    glutPostRedisplay();     // 请求重绘
}
```

全局变量的应用

- 所有GLUT回调函数的调用形式是固定的
  - `void display();`
  - `void mouse(int button, int state, int x, int y);`
- 为了向回调函数传递信息，必须应用全局变量

```
float t; //全局变量
void display() {
        // 根据t绘制图形
}
```

第四章 输入与交互

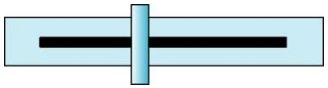
4.1 输入设备

4.2 事件驱动编程

4.3 弹出式菜单

4.4 更多的交互

构件(widgets)

- 许多窗口系统提供了一个工具包或者一组库函数用来建立用户界面，界面中用到了一些特殊类型的窗口，称为构件(widgets)
- 构件集中包含下述工具:
  - 菜单
  - 滑动条 
  - 对话框
  - 文本输入框
- 但上述构件通常都是与平台相关的
- GLUT只提供了包含菜单在内的很少几个构件

菜单

- GLUT支持弹出式菜单
  - 可以有子菜单
- 创建弹出式菜单的三个步骤
  - 定义菜单内各条目
  - 定义每个菜单项的行为
    - 如果条目被选择执行的操作
  - 把菜单连接到鼠标按钮上

菜单函数

int **glutCreateMenu**(void (*f)(int value))

- 创建一个使用回调函数f()的菜单，并返回菜单的整数标识符

void **glutAddMenuEntry**(char *name, int value)

- 为当前菜单增加一个名为name的菜单项；value值在选中时返回给菜单回调函数

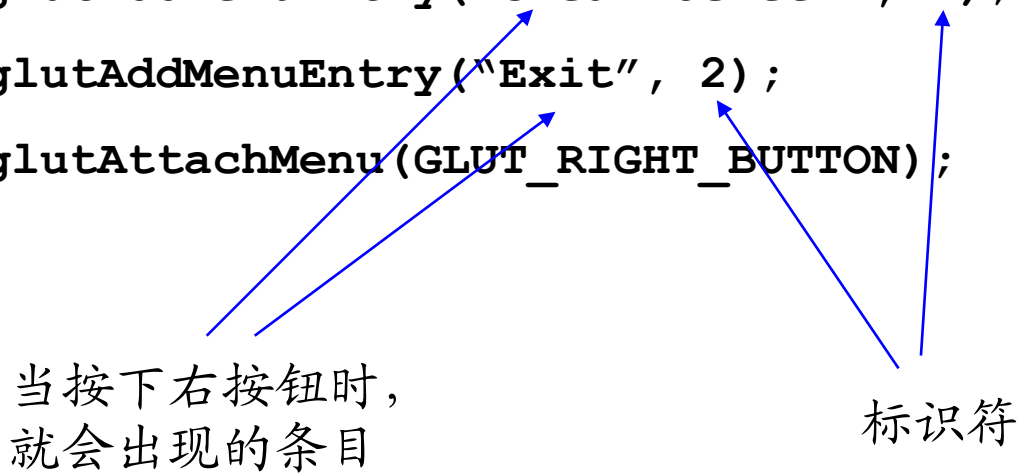
void **glutAttachMenu**(int button)

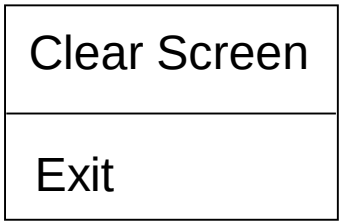
- 将当前菜单关联到鼠标按钮button上 (GLUT_RIGHT_BUTTON、GLUT_MIDDLE_BUTTON、GLUT_LEFT_BUTTON)

定义一个简单菜单

- 在main()函数中

```
menu_id = glutCreateMenu(mymenu);
glutAddMenuEntry("Clear Screen", 1);
glutAddMenuEntry("Exit", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```





菜单的行为

- 菜单回调函数

```
void mymenu(int value) {
        if(value==1) glClear(GL_COLOR_BUFFER_BIT);
        if(value==2) exit(0);
}
```

