OpenGL 3D

入门与提高

李 薇 徐国标 等 编

西南交通大学出版社

入门与提高
系列丛书

OpenGL 3D 入门与提高

明星谷电脑工作室　策划

李　薇　徐国标　等编
尹　皓　李　果

西南交通大学出版社

内 容 简 介

OpenGL 是输出到图形硬件的一个软件编程接口界面，该接口界面包括大约 120 条不同的命令，用来定义 3D 物体和交互式 3D 应用的各种操作。OpenGL 被设计成独立于硬件，流水线工作方式，可以在许多不同的硬件平台上实现。

本书详细介绍了 OpenGL 基本原理、计算机图形和图像方面的基础知识，并精遣了 8 个编程实例，以帮助读者有效地掌握 OpenGL。

OpenGL 3D 入门与提高

——入门与提高系列丛书

[illegible] 等编

责任编辑 吴晓[illegible]

西南交通大学出版社出版发行

（成都二环路北一段 610031）

郫县印刷厂印刷

*

开本：787×1092 1/16 印张：31

字数：786 千字 印数：1—5000 册

1998 年 7 月第 1 版 1998 年 7 月第 1 次印刷

ISBN 7-81057-180-X/T · 270

定价：50.00 元

目　录

第一章　OpenGL 简介

[本章目的]

通过阅读本章，你将能够达到：

- 了解 OpenGL 所提供的一般功能
- 理解绘制不同程序的复杂性
- 理解 OpenGL 程序的基本功能
- 认识 OpenGL 命令语法结构
- 了解如何利用 OpenGL 程序实现动画的一般概念

[本章内容]

本章介绍了 OpenGL 的一般知识。它有以下几个主要部分：

- “什么是 OpenGL”一节解释了 OpenGL 的概念和功能，即它能用来干什么，不能干什么以及如何使 OpenGL 工作。
- “简单的 OpenGL 程序”一节给出了一个较小的 OpenGL 程序实例，并简要分析讨论了该程序。这一节中也定义了一些基本的计算机图形学术语。
- “OpenGL 命令语法”一节解释了 OpenGL 命令常用的一些习惯和语法规范。
- “OpenGL 状态机制”一节讨论了 OpenGL 中的状态变量使用和与之相关的状态查询，激活/关闭该状态的命令。
- “与 OpenGL 有关的库文件”讨论了与 OpenGL 有关的库程序集合，包括本书中经常用到的辅助程序库的使用。
- “动画”一节解释如何在屏幕上创建图片并使之移动或动画的一般概念。

§1.1　什么是 OpenGL

OpenGL 是输出到图形硬件的一个软件编程接口界面。该接口界面包括大约 120 条不同的命令，用来定义 3D 物体和交互式 3D 应用的各种操作。

OpenGL 可以工作于网络模式下，即你所创建的图形显示可以不与运行该程序的机器相同。在多台机器互连的网络环境中，运行 OpenGL 程序的机器称为客户端（Client），接收绘制命令执行绘图工作的机器称为服务器端（Server）。传输 OpenGL 命令的数据格式是一样的（也称之为协议），因此即使在不同类型的机器上运行客户端或服务器端程序，OpenGL 程序也能正确运行。如果 OpenGL 程序不在客户/服务器方式运行时，该机器是客户也是服务器。

OpenGL 被设计成独立于硬件，流水线工作方式，可以在许多不同的硬件平台上实现。为

了达到这个目的，在 OpenGL 中没有任何窗口操作和获取用户输入等命令函数，你必须在所使用的特定硬件平台上考虑窗口操作细节。同样地，OpenGL 也没有提供高级命令函数来定义复杂 3D 物体，像汽车、身体的某个部分、飞机等。使用 OpenGL，你必须从几何图元（Geometric primitives）的点、线和多边形来构造你希望的模型。OpenGL 的高级图形库版本 Open Inventor 实现了一些复杂物体定义功能，后面章节对 Open Inventor 有进一步的介绍。

