OpenGL 游戏程序设计

第一章 OpenGL概述

一、OpenGL的特点

1、与C语言紧密结合。

OpenGL命令最初就是用C语言函数来进行描述的,对于学习过C语言的人来讲,OpenGL是容易理解和学习的。如果你曾经接触过TC的graphics.h,你会发现,使用OpenGL作图甚至比TC更加简单。

• 2、强大的可移植性

微软的Direct3D虽然也是十分优秀的 图形API, 但它只用于Windows系统(现 在还要加上一个XBOX游戏机)。而 OpenGL不仅用于 Windows, 还可以用于 Unix/Linux等其它系统,它甚至在大型计 算机、各种专业计算机(如: 医疗用显 示设备)上都有应用。并且, OpenGL 的 基本命令都做到了硬件无关, 甚至是平 台无关。

• 3、高性能的图形渲染。

OpenGL是一个工业标准,它的技术紧跟时代,现今各个显卡厂家无一不对OpenGL提供强力支持,激烈的竞争中使得OpenGL性能一直领先

OpenGL程序准备工作

第一步,选择一个编译环境

 现在Windows系统的主流编译环境有 Visual Studio, Broland C++ Builder等都 是支持OpenGL的。但这里我们选择 Visual C++作为学习OpenGL的环境。

- 第二步,安装GLUT工具包
- GLUT不是OpenGL所必须的,但它会给我们的 学习带来一定的方便,推荐安装。
- Windows环境下安装GLUT的步骤:
- 1、将下载的压缩包解开,将得到5个文件
- 2、在"我的电脑"中搜索"gl.h",并找到其 所在文件夹。把解压得到的glut.h放到这个文件 夹。
- 3、把解压得到的glut.lib和glut32.lib放到静态函数库所在文件夹(如果是Visual C++,则应该是其安装目录下面的"VC\lib"文件夹)。
- 4、把解压得到的glut.dll和glut32.dll放到操作系统目录下面的system32文件夹内。(典型的位置为: C:\Windows\System32)

- · 第三步,建立一个OpenGL工程
- 这里以Visual C++ 6.0为例。
- 选择File->New->Project, 然后选择Win32 Console Application, 选择一个名字, 然后按OK。
- 在谈出的对话框左边点Application Settings,找到Empty project并勾上,选择Finish。
- 然后向该工程添加一个代码文件。

第一个OpenGL程序

见文本文件"第一个OpenGL文本文件"

第二章 基本图元绘制

OpenGL函数形式

• 字母表示参数的类型:

s表示16位整数(OpenGL中将这个类型定义为GLshort);

i表示32位整数(OpenGL中将这个类型定义为GLint和GLsizei);

f表示32位浮点数(OpenGL中将这个类型定义为GLfloat和GLclampf);

d表示64位浮点数(OpenGL中将这个类型定义 为GLdouble和GLclampd)。

v表示传递的几个参数将使用指针的方式,见 下面的例子。

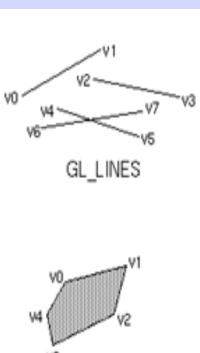
这些函数除了参数的类型和个数不同以外,功能是相同的。

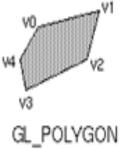
- 例如,以下五个代码段的功能是等效的:
 - (一) glVertex2i(1, 3);
 - (\Box) glVertex2f(1.0f, 3.0f);
 - (Ξ) glVertex3f(1.0f, 3.0f, 0.0f);
 - (四) glVertex4f(1.0f, 3.0f, 0.0f, 1.0f);
 - (五) GLfloat VertexArr3[] = $\{1.0f, 3.0f, 0.0f\}$;

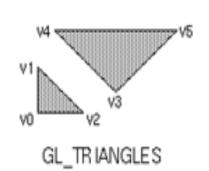
glVertex3fv(VertexArr3);

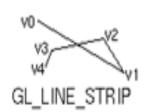
• OpenGL要求:指定顶点的命令必须包含在glBegin函数之后,glEnd函数之前(否则指定的顶点将被忽略)。并由glBegin来指明如何使用这些点。

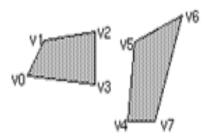
glBegin支持的方式除了GL_POINTS和GL_LINES,还有GL_LINE_STRIP,GL_LINE_LOOP,GL_TRIANGLES,GL_TRIANGLE_STRIP,GL_TRIANGLE_STRIP,GL_TRIANGLE_FAN等,每种方式的大致效果见下图:



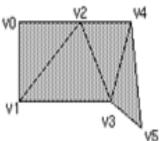








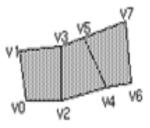
GL_QUADS



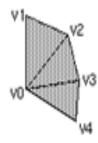
GL_TRIANGLE_STRIP



GL_LINE_LOOP



GL_QUAD_STRIP



GL_TRIANGLE_FAN

• 三个课堂练习,见三个相关文本文件

第三章 图元绘制的深入

- 1、关于点
- · 点的大小默认为1个像素,但也可以改变之。 改变的命令为glPointSize,其函数原型如下:
- void glPointSize(GLfloat size);
- size必须大于0.0f, 默认值为1.0f, 单位为"像素"。
- 注意:对于具体的OpenGL实现,点的大小都有个限度的,如果设置的size超过最大值,则设置可能会有问题。

- 2、关于直线
- (1) 直线可以指定宽度:
- void glLineWidth(GLfloat width);
- 其用法跟glPointSize类似。
- (2)画虚线。
- 首先,使用glEnable(GL_LINE_STIPPLE);来启动虚线模式(使用glDisable(GL_LINE_STIPPLE)可以关闭之)。
- 然后,使用glLineStipple来设置虚线的样式。
- void glLineStipple(GLint factor, GLushort pattern);
- pattern是由1和0组成的长度为16的序列,从最低位开始看,如果为1,则直线上接下来应该画的factor个点将被画为实的;如果为0,则直线上接下来应该画的factor个点将被画为虚的。

PATTERN	FACTOR
0x00FF	1 ——— ———
0x00FF	2 ———
0x0C0F	1 — — — — —
0x0C0F	3 ———
OxAAAA	1
0xAAAA	2
0xAAAA	3 — — — — — —
0xAAAA	4 — — — — —

- 3、关于多边形
 - (1) 多边形的两面以及绘制方式。
- 虽然我们目前还没有真正的使用三维坐标来画图,但是建立一些三维的概念还是必要的。
- 从三维的角度来看,一个多边形具有两个面。每一个面都可以设置不同的绘制方式:填充、只绘制边缘轮廓线、只绘制顶点,其中"填充"是默认的方式。可以为两个面分别设置不同的方式。
- glPolygonMode(GL_FRONT, GL_FILL); // 设置正面为填充方式
- glPolygonMode(GL_BACK, GL_LINE); // 设置反面为边缘绘制方式
- glPolygonMode(GL_FRONT_AND_BACK, GL POINT); // 设置两面均为顶点绘制方式

- (2) 反转
- 一般约定为"顶点以逆时针顺序出现在屏幕上的面"为"正面", 另一个面即成为"反面"。生活中常见的物体表面,通常都可以 用这样的"正面"和"反面","合理的"被表现出来(请找一 个比较透明的矿泉水瓶子,在正对你的一面沿逆时针画一个圆, 并标明画的方向,然后将背面转为正面,画一个类似的圆,体会 一下"正面"和"反面"。你会发现正对你的方向,瓶的外侧是 正面,而背对你的方向,瓶的内侧才是正面。正对你的外侧则是反面。这样一来,同样属于"瓶的外侧"这个表 面,但某些地方算是正面,某些地方却算是反面了)。
- 可以通过glFrontFace函数来交换"正面"和"反面"的概念。
- glFrontFace(GL_CCW); // 设置CCW方向为"正面",CCW即CounterClockWise,逆时针
- glFrontFace(GL_CW); // 设置CW方向为"正面",CW即ClockWise,顺时针

- (3)剔除多边形表面
- 在三维空间中,一个多边形虽然有两个面,但我们无法看见背面的那些多边形,而一些多边形虽然是正面的,但被其他多边形所遮挡。如果将无法看见的多边形和可见的多边形同等对待,无疑会降低我们处理图形的效率。在这种时候,可以将不必要的面剔除。
- 首先,使用glEnable(GL_CULL_FACE);来启动剔除功能(使用glDisable(GL_CULL_FACE)可以关闭之)
- 然后,使用glCullFace来进行剔除。
- glCullFace的参数可以是GL_FRONT, GL_BACK或者 GL_FRONT_AND_BACK, 分别表示剔除正面、剔除 反面、剔除正反两面的多边形。
- 注意:剔除功能只影响多边形,而对点和直线无影响。例如,使用glCullFace(GL_FRONT_AND_BACK)后,所有的多边形都将被剔除,所以看见的就只有点和直线。