- 添加子菜单

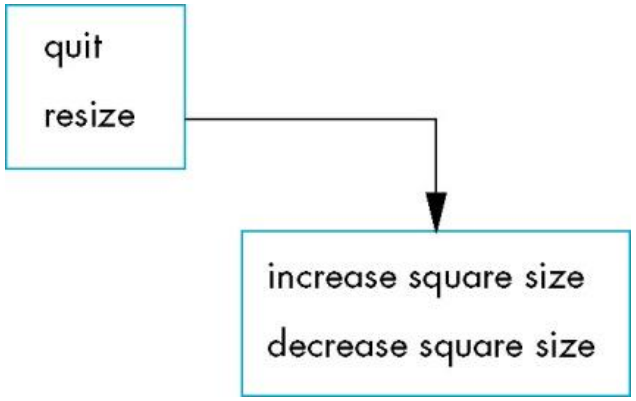
```
void glutAddSubMenu(char *submenu_name,
    int submenu_id)
```

  - 增加一个子菜单项submenu_name作为当前菜单的一项，子菜单创建时返回的标识符为submenu_id
  - 必须先创建子菜单

父菜单中一项

子菜单

在main()函数中创建一个层级菜单




```
int sub_menu;

sub_menu = glutCreateMenu(processSizeMenu);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("decrease square size", 3);

glutCreateMenu(processTopMenu);
glutAddMenuEntry("quit", 1);
glutAddSubMenu("resize", sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

子窗口与多窗口

- int **glutCreateWindow**(char *name)
  - 创建一个顶层窗口name，并为其返回一个整数标识符
- void **glutDestroyWindow**(int id)
  - 销毁标识符为id的窗口
- void **glutSetWindow**(int id)
  - 把当前窗口设为标识符为id的窗口
- int **glutCreateSubWindow**(int parent, int x, int y, int width, int height)
  - 为parent窗口创建一个子窗口，返回子窗口的标识符。子窗口原点位于(x,y)，宽度为width，高度为height
- void **glutPostWindowRedisplay**(int id)
  - 通知标识符为id的窗口重新显示

GLUT中其它的函数

- 动态窗口
  - 在执行期间创建或关闭窗口
- 在执行期间改变回调函数
- 计时器
- 字体
  - glutBitmapCharacter
  - glutStrokeCharacter
- GLUT教程
  http://www.lighthouse3d.com/opengl/glut/

4.1 输入设备

4.2 事件驱动编程

4.3 弹出式菜单

4.4 更多的交互

4.4 更多的交互

- 选择与拾取
- 橡皮条操作
- 显示列表

拾取(picking)

- 从屏幕上显示结果中识别用户定义的对象
- 原则上说要达到这个目标是很简单的，因为鼠标可以提供位置信息，我们应该能够根据位置确定对应的是哪个对象
- 实际操作中的困难
  - 流水线图形体系结构是单向的，很难从二维屏幕返回到三维世界
  - 二维屏幕，三维世界，也使问题变得复杂
  - 离位置多近可以认为选择了对象？

三种方法

- 命中列表(hit list)
  - 最一般性的方法，实现也最困难
  - OpenGL 3.1及更高版本已废弃该功能
- 矩形映射（包围盒方法）
  - 容易实现
- 利用后缓冲区或其它缓冲区，用不同颜色绘制不同的对象，通过颜色来识别对象（推荐方法）

绘制模式

- OpenGL使用三种模式进行绘制

GLint ***glRenderMode****(GLenum mode)*

  - GL_RENDER 渲染模式：正常绘制到帧缓冲区中(缺省模式)
  - GL_FEEDBACK 反馈模式：提供已绘制的图元列表，但并不输出到帧缓冲区中
  - GL_SELECT 选择模式：视景体中的每个图元产生一个命中记录(hit record)，这个记录放到名字堆栈(name stack)中，稍后要被检测

选择模式中用到的函数

void ***glSelectBuffer****(GLsizei size, GLuint *buffer);*

- 指定用于返回选择数据的数组。
  - *buffer* 指向的无符号整型数组用于存放选择数据（命中纪录）
  - *size* 表示数组的大小
  - 在进入选择模式前，必须调用glSelectBuffer()

选择模式中用到的函数

void ***glInitNames****(void);*

- 初始化名字堆栈，使其成为空栈

void ***glPushName****(GLuint name);*

- 把*name*压入到名字堆栈。名字堆栈的深度因实现而异，至少能容纳64个名字。

void ***glPopName****(void);*

- 从名字堆栈中弹出栈顶名字

void ***glLoadName****(GLuint name)*

- 用*name*替换名字堆栈的栈顶元素。应该至少调用一次glPushName()函数在堆栈中压入一些东西，然后再调用glLoadName()以避免堆栈是空的。
- 如果要用程序并不是处于选择模式下，对glPushName(), glPopName() 和glLoadName()的调用会被忽略

创建名字栈