上面你已经知道 OpenGL 不能做什么，这里是它能够干的事情。让我们来看书中的插图，它们演示了 OpenGL 典型使用的功能。下面简单地讨论了这些图像的生成方法。

1. 构造三维物体的形状。需要各种几何图元和数学描述信息来合成，在 OpenGL 中，图元包括点、线、多边形、图像和位图等。

2. 在三维空间中放置该物体。其中包括构成合成场景时所选择的视点、视角等信息。

3. 计算出所有物体的颜色。物体表面的颜色可以由应用程序显式给出，并根据光照条件计算得到，也可以通过给物体施加纹理来实现。

4. 把物体的数学描述和相应的色彩信息转换成屏幕上的像素值。OpenGL 把这种处理称之为光栅化（Rasterization）。

在上述步骤中，OpenGL 也执行了其它的一些操作，诸如消去物体因被其前面物体遮住而不可见的部分，这样省去隐藏面的绘制，可以加快绘制速度，提高系统的性能。另外，在绘制图像光栅化后在屏幕上显示出来之前，你也可以根据需要操纵像素数据。

§ 1.2 简单的 OpenGL 程序

既然 OpenGL 图形系统可以做如此众多的工作，因而 OpenGL 程序也会是相当复杂。然而，一个 OpenGL 程序的基本结构可以非常简单，它的任务是初始化某些状态变量来控制 OpenGL 如何绘制以及定义要绘制的物体。

在你阅读 OpenGL 例子程序之前，让我们来学习几个术语。

- 绘制（Rendering）指的是计算机根据模型创建三维图像的处理过程。
- 模型（Models）也称物体（Objects），是由几何图元组成的，几何图元包括点、线、多边形，所有这些都由顶点（Vertices）来定义。

最后绘制的图像是屏幕上的一系列像素值。因此一个像素（Pixel）点是图像元素的简称，是显示设备能够显示到屏幕上的最小可见单位。有关像素的信息，如它们支持什么颜色，由系统显示存储器的位面数来决定。一个位面（Bitplane）是内存中的一块区域，它保留每个像素点在屏幕上的一位信息，该位可以决定某一像素点应该显示红色或者其它颜色。位面的总和构成帧存（Framebuffer），其中保存了图形显示所需要的屏幕上所有像素值。

现在让我们来看一个 OpenGL 程序。程序清单 1－1 程序在黑色背景下绘制一白色矩形，结果如图 1－1 所示。

程序清单 1－1　一个简单的 OpenGL 程序

```
#include <whatever You Need.h>
main () {
    Open WindowPlease ();
    glClear (0.0, 0.0, 0.0, 0.0);
```

```
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
    glBegin (GL_POLYGON);
    glVertex2f (-0.5, -0.5);
    glVertex2f (-0.5, 0.5);
    glVertex2f (0.5, 0.5);
    glVertex2f (0.5, -0.5);
  glEnd ();
  glFlush ();
  Keep_The_Window_On_The_Screen_For_A_While ();
}
```

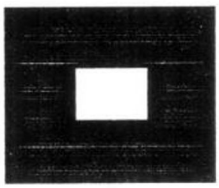

图 1-1 黑色背景下绘制一白色矩形

程序中 Main () 函数第一行表示在屏幕上打开一个窗口，Open WindowPlease () 函数是与窗口系统有关的程序，与运行的系统平台有关，不是 OpenGL 的命令函数。下面两行 OpenGL 命令其作用是把窗口清成黑色背景：glClearColor () 函数设定清除窗口背景的颜色，glClear () 完成清除窗口工作。一旦清除的颜色被设置，以后调用 glClear () 函数时窗口都会被清除该颜色。因此，清屏颜色也可以通过调用 glClearColor () 函数改成其它颜色。同样地，glColor3f () 函数创建用来绘制物体的颜色，在本例中，该颜色被置成白色。以后绘制物体都用该颜色绘制，直到重新用其它颜色设置 glColor3f () 函数调用。