- (4) 镂空多边形
- 直线可以被画成虚线,而多边形则可以进行镂空。
- 首先,使用glEnable(GL_POLYGON_STIPPLE);来启动 镂空模式(使用glDisable(GL_POLYGON_STIPPLE)可 以关闭之)。
- 然后,使用glPolygonStipple来设置镂空的样式。
- void glPolygonStipple(const GLubyte *mask);
- 其中的参数mask指向一个长度为128字节的空间,它表示了一个32*32的矩形应该如何镂空。其中:第一个字节表示了最左下方的从左到右(也可以是从右到左,这个可以修改)8个像素是否镂空(1表示不镂空,显示该像素;0表示镂空,显示其后面的颜色),最后一个字节表示了最右上方的8个像素是否镂空。

第四章 颜色模型

1. RGBA颜色

RGBA模式中,每一个像素会保存以下数据: R值(红色分量)、G值(绿色分量)、B值(蓝色分量)和A值(alpha分量)。其中红、绿、蓝三种颜色相组合,就可以得到我们所需要的各种颜色,而alpha不直接影响颜色,它将留待以后介绍。

在RGBA模式下选择颜色是十分简单的事情,只需要一个函数就可以搞定。

glColor*系列函数可以用于设置颜色,其中三个参数的版本可以指定R、G、B的值,而A值采用默认;四个参数的版本可以分别指定R、G、B、A的值。例如:void glColor3f(GLfloat red, GLfloat green, GLfloat blue);void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

将浮点数作为参数,其中0.0表示不使用该种颜色,而1.0表示将该种颜色用到最多。例如:glColor3f(1.0f, 0.0f, 0.0f);表示不使用绿、蓝色,而将红色使用最多,于是得到最纯净的红色

glColor3f(0.0f, 1.0f, 1.0f); 表示使用绿、蓝色到最多,而不使用红色。混合的效果就是浅蓝色。

glColor3f(0.5f, 0.5f, 0.5f); 表示各种颜色使用 一半, 效果为灰色。

注意: 浮点数可以精确到小数点后若干位,这并不表示计算机就可以显示如此多种颜色。实际上, 计算机可以显示的颜色种数将由硬件决定。如果OpenGL找不到精确的颜色, 会进行类似"四舍五入"的处理。

```
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 1.0f, 1.0f);
    glRectf(-0.5f, -0.5f, 0.5f, 0.5f);
    glFlush();
}
```

注意: glColor系列函数,在参数类型不同时,表示"最大"颜色的值也不同。

采用f和d做后缀的函数,以1.0表示最大的使用。

采用b做后缀的函数,以127表示最大的使用。 采用ub做后缀的函数,以255表示最大的使用。 采用s做后缀的函数,以32767表示最大的使用。 采用us做后缀的函数,以65535表示最大的使用。

2、索引颜色

在索引颜色模式中,OpenGL需要一个颜色表。这个表就相当于画家的调色板:虽然可以调出很多种颜色,但同时存在于调色板上的颜色种数将不会超过调色板的格数。试将颜色表的每一项想象成调色板上的一个格子:它保存了一种颜色。

在使用索引颜色模式画图时,我说"我把第i种颜色设置为某某",其实就相当于将调色板的第i格调为某某颜色。"我需要第k种颜色来画图",那么就用画笔去蘸一下第k格调色板。

颜色表的大小是很有限的,一般在256~4096之间,且 总是2的整数次幂。在使用索引颜色方式进行绘图时, 总是先设置颜色表,然后选择颜色。

2.1、选择颜色

使用glIndex*系列函数可以在颜色表中选择颜色。其中最常用的可能是glIndexi, 它的参数是一个整形。void glIndexi(GLint c);

使用glShadeModel函数可以关闭这种计算,如果顶点的颜色不同,则将顶点之间的其它点全部设置为与某一个点相同。 (直线以后指定的点的颜色为准,而多边形将以任意顶点的颜色为准,由实现决定。)为了避免这个不确定性,尽量在多边形中使用同一种颜色。

```
glShadeModel的使用方法:
glShadeModel(GL_SMOOTH); // 平滑方式, 这也是默认方式
glShadeModel(GL_FLAT); // 单色方式
```

第五章 变换

我们生活在一个三维的世界——如果要观察一个物体,我们可以:

- 1、从不同的位置去观察它。(视图变换)
- 2、移动或者旋转它,当然了,如果它只是计算机里面的物体,我们还可以放大或缩小它。 (模型变换)
- 3、如果把物体画下来,我们可以选择:是否需要一种"近大远小"的透视效果。另外,我们可能只希望看到物体的一部分,而不是全部(剪裁)。(投影变换)
- 4、我们可能希望把整个看到的图形画下来, 但它只占据纸张的一部分,而不是全部。(视 口变换)

- OpenGL变换实际上是通过矩阵乘法来实现。 无论是移动、旋转还是缩放大小,都是 通过在当前矩阵的基础上乘以一个新的 矩阵来达到目的。
- OpenGL可以在最底层直接操作矩阵,不过 作为初学,这样做的意义并不大。