```
glInitNames();
glPushName(0);

glPushMatrix();   /* save the current transformation state */

    /* create your desired viewing volume here */

    glLoadName(1);
    drawSomeObject();
    glLoadName(2);
    drawAnotherObject();
    glLoadName(3);
    drawYetAnotherObject();
    drawJustOneMoreObject();

glPopMatrix ();   /* restore the previous transformation state*/
```

命中纪录(hit record)

- 在选择模式下，与视景体相交的图元产生一次选择命中(selection hit)，OpenGL会在选择数组中写入一条命中纪录(hit record)
  - 共享相同名字的图元（例如某个对象由多个图元构成）不会产生多条命中纪录
- 命中纪录由4部分组成，按顺序如下：
  - 命中发生时，名字堆栈中的名称数量
  - 自从上一次命中纪录后，与视景体相交的图元的所有顶点的最小和最大窗口坐标z值(深度值)。深度值（范围在[0,1]）与2^{32}-1相乘后四舍五入到最接近的无符号整数
  - 名字堆栈的内容，从底部元素开始

使用选择模式

使用选择机制，需要执行如下步骤：(select.c)

- 用glSelectBuffer()函数指定用于返回命中纪录的数组
- 调用glRenderMode(GL_SELECT)进入选择模式
- 使用glInitNames()和glPushName()对名字栈进行初始化
- 定义用于选择的视景体。通常该视景体不同于最初用于渲染场景的视景体，因此需要调用glPushMatrix()和glPopMatrix()函数保存和恢复当前变换状态。
- 交替调用图元绘制函数和名字栈操纵函数，为每个感兴趣图元分配一个适当的名称
- 再次调用glRenderMode()函数退出选择模式，并处理返回的旋转数据（命中纪录）

选择模式与拾取

- 在选择模式中不能进行拾取(picking)，因为在视景体中的每个图元都会产生一个命中
- 拾取和常规选择的区别:
  - 拾取通常是由输入设备（例如鼠标左键）触发
  - 使用工具函数gluPickMatrix()来把一个特殊的拾取矩阵与当前投影矩阵相乘。该函数应该在乘以标准的投影矩阵（例如调用glOrtho() 或gluPerspective()）之前调用

设置拾取矩阵

void ***gluPickMatrix****(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4])*

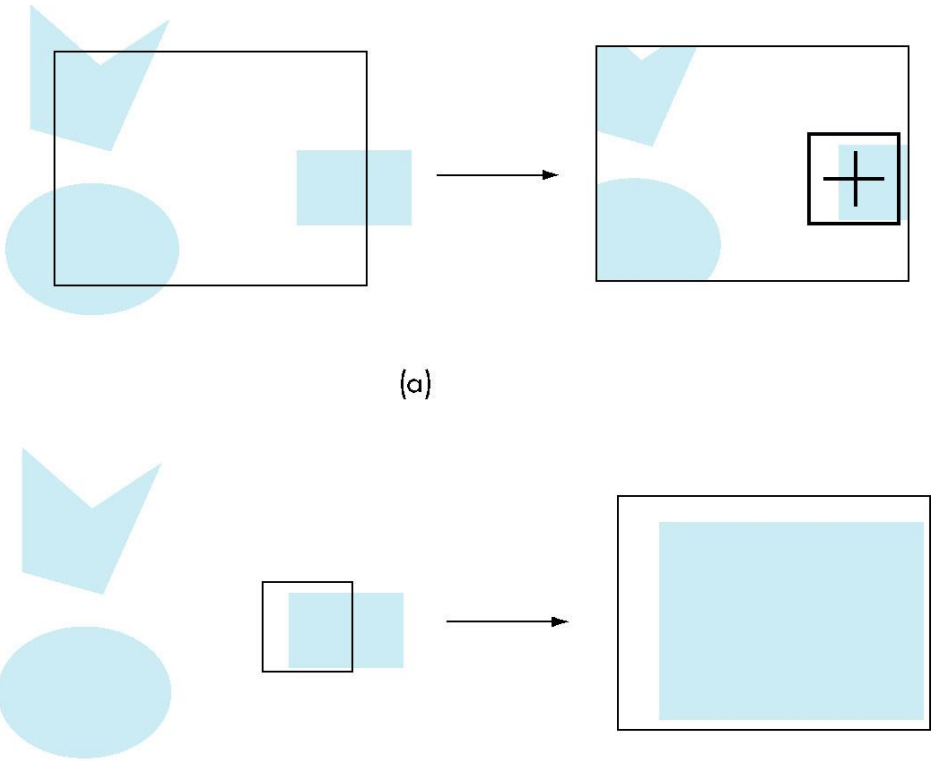
- 创建一个投影矩阵把绘图限制在视口的一个小区域内，并把该矩阵与当前的投影矩阵相乘
- 拾取区域以窗口坐标(*x*,*y*)为中心，宽度为*width*，高度为*height*（屏幕坐标）
- *viewport[]*表示当前的视口边界，可以通过如下方式来获取:

```
glGetIntegerv(GL_VIEWPORT, GLint *viewport);
```

定义拾取矩阵


```
glMatrixMode (GL_PROJECTION);
glPushMatrix ();
glLoadIdentity ();
gluPickMatrix (...);
gluPerspective, glOrtho, gluOrtho2D, or glFrustum
   /* ... draw scene for picking ; perform picking ... */
glPopMatrix();
```

二维拾取的例子



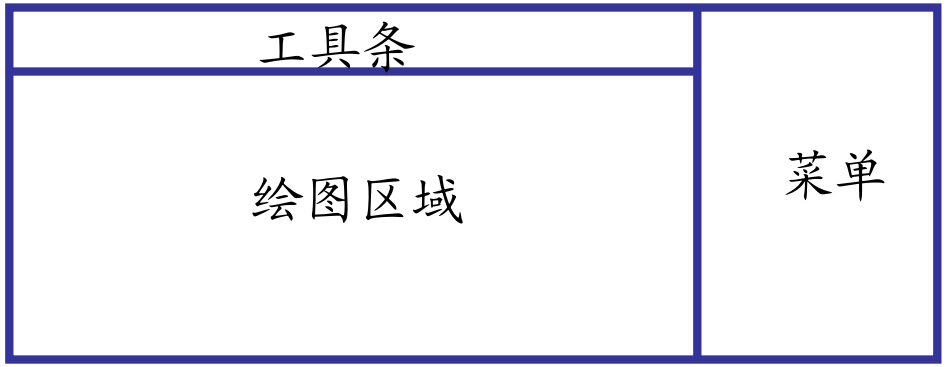
(a)

(b)

- demo: pick.c

应用屏幕上的区域

- 在许多应用程序中，屏幕采用简单的矩形分划
  - 例如：



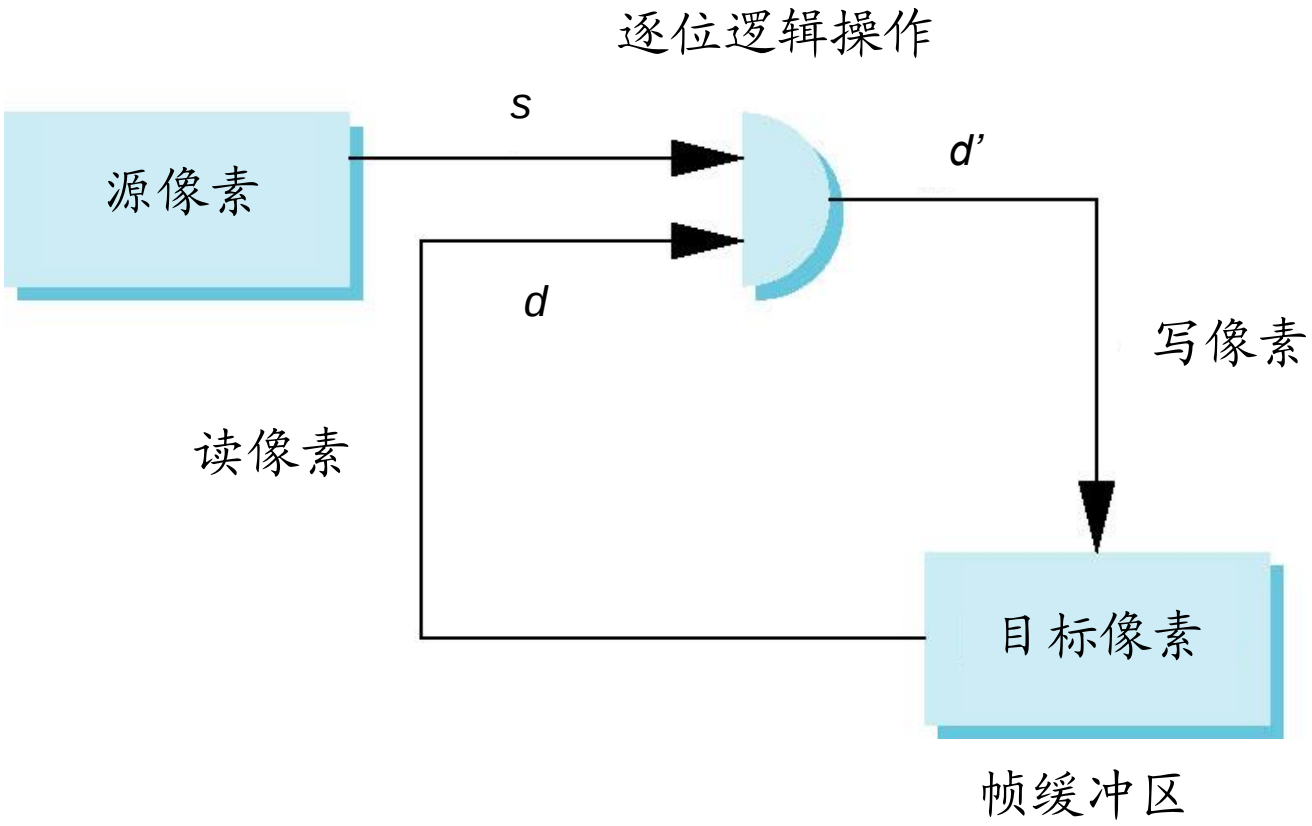
- 这时，相比于应用选择模式进行拾取而言，通过查看鼠标位置，确定屏幕的区域就更简单了

通过另外的缓冲区和颜色进行拾取

- 对于很少数目的对象，可以给每个对象赋以唯一的颜色(有时是在颜色索引模式中)
- 把场景绘制到不是前缓冲区的一个颜色缓冲区中，这样就不会看到显示出来的结果
- 获取鼠标位置，利用glReadPixels()读取缓冲区中鼠标位置对应的颜色
- 根据返回的颜色确定是哪个对象

- OpenGL 3.1及更高版本的推荐方式

像素的写入模式



XOR写入模式

- 缺省的写入模式是复制模式(copy)或替换模式(replacement)，直接用源像素取代目标像素，即 *d*'=*s*
  - 用这种方法不能绘制一条临时直线，因为我们不能用快速简单的方法恢复在临时直线下面的内容
- 异或操作(XOR)：$d' = d \oplus s$
  - 相同值为0，不同值为1
  - $(d \oplus s) \oplus s = d$
  - 因此如果应用XOR模式画一条直线，那么只要在原地再画一遍这条直线就可以删除这条直线

16种写入模式