接下来一行 OpenGL 命令 glOrtho () 定义了坐标系统。OpenGL 由此决定最终的图像以及如何把该图像映射到屏幕中。下一段语句，用 glBegin () 和 glEnd () 括起部分，用来定义要绘制的物体。在本例子中，是一个由四个顶点组成的多边形。其中多边形的角由 glVertex2f () 函数来定义。正如你所猜想的那样，其中的参数是 (x, y) 坐标。不难看出，该多边形是个矩形。

最后，glFlush () 参数确认所有的绘制命令已经被实际执行，而没有被暂时存在缓冲区中等待其它 OpenGL 命令输入。

KeepTheWindowOnTheSoreenForAWhile () 函数也不是 OpenGL 命令，它涉及用户界面输出，该例程目的是让该图像在屏幕中显示停留一段时间，而不会马上消失。

§ 1.3 OpenGL 命令语法

也许你从上节的程序例子中不难发现，OpenGL 命令函数均使用 gl 作为前缀首字母大写而构成。如 glClearColor () 等语句。同样地，OpenGL 以 GL_为前缀定义常量，使用大写字

母方式，中间用下划线作单词间的分隔符，象 GL_COLOR_BUFFER_BIT 等。

你可能也发现一些语句中出现类似无关紧要的命令符号，如 glColor3f () 命令函数中的 3f 等。确定命令名中的 Color 部分地决定该命令用来设置当前的颜色。然而，事实上不止一种命令定义用来设置颜色，其中的 3 后缀表示给出了三个参数，另外一条 glColor 命令可以使用四个参数。f 定义了参数所使用的是浮点数。某些 OpenGL 命令可以在其参数中接收多达八种不同的数据类型。这里后缀字符对应的 ANSI C 数据类型表见 1－1 所示。需要注意的是，OpenGL 的实现版本不同，并不一定遵从该规范，如 C＋＋或者 Ada 版本等。

表 1－1 命令后缀和参数数据类型

OpenGL 后缀符	数据类型	通常相应的 C 语言定义	OpenGL 类型定义
b	8 位整数	unsigned char	GLbyte
s	16 位整数	short	GLshort
i	32 位整数	long	GLint, GLsizei
f	32 位浮点数	float	GLfloat, GLclampf
d	64 位浮点数	double	GLdouble, GLclampd
ub	8 位无符号整数	unsigned char	GLubyte, GLboolean
us	16 位无符号整数	unsigned short	GLushort
ui	32 位无符号整数	unsigned long	GLunit, GLenum GLbitfield

从上表不难看出，命令

glVertex2i (1, 3);

和命令

glVertex2f (1.0, 3.0);

功能是相同的，其不同之处是第一条语句以 32 位整数方式定义顶点坐标，而第二条语句以单精度浮点数方式给出参数。

有一些 OpenGL 命令其最后字符为 v，表示命令接收参数是向量的指针而不是一系列单独的参数。在 OpenGL 语言中，有许多命令既有矢量格式又有非矢量格式的定义方式，还有其它的命令只能提供单个参数的格式或者至少部分参数必须以向量的格式给出。

下面的语句行说明了你既可以用向量参数也可以由非向量参数格式来设置当前的颜色。

```
glColor3f (1.0, 0.0, 0.0);
float color_array [] = {1.0, 0.0, 0.0};
glColor3fv (color_array);
```

考虑到讨论的简洁性，在本章的其余章节中（实际代码例子除外），OpenGL 命令格式将只给出其基本名字部分，＊号用来指示该命令可能有多种定义方式。例如，glColor＊() 命令函数代表用来设置命令的所有变种定义方式。如果我们需要指定一命令的某一格式，讨论时使用了必要的后缀来定义。例如，glVertex＊v () 命令代表定义顶点坐标的所有向量参数定义格式。