1、模型变换和视图变换

从"相对移动"的观点来看,改变观察点的位置与方向和改变物体本身的位置与方向具有等效性。在OpenGL中,实现这两种功能甚至使用的是同样的函数。

由于模型和视图的变换都通过矩阵运算来实现, 在进行变换前,应先设置当前操作的矩阵为 "模型视图矩阵"。设置的方法是以

GL_MODELVIEW为参数调用glMatrixMode函数,像这样:

glMatrixMode(GL_MODELVIEW);

通常,我们需要在进行变换前把当前矩阵设置为单位矩阵。这也只需要一行代码:glLoadIdentity();

进行模型和视图变换,主要涉及到三个函数:

glTranslate*,把当前矩阵和一个表示移动物体的矩阵相乘。三个参数分别表示了在三个坐标上的位移值。

glRotate*,把当前矩阵和一个表示旋转物体的矩阵相乘。物体将绕着(0,0,0)到(x,y,z)的直线以逆时针旋转,参数angle表示旋转的角度。glScale*,把当前矩阵和一个表示缩放物体的矩阵相乘。x,y,z分别表示在该方向上的缩放比例。

2、投影变换

投影变换就是定义一个可视空间,可视空间以外的物体不会被绘制到屏幕上。(注意,从现在起,坐标可以不再是-1.0到1.0了!)

OpenGL支持两种类型的投影变换,即透视投影和正投影。投影也是使用矩阵来实现的。如果需要操作投影矩阵,需要以GL_PROJECTION为参数调用。dMatrixMode函数。

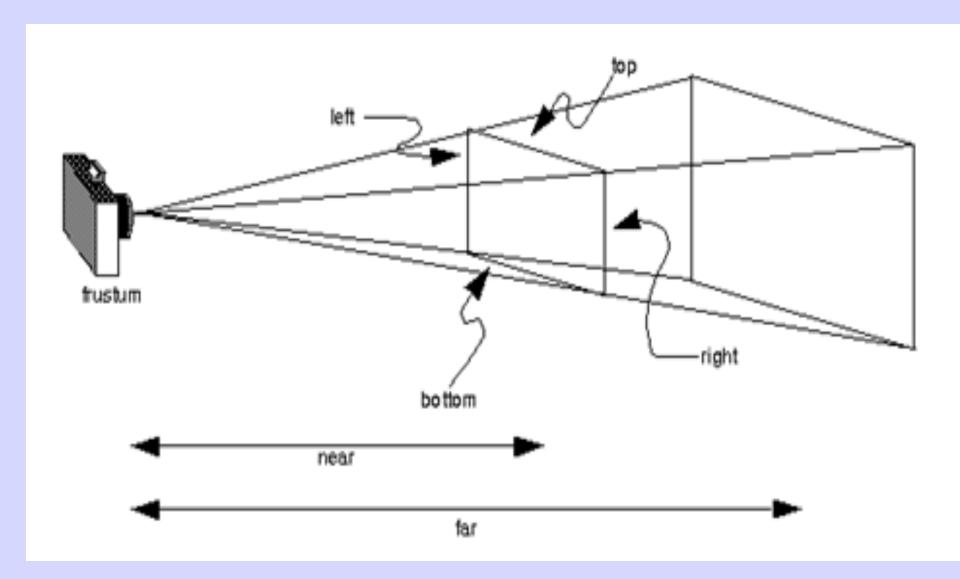
glMatrixMode(GL PROJECTION):

通常,我们需要在进行变换前把当前矩阵设置为单位 矩阵。

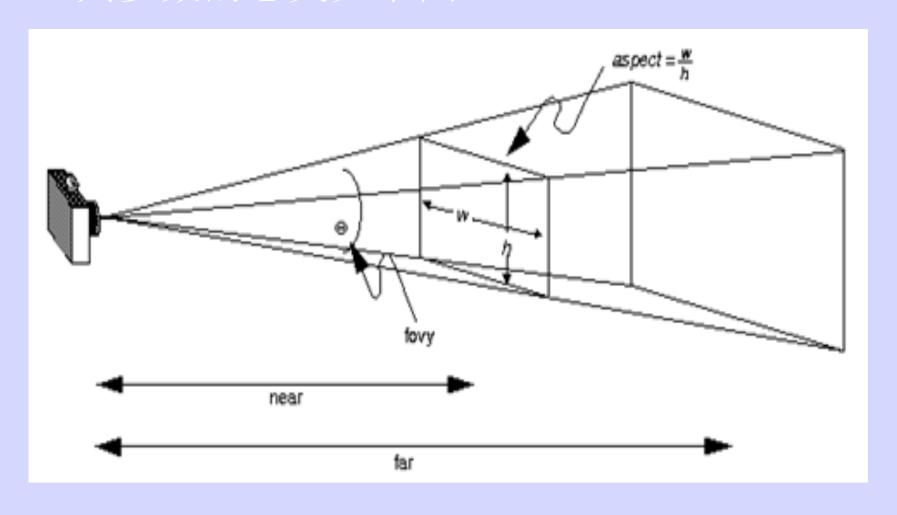
glLoadIdentity();