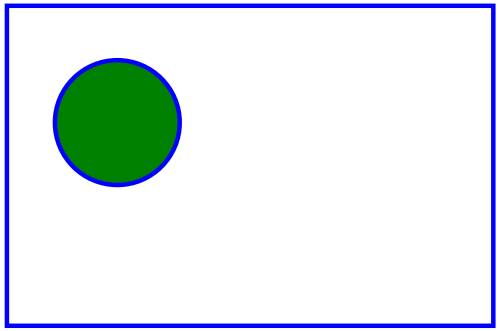
s	d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

替换　　XOR

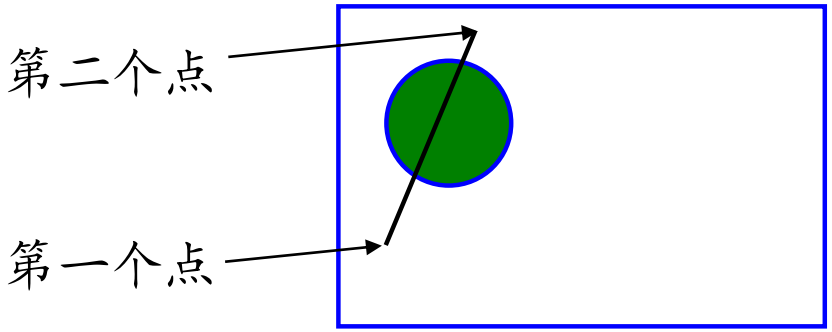
橡皮条式绘图

- 切换到XOR写入模式
- 绘制对象
  - 利用第一次单击鼠标固定线段的一个端点，然后再利用鼠标移动的回调函数连续更新第二个端点
  - 每次鼠标移动的时候，重新画一遍原来的直线从而删除它，然后再从固定的第一个端点到新的第二个端点之间画一条直线
  - 最后切换回到正常的绘图模式并绘制直线
  - 也适用于其它对象：矩形、圆

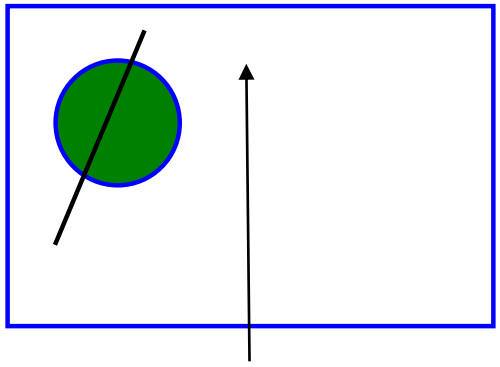
橡皮条型直线



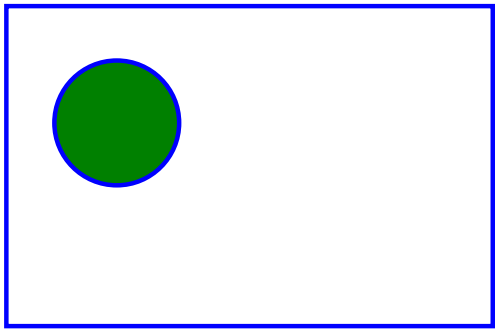
初始显示结果



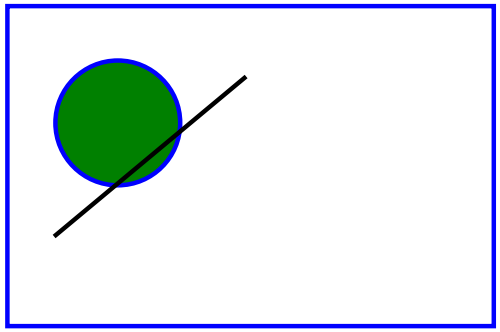
利用鼠标在XOR模式中画一条直线



把鼠标移到新的位置



原来的直线利用XOR重新画一遍



新的直线利用XOR画出来

OpenGL的逻辑操作

- 在两位之间有16种可能的逻辑操作
- OpenGL支持所有的这16种操作
  - 必须首先启动逻辑操作
    - glEnable(GL_COLOR_LOGIC_OP)
  - 然后选择逻辑操作类型
    - glLogicOp(GL_XOR)
    - glLogicOp(GL_COPY) (默认值)

- demo: rubberbanding.c

即时模式与保留模式

- 在标准的OpenGL程序中，一旦某个对象被显示输出，与它相关的信息不会再系统中保留；为了重新显示它，需要再执行一次相应的代码
  - 这就是即时模式(immediate mode)图形
  - 如果对象相当复杂，而且通过网络传输，那么这就可能导致系统非常慢
- 另外一种模式就是首先定义对象，然后以某种形式把它保存下来，从而更容易重新显示
  - 这就是保留模式(retained mode)图形