最后，OpenGL 定义了特殊常量 GLvoid。如果你使用 C 语言进行 OpenGL 编程，可以使用 GLvoid 常量来代替 void。

§ 1.4 OpenGL 的状态机制

OpenGL 的工作方式就像模拟状态机器一样。你可以通过各种状态或模式设置，在你重新改变状态之前它们将一直有效。正如在上一节中所述，当前设置的颜色就是一状态变量。你可以设置当前颜色为白色、红色或者别的颜色，而后所画的每个对象将一直用该颜色绘制，除非你又改用了其它颜色为止。这里讨论的当前颜色仅仅是 OpenGL 保护的许多状态变量中的一种。其它的控制状态包括当前视图和投影变换，线和多边形绘制模式，光源的位置和特性以及要绘制对象的材料特性等。大多数状态变量可以用命令 glEnable () 或 glDisable () 来打开或关闭。

在 OpenGL 中，每个状态变量都有其默认值，并且在程序的任何地方你都可以查询到系统中每个变量的当前值。通常情况下，你可以用下列四个命令来获取其状态变量的值：glGetBooleanv ()，glGetDoublev ()，glGetFloatv () 和 glGetIntegerv ()。到底选择哪个命令应该根据你要获取返回值的数据类型来决定。有一些决定变量有其特定的查询命令，如 glGetLight * ()，glGetError () 和 glGetPolygonStipple () 命令等。另外，使用 glPushAttrib ()，glPopAttrib () 命令，你可以存储和恢复最近的状态变量值。只要有可能，你都应该使用这些查询命令，因为它们比其它的查询语句更有效。

在附录 B 中完整地列出了 OpenGL 状态变量。对每个变量，同时还列出了 glGet * () 命令和变量返回值，所属的属性类和变量返回值。

§ 1.5 与 OpenGL 有关的库文件

OpenGL 提供了功能强大的图元绘制命令，所有高级的目标绘制都通过这些命令来实现。因此，你也许想在 OpenGL 基础上编写自己的库文件来简化编程工作。比如说，你可能会考虑编写一些例程以便让 OpenGL 程序与你的窗口系统相挂接。实际上 OpenGL 提供了几个库例程来提供特殊的功能。在下列讨论的库文件中，头两个库文件在每个 OpenGL 版本中都已经提供，第三个库结合讨论教学使用，可以通过 ftp 获取，第四个库 Open Inventor 是独立的产品。

• OpenGL 功能库 (GLU)：包含几个使用低级 OpenGL 命令的例程，可以用来创建指定视角的投影模型，执行多边形镶嵌和着色表面。该库作为 OpenGL 实现版本的一部分。GLD 例程使用 glu 命令前缀。

• OpenGL X 窗口系统扩展库 (GLX)：提供创建 OpenGL 上下文和相关的 X 窗口系统的可画窗口。它提供了 OpenGL 与 X 窗口系统的相挂接功能。GLX 例程使用 glx 命令前缀。

• OpenGL 编程指南辅助库：它是为了使程序例子更简单，便于讨论而设计的。它使用 aux 命令前缀，后面将对它作进一步讨论。

• Open Inventor 是基于 OpenGL 的面向对象工具包：它提供对象和创建交互式三维图形应用的方法等。该产品由 SGI 公司用 C++编写而成，Window NT 版本由 TGS 公司提供。

Open Inventor 提供预建的对象和用于交互的内置事件模型，用于创建和编辑三维场景的高级应用部件，以及打印对象和与其它图形格式交互数据的能力。

一、OpenGL 编程辅助库

正如你从上面看到的那样，OpenGL 包含了各种 3D 绘制命令，但它独立于任何窗口系统和操作系统。因此，它没有任何打开窗口和从键盘或鼠标中获取输入的命令函数。然而，编写一个完整的图形应用程序不可能没有窗口操作，大多数有趣的程序也需要有用户输入和调用操作系统或窗口系统提供的功能。