透视投影所产生的结果类似于照片,有近大远小的效果,比如在火车头内向前照一个铁轨的照片,两条铁轨似乎在远处相交了。

使用glFrustum函数可以将当前的可视空间设置为透视投影空间。其参数的意义如下图:

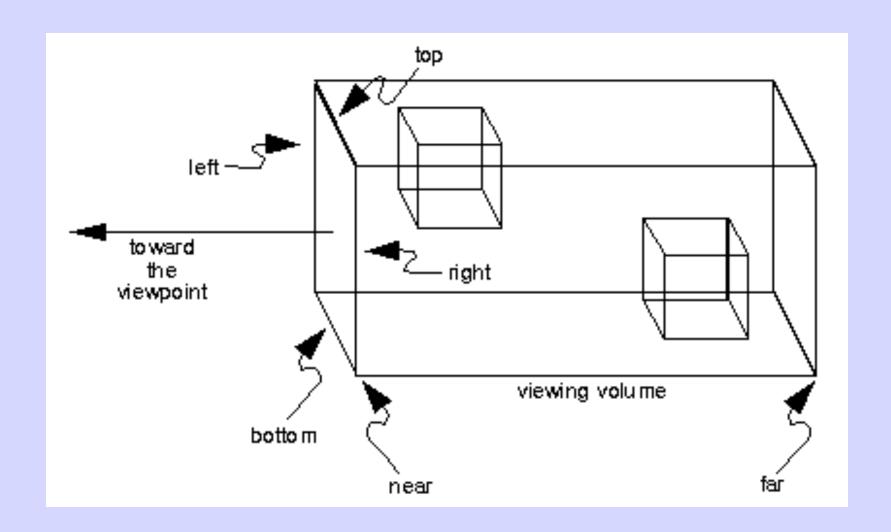


也可以使用更常用的gluPerspective函数。 其参数的意义如下图:



正投影相当于在无限远处观察得到的结果, 它只是一种理想状态。但对于计算机来 说,使用正投影有可能获得更好的运行 速度。

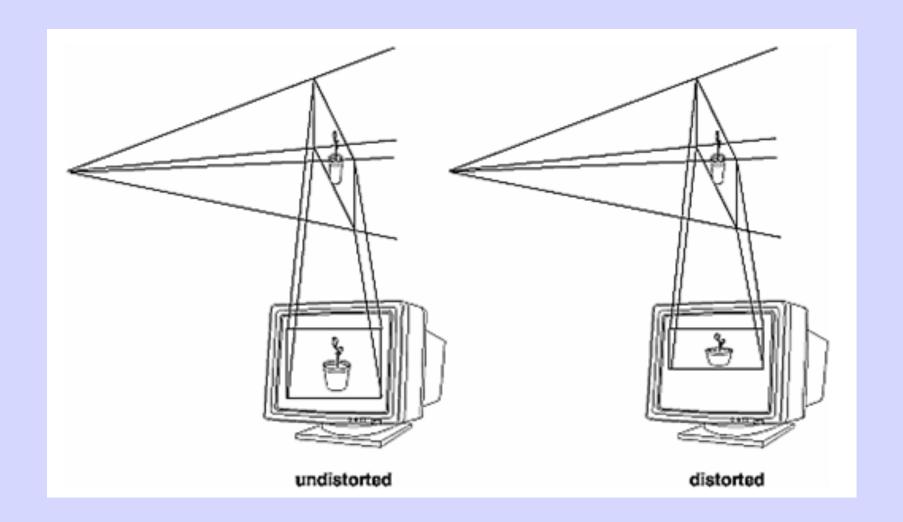
使用glOrtho函数可以将当前的可视空间设置为正投影空间。其参数的意义如下图:



如果绘制的图形空间本身就是二维的,可以使用gluOrtho2D。他的使用类似于glOrgho。

3、视口变换

当一切工作已经就绪,只需要把像素绘制到屏幕上了。这时候还剩最后一个问题:应该把像素绘制到窗口的哪个区域呢?通常情况下,默认是完整的填充整个窗口,但我们完全可以只填充一半。(即:把整个图象填充到一半的窗口内)