  - 在OpenGL中是通过显示列表(display lists)实现的

显示列表

- 从概念上类似于图形文件
  - 必须首先定义列表的名称并创建它
  - 向列表中添加内容
  - 关闭列表
- 在客户-服务的体系环境中，显示列表是放在服务器(显示服务器)一方
  - 这样不必通过网络每次发送原来的图元定义就可以重新显示

显示列表相关函数

*GLuint **glGenLists**(GLsizei range);*

- 分配*range*个连续的以前未分配的显示列表索引。
  - 如果返回值为n，则索引值为n,n+1,…,n+range-1
  - 返回的所有索引都被标记为空和已使用，后续的glGenLists()调用不会再返回这些值，除非它们已被删除

*void **glNewList** (GLuint list, GLenum mode);*

- 创建或替换显示列表*list*。在它之后的OpenGL函数将依序存储在一个显示列表中，直到调用glEndList()。有些OpenGL函数无法编译到显示列表中，而将立即执行。
  - *list*是正整数，唯一地标识这个显示列表
  - *mode*是编译模式：GL_COMPILE表示把显示列表放到服务器，但不显示；GL_COMPILE_AND_EXECUTE指创建时即被显示

*void **glEndList** (void);*

- 标识一个显示列表定义的结束

显示列表相关函数

void ***glCallList*** *(GLuint list);*

- 执行名为*list*的显示列表。如果*list*并未定义，这个函数不执行任何任务。

GLboolean ***glIsList****(GLuint list);*

- 如果*list*已被用于定义显示列表，函数返回GL_TURE;否则返回GL_FALSE

void ***glDeleteLists****(GLuint list, GLsizei range);*

- 删除自*list*开始的*range*个显示列表
  - 显示列表一经创建便不可修改，只能删除

显示列表的使用

- 创建显示列表