另外，由于 OpenGL 的绘制命令限于几种简单几何图元的生成，像点、线和多边形等，因此在辅助库中包含了几个例程用来创建更复杂的三维物体，有球体、圈和茶壶。这样讨论的程序输出将会更加生动有趣。在多数情况下，本书中使用的辅助库程序可以在任何带有辅助库的系统下运行而不必作任何修改。

辅助库有意把它写得很简单，因此在它上面去创建庞大复杂的应用程序是不现实的。但是它可以作为学习 OpenGL 编程的最佳起点。下面我们简要地讨论一下辅助库选用的命令以便更好地阅读本书中其它程序例子。

二、窗口管理功能函数

辅助库提供了三个例程来初始化和打开窗口：

• auxInitWindow () 在屏幕上打开一窗口。该窗口按 ESC 键可以退出，并且把窗口背景色设置成黑色。

• auxInitPosition () 告诉 auxInitWindow () 函数在屏幕上的哪一点定位窗口。

• auxInitDisplayMode () 告诉 auxInitWindow () 函数创建的是 RGBA 还是颜色索引窗口。你也可以指定单/双缓冲窗口。在颜色索引方式下，你需要载入某种颜色到颜色查找表中可以使用 auxSetOneColor () 命令来实现。另外，你还可以用该函数来指定窗口的深度、模板等信息。

三、用户输入处理函数

下列例程用来登记指定事件发生时要调用的回调函数。

• auxReshapeFunc () 当窗口大小改变、移动或暴露时应该执行的命令函数。

• auxKeyFunc () 和 auxMouseFunc () 允许你在键盘或鼠标输入时对相应的键作出处理。

四、三维物体绘制函数

辅助库包括绘制下列三维物体的例程：

球体	立方体	十二面体
圈	柱体	二十四面体
锥体	八面体	茶壶

上述物体你既可以绘制成网格（wireframe）方式也可以是带平面法矢量的实体方式。下面是绘制球体和图的相应命令函数原型：

```
void auxWireSphere (GLdouble radius);
void auxSolidSphere (GLdouble radius);
void auxWireTorus (GLdouble innerRadius, GLdouble outer Radius);
```

```
void auxSolidTorus (GLdouble innerRadius, GLdouble outer Radius);
```

所有这些物体都以窗口中心点为原点绘出。当用单位缩放因子绘制时，这些模型都可以放入到－1 到 1 的立方体内。使用命令参数可以缩放该物体的大小。

五、后台进程管理函数

你可以指定没有事件发生时执行某些功能，auxIdleFun () 命令函数接收函数指针作了参数。传入参数 0 可以使该功能执行停止。

六、程序运行函数

在程序 main () 中，调用 auxMainLoop () 时传递绘制 3D 目标的例程指针。程序清单 1－2 演示了使用辅助库如何正确实现程序清单 1－1 的具体功能：

程序清单 1－2 使用 Aux 库改写的简单 OpenGL 程序清单

```
#indude <GL/gl. h>
#include " aux. h"
int main (int argc, char * * argv)
{
    auxInitDisplay_Mode (AUX_SINGLE|AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv [0]);
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glMatrixMode (GL_PROJECTION);
    glLoad Identity ();
    glOrtho (－1.0, 1.0, －1.0, 1.0, －1.0, 1.0);
    glBegin (GL_POLYGON)
        glVertex2f (－0.5, －0.5);
        glVertex2f (－0.5, 0.5);
        glVertex2f (0.5, 0.5);
        glVertex2f (0.5, －0.5);
        glEnd ();
        glFlush ();
        Sleep (10);
}
```

§1.6 动　　画

在图形系统上你所能做的最激动人心的事情是使画图运动起来。无论是你是工程师，正尝试检查你设计的机械另件的各个面，还是正在利用模拟机学习起飞的飞机驾驶员，或者只是个电脑游戏迷，很明显动画在图形显示系统中占有相当重要的部分。