使用glViewport来定义视口。其中前两个参数定义了视口的左下脚(0,0表示最左下方),后两个参数分别是宽度和高度。

4、操作矩阵堆栈

我们在进行矩阵操作时,有可能需要先保存某个矩阵,过一段时间再恢复它。当我们需要保存时,调用glPushMatrix函数,它相当于把矩阵(相当于盘子)放到堆栈上。当需要恢复最近一次的保存时,调用glPopMatrix函数,它相当于把矩阵从堆栈上取下。OpenGL规定堆栈的容量至少可以容纳32个矩阵,某些OpenGL实现中,堆栈的容量实际上超过了32个。因此不必过于担心矩阵的容量问题。

通常,用这种先保存后恢复的措施,比先变换再逆变换要更方便,更快速。

注意:模型视图矩阵和投影矩阵都有相应的堆栈。使用glMatrixMode来指定当前操作的究竟是模型视图矩阵还是投影矩阵。

综合举例(robot.c)

第六章 动画

想必大家都知道电影和动画的工作原理吧?是的, 快速的把看似连续的画面一幅幅的呈现在人们 面前。一旦每秒钟呈现的画面超过24幅,人们 就会错以为它是连续的。 我们通常观看的电视,每秒播放25或30幅画面。 但对于计算机来说,它可以播放更多的画面, 以达到更平滑的效果。如果速度过慢。画面不

以达到更平滑的效果。如果速度过慢,画面不够平滑。如果速度过快,则人眼未必就能反应得过来。对于一个正常人来说,每秒60~120幅图画是比较合适的。具体的数值因人而异。

双缓冲技术

在计算机上的动画与实际的动画有些不同:实际的动画都是先画好了,播放的时候直接拿出来显示就行。计算机动画则是画一张,就拿出来一张,再画下一张,再拿出来。如果所需要绘制的图形很简单,那么这样也没什么问题。但一旦图形比较复杂,绘制需要的时间较长,问题就会变得突出。

让我们把计算机想象成一个画图比较快的人,假如他直接在屏幕上画图,而图形比较复杂,则有可能在他只画了某幅图的一半的时候就被观众看到。而后面虽然他把画补全了,但观众的眼睛却又没有反应过来,还停留在原来那个残缺的画面上。也就是说,有时候观众看到完整的图象,有时却又只看到残缺的图象,这样就造成了屏幕的闪烁。

如何解决这一问题呢?我们设想有两块画板,画图的人在旁边画,画好以后把他手里的画板与挂在屏幕上的画板相交换。这样以来,观众就不会看到残缺的画了。这一技术被应用到计算机图形中,称为双缓冲技术。即:在存储器(很有可能是显存)中开辟两块区域,一块作为发送到显示器的数据,一块作为绘画的区域,在适当的时候交换它们。由于交换两块内存区域实际上只需要交换两个指针,这一方法效率非常高,所以被广泛的采用

注意:虽然绝大多数平台都支持双缓冲技术,但这一技术并不是OpenGL标准中的内容。OpenGL为了保证更好的可移植性,允许在实现时不使用双缓冲技术。当然,我们常用的PC都是支持双缓冲技术的。

这里的"在CPU空闲的时间绘制"和我们在第一课讲的"在需要绘制的时候绘制"有些共通,都是"在XX时间做XX事",GLUT工具包也提供了一个比较类似的函数: glutIdleFunc,表示在CPU空闲的时间调用某一函数。其实GLUT还提供了一些别的函数,例如"在键盘按下时做某事"等。