```
GLuint id;
void init(void) {
        id = glGenLists(1);
        glNewList(id, GL_COMPILE);
        //其它OpenGL绘图子程序或者绘图命令
        glEndList();
}
```

- 调用已创建的显示列表

```
void display(void) {
        glCallList(id);
}
```

- demo: list.c

显示列表中的状态

- 绝大多数OpenGL函数都可放在显示列表中
- 在显示列表中发生的状态改变在列表执行结束后仍然起作用
- 可以在进入显示列表时调用glPushAttrib(GL_ALL_ATTRIB_BITS)和glPushMatrix()把属性和变换矩阵压入各自的栈中；在离开显示列表前调用glPopAttrib() 和glPopMatrix()恢复属性和变换矩阵来避免这个问题

层级显示列表

- 层级显示列表：在glNewList()和glEndList()之间通过调用glCallList()执行其它显示列表的显示列表
  - 显示列表的嵌套层次有限制，至少是64，取决于实现
- 考虑一个汽车模型
  - 为底盘创建显示列表
  - 为车轮创建显示列表

```
glNewList( CAR, GL_COMPILE );
      glCallList( CHASSIS );
      glTranslatef( … );
      glCallList( WHEEL );
      glTranslatef( … );
      glCallList( WHEEL );
            …
glEndList();
```