众所周知，电影运动画面是由一系列静止的画面组成，每秒前进 24 格，即以每秒 24 幅画面的速度投影到屏幕上。每一帧图画完全移到镜头后面时，快门打开，该帧画面被投影到

屏幕上，而在下一帧移到镜头后面的前进过程中，快门暂时关闭，如此循环反复。尽管每秒钟你所看到的是 24 幅静止不同的图像，由于人脑中的暂留视觉效果使人感到连续平滑的运动效果。在现代电影放映机中通常每幅画面投影 2 次，即以每秒 48 幅的速度连续放映以减少闪烁感。而计算机图形屏幕刷新大约是每秒 60～76 次，有些甚至达到每秒 120 次。显然，6 次/秒没有 30 次/秒平滑，而 120 次/秒则更好。刷新率超过 120 次/秒则已经超过了消失返回点人眼所能感觉的范围。

制作运动图像工作关键之处是当它显示时，每一帧都需彻底重画。如果要为数百万帧电影用电脑制作动画，那么程序流程可用下面的语句来表示：

```
open_window ();
    for (i=0; i<1000000; i++) {
    clear_the_window ();
    draw_frame (i);
    wait_until_a_24th_of_a_second_is_over ();
}
```

清除窗口和绘制帧画面需要额外的系统开销时间，该时间与 1/24 秒的接近程度将直接影响系统显示性能。假设整个绘制时间耗时接近 1/24 秒，第一帧画面显示后，在下一帧画面显示前都需要不断地被清除，因此通常会显示出不稳定的闪烁感。原因是在 1/24 秒的多数时间里你的眼睛所看到的是清除了背景的图像而不是绘制出来的画面。问题的根源是程序不显示完整的画面，而你所看到的是绘制的过程。

解决这种问题的简单方案是双缓冲机制，采用硬件或软件来提供两个完整的颜色缓冲区。一个缓冲区显示时另外一个用来绘制。当绘制帧完成后，两个缓冲区相互交换，原来显示的缓冲区现在用来绘制下一帧画面。这个过程好象是一个只有两帧画面的电影放映机。当一帧被投向屏幕时，艺术家快速擦除和重画不被显示的那个画面。只要这个工作足够快，观众将感觉不到这种装置与所有帧都已画好简单播放的电影放映机有何不同。有了双缓冲机制，每一帧都只有当绘制完后才显示出来，观察者却不能看到绘制的过程。

对前面例子采用双缓冲存储机制动态平滑地显示图像的程序应改为：

```
Open_window_in_doubl_buffer_mode ();
for (i=0; i<1000000; i++) {
    Clear_the_window ();
    draw_frame (i);
    swap_the_buffers ();
}
```

除了简单地倒换可见帧和可画帧缓冲区外，swap_the_buffers () 例程等待当前屏幕刷新周期结束时才切换前后屏，这样以前的缓冲区内容才被保证完整地显示。假设系统的显示刷新速度是每秒 60 次，那么每秒钟你可以达到的最大帧率只能是 60 帧，并且如果所有帧能够在 1/60 秒内清除和画出，则动画在该速率下能够平滑显示。

然而在系统实现时最容易发生的是帧内容太复杂无法在 1/60 秒内完成，因此每帧显示不止一次。举例来说，如果需要 1/45 秒绘制完一帧，每秒钟达到 30 帧的显示速度，则每帧图

形的空闲时间有 1/30－1/45＝1/90 秒。虽然 1/90 浪费不算大，然而在每隔 1/30 秒都有 1/90 秒的浪费，因此实际上有 1/3 的时间被浪费掉了。