到现在,我们已经可以初步开始制作动画了。 好的,就拿上次那个"太阳、地球和月亮"的 程序开刀,让地球和月亮自己动起来。

3、关于垂直同步

几乎所有的显卡都支持"垂直同步"这 一功能。

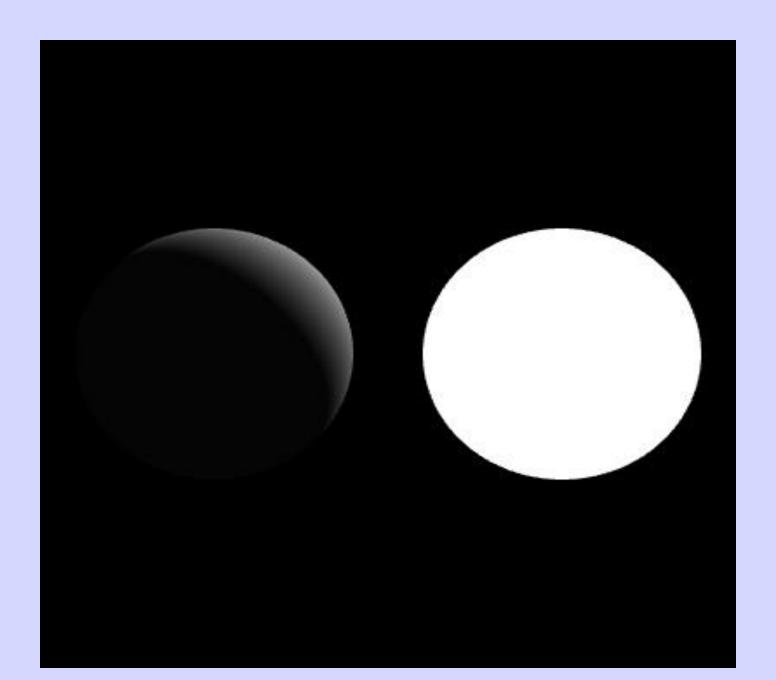
4、计算帧速

第七章 光照

从生理学的角度上讲,眼睛之所以看见各种物体,是因为光线直接或间接的从它们那里到达了眼睛。人类对于光线强弱的变化的反应,比对于颜色变化的反应来得灵敏。因此对于人类而言,光线很大程度上表现了物体的立体感。

请看图1,图中绘制了两个大小相同的白色球体。其中 右边的一个是没有使用任何光照效果的,它看起来就 像是一个二维的圆盘,没有立体的感觉。左边的一个 是使用了简单的光照效果的,我们通过光照的层次, 很容易的认为它是一个三维的物体。

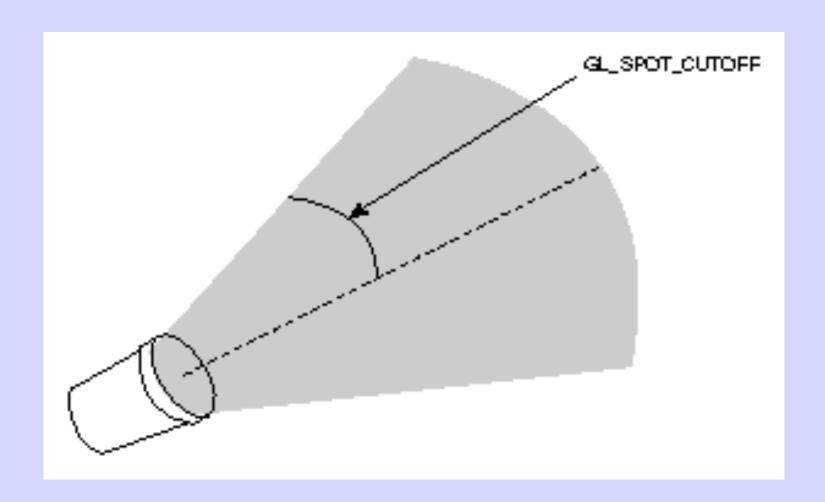
OpenGL对于光照效果提供了直接的支持,只需要调用某些函数,便可以实现简单的光照效果。



一、建立光照模型

在现实生活中,某些物体本身就会发光,例如太阳、电灯等,而其它物体虽然不会发光,但可以反射来自其它物体的光。这些光通过各种方式传播,最后进入我们的眼睛——于是一幅画面就在我们的眼中形成了。

二、法线向量



四、控制材质

- (2) GL_SHININESS属性。该属性只有一个值,称为"镜面指数",取值范围是0到128。该值越小,表示材质越粗糙,点光源发射的光线照射到上面,也可以产生较大的亮点。该值越大,表示材质越类似于镜面,光源照射到上面后,产生较小的亮点。
 - 3) GL_EMISSION属性。该属性由四个值组成,表示一种颜色。OpenGL认为该材质本身就微微的向外发射光线,以至于眼睛感觉到它有这样的颜色,但这光线又比较微弱,以至于不会影响到其它物体的颜色。
 - (4) GL_COLOR_INDEXES属性。该属性仅在颜色索引模式下使用,由于颜色索引模式下的光照比RGBA模式要复杂,并且使用范围较小,这里不做讨论。

五、选择光照模型

- 以上四方面的内容都通过同一个函数glLightModel*来进行设置。该函数有两个参数,第一个表示要设置的项目,第二个参数表示要设置的值。
 - GL LIGHT_MODEL_AMBIENT表示全局环境光线强度,由四个值组成。
 - GL LIGHT MODEL LOCAL VIEWER表示是否在近处观看,若是则设置为GL_TRUE,否则(即在无限远处观看)设置为GL_FALSE。
 - GL_LIGHT_MODEL_TWO_SIDE表示是否执行双面光照计算。如果设置为GL_TRUE,则OpenGL不仅将根据法线向量计算正面的光照,也会将法线向量反转并计算背面的光照。
 - GL LIGHT MODEL COLOR CONTROL表示颜色计算方式。如果设置为GL SINGLE COLOR,表示按通常顺序操作,先计算光照,再计算领理。如果设置为
 - GL SEPARATE SPECULAR COLOR,表示将GL SPECULAR属性分离出来,先计算光照的其它部分,待纹理操作完成后再计算GL SPECULAR。后者通常可以使画面效果更为逼真(当然,如果本身就没有执行任何纹理操作,这样的分离就没有任何意义)。

示例

第八章 纹理映射

1、启用纹理和载入纹理 就像我们曾经学习过的OpenGL光照一样。 在使用纹理前,必须启用它。OpenGL支 持一维纹理、二维纹理和三维纹理,这 里我们仅介绍二维纹理。可以使用以下 语句来启用和禁用二维纹理:

glEnable(GL_TEXTURE_2D); // 启用二维 纹理

glDisable(GL_TEXTURE_2D); // 禁用二维 _ 纹理

第四、五个参数是二维纹理像素的宽度和高度。使用纹理时要特别注意其大小。尽量使用大小为2的整数次方的纹理,当这个要求无法满足时,使用gluScaleImage函数把图象缩放至所指完的大小

无论旧版本还是新版本,都限制了纹理大小的最大值,例如,某OpenGL实现可能要求纹理最大不能超过1024*1024。可以使用如下的代码来获得OpenGL所支持的最大纹理:

GLint max; glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max);

- 第六个参数是纹理边框的大小,我们没有使用纹理边框,因此这里设置为零。
- 第七个参数表示读取的内容,例如: GL RGB就会依次读取像素的红、绿、蓝三种数据, GL RGBA则会依次读取像素的红、绿、蓝、alpha四种数据, GL RED则只读取像素的红色数据(类似的还有GL_GREEN, GL_BLUE, 以及GL ALPHA)。
- 第八个参数表示读取的内容保存到内存时所使用的格式,例如: GL_UNSIGNED_BYTE会把各种数据保存为GLubyte, GL_FLOAT会把各种数据保存为GLfloat等。
- 第九个参数表示一个指针,像素数据被读取后, 将被保存到这个指针所表示的地址。

举个例子,如果有一幅大小为width*height, 格式为Windows系统中使用最普遍的24位 BGR,保存在pixels中的像素图象。则把 这样一幅图象载入为纹理可使用以下代 码:

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR_EXT, GL_UNSIGNED_BYTE, pixels);

2、纹理坐标

3、纹理参数

在使用纹理前还有某些参数是必须设置的。 使用glTexParameter*系列函数来设置纹理参数。通常 需要设置下面四个参数:

GL_TEXTURE_MAG_FILTER: 指当纹理图象被使用到一个大于它的形状上时(即:有可能纹理图象中的一个像素会被应用到实际绘制时的多个像素。例如将一幅256*256的纹理图象应用到一个512*512的正方形),应该如何处理。可选择的设置有GL_NEAREST和GL_LINEAR,前者表示"使用纹理中坐标最接近的一个像素的颜色作为需要绘制的像素颜色",后者表示"使用纹理中坐标最接近的若干个颜色,通过加权平均算法得到需要绘制的像素颜色"。

4、纹理对象

前面已经提到过,载入一幅纹理所需要的时间是比较多的。因此应该尽量减少载入纹理的次数。如果只有一幅纹理,则应该在第一次绘制前就载入它,以后就不需要再次载入了。

但是,在每次绘制时要使用两幅或更多幅的纹理时,这个办法就行不通了。你可能会编写下面的代码:

glTexImage2D(/* ... */); // 载入第一幅纹理 // 使用第一幅纹理 glTexImage2D(/* ... */); // 载入第二幅纹理 // 使用第二幅纹理 // 当纹理的数量增加时,这段代码会变得更加复杂。

在绘制动画时,由于每秒钟需要将画面绘制数十次,因此如果使用上面的代码,就会反复载入纹理,这对计算机是非常大的负担,以目前的个人计算机配置来说,根本就无法让动画能够流畅的运行。因此,需要有一种机制,能够在不同的纹理之间进行快速的切换。

使用一个正整数来作为纹理对象的编号。 在使用前,可以调用glGenTextures来分 配纹理对象。该函数有两种比较常见的 用法:

GLuint texture_ID; glGenTextures(1, &texture_ID); // 分配一 个纹理对象的编号

或者

GLuint texture_ID_list[5]; glGenTextures(5, texture_ID_list); // 分配5 个纹理对象的编号 在分配了纹理对象编号后,使用glBindTexture函数来指定"当前所使用的纹理对象"。然后就可以使用glTexImage*系列函数来指定纹理像素、使用glTexParameter*系列函数来指定纹理参数、使用glTexCoord*系列函数来指定纹理坐标了。如果不使用glBindTexture函数,那么glTexImage*、glTexParameter*、glTexCoord*系列函数默认在一个编号为0的纹理对象上进行操作。

glBindTexture函数有两个参数,第一个参数是需要使用纹理的目标,因为我们现在只学习二维纹理,所以指定为GL TEXTURE_2D,第二个参数是所使用的纹理的编号。