另外，由于视频刷新率是常量，可能带有一些不可预料的性能影响。例如，在 1/60 秒刷新率的显示器上，你可以运行每秒 60 帧，30 帧，20 帧，15 帧，12 帧等的图像（60/1，60/2，60/3，60/4，60/5，…）。这意味着如果你正编写一应用程序并且逐步添加新特征（譬如设计飞行模拟机程序，你添加了地面视景部分），刚开始时添加的每个特征对整个性能没有影响，你始终可以获得每秒 60 帧的速率。然后，突然之间，由于你添加一新的显示特征，显示性能减半了，因为系统不能在 1/60 秒内完成整个视景的绘制。同样地当绘制工作量每帧超过 1/30 秒时，显示性能只能达到 20 帧/秒，下降了 33%。

另一种麻烦的情况是场景复杂性接近任何整数倍时间时（1/60 秒，2/60 秒，3/60 秒等等），那么由于随机性，有些帧稍高于该时间而某些帧又低于该时间，从而导致帧率不稳定，造成视觉上的抖动现象。在这种情况下，如果你不能简化场景提高帧率时，其它的解决方案只能为所有帧给某些延时以保证每一帧在某一较低帧率下都能正常显示。如果帧间有剧烈不同的复杂度，则需要有更高级的专门的解决方案。

实际的动画程序实现结构与上述描述并没有什么不同。通常每帧的整个显示缓冲区都被重画，因为，这比要指出屏幕中的哪一部分需要重画容易得多。尤其像三维飞行模拟机中由于飞机方向的轻微改变造成窗口外视景的所有位置都有轻微的改变。

在很多动画程序中，3D 场景上的物体只是需要简单地用不同的变换重画，像观察视点的移动，汽车走下坡或物体轻微地转动等。如果显示结构有相当大的变化并需要有相当多的重新计算时，帧率通常要下降。不过需要记住的是，swap_the_buffers () 例程后的空闲时间通常可用于这种计算。

OpenGL 并没有 swap_the_buffers () 命令函数，因为该特征并不是所有的硬件都拥有，并且它与实现的窗口系统密不可分。然而在 Xwindows 系统中，GLX 提供了这条命令，其函数原型为：

```
void glXSwapBuffers (Display * dpy, Window window);
```

程序清单 1－3 演示了利用 glXSwapBuffers () 函数连续不断地旋转一正方形的例子，结果如图 1－2 所示：

程序清单 1－3 双缓冲程序例子：旋转的正方形

```
#include<GL/gl.h>
#include<GL/glu.h>
#include<GL/glx.h>
#include" aux.h"
static GLfloat spin=0.0;
void display (void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glPushMatrix ();
    glRotatef (spin, 0.0, 0.0, 1.0);
    glRectf (－25.0, －25.0, 25.0, 25.0);
    glPopMatrix ();
```

```
    glFlush ();
    glxSwapBuffers (auxXDisplay (), auxXWindow ());
}
void spinDisplay (void)
    {
    spin=spin+2.0;
    if (spin>360.0
    spin=spin-360.0;
    display ();
}
void startIdleFunc (AUX_EVENTREC *event)
{
    auxIdleFunc (spinDisplay);
}
void stopIdleFunc (AUX_EVENTREC *event)
{
    auxIdleFunc (0);
}
void myinit (void)
    {
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f (1.0, 1.0, 1.0);
    glShadeModel (GL_FLAT);
}
void myReshape (GLsizei w, GLsizei h)
    {
    glViewport (0, 0, w, h);
    glMatrixMode (GL_PROJECTION);
    glLoodIdentity ();
    if (w<=h)
        glOrtho (-50.0, 50.0, -50.0* (GLfloat) h/ (GLfloat) w,
          50.0* (GLfloat) h/ (GLfloat) w, -1.0, 1.0);
else
glOrtho (-50.0* (GLfloat) w/ (GLfloat) h, 50.0* (GLfloat) w/ (GLfloat) h, -50.0, 50.0, -
        1.0, 1.0);
glMatrixMode (GL_MODELVEW);
glLoodIdentity ();
}
int main (int argc, char * * argv)
{
    auxInitDisplayMode (AUX_DOUBLE|AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv [0]);